PRG (E.T.S. de Ingeniería Informática) - Curso 2014-2015

Práctica 4. Tratamiento de excepciones y ficheros

(3 sesiones)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Contexto y trabajo previo **2**

2. Planteamiento del problema **2**

3. Detección de errores lógicos y en tiempo de ejecución **3**

4. Tratamiento de excepciones predefinidas **4**

5. Tratamiento de excepciones definidas por el usuario **6**

6. Leer/escribir desde/en un fichero de texto **7**

7. Leer/escribir desde/en un fichero binario de objetos **10**

8. Evaluación **13**

1. Contexto y trabajo previo

En el marco académico, esta práctica corresponde al "*Tema 3. Elementos de la POO: herencia y tratamiento de excepciones*" y al "*Tema 4. E/S: ficheros y flujos*". El objetivo principal que se pretende alcanzar con ella es que refuerces y pongas en práctica los conceptos introducidos en las clases de teoría sobre el tratamiento de excepciones y la gestión de la E/S mediante ficheros y flujos. En particular, al finalizar esta práctica debes ser capaz de:

- Lanzar, propagar y capturar excepciones local y remotamente, tanto excepciones predefinidas como excepciones definidas por el usuario.
- Leer/escribir desde/en un fichero de texto.
- Leer/escribir desde/en un fichero binario de objetos.
- Tratar las excepciones relacionadas con la E/S.

Para ello, durante las tres sesiones de prácticas, se va a trabajar con una aplicación que gestiona cuentas bancarias, experimentando con capturar y lanzar excepciones, y añadiendo la posibilidad de realizar entrada y salida de datos con ficheros.

Para que aproveches al máximo las sesiones de prácticas, te aconsejamos que, antes de la primera sesión realices una lectura comprensiva de las secciones 2 a 5 de este boletín e intentes resolver las actividades 1 y 2. Y antes de la sesión 2 completes la lectura del boletín.

2. Planteamiento del problema

Como ya hemos adelantado, en esta práctica se trabajará con una aplicación que simula la gestión de las cuentas de un banco. La aplicación, inicialmente, consta de las siguientes clases:

- La clase de utilidades `LecturaValida` que permite realizar la lectura validada de datos de tipos primitivos desde la entrada estándar.
- La clase `Cuenta` que permite representar una cuenta bancaria mediante los atributos `saldo` (`double` que indica el saldo disponible en la misma) y `numCuenta` (`int` que representa el número de cuenta). De una cuenta se pueden consultar sus datos, obtener una `String` de los mismos, ingresar y retirar una cantidad de dinero.
- La clase `Banco` que permite representar un número máximo de cuentas (`MAX_CUENTAS`) mediante los atributos `cuentas` (un array de objetos `Cuenta`) y `numCuentas` (`int` que indica el número actual de cuentas en el banco y también el primer índice libre del array `cuentas`). De un banco se puede consultar cuántas cuentas tiene, comprobar si existe alguna cuenta con un número dado, añadir una nueva cuenta y obtener una `String` con la información de todas sus cuentas.
- La clase `GestorBanco` que permite gestionar los ingresos y retiradas de dinero de las cuentas de un banco. De todas ellas, en un momento determinado de la ejecución, solo puede trabajar con una, la cuenta activa. Así, esta clase permite crear o buscar una cuenta bancaria, que pasa a ser la cuenta activa; hacer ingresos y retiradas de dinero de la cuenta activa, consultar sus datos y los de todas las cuentas del banco.

Para empezar

- Crea un proyecto *BlueJ* `pract4` específico para esta práctica. Copia en `pract4`, desde `Recursos/Laboratorio/Práctica 4` de `PoliformaT` de `PRG`, los ficheros de código `SaldoInsuficienteException.java`, `LecturaValida.java`, `Cuenta.java`, `Banco.java` y `GestorBanco.java`, y los ficheros de texto `cuentas.txt` y `badinput.txt`.

3. Detección de errores lógicos y en tiempo de ejecución

Una vez resueltos los errores de compilación, es posible que en nuestro código tengamos errores de ejecución que provoquen un mal funcionamiento del mismo. Se distingue entre los denominados errores en tiempo de ejecución y errores lógicos. Provocando los primeros la detención de la ejecución, mientras que los segundos, los más difíciles de descubrir, consisten en que los resultados obtenidos, o los procesos realizados, por el programa o una parte del mismo no son correctos aunque el programa puede parecer que funciona correctamente.

Actividad 1: detección de errores lógicos en la clase `Cuenta`

- Revisa la clase `Cuenta` y fíjate en las precondiciones del constructor y de los métodos `ingresar(double)` y `retirar(double)`.
- Crea, en el *banco de objetos* (*Object Bench*) de *BlueJ*, un objeto `Cuenta` con un número de cuenta de 3 dígitos y un saldo negativo. ¿Puedes hacerlo?
- Invoca al método `ingresar(double)` con una cantidad negativa. ¿Es posible?
- Invoca al método `retirar(double)` con una cantidad superior al saldo de la cuenta. ¿Qué ocurre?

Son errores lógicos que dan lugar a resultados no deseados. Para evitarlos, debemos asegurarnos de que los argumentos cumplen las precondiciones. En la clase `LecturaValida` dispones de algunos métodos para realizar la lectura, desde la entrada estándar, de datos válidos de tipos `int` y `double`.

Actividad 2: detección de errores en tiempo de ejecución en la clase `LecturaValida`

- Fíjate en el método `leerDoublePos(Scanner,String)` que permite realizar la lectura de un valor de tipo `double` ≥ 0, con un bucle `do-while` que se repite mientras el valor leído sea < 0. El segundo argumento de tipo `String` es el mensaje de petición del valor.
- Prueba este método en la *zona de código* (*Code Pad*) de *BlueJ*. Para ello, ejecuta las instrucciones siguientes:

```
import java.util.*;
Scanner t = new Scanner(System.in).useLocale(Locale.US);
double realPos = LecturaValida.leerDoublePos(t,"Valor: ");
```

- Desde la *ventana del terminal* de *BlueJ*, introduce valores reales negativos. Observa que la ejecución del método no acaba hasta que no introduces un valor positivo o cero.

- Vuelve a ejecutarlo. ¿Qué ocurre si no introduces un valor de tipo `double`, por ejemplo, si introduces el carácter '`k`'?

La ejecución se detiene y se muestra un mensaje indicando qué ha ocurrido y en qué instrucción del código. Se trata de un error en tiempo de ejecución o *excepción*. En las secciones que siguen veremos cómo gestionar este tipo de errores.

4. Tratamiento de excepciones predefinidas

Como ya sabes, cuando se produce un fallo o una anomalía en la ejecución de un programa Java, ocurre una *excepción*. El código que causó dicho fallo inmediatamente deja de ejecutarse y Java intenta manejar la excepción, buscando el código que puede resolverlo (en el método donde se produjo el fallo o hasta encontrar un método que lo resuelva). Si no puede encontrar ese código, el programa se detiene y se muestra un mensaje describiendo qué falló y dónde ocurrió el problema.

En Java se puede distinguir entre excepciones *checked* o comprobadas, cuyo tratamiento (captura o propagación) es obligatorio, y excepciones *unchecked* o no comprobadas, en las que su tratamiento no es obligatorio pero sí es posible. El tratamiento de excepciones es una forma de que el programa se pueda recuperar del fallo o, al menos, terminar de una manera aceptable.

En esta sección te proponemos una serie de actividades que te guiarán en el tratamiento de algunas excepciones predefinidas en Java, mediante su captura utilizando un bloque `try-catch-finally`.

Actividad 3: análisis del método `leerInt(Scanner,String)`

- Fíjate ahora en el método `leerInt(Scanner,String)` de la clase `LecturaValida` que permite realizar la lectura de un valor de tipo `int`. La lectura se realiza con el método `nextInt()` de la clase `Scanner`, que puede lanzar la excepción `InputMismatchException` si el valor introducido por el usuario no es un entero.

- Consulta la documentación del método `nextInt()` (y las excepciones que puede generar) en el API de Java de la clase `Scanner`:
`http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

- Consulta también la documentación de la clase `InputMismatchException`, comprobando que es una excepción *unchecked* (derivada de `RuntimeException`) y que su situación en la jerarquía de clases coincide con la que se muestra a continuación.

```
java.lang.Object
 |
 +--java.lang.Throwable
     |
     +--java.lang.Exception
         |
         +--java.lang.RuntimeException
            |
            +--java.util.NoSuchElementException
               |
               +--java.util.InputMismatchException
```

- El método `leerInt(Scanner,String)` captura (`catch`) este tipo de excepción, mostrando un mensaje de error para indicar al usuario qué acción correctiva es necesaria.
- En la *zona de código* de *BlueJ*, ejecuta el método `leerInt(Scanner,String)` y, desde la *ventana del terminal*, introduce un valor no entero, por ejemplo, un valor real. Observa el mensaje mostrado y que la ejecución no acaba hasta que no introduzcas un valor entero.
- Fíjate ahora en la cláusula `finally` del método `leerInt(Scanner,String)`. Incluso cuando se produce un error en un método, puede haber instrucciones que se requieren antes de que el método o programa termine. La cláusula `finally` se ejecuta si todas las instrucciones del bloque `try` se ejecutan (y ningún bloque `catch`) o si se produce una excepción y uno de los bloques `catch` se ejecuta. En el método `leerInt(Scanner,String)`, para cualquier posible lectura, siempre se ejecuta la instrucción `tec.nextLine()` de la cláusula `finally`, permitiendo descartar el salto de línea que se almacena en el buffer de entrada cuando el usuario pulsa la tecla *Enter* o el token incorrecto que hace que se lance la excepción `InputMismatchException`, evitando que el método entre en un bucle infinito.

Actividad 4: tratamiento de excepciones en `leerDoublePos(Scanner,String)`

- Completa el método `leerDoublePos(Scanner,String)` de la clase `LecturaValida` para que capture la excepción `InputMismatchException` si el valor introducido por el usuario no es un `double`, de manera similar al método `leerInt(Scanner,String)`, mostrando un mensaje de error apropiado en lugar de abortar la ejecución.

Lo que acabas de hacer es añadir un controlador de excepciones para detectar una excepción a nivel local, es decir, en el mismo método en donde se produce el fallo.

Actividad 5: tratamiento de excepciones en `leerInt(Scanner,String,int,int)`

- Completa el método `leerInt(Scanner,String,int,int)` de la clase `LecturaValida` para que capture la excepción `InputMismatchException` si el valor introducido por el usuario no es un `int`, de manera similar al método `leerInt(Scanner,String)`, mostrando un mensaje de error apropiado en lugar de abortar la ejecución.
- Este método, además, ha de controlar que el valor introducido está en el rango `[lInferior,lSuperior]`. Hay dos formas de realizar este control: la primera consiste en añadir la condición apropiada en la guarda del bucle, como en el caso del método `leerDoublePos(Scanner,String)`, y la segunda en lanzar una excepción. Optamos por esta última. Añade una instrucción condicional tal que si el valor introducido no está en el rango anterior, lance la excepción `IllegalArgumentException`, usando una instrucción `throw`, con un mensaje que indique que el valor leído no está en dicho rango.
- Ahora, añade una cláusula `catch` para capturar localmente dicha excepción, de forma similar a la captura de la excepción `InputMismatchException`, mostrando el mensaje de la excepción mediante el método `getMessage()` (heredado de la clase `Throwable`).

Lo que acabas de hacer es un bloque multi-catch. Recuerda que siempre que aparece una cláusula `try`, debe existir al menos una cláusula `catch` o `finally`. Nota que, para una única cláusula `try`, pueden existir tantas `catch` como sean necesarias para tratar las excepciones que

se puedan producir en el bloque de código del `try`. Cuando en el bloque `try` se lanza una excepción, los bloques `catch` se examinan en orden, y el primero que se ejecuta es aquel cuyo tipo sea compatible con el de la excepción lanzada. Así pues, el orden de los bloques `catch` es importante. La excepción más específica debe aparecer antes que cualquier excepción de una clase antecesora. Aunque, en este caso, el orden no es relevante, en una actividad posterior relacionada con el tratamiento de ficheros verás un ejemplo en el que sí que lo es.

Recuerda también que a partir de la versión 7.0 de Java, un único bloque `catch` puede utilizarse para capturar más de un tipo de excepción, separándolas por una barra vertical (|), siempre que no haya ninguna que sea subclase de otra.

Actividad 6: tratamiento de excepciones en la clase `GestorBanco`

- Ejecuta la clase `GestorBanco` y, mediante el menú, elige la opción 2 para ingresar dinero (a pesar de que no se ha creado ninguna cuenta, esto es, no hay ninguna cuenta activa). Fíjate que la ejecución se aborta: no hay una cuenta activa para hacer el ingreso.

- Echa un vistazo al código de la clase `GestorBanco`. Se utiliza una variable `cuenta` de tipo `Cuenta` para representar la cuenta activa, que inicialmente es `null`.

- En el `case 2` de la instrucción `switch`, observa que al tratar de ingresar dinero en la cuenta, ya que la variable `cuenta` es `null` inicialmente, se lanza la excepción `NullPointerException`.

- Completa el `case 2` de la instrucción `switch` para capturar este tipo de excepción, mostrando el mensaje "*ERROR: ¡No hay ninguna cuenta activa! Primero búscala o crea una nueva cuenta*".

- Completa del mismo modo las opciones 3 y 5 de la instrucción `switch`.

Actividad 7: prueba de la clase `GestorBanco`

- Ejecuta la clase `GestorBanco` y prueba las distintas opciones con datos que conducirían a situaciones erróneas y comprueba que el tratamiento de errores realizado evita su mal funcionamiento.

5. Tratamiento de excepciones definidas por el usuario

Hasta ahora hemos estado usando las excepciones que proporciona Java, pero también podemos crear nuestras propias excepciones. A modo de ejemplo, vamos a cambiar el tratamiento de uno de los errores lógicos considerados, el que se produce al retirar una cantidad mayor que el saldo disponible en la cuenta activa. Observa que lo que se ha hecho en este caso, ha sido poner como precondición `cantidad <= saldo` y comprobarla antes de efectuar la invocación al método `retirar(double)` en la clase `GestorBanco`. En la actividad que sigue se utilizará una excepción de usuario para realizar este tratamiento. El método `retirar(double)` lanzará y propagará la excepción que se capturará en `GestorBanco`.

Actividad 8: excepción de usuario en `retirar(double)` de la clase `Cuenta`

- Echa un vistazo a la clase `SaldoInsuficienteException`. Fíjate que el cuerpo de la clase es bastante trivial, porque la clase `Exception` de Java hace el trabajo por nosotros.
- ¿Cómo? En el encabezado de la clase aparece `extends Exception`, lo que indica que la excepción `SaldoInsuficienteException` es una clase heredera de `Exception`. Y, por otro lado, el constructor invoca al de la clase `Exception` con el mensaje que se le pasa como parámetro. Vamos a probar esta nueva excepción.
- En el método `retirar(double)` de la clase `Cuenta`, añade código para comprobar si la cantidad que se le pasa como parámetro es superior al saldo de la cuenta. Si lo es, entonces lanza (`throw`) la excepción `SaldoInsuficienteException` con el mensaje "*¡No se puede retirar más dinero del que hay en la cuenta!*".
- Además, puesto que `SaldoInsuficienteException` es una excepción *checked*, debes añadir una cláusula `throws` a la cabecera del método `retirar`, como sigue:

  ```
  public void retirar(double cantidad) throws SaldoInsuficienteException
  ```

  indicando al compilador que la excepción podría ser lanzada y que el método `retirar` no tratará de solucionar el problema. En su lugar, se propaga la excepción al método que llama a `retirar`.
- A continuación, esta excepción se capturará remotamente en el método `main` de la clase `GestorBanco`. Modifica el código del `case 3` de la instrucción `switch`, eliminando la instrucción condicional y añadiendo una cláusula `catch` para capturar este tipo de excepción. Todo lo que tienes que hacer es mostrar un mensaje de error junto con el mensaje de la excepción.
- Vuelve a ejecutar el programa. Crea una cuenta e intenta retirar una cantidad mayor que el saldo disponible. Ahora tampoco debería ser posible hacerlo.

Fíjate que capturamos la excepción en el método `main` de la clase `GestorBanco`, en lugar de en el método donde ocurre el problema. Cuando se detecta una excepción de esta manera, es decir, en un método diferente de aquel en el que ocurrió el problema, se dice que la captura se realiza de manera remota. Las excepciones pueden ser tanto capturadas localmente como de forma remota dependiendo de dónde tenga más sentido tratar de resolver el problema.

6. Leer/escribir desde/en un fichero de texto

En esta sección, vamos a ver cómo leer/escribir cuentas bancarias desde/en un fichero de texto, realizando el tratamiento de las excepciones relacionadas. Para ello implementarás dos métodos en la clase `Banco` y ampliarás los casos del método `main` de la clase `GestorBanco`.

Actividad 9: lectura de las cuentas guardadas en un fichero de texto

- Las cuentas bancarias están guardadas en un fichero con un formato determinado. El formato indica el orden y disposición de la información en el fichero. Cuando se procesan

ficheros, debemos ser capaces de leer ficheros válidos que siguen las especificaciones de formato (y rechazar los ficheros no válidos). En nuestro caso, cada línea del fichero tiene dos elementos de información, un número de cuenta de tipo `int` seguido de un saldo de tipo `double` (por ejemplo, véase `cuentas.txt`, que se ha descargado anteriormente). Los ficheros de entrada pueden tener cualquier número de líneas. Se puede utilizar el método `hasNext()` de `Scanner` para leer mientras quede texto en el fichero por leer. Para cada línea, tendrás que separar el número de cuenta y el saldo. Hay dos formas de hacerlo:

Primera aproximación La manera más fácil de leer información de un fichero requiere que los valores estén separados por espacios en blanco, tabulaciones o saltos de línea. El fichero `cuentas.txt` está formateado de esta manera por lo que se puede utilizar `Scanner` para leer cada fragmento de información por separado (es decir, usando `nextInt()` y `nextDouble()`). Este enfoque funciona bien siempre y cuando se tenga cuidado con el procesamiento del salto de línea, que marca el final de cada línea.

Segunda aproximación Si los valores en un fichero formateado están separados por algún otro delimitador, como una coma, es mejor leer cada línea completa del fichero por separado (usando `nextLine()`) y luego dividir la línea en los fragmentos apropiados. Hay varias maneras de hacer esto, pero te sugerimos que utilices el método `split` de la clase `String`. Tras obtener las cadenas de caracteres con los números, se pueden usar los métodos `Integer.parseInt(String)` y `Double.parseDouble(String)` para convertirlos a formato numérico.

- Añade a la clase `Banco` el método `public void cargarFormatoTexto(Scanner f)`. Mientras haya algo que leer del `Scanner f`, este método lee el número de cuenta (`int`) y el saldo (`double`) de una línea del fichero siguiendo una de las dos aproximaciones anteriores. Con estos datos, crea un objeto de tipo `Cuenta` y lo añade al banco (`this`) usando el método `añadir(Cuenta)`.

- Añade un `case 7` al `switch` del `main` de la clase `GestorBanco` en el que se pida al usuario el nombre del fichero en el que están guardadas las cuentas. Con dicho nombre creará un objeto `File`, a partir del que se creará un objeto de tipo `Scanner` para realizar la lectura. Al crear este último objeto, se intenta localizar un fichero en el directorio en el que se ejecuta la aplicación. Esto puede producir una excepción de tipo `FileNotFoundException`, entre otras razones, porque no existe el fichero o no se poseen los permisos de acceso apropiados. Debes capturar la excepción, informando al usuario de que no se ha localizado el fichero y salir del `case` sin realizar ninguna acción. Si el fichero se ha localizado con éxito, se invoca al método `cargarFormatoTexto(Scanner)` aplicándolo sobre el banco que se define en el `main`. Recuerda que, en la cláusula `finally` del `try`, hay que comprobar si se ha creado el `Scanner` y, si es así, cerrarlo.

- Modifica el método `menu(Scanner)` de la clase `GestorBanco` añadiendo la opción *7) Cargar banco desde fichero*. Ten en cuenta que has añadido una opción y que en la llamada a `leerInt(Scanner,String,int,int)` se ha de incrementar el rango de valores permitido.

- Para probar la lectura desde fichero de texto, ejecuta el `main` de la aplicación, elige la opción de cargar el fichero `cuentas.txt` y muestra todas las cuentas del banco (opción 6 del menú) para comprobar que se han leído correctamente.

- Vuelve a ejecutar la opción de cargar el fichero `cuentas.txt` y muestra todas las cuentas. ¿Qué ocurre? Se han vuelto a cargar las mismas cuentas obteniendo cuentas duplicadas. Intenta arreglarlo comprobando previamente si la cuenta existe en el array `cuentas`, usando el método `getCuenta(int)`.

- Ejecuta de nuevo la opción de cargar el fichero pero ahora con el fichero `badinput.txt`. Verás que se detiene la ejecución al leer la línea que contiene un número de cuenta que no es un `int`. Para que la ejecución no se detenga si el valor leído no es un `int` o un `double`, captura la excepción `InputMismatchException` en el método `cargarFormatoTexto(Scanner)`.

- Ejecuta otra vez la opción de cargar el fichero con el fichero `badinput.txt` y muestra todas las cuentas. ¿Qué ocurre? Se han cargado cuentas no válidas. Modifica el método `cargarFormatoTexto(Scanner)` de modo que si la línea que se lee no es válida, esto es, si el número de cuenta no tiene 5 dígitos o si el saldo es negativo, no se cargue dicha cuenta en el banco.

Actividad 10: escritura de las cuentas en un fichero de texto usando el método `toString()` de la clase `Cuenta`

- Añade el método `public void guardarFormatoTexto(PrintWriter f)` a la clase `Banco`. Este método debe escribir para cada cuenta del banco una línea, en el fichero representado por el objeto `PrintWriter f` dado, con su número de cuenta y su saldo, en este orden, y separados por un espacio en blanco. Puedes usar el método `println(String)` de la clase `PrintWriter` y el método `toString()` de la clase `Cuenta`.

- Añade un `case 8` al `switch` del `main` de la clase `GestorBanco` en el que se pida al usuario el nombre del fichero en el que se guardarán las cuentas. Con dicho nombre creará un objeto `File`, a partir del que se creará un objeto de tipo `PrintWriter` para realizar la escritura. Al crear este objeto, se intenta crear un fichero en el directorio en el que se ejecuta la aplicación y se puede producir una excepción de tipo `FileNotFoundException`, entre otras razones, por falta de espacio en disco o por la no posesión de permisos de escritura. Debes capturar la excepción, informando al usuario de que no se ha creado el fichero. Si se ha creado con éxito, invoca al método `guardarFormatoTexto(PrintWriter)` aplicado sobre el banco que se define en el `main`. Recuerda que, en la cláusula `finally` del `try`, hay que comprobar si se ha creado el `PrintWriter` y, en caso afirmativo, cerrarlo.

- Modifica el método `menu(Scanner)` de la clase `GestorBanco` añadiendo la opción *8) Guardar banco en fichero*. Ten en cuenta que has añadido una nueva opción y que en la llamada a `leerInt(Scanner,String,int,int)` se ha de incrementar el rango de valores permitido.

- Para probar la escritura en fichero de texto, ejecuta el `main` de la aplicación, crea un par de cuentas nuevas y guarda las cuentas del banco en un fichero. Abre el fichero desde el terminal y comprueba que para cada cuenta contiene una línea con el número de cuenta y saldo que has introducido previamente. Además, comprueba que los valores numéricos están separados por un espacio en blanco.

Actividad 11: escritura de las cuentas en un fichero de texto usando el método `toString()` de la clase `Banco`

En esta actividad vas a modificar el método `guardarFormatoTexto(PrintWriter f)` de la clase `Banco` para obtener una implementación más sencilla, aprovechando el hecho de que el método `toString()` de la clase `Banco` hace uso del `toString()` de la clase `Cuenta`, devolviendo un `String` con tantas líneas como cuentas tiene el banco, cada línea con un número de cuenta y un saldo separados por un espacio en blanco. ¡Justamente el formato del fichero `cuentas.txt`!

- Modifica[1] el método `guardarFormatoTexto(PrintWriter f)` para que se escriban de una sola vez todas las cuentas del banco, usando el método `print(String)` de la clase `PrintWriter` y el método `toString()` de la clase `Banco`.
- Ejecuta la aplicación, crea algunas cuentas y elige la opción de guardar en un fichero. Desde el terminal, abre el fichero que se ha creado y comprueba que contiene la misma información que antes y con el mismo formato.
- Ejecuta de nuevo la aplicación, pero esta vez no crees ninguna cuenta antes de guardar las cuentas. Comprueba el contenido del fichero creado. ¿Qué se ha guardado? Si se intentan cargar las cuentas de este fichero se producirá un error de formato. Esto se debe a que contiene la frase que devuelve `toString()` cuando un banco no tiene ninguna cuenta ("*No hay cuentas en el banco*"). Para solucionarlo bastará considerar que sólo se debe guardar algo en el fichero si la cantidad de cuentas (`numCuentas`) es mayor que cero.
- ¿Con cuántas líneas de código has implementado el método?

7. Leer/escribir desde/en un fichero binario de objetos

En la sección anterior hemos visto lo fácil que resulta la implementación del método `guardarFormatoTexto(PrintWriter)` usando el mismo formato en los ficheros a guardar que en los `String` resultado de los métodos `toString()`. Estos métodos se usan para devolver información contenida en los objetos en un formato especificado por el diseñador. El lenguaje Java proporciona una forma de automatizar este proceso usando una clase especial denominada `Serializable`. Esta clase permite obtener la información de un objeto en un formato estándar en Java siempre que todo su contenido, incluidos los objetos referenciados, utilicen el mismo formato. Para obtener esta utilidad del lenguaje, se debe añadir `implements Serializable` tras el nombre de la clase en su definición. En esta sección vamos a usar este formato estándar para recuperar/guardar desde/en un fichero la información de todas las cuentas de un banco a las que referencia el array `cuentas` de la clase `Banco`. Para ello implementarás dos nuevos métodos en la clase `Banco` y modificarás las opciones 7 (cargar) y 8 (guardar) del `switch` del `main` de la clase `GestorBanco`.

Actividad 12: escritura de objetos `Cuenta` en un fichero binario de objetos

- Añade `implements Serializable` en la cabecera de la clase `Cuenta` y la instrucción `import java.io.Serializable` al comienzo de la misma.

[1] Comenta primero las instrucciones del cuerpo del método implementado en la actividad 10.

- Añade el método `public void guardarFormatoObjeto(ObjectOutputStream f)` a la clase `Banco`. Este método debe escribir cada uno de los objetos `Cuenta` del array `cuentas` en el fichero representado por el flujo `f`, usando el método `writeObject(Object)`. Este método propaga la excepción `IOException`; añade la cláusula necesaria en la cabecera de `guardarFormatoObjeto(ObjectOutputStream)` para que también propague dicha excepción.

- Modifica[2] el `case 8` del `switch` de la clase `GestorBanco` para que después de leer el nombre de un fichero y crear objetos de las clases `FileOutputStream` y `ObjectOutputStream`, invoque al método `guardarFormatoObjeto(ObjectOutputStream)`. Debes tratar las siguientes excepciones, informando al usuario del error ocurrido en el proceso de escritura:

  - `FileNotFoundException` que se lanza si el objeto `FileOutputStream` no localiza el fichero que se le pasa como parámetro.
  - `IOException` que puede lanzarse por el objeto `ObjectOutputStream` o ser propagada por el método `guardarFormatoObjeto(ObjectOutputStream)`.

- Recuerda comprobar si el `ObjectOutputStream` se ha creado y, si es así, cerrarlo en la cláusula `finally` del `try`. En este caso, dicho cierre requiere tratar dentro de la misma cláusula una `IOException` propagada por el método `close()`.

- Ejecuta el `main` de la aplicación, elige la opción de cargar el fichero `cuentas.txt` y, a continuación, elige la opción 8 para guardar las cuentas, en este caso, en un fichero binario de objetos de nombre `cuentas.obj`. Ahora tienes dos ficheros con la misma información pero en distinto formato. El fichero de texto es accesible mediante cualquier editor de textos mientras que el fichero binario requiere de instrucciones Java para su lectura.

Actividad 13: lectura de las cuentas de un banco desde un fichero binario de objetos

- Añade el método `public void cargarFormatoObjeto(ObjectInputStream f)` a la clase `Banco`. Este método debe leer los objetos `Cuenta` almacenados en un fichero binario de objetos. La lectura se debe realizar mediante un bucle cuya terminación vendrá dada por el acceso al final del fichero, provocado por la excepción `EOFException` que debe ser tratada. Así, mientras haya algo que leer del `ObjectInputStream f`, este método lee un objeto `Cuenta` con el método `readObject()` y lo añade al banco (`this`) usando el método `añadir(Cuenta)`.

- Compila la clase `Banco`. ¿A qué se debe el error que se produce? Soluciona el problema forzando la conversión del tipo del objeto que devuelve el método `readObject()` al tipo `Cuenta`.

- Al igual que ocurría con los ficheros de texto, debes controlar que no existan cuentas con números de cuenta duplicados. Para ello comprueba si ya existe la cuenta usando el método `getCuenta(int)` pasándole como parámetro el numero de la cuenta leída del fichero.

[2]Comenta primero las instrucciones de la escritura con ficheros de texto implementadas en la actividad 10.

- La instrucción `readObject()`, además de la excepción `EOFException` comentada anteriormente, puede lanzar las excepciones `ClassNotFoundException` e `IOException` que se propagarán para ser tratadas en la clase `GestorBanco`.
- Modifica[3] el `case 7` del `switch` de la clase `GestorBanco` para que después de leer el nombre de un fichero y crear objetos de las clases `FileInputStream` y `ObjectInputStream`, invoque al método `cargarFormatoObjeto(ObjectInputStream)`. En este caso, debes tratar las tres excepciones que siguen, informando al usuario del error ocurrido en el proceso de lectura del fichero:
  - `FileNotFoundException` que se lanza si el objeto `FileInputStream` no localiza el fichero que se le pasa como parámetro.
  - `IOException` que puede lanzarse por el objeto `ObjectInputStream` o ser propagada por el método `cargarFormatoObjeto(ObjectInputStream)`.
  - `ClassNotFoundException` que también puede ser propagada por el método anterior si no se puede determinar la clase del objeto que se intenta leer.
- ¿En qué orden debes capturar las excepciones anteriores? Recuerda que la excepción más específica debe aparecer antes que cualquier excepción de una clase antecesora. Para asegurarte que realizas su captura en el orden adecuado, consulta en el API de Java la documentación de las clases `ClassNotFoundException`, `FileNotFoundException` e `IOException`. Su situación en la jerarquía de clases de Java es la que sigue:

```
java.lang.Object
 |
 +--java.lang.Throwable
     |
     +--java.lang.Exception
         |
         +--java.lang.ClassNotFoundException
         |
         +--java.io.IOException
            |
            +--java.io.FileNotFoundException
```

- Recuerda comprobar si el `ObjectInputStream` se ha creado y, si es así, cerrarlo en la cláusula `finally` del `try`. También en este caso, dicho cierre requiere tratar dentro de dicha cláusula una `IOException` lanzada por el método `close()`.
- Para probar la lectura desde fichero binario de objetos, ejecuta el `main` de la aplicación, elige la opción de cargar el fichero `cuentas.obj` y muestra todas las cuentas del banco (opción 6 del menú) para comprobar que se han leído correctamente.

Actividad 14: escritura de un objeto `Banco` en un fichero binario de objetos

En esta actividad vas a modificar el método `guardarFormatoObjeto(ObjectOutputStream)` de la clase `Banco` para guardar directamente el propio objeto `Banco` en el fichero.

[3]Comenta primero las instrucciones de la lectura con ficheros de texto implementadas en la actividad 9.

- Modifica[4] el método `guardarFormatoObjeto(ObjectOutputStream)` de la clase `Banco` para que escriba el objeto `Banco this` en el fichero representado por el flujo `f` usando el método `writeObject(ObjectOutputStream)`. De igual modo que en la versión anterior (Actividad 12), debes propagar la excepción `IOException`.
- Ejecuta la aplicación, crea un par de cuentas nuevas y guarda el banco. Se produce un error de ejecución debido a que la clase `Banco` no sigue el formato `Serializable`. Añade `implements Serializable` a esta clase.
- Ejecuta de nuevo la aplicación, elige la opción de cargar el fichero `cuentas.obj` y, a continuación, elige la opción 8 para guardar el banco en un fichero binario de objetos de nombre `banco.obj`.

Actividad 15: lectura de un banco desde un fichero binario de objetos

En esta actividad vas a modificar el método `cargarFormatoObjeto(ObjectInputStream)` de la clase `Banco` para leer un único objeto `Banco` desde fichero.

- Modifica[5] el método `public void cargarFormatoObjeto(ObjectInputStream f)` para que realice la lectura de un objeto `Banco b` usando el método `readObject()`. Recuerda que debes forzar la conversión del tipo del objeto que devuelve el método `readObject()` al tipo `Banco`.
- La variable `b` referencia a un objeto `Banco` con todas sus cuentas, pero se trata de un objeto nuevo, distinto al `Banco this`. Recorre el array `cuentas` de `b` y, usando el método `añadir(Cuenta)`, añade las cuentas de `b` al banco `this`. Para cada cuenta comprueba si ya existe en `this` usando el método `getCuenta(int)` pasándole como parámetro el número de la cuenta de `b`.
- De igual modo que en la versión anterior (Actividad 13), debes propagar las excepciones que puede generar el método `readObject()` (`ClassNotFoundException` e `IOException`) para ser tratadas en la clase `GestorBanco`.
- Ejecuta la aplicación, elige la opción de cargar el fichero `banco.obj` y muestra todas las cuentas del banco (opción 6 del menú) para comprobar que se han leído correctamente.

8. Evaluación

Esta práctica forma parte del segundo bloque de prácticas de la asignatura que será evaluado en el segundo parcial de la misma. El valor de dicho bloque es de un 60 % con respecto al total de las prácticas. El valor porcentual de las prácticas en la asignatura es de un 20 % de su nota final.

[4] Comenta primero las instrucciones del cuerpo del método implementado en la actividad 12.
[5] Comenta primero las instrucciones del cuerpo del método implementado en la actividad 13.