1. Introducción

Introducción

*Una de las características más importantes de las empresas informáticas modernas, es la de formar continuamente a sus empleados para que su trabajo se adapte a las, cada vez más novedosas, necesidades de sus clientes. En una empresa pequeña como puede ser **SI Andalucía**, no renuncian a que sus clientes tengan el mejor servicio y se dedican a la autoformación, aunque cuando hay miembros expertos en algún tema, siempre se dedica a orientar a sus compañeros. Es el caso de **José** que tiene grandes conocimientos en programación con Java, pero que evidentemente no puede hacerlo todo. Por ello se dedica a explicar algunas técnicas de programación a **María** y **Víctor**, que colaboran con él en el departamento de programación.*

En la unidad anterior introducíamos el uso de estructuras estáticas de datos en Java, comenzando con las cadenas de caracteres y los arrays. Pero nos queda otra estructura estática de datos importante, que son los ficheros. Básicamente todos sabemos, al menos de forma aproximada, lo que es un fichero pero no viene mal repasar algunos conceptos, y sobre todo es necesario que expliquemos las peculiaridades del manejo de ficheros en Java. Ten en cuenta que es una estructura de datos con la suficiente entidad como para que te la presentemos en una unidad independiente.

No obstante, el **uso de ficheros en Java exige manejar excepciones**, por lo que a modo de concepto previo, te vamos a introducir en los fundamentos de esta potente característica de Java, que facilita enormemente el tratamiento de errores, y contribuye a que se creen programas más robustos, más claros y que toleren mejor los fallos, con relativamente poco esfuerzo por parte del programador. En el siguiente apartado explicamos estos conceptos.

2. Manejo de Excepciones

Manejo de Excepciones

*Una de estas técnicas es la del manejo de excepciones para controlar la aparición de errores durante la ejecución de un programa. **José** les explica que hay ocasiones en las que de alguna manera, podemos esperar un error inevitable y que al menos, debemos gestionarlo para evitar la ruptura del programa y orientar al usuario para que no vuelva a aparecer ese error.*
***Carmen** sabe de qué está hablando, cuando estudiaba el ciclo formativo en el Instituto ya le acostumbraron al manejo de excepciones y eso es algo de lo que ya no prescinde en sus programas, especialmente cuando se trata de trabajar con estructuras externas de datos, como son los ficheros. Por ejemplo hay situaciones en las que un programa intenta escribir en un fichero que es de sólo lectura, o borrar un fichero que ya ha sido borrado, o simplemente leer de un fichero ubicado en otra carpeta o directorio y que no encuentra.*

Piensa la cantidad de errores que se pueden presentar en cualquier programa. Hasta ahora, nosotros hemos venido comprobando y controlando en nuestros programas numerosos errores, tanto en los ejemplos que te ofrecemos como en los que te pedimos que realices como ejercicios. En casi todos ellos es necesario comprobar determinadas circunstancias que pueden provocar fallos en el programa. Como ejemplos, puedes pensar en la necesidad de controlar que no se realicen divisiones por cero, o evitar que se acceda a una posición de un vector que no existe, o impedir que se intente comprobar un carácter de un `String` más allá del último (o anterior al primero), introduciendo un valor negativo para la posición dentro del `String` o el índice del vector.

También tenemos que comprobar cada vez que leemos un número desde teclado (usando la clase `ES` que ya hemos creado) que lo que se teclea se puede considerar un número, y se puede convertir en número sin que el programa aborte cuando metemos caracteres extraños. Por ejemplo **Xt56?e4** no se podrá convertir en número, por más que nos empeñemos, así que tenemos que asegurarnos de que cuando se espere un número y se teclee esa cadena, el error no va a suponer que nuestro programa termine bruscamente, abortando.

En general existen dos planteamientos posibles, a la hora de manejar los errores:

Prevenirlos

Algo muy común. Supone incluir líneas de código adicionales, intercaladas con el código de la aplicación, en

los puntos en los que pueden ocurrir los errores, de forma que prevengan y eviten esos errores. Esto tiene ventajas e inconvenientes:

> ***Ventaja: El programador que lee el código puede ver claramente si en ese punto se procesó o no el error, y si se comprobó o no correctamente.***
> ***Inconveniente: El código se "complica" con el procesamiento de errores****. Para el programador que lee la aplicación intentando comprender su funcionamiento, las líneas de control de errores le distraen de la lógica principal del sistema, dificultando la comprensión de la aplicación.*

Esta es la forma elegida en los ejemplos sobre cadenas de caracteres y arrays, para impedir que se pueda acceder a posiciones que no existen. En todos esos casos hemos usado la función **`length`** o el método **`length()`** para impedir accesos más allá del último elemento del array o el **`String,`** respectivamente. Por otro lado, hemos evitado usar números negativos para dimensionar un array o para acceder a posiciones de un **`String,`** mediante el uso del método **`ES.leeNº(String mensaje, int minimo)`**. Por ejemplo, al leer de teclado el número de elementos a los que se va a dimensionar un array, usaremos

```
int dimension=ES.leeNº("Introduce la dimensión del vector: ", 1);
```

De esta forma, al dimensionar el array, nos aseguramos de que al menos tenga un elemento, y evitamos que el programa aborte.

Tratarlos adecuadamente una vez que ocurren

No se impide que se produzca el error, por que no se pueden tener en cuenta todas las circunstancias que lo pueden provocar, o sencillamente porque supone complicar excesivamente el código. Una vez que el error (o excepción) ha ocurrido, se "captura", y se pasa el control del flujo del programa a una zona concreta que llamamos manejador de la excepción, de forma que se corrige ese error, se avisa al usuario de lo que ha ocurrido, o sencillamente se termina el programa de forma ordenada impidiendo que aborte. Como vimos en el apartado 5.4 de la unidad 5, el manejo de excepciones aparentemente rompe el flujo del programa, con un salto incondicional a otra zona de código. Pero si esto se hace como en Java, transfiriendo el control hacia delante, a una zona cercana y bien definida del código, no tiene porqué ser considerado como una violación de los principios de la programación estructurada.

Esta es la forma elegida para controlar los errores de Entrada/Salida en la clase **`ES`**. Concretamente, para leer números enteros o reales desde teclado, se lee una cadena de caracteres, que se intenta **`(try)`** convertir a número. Si se produce un error (una **`NumberFormatException`**), debido a que la cadena introducida no se corresponde con el formato numérico de los literales del tipo int que se están leyendo, se captura **`(catch)`** la excepción, se escribe un mensaje indicando el error y solicitando un nuevo valor. Esto se repite dentro de un ciclo, mientras que el valor introducido sea incorrecto.

> ***Ventajas: Permite al programador quitar el código que maneja los errores del código principal, quedando un hilo de ejecución más limpio y claro, de forma que los programas así escritos son más fáciles de modificar y mantener.*** *Además, es congruente con los criterios de modularidad. La tarea de tratar los errores es algo distinto al propósito principal del programa, y tiene sentido que se haga en un módulo distinto, que es el manejador de la excepción.* ***Por otro lado, una vez que se produce una excepción, no se puede ignorar, y debe haber un código que la maneje.***
> ***Inconveniente: Las líneas de código que gestionan los errores pueden estar físicamente bastante separadas de las líneas en las que se produjo el error,*** *por lo que para saber cómo se maneja un error determinado puede obligarse al programador a revisar el código en su búsqueda. Pero este inconveniente es mínimo, ya que los manejadores de excepciones están bien identificados, y siempre detrás de los bloques try correspondientes que contienen el código potencialmente generador de errores.*

La filosofía del manejo de excepciones en Java contempla la posibilidad de "capturar el error", y en lugar de abortar el programa, "manejar el error".

¿En qué consiste manejar el error?

- **Si el error no es excesivamente grave, se podrá dar otra oportunidad al usuario para corregirlo**, por ejemplo introduciendo un nuevo valor para el número que se solicita, o volviendo a llamar al método que provocó el error, pero con otros parámetros más adecuados.
- **Si el error es un fallo realmente grave, un error irrecuperable, al menos tendremos la oportunidad de avisar convenientemente al usuario de lo que ha pasado, podremos salvar lo**

que sea aprovechable de nuestra aplicación, y terminar el programa ordenadamente, en vez de con un final brusco, como cuando el programa aborta.

Autoevaluación

2.1. Excepciones básicas

Excepciones básicas

Antes de entrar en los detalles sobre cómo capturar y tratar excepciones y cómo generar y lanzar excepciones propias, ¿te has preguntado cómo define Java las excepciones y cuando se usará el manejo de excepciones para manejar los errores del programa?

En general, **por excepción entenderemos una situación anómala en la ejecución de un programa, que impide que se siga ejecutando el flujo normal del programa, sin que en el ámbito en que se produce esa situación de error tengamos información suficiente para corregir el error**, por lo que debemos pasar el control del flujo del programa a otro ámbito en el que quizás sea posible manejar ese error disponiendo de más información. Así se propaga la excepción al método desde el que se invocó el código que generó la excepción, que si tampoco la sabe tratar la pasará a su vez al método que lo invocó, y así sucesivamente, hasta en el peor de los casos llegar al método `main()`, que le pasará la excepción a la máquina virtual Java, que escribirá un mensaje de error indicando el error ocurrido, escribirá la lista de invocaciones a métodos que han producido la excepción, y terminará la ejecución del programa. Esta es la situación menos deseable, pero en cualquier caso garantiza que cualquier excepción va a ser tratada, y no va a poder ignorarse sin más.

En Java existe la clase `Throwable`, y cualquier situación anómala en la ejecución de un programa genera directa o indirectamente un objeto de la clase `Throwable.`

La clase `Throwable` tiene dos subclases definidas en Java, cada una para un tipo de error o situación anómala en la ejecución:

- `Error`**. Indica que se ha producido un fallo irrecuperable, para el que es imposible recuperar y continuar la ejecución del programa**. La máquina Virtual Java presenta un mensaje en el dispositivo de salida, y concluye la ejecución del programa. Para este tipo de errores, no hay nada que el programador pueda hacer, por lo que no nos vamos a ocupar más de ellos. Te remitimos a la información adicional sobre sus posibles subclases que puedes encontrar en la especificación de la API de Java.
- `Exception`. **Indica una situación anormal, pero que puede corregirse y reconducirse para que el programa no tenga que terminar forzosamente**. Java además nos proporciona cerca de 70 subclases directas de la clase `Exception`, ya predefinidas, cada una para un tipo de error concreto. Además, cada una de estas subclases suele tener a su vez bastantes otras subclases más, que nos definen cada vez errores más concretos y específicos.

Así por ejemplo, una de las subclases de la clase `Exception` es `IOException`, encargada de capturar errores de Entrada/Salida. Tiene a su vez 26 subclases, entre las que se encuentra `EOFException,` que se lanza cuando se alcanza inesperadamente el final de un fichero realizando una entrada de datos desde un fichero o flujo.

Además de la inmensa cantidad de subclases de `Exception` que ya nos da definidas el lenguaje Java, y que abarcan la práctica totalidad de situaciones problemáticas que puedas imaginar, **el programador puede definir, lanzar y usar sus propias excepciones en Java**, por lo que las posibilidades de capturar y tratar errores son infinitas.

En general, la sintaxis del manejo de excepciones en Java incluye el uso de las siguientes palabras reservadas: `try, catch, finally, throw, throws`

En los apartados siguientes se analiza el uso de cada una de ellas.

2.2. Capturar una excepción (try - catch - finally)

Capturar una excepción (try - catch - finally)

Ha llegado el momento de ver con detalle la sintaxis del manejo de excepciones en Java.

¿Cómo identificar en el programa el código que puede generar excepciones?

- **De ello se encarga el bloque de código que comienza con la palabra reservada `try`**, seguida de un bloque de sentencias entre llaves, que **son el código que puede generar el error**.

¿Cómo indicar el código al que se transfiere el control en el caso de que se produzca la excepción?

- **Es la misión del bloque iniciado por la palabra reservada `catch` seguida de un paréntesis que contiene la declaración del objeto de tipo `Exception` (o subclase de `Exception`) que se creará si se produce el error**, y que recibe la cláusula `catch` como si de un parámetro se tratara. El paréntesis va seguido de un bloque de sentencias contenidas entre llaves, que **constituyen el manejador de la excepción**, indicando lo que debe hacerse en el caso de que se produzca el error. El bloque `catch` irá siempre después del bloque `try`.

Puede haber varios bloques `catch` para cada bloque `try`, cada uno encargado de indicar lo que debe hacerse en el caso de que se presente un determinado tipo distinto de excepción dentro de las sentencias del bloque `try`. Lo que no es posible es tener un bloque `try` sin ningún bloque `catch`, o viceversa. Sólo se permite un bloque `try` sin ningún bloque `catch` si se ha incluido un bloque `finally`. En definitiva, **debe haber siempre un código que se ejecute cuando se lanza la excepción, que no puede quedar sin tratamiento.**

- Por último **podemos colocar opcionalmente un bloque `finally`**, que encerrará entre llaves un grupo de sentencias que se ejecutarán siempre, tanto si se produce una excepción como si no. Siempre que se incluya una cláusula `finally`, debe colocarse detrás del último bloque `catch`. Normalmente la cláusula `finally` no es muy utilizada, pero su principal utilidad es garantizar la liberación de recursos que el bloque `try` ha acaparado con uso exclusivo, y que podrían quedar definitivamente retenidos por él si no se ejecutan las sentencias que los liberan debido a que una excepción se produce antes de las sentencias encargadas de esa liberación del recurso, transfiriendo el control al bloque `catch`.

Por ejemplo, si en el bloque `try` se reserva un fichero para escritura, ningún otro programa podrá escribir en ese fichero hasta que el bloque `try` concluya y libere ese recurso. Pero si se produce una excepción en el bloque `try` antes de haber alcanzado la sentencia que libera el recurso, debe tenerse en cuenta e incluir en el bloque `catch` correspondiente el código necesario para liberar el fichero en cuestión. Para no tener que repetir ese código que libera el recurso tanto en el bloque `try` como en el bloque `catch`, Java crea la cláusula `finally`, que se ejecutará siempre, tanto si se produce la excepción como si no. Si colocamos las sentencias que liberan el recurso en la cláusula `finally`, estamos garantizando que pase lo que pase, el recurso se liberará correctamente. Veremos un uso más o menos lógico de la cláusula `finally` cuando trabajemos con ficheros en esta misma unidad.

Otro ejemplo de recurso que debería ser liberado una vez que no es necesario puede ser una conexión a una base de datos. La mayoría de las bases de datos tendrán establecido un número máximo de usuarios que pueden acceder simultáneamente a los datos. Si nuestra aplicación establece la conexión, y debido a un error, no la cierra, puede estar impidiendo que otro usuario pueda conectarse a la base de datos, aunque nuestra aplicación ya no esté usando esa conexión.

Veamos como sería el esquema del código para capturar y manejar una excepción, usando como base el método `leeNº()` de la clase `ES`, al que se le han añadido algunos mensajes para explicar el funcionamiento de cada cláusula. También se le ha añadido la captura de una excepción genérica, `Exception,` para mostrar la forma de capturar más de una excepción para un único bloque `try`. Finalmente, se incluye una cláusula `finally`, cuyo único propósito es mostrar un mensaje para que se pueda comprobar que siempre se ejecuta el bloque `finally,` tanto si se produce el error como si no.

Puedes sustituir el método en cuestión de la clase `ES` por esta versión, y comprobar los resultados usándolo en cualquiera de los programas que leen números desde teclado.

```
public static int leeNº(String mensaje) {
     int numero=0;
     boolean incorrecto=true;
      while(incorrecto) {
         System.out.println(mensaje);
```

```
            try {

              /* Código que potencialmente puede producir errores. Concretamente el método
               * parseInt() puede no ser capaz de convertir el String que recibe como
               * parámetro a número entero porque no se corresponda con el formato numérico
               * requerido. En ese caso se lanza una NumberFormatException.
               */

               numero=Integer.parseInt(leeDeTeclado().trim());
               incorrecto=false;

            } catch(NumberFormatException e) {

              /* Si se produce la excepción del tipo NumberFormatException en el código
               * incluido en el bloque try, la capturamos y le decimos qué debe hacerse. En
               * este caso darle el valor adecuado a la variable incorrecto para que el bucle
               * while encargado de leer el número siga ejecutándose, y enviar un mensaje que
               * aclare lo que ha pasado, y lo que debe hacer el usuario.
               */

               incorrecto=true;
               System.err.println("NO ES UN NÚMERO ENTERO VÁLIDO: Vuelve a intentarlo.");

            }catch (Exception e2){

               /* Se captura cualquier otra posible excepción que se pueda producir, pero en
                * este caso no se indica que se haga nada especial, sólo evitar que el
                * programa aborte.
                */
            }
           /* Aquí podrían situarse otros bloques catch para otras posibles excepciones que
            * pudiera generar el bloque try, aunque en este caso prácticamente no es posible.
            */

          finally {

            /* Las sentencias que incluyamos en este bloque, se van a ejecutar tanto si se
             * lanza una excepción como si no.
             */

              System.out.println("Sentencia que siempre se ejecuta");
          }
        }
        return numero;
    }
```

Autoevaluación

2.3. Normas para el manejo de excepciones

Normas para el manejo de excepciones

Podemos resumir un poco lo dicho anteriormente en una serie de normas. A la hora de capturar excepciones, hay varias cosas que debes tener en cuenta:

- **Cuando hay una excepción que se lanza desde el bloque `try`, se transfiere el control al primer bloque `catch` cuyo parámetro coincida con el tipo de excepción que se ha lanzado**.
- **Si se coloca primero un bloque `catch` para una excepción más genérica, siempre se ejecutará ese bloque `catch`, y los que aparezcan detrás, correspondientes a subclases de excepciones de la anterior, no se alcanzarán ni se ejecutarán nunca**. Por ejemplo, si el primer bloque `catch` captura una `Exception`, poner detrás una captura de `NumberFormatException` es inútil, ya que es una subclase. Para Java una `NumberFormatException` no deja de ser un tipo particular de `Exception` (una subclase de `Exception`) por lo que se considerará que es el tipo adecuado, y se ejecutará el manejador para la clase `Exception`. Si posteriormente hay un `catch` para capturar `NumberFormatException`, no se ejecutará nunca, porque se considera que la excepción ya ha sido tratada convenientemente por el manejador de `Exception`. De hecho, el compilador avisará diciendo que la excepción `NumberFormatException` ha sido ya capturada. Sí se pueden colocar en orden contrario, como en el ejemplo del apartado anterior, de forma que si se produce `NumberFormatException`, se ejecuta el código adecuado de su manejador, y si se produce

cualquier otro tipo de excepción, será el manejador genérico para **`Exception`** el que se ejecutará.

- Cuando se ejecutan todas las sentencias del manejador de la excepción, se continúa con la sentencia siguiente, es decir, **en el manejador de la excepción nunca se produce un salto hacia atrás, a la sentencia que produjo la excepción**.
- **Si la excepción que se lanza desde el bloque `try` no se captura por ningún `catch` adecuado, se relanza hacia el método anterior en la cadena de llamadas, hasta llegar a un lugar donde se capture y se trate adecuadamente**.
- **En el peor de los casos será la propia máquina virtual la que se encargue de tratar la excepción**, escribiendo un mensaje de error y terminando la ejecución del programa. Es lo único que puede hacer la máquina virtual de forma genérica para cualquier excepción que le llegue sin haber sido capturada adecuadamente, ya que no tiene más información que le permita intentar una recuperación del error.
- **Si se escribe el bloque opcional `finally`, irá después del último `catch`, y se ejecutará siempre,** se produzca o no alguna excepción, y con independencia del tipo de excepción de la que se trate.
- Si en las sentencias del bloque **`catch`** se produjera una nueva excepción, sería esta última la única que se propagaría hacia los métodos llamantes.
- Aunque se puede transferir el control desde dentro del bloque **`try`** hacia otras zonas del código, usando **`break, continue o return`**, no es demasiado aconsejable hacerlo, y se debe tener presente que **si se ha especificado una cláusula `finally`, primero se ejecutará la cláusula `finally`, y luego se transferirá el control a la zona del programa que corresponda**.
- En el bloque **`catch`** se recibe como parámetro el objeto **`Exception`** del tipo de la excepción que se ha producido en el bloque **`try,`** y se le pueden enviar una serie de métodos para obtener información sobre la excepción que se ha producido. Algunos de esos métodos son:
  - **`getMessage()`** , que recoge el mensaje de error asociado al tipo de excepción que se ha producido.
  - **`printStackTrace(),`** que escribe la cadena de llamadas que han generado la excepción, es decir el método que produjo la excepción, indicando desde qué otro método fue llamado, y a su vez desde qué otro método fue llamado éste, y así sucesivamente hasta llegar al método **`main ()`**,desde el que se comienza la ejecución de todo el programa.

Autoevaluación

2.4. Creación de excepciones propias. Cláusulas throw y throws

Creación de excepciones propias. Cláusulas throw y throws

¿Sería interesante poder definir excepciones propias por parte del programador? ¿Permite Java hacer eso? Pues claro.

A pesar de que Java nos proporciona una variedad suficientemente amplia de excepciones para todo tipo de situaciones posibles de error, también es interesante poder crear excepciones propias, de forma que los mensajes de error estén personalizados, o que podamos decidir las condiciones bajo las que deben lanzarse esas excepciones. Java una vez más nos ofrece esta posibilidad, aumentando la flexibilidad, y lo hace mediante las cláusulas **`throw`** y **`throws`**.

- **Debemos definir una clase que declare la excepción que queremos crear como una subclase de alguna de las excepciones definidas por el lenguaje**. Naturalmente puede ser una subclase de **`Exception`**.
- **`throw` Se usa para indicar en qué lugar del código de nuestro método se lanza esa excepción, y para lanzarla, de hecho.**
- **`throws` Se usa en la cabecera del método para avisar a los programadores usuarios de nuestro método de que lanza un determinado tipo de excepción, y que deberán preocuparse de capturarla y manejarla convenientemente en los programas que usen ese método.**

Un ejemplo puede ayudar a entender mejor estos conceptos.

En una empresa tienen almacenado en la base de datos un catálogo de puestos de trabajo disponibles. Cuando se da de alta un nuevo trabajador, se le debe asignar un puesto de trabajo, mediante el método siguiente :

```
asignaPuestoTrabajo(String puestoAsignado)
```

Y es importante que el puesto de trabajo sea correcto, uno de los que realmente existen, ya que de él dependerá el sueldo a cobrar, los días de descanso y vacaciones, y algunas otras cuestiones importantes para el trabajador.

Asignarle un puesto erróneo originaría problemas, ya que no se sabría cuanto debe cobrar ese trabajador, o qué días de descanso le corresponden. Por eso, cuando se de de alta un nuevo trabajador, hay que comprobar que el puesto de trabajo que se le intenta asignar es correcto, y de eso se va a encargar el método:

```
analizaPuestoTrabajo(String[] puestosTrabajo, String puestoAsignado)
```

Para eso, se haría una consulta a la base de datos (eso ya verás cómo hacerlo en la unidad 20, dedicada al acceso a Bases de Datos). Como resultado de esa consulta obtendríamos un array de String llamado **`puestosTrabajo`**, que contendría en cada una de sus posiciones el nombre de uno de esos puestos de trabajo. Una vez generado ese array, se le pasa como parámetro al método **`analizaPuestoTrabajo()`** junto al puesto que se quiere asignar. Si **`puestoAsignado`** no aparece en la lista del array **`puestosTrabajo`**, el método **`analizaPuestoTrabajo()`** devuelve **`false`** (falso), y en caso contrario **`true`** (verdadero). En nuestro ejemplo, el array **`puestosTrabajo`** se declarará e inicializará explícitamente, con el propósito de simplificar el problema.

El método **`asignaPuestoTrabajo()`**, llama al método **`analizaPuestoTrabajo()`** antes de asignar el puesto de trabajo al nuevo trabajador. Si éste devuelve el valor falso, no hace la asignación, si no que lanza una excepción, de tipo **`PuestoTrabajoInexistenteException`**, **que ha sido creada y definida por nosotros**, de forma que le avise adecuadamente del error que se ha producido, y permita subsanarlo sin que se genere un malfuncionamiento del programa.

Un programa que se encarga de resolver esa parte del problema, puede ser el que se compone de las clases disponibles en los enlaces que siguen. Naturalmente tendrás que añadir la ya tradicional clase **`ES`** contenida en el fichero **`ES.java`**. Por otra parte, para no complicar las cosas más, en la clase **`GestionTrabajadores`** se crea el array **`puestosTrabajo`** como una variable declarada de forma explícita, con una lista de puestos de trabajo disponibles. De esta manera no es necesario complicar el ejemplo con cuestiones relativas al acceso a Bases de datos, que todavía son prematuras.

- La clase **`GestionTrabajadores`** es la que contiene el método **`main()`**, y la que al crear los trabajadores deberá capturar la posible excepción de intentar asignar un puesto de trabajo inexistente.
- La clase **`Trabajador`** contiene la definición de lo que es un trabajador, y entre otras cosas dispondrá de los constructores y de métodos de modificación de los datos de un trabajador. Desde esta clase se llamará la método **`asignaPuestoTrabajo()`**, que genera la excepción
- La clase **`PuestoTrabajoInexistenteException`** define el nuevo tipo de excepción como una subclase de **`Exception`**.

Descarga el archivo Trabajador.java

Descarga el archivo PuestoTrabajoInexistenteException.java

Descarga el archivo GestionTrabajadores.java

Descarga el archivo ES.java

Una vez más, te recomendamos que leas con atención el código del ejemplo, junto a los comentarios explicativos que en él se incluyen. Esos comentarios, más que documentar el código, pretenden explicar sobre el ejemplo los conceptos destacados del manejo de excepciones.

Existe la posibilidad de copiar en la carpeta que uses para tus proyectos hechos con NetBeans, directamente la carpeta que contiene todo el proyecto, que en el caso del ejemplo hemos llamado **`PuestosTrabajo.`** De esta forma, se puede seleccionar en el entorno la opción abrir proyecto, y tener directamente cargadas todas las clases del ejemplo, listas para ejecutarlo sin nada más. Puedes observar que en este caso, NetBeans crea un paquete llamado puestostrabajo para el proyecto, y que todas las clases que lo forman contienen como primera sentencia **`package puestostrabajo;`**

Un paquete no es más que una carpeta en la que se guardan clases que tienen alguna relación entre sí. En este caso la relación es que pertenecen al mismo proyecto, pero Java usa también los paquetes como parte

de la estrategia de control de acceso y ocultación de la información. Estos conceptos quedarán más claros en próximas unidades, dedicadas a la programación orientada a objetos

A continuación tienes un enlace a la carpeta que contiene todo el proyecto para el ejemplo anterior.

Descarga el zip con la carpeta completa del Proyecto Puestos Trabajo

> ***A partir de ahora en algunos ejercicios de la tarea será necesario que entreguéis la carpeta completa del proyecto, de forma que se facilite su ejecución en otros ordenadores, especialmente en el del tutor que debe corregir los ejercicios.***

DEMO: Visualiza cómo abrir y ejecutar un proyecto Java en NetBeans

3. Ficheros

Ficheros

> *Una vez que **Víctor** y **Carmen** han asimilado correctamente el uso de las excepciones, **José** pretende iniciarles en el correcto uso de los ficheros para la programación de aplicaciones con Java. **Víctor** comenta que eso está algo anticuado y que puede pasar directamente a comentarle cómo se utilizan las bases de datos en Java, que es lo que se utiliza en la actualidad. **José** le explica que para programar en cualquier lenguaje, es indispensable dominar el tratamiento de los ficheros como utilísimas estructuras de datos externas, ya sean bases de datos completas o un simple contador de visitas de un sitio Web.*
> ***José** les comenta la existencia de ciertas clases y paquetes específicos que Java reserva para el uso de los ficheros. Explica el concepto de flujos y cómo los utiliza el lenguaje para comunicar dispositivos (unos reciben datos y otros los envían). A **Víctor** le sorprende que tenga el mismo tratamiento un teclado o una impresora, que un fichero, porque para Java no son más que consumidores o productores de información.*

Recuerda que el objetivo de esta unidad es el estudio de los ficheros en Java. Hasta ahora hemos visto el manejo de excepciones como herramienta necesaria para trabajar con ficheros, pero son éstos el objetivo principal de la unidad.

¿Recuerdas cómo definíamos fichero o archivo en la unidad 2?

Un fichero o archivo es un conjunto de información sobre un mismo tema, tratada como unidad de almacenamiento y organizada de forma estructurada para la búsqueda y recuperación de un dato individual. Es por tanto la estructura que necesitamos usar para poder guardar los datos e informaciones en soportes de almacenamiento masivo o permanente, tales como discos duros, disquetes, CD's, memorias Flash, cintas magnéticas, etc. y poder recuperarlos cada vez que quiera usarlos en una aplicación sin tener que volver a introducirlos de nuevo.

Lo primero que debemos mencionar es que en las aplicaciones de gestión habituales, cada vez se usan menos los ficheros como soporte de almacenamiento de los datos de la empresa, y que esta función suelen hacerla las bases de datos, debido a que así se independizan más los datos de las aplicaciones que los usan. Piensa en cualquier empresa que manipule gran cantidad de datos sobre sus clientes: Un banco o una compañía de seguros, por ejemplo. Es muy probable que estas empresas necesiten cambiar las aplicaciones que usan, incluso cambiando de lenguaje de programación, pero lo que no se pueden permitir es tener que empezar a fichar de nuevo a todos sus cientos de miles de clientes, y sus millones de operaciones. Las aplicaciones nuevas deben ser capaces de utilizar los mismos datos que usaban las anteriores.

Puesto que la organización de los ficheros depende de cada lenguaje, y la forma de guardar en ellos la información a su vez depende de cada aplicación concreta, es muy posible que si se cambia de aplicación, hubiera que crear de nuevo los ficheros con la información para la nueva aplicación. ¡A fichar de nuevo todos los datos! Eso no es admisible desde el punto de vista empresarial, ya que podría suponer dejar inoperativa a la empresa durante meses.

En cambio, un buen gestor de bases de datos, almacena la información de manera estándar, e independiente de aplicaciones y lenguajes, y establece mecanismos también estándares para que cualquier aplicación escrita en cualquier lenguaje pueda acceder a modificar y consultar los datos que contiene. Podemos cambiar la aplicación tantas veces como queramos. Los datos de nuestra base de datos nos van a

resultar siempre útiles, sin ningún cambio adicional.

No obstante, las bases de datos también se guardan en ficheros, y los programadores que diseñan un nuevo gestor de bases de datos, evidentemente, tienen que vérselas con la programación de ficheros. Además, cualquier aplicación puede tener necesidades de guardar algún dato al margen de los datos de la base de datos, tal como breves comentarios recordatorios de tareas a realizar, o algún comentario provisional sobre algún pedido, etc. Algo que no queremos almacenar en la base de datos, por que no es relevante, pero que queremos tener guardado a mano en algún lugar. Para eso también es posible necesitar ficheros.

3.1. Nociones básicas sobre ficheros y organización de ficheros

Nociones básicas sobre ficheros y organización de ficheros

En el apartado anterior hemos mencionado qué es un fichero. ¿Qué destacarías de la definición?

Lo más destacable es el hecho de que **proporcionan soporte para el almacenamiento de la información en dispositivos de memoria masiva de forma permanente**.

Por tanto, si los ficheros deben guardarse en estos soportes de almacenamiento, deberán estar "adaptados a ellos". Es decir, cada soporte está asociado a un dispositivo físico de almacenamiento (una lectora de DVD, una regrabadora, un disco duro, una disquetera, etc.). Cada uno de estos dispositivos trabaja de forma distinta y también almacena la información de forma distinta.

Cuando como programador quiero almacenar desde mi aplicación la información en un fichero, ¿debería tener en cuenta el dispositivo que se va a usar para ello? ¿Debo tenerlo en cuenta también para leer la información desde el dispositivo?

Si quiero leer un registro correspondiente a los datos de un empleado concreto desde el fichero, no será igual si el fichero está almacenado en una unidad de cinta (tendré que rebobinar hacia delante o hacia atrás hasta encontrar el registro que busco, **accediendo secuencialmente a la información**) que si el fichero está en un disco duro (la cabeza lectora del disco puede situarse directamente encima de la zona donde están almacenados los datos del registro del trabajador que me interesa, **accediendo directamente a la información**).

En principio sí es distinto y sí debería tenerlo en cuenta, pero en la práctica no es así. **Nuestra aplicación hará la petición de trabajar con ficheros tanto para lectura como para escritura, al Sistema Operativo, y éste se encargará de enmascarar los detalles de funcionamiento del dispositivo de entrada/salida,** de forma que la aplicación siempre leerá o escribirá sobre ficheros más o menos de la misma forma, y es el Sistema Operativo el que se "peleará" con las características concretas del dispositivo que se use en cada caso. Es algo que los programadores tenemos que agradecer a los buenos Sistemas Operativos.

Existe toda una teoría de ficheros, relacionada con estas cuestiones, y que establece la forma de elegir la mejor organización de ficheros para cada caso, y cómo gestionarla. No vamos a entrar en demasiados detalles, porque como ya hemos comentado, en las aplicaciones de gestión que pretendemos enseñarte a programar, cada vez se usan menos los ficheros, y prácticamente podemos obviar esas cuestiones.

Pero es algo que sí resulta útil para otro tipo de aplicaciones, y sobre lo que nunca estaría mal que te informaras.

Para saber más
Accediendo a este enlace, tendrás una visión más específica sobre la organización de ficheros
Organización de Ficheros y Métodos de Enlace **[Versión en caché]**
En este enlace encontrarás, de forma detallada, el uso de ficheros secuenciales y aleatorios en Java
Uso de Ficheros en Java **[Versión en caché]**

3.2. Flujos (Stream)

Flujos (Stream)

¿Cómo implementa Java el trabajo con ficheros?

Mediante una abstracción más amplia, que es el concepto de Flujo o Stream.

Para Java, los datos son algo que fluye en una corriente desde un origen (productor) hasta un destino (consumidor).

Básicamente considera dos grandes tipos de flujos de datos:

- **De entrada (`InputStream`).** En ellos nuestra aplicación recibe los datos. Es el **consumidor** que retira los datos del flujo, que puede provenir de un teclado, o de un fichero, por ejemplo.
- **De salida (`OutputStream`).** En ellos nuestra aplicación **produce** los datos, y los envía al flujo para que lleguen hasta su destino, que puede ser el monitor, la impresora, o un fichero, por ejemplo.

En Java, en los flujos se escriben o leen bytes, en principio, de forma que cualquier información que quiero escribir en un flujo tengo que ir descomponiéndola en bytes, y escribiendo cada uno de ellos en el flujo de salida. De la misma forma, para recuperar esa información, tendré que ir leyendo uno a uno los bytes que me va proporcionando el flujo, y a partir de ellos recomponer la información original.

Este proceso, que en principio es complicado, se puede simplificar. Por ahora debes tener claro que el flujo es como una especie de "manguera unidireccional", en la que la aplicación va metiendo la información byte a byte, si el flujo es de salida (escritura) o de la que la aplicación va recibiendo la información byte a byte si el flujo es de entrada (lectura).

La aplicación siempre ve el mismo extremo de la manguera, que siempre es igual. Pero la manguera en sí puede conectarse por el otro extremo a muy distintos orígenes o destinos de datos, a muy distintos dispositivos, sin que la aplicación tenga por qué saber nada sobre ellos, ni sobre sus características, ni sobre su forma de trabajar. Sólo debe tratar con el flujo, escribiendo bytes en la manguera de salida, o leyendo bytes de la manguera de entrada.

DEMO: Visualiza un ejemplo de cómo se pasa la información del programa a un soporte

Tanto `InputStream` como `OutputStream` son clases abstractas (más adelante verás su significado exacto) que definen una serie de características comunes para todo flujo de entrada o de salida, respectivamente. Así, al ser clases abstractas, no podré crear directamente objetos `InputStream,` pero todas las subclases que definen y crean flujos de entrada, heredan de `InputStream` algunos campos, y algunos métodos, y están obligadas a implementar el método `read()` que proporciona el siguiente byte de un flujo de entrada. Algo similar ocurre para `OutputStream,` que es la superclase de cualquier clase que defina flujos de salida, y que obliga a implementar el método `write()`para poner en el flujo un byte.

El concepto de flujo permite independizar aún más la aplicación de los dispositivos de entrada o salida.

La aplicación ya no se comunica con el sistema operativo cuando quiere realizar alguna operación de entrada o de salida. Siempre se lee de un flujo o se escribe en un flujo, y los flujos se definen por el lenguaje siempre de la misma forma. Será el flujo el que tenga que comunicarse con el Sistema Operativo concreto y "entendérselas" con él. De esta manera**, para que las operaciones de entrada/salida de nuestro programa funcionen usando otro dispositivo de entrada/salida o en otro sistema operativo (en cualquier otro sistema operativo) no tenemos que cambiar absolutamente nada en nuestra aplicación, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es importante en un lenguaje multiplataforma y tan altamente portable como Java.**

Debes saber también que cuando ejecutas cualquier aplicación Java se crean automáticamente tres objetos que son flujos:

- **System.err** Es un objeto de tipo `PrintStream,` que a su vez es una subclase de `FilterOutputStream,` que a su vez es una subclase de `OutputStream.` En definitiva, es un flujo de salida definido en la clase `System` y que representa la salida de error estándar. Por defecto, es el monitor, aunque es posible redireccionarlo a otro dispositivo de salida.
- **System.out** También es un objeto de tipo `PrintStream.` Es el flujo de salida definido en la clase `System` que representa la salida estándar. Por defecto también es el monitor, aunque puede

redireccionarse a otro dispositivo.

- **System.in** Es un objeto de tipo **`InputStream,`** y está definido en la clase **`System`** como flujo de entrada estándar. Por defecto es el teclado, pero puede redirigirse para cada host o cada usuario, de forma que se corresponda con cualquier otro dispositivo de entrada.

Esos tres flujos se usan en distintos métodos de la clase **`ES`**, tanto para leer desde teclado, como para enviar mensajes a la pantalla (que se escriben en rojo si se envían mediante **`System.err`**). Como ves, usar flujos no tiene porqué resultar complicado.

Autoevaluación

3.3. Jerarquía de clases para Flujos en Java

Jerarquía de clases para Flujos en Java

Ya hemos nombrado bastantes clases relacionadas con los flujos, ¡y todavía no hemos empezado a trabajar con ficheros! ¿Son muchas las clases de tipo Stream que define Java?

Sí, realmente hay toda una jerarquía de clases y subclase en Java. De hecho hay dos "familias de jerarquías" para flujos (Streams). Además de las clases bases para entrada y salida orientada a byte, que son **`InputStream`** y **`OutputStream`**, existen otras dos clases base para entrada y salida orientada a caracteres y texto, y que es recomendable usar cuando los datos que se manipulan son de tipo texto. Esas clases son **`Reader`** para entrada o lectura de caracteres y **`Writer`** para salida o escritura de caracteres. Estas clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por **dos bytes**.

De entre todas las clases que componen cada jerarquía, sólo unas pocas identifican orígenes o destinos de datos, es decir, distintos "dispositivos" con los que se conecta el Stream. Las hemos marcado en las tablas siguientes con un sombreado de color. Las otras clases sólo añaden características que permiten modificar la forma de trabajar con el flujo, y la forma de enviar los datos hacia o desde el flujo.

Jerarquía de clases InputStream

- InputStream
  - **FileInputStream**
  - PipedInputStream
  - ByteArrayInputStream
  - StringBufferInputStream
  - SequenceInputStream
  - FilterInputStream
    - DataInputStream
    - LineNumberInputStream
  - **BufferedInputStream**
    - PushbackInputStream
  - **ObjectInputStream**

Jerarquía de clases OutputStream

- OutputStream
  - **FileOutputStream**
  - PipedOutputStream
  - ByteArrayOutputStream
  - FilterOutputStream
    - DataOutputStream
    - PrintStream
  - **BufferedOutputStream**
    - PushbackOutputStream
  - **ObjectOutputStream**

Jerarquía de clases Reader

- Reader
  - CharArrayReader

Jerarquía de clases Writer

- Writer
  - CharArrayWriter

PipedReader	PipedWriter
StringReader	StringWriter
BufferedReader	**BufferedWriter**
LineNumberReader	OutputStreamWriter
InputStreamReader	**FileWriter**
FileReader	FilterWriter
FilterReader	PrintWriter
PushbackReader	

1. Cada nuevo margen indica que son subclases de la clase que hay encima en el nivel anterior. Así, `BufferedInputStream` es subclase de `FilterInputStream,` que a su vez es subclase de `InputStream.`
2. **Las clases sombreadas de color, son las que definen orígenes o destinos distintos de datos.** Así...
    1. `FileInputStream` establece un **fichero** como origen de datos.
    2. `PipedInputStream` establece un origen de datos desde **otra aplicación**, que puede estar ejecutándose concurrentemente en otro thread o hilo de ejecución.
    3. `ByteArrayInputStream` define como origen de datos la memoria, y ...
    4. `StringBufferInputStream` define un origen de datos consistente en un flujo de caracteres desde un String. Esta clase está marcada en la API de Java como "Deprecated". Es una forma de decir que está considerada obsoleta, que se mantendrá en futuras versiones del JDK para mantener la compatibilidad con aplicaciones que la usen, pero que en algún momento saldrá un nuevo JDK que ya no dispondrá de ella, por lo que se recomienda no usarla en las nuevas aplicaciones. En su lugar se recomienda usar la clase `StringReader,` más adecuada para trabajar con caracteres y texto.

Análogamente se pueden considerar como destinos de datos las clases similares de la jerarquía `OutputStream,` o las clases similares de las jerarquías `Reader` y `Writer.`

- **Las clases que no están sombreadas de color no definen flujos nuevos, si no que se "montan" sobre otros flujos para modificar la forma de trabajar con ellos.** Por ejemplo, con `BufferedInputStream` podemos añadir un buffer a un flujo `FileInputStream,` de forma que se mejore la eficiencia de los accesos a los dispositivos en los que se almacena el fichero con el que conecta el flujo.
- **Las clases en negrita son las directamente implicadas en la lectura/escritura en ficheros.**
- **Las clases `ObjectInputStream` y `ObjectOutputStream` se usan para la Serialización,** que consiste en modificar la forma de leer y escribir en los flujos de tipo `FileInputStream` y `FileOutputStream.` Usando las clases "Object..." podemos escribir y leer objetos en el flujo, en vez de bytes, lo cual simplifica mucho el trabajo.
- **Además de las clases que definen flujos, debemos usar para trabajar con ficheros la clase `File`, que nos permite definir una ruta hasta un archivo o un directorio. Un objeto `File` no es más que un path para el Sistema operativo.**

De todas las clases incluidas en estas jerarquías, sólo nos interesa por ahora ver las que tienen relación con los ficheros, que son las que estudiaremos en los apartados siguientes.

3.4. Clase File

Clase File

¿Todas las clases que proporciona Java para trabajar con flujos son directamente utilizables, de la misma manera que la clase `String`? La respuesta es no. La clase `String` está en un paquete que se importa automáticamente, pero no ocurre lo mismo con las clases relacionadas con ficheros.

Quizás sea conveniente, antes de nada, aclarar que todas las clases que nos permiten trabajar con ficheros en Java se encuentran disponibles en el paquete **`java.io,`** por lo que **lo primero que debemos hacer en toda aplicación que use las clases involucradas en la entrada/salida de ficheros es escribir la sentencia...**

```
import java.io.*;
```

Indica que deben importarse todas las clases del paquete **`java.io.`** Así estarán disponibles todas las clases, de forma que si en nuestra aplicación aparece alguna referencia a alguna clase que no está definida en nuestro programa, antes de que el compilador la dé por errónea, la buscará en el paquete **`java.io,`** y si la encuentra, importará el código que necesite desde ella.

Entre ellas se encuentra la clase **`File,`** que define una ruta hasta un fichero o un directorio, y nos permite consultar mediante toda una serie de métodos que define, información sobre el fichero o directorio que especifica esa ruta. Lo primero que podemos comprobar es si ese fichero existe o no.

En el enlace siguiente encontrarás un ejemplo en el que se puede ver el funcionamiento de algunos de los métodos principales que proporciona la clase **`File,`** y si ejecutas el ejemplo con distintas rutas, y para ficheros que existan y que no, y también para directorios, podrás apreciar la utilidad de cada uno.

Descarga el archivo Ejemplo_file.java

La clase **`File`**proporciona bastantes métodos más, además de los que aparecen en este ejemplo, para obtener la lista de ficheros de un directorio, para crear un nuevo directorio o un nuevo fichero, etc.

Para una referencia completa de los métodos disponibles en la clase **`File,`** consulta como siempre la API de Java.

3.5. Interfaz FilenameFilter

Interfaz FilenameFilter

Cuando se trabaja con ficheros, es importante poder ver la lista de todos los ficheros que contiene un directorio. En muchos casos, puede interesarnos no ver la lista completa, sino los que encajen con una cierta plantilla.

Por ejemplo, para ver sólo los ficheros relativos a trabajadores. Si todos esos ficheros tienen nombres que comienzan por "trab" nos interesa aplicar un filtro que sólo nos muestre los ficheros cuyo nombre comience por esas letras.

Podemos imaginar muchos filtros, para ver los ficheros creados o modificados después de una fecha, o los que tienen un tamaño mayor que el que indiquemos, etc.

El interface **`FilenameFilter`** se puede usar para crear filtros que establezcan condiciones de filtrado relativas al nombre de los ficheros. Obliga a cualquier clase que la implemente a definir e implementar el método...

```
boolean accept(File dir, String name)
```

Este método devolverá verdadero sólo en el caso de que el fichero cuyo nombre se indica en el parámetro **`name,`** aparezca en el listado de los ficheros del directorio indicado por el parámetro **`dir.`**

En el ejemplo que incluimos en el fichero **`ListadoDirectorio.java`** se muestra el uso del interface **`FilenameFilter`** y de la clase **`File.`** En el mismo fichero hay definidas dos clases:

- **`ListadoDirectorio,`** que tiene el método **`main(),`** establece un path mediante un objeto de tipo **`File`** que representa un directorio del que mostrar la lista de ficheros que encajen en la plantilla que se establece.
- **`Filtro,`** que implementa el interface **`FilenameFilter`** y que establece en el método **`accept()`** la condición que debe cumplir un fichero de ese directorio para aparecer en el listado.

El ejemplo está pensado para que se ejecute **desde la línea de comandos** (no desde NetBeans), pasándole como parámetro en la ejecución al método `main()` el `String` que vamos a usar como plantilla. Si no se especifica ningún parámetro, nos mostrará el listado de todos los ficheros y directorios de la ruta establecida por defecto en el programa, que inicialmente es el directorio actual. Te aconsejo que cambies esa ruta y pruebes distintas ejecuciones del programa, para entender mejor cómo funciona.

Un posible ejemplo de ejecución sería el que aparece en la imagen siguiente. Ten en cuenta que si tú lo ejecutas en tu equipo, el directorio de trabajo será otro, y tendrá otros ficheros distintos, por lo que el resultado de la ejecución no será el mismo:

Observa que en la primera ejecución:

`java ListadoDirectorio` nos muestra la lista de todos los archivos y directorios del directorio actual, que en este caso es **C:\PruebasJava**. Observa que aparece en la lista "**Litografias**", que es un directorio.

En la siguiente ejecución:

`java ListadoDirectorio Lit`, nos muestra la lista de todos los ficheros cuyo nombre contenga la cadena "**Lit**". Observa que en este caso Litografias no se muestra porque en la clase `Filtro` hemos establecido que el método `accept()` sólo devuelva verdadero si además de incluir la cadena pasada como parámetro en la ejecución, se trata de un fichero, comprobándolo con el método `isFile().`

Descarga el archivo ListadoDirectorio.java

Existe también un interface `FileFilter,` que se usa para establecer filtros más generales, que afecten a otras características de los ficheros, además del nombre. La filosofía es la misma, y nos obliga a implementar también un método `accept().` Para más información, como siempre, debes dirigirte a la API de Java.

A continuación te proponemos otro ejemplo en el que se maneja la clase `File` para trabajar con directorios (crearlos, borrarlos y renombrarlos.) También define un método `fechaEnEspañol()` que nos muestra la fecha de creación o última modificación de un fichero o directorio de una forma comprensible para nosotros, a partir del número long que devuelve el método `lastModified()` de la clase `File.` Recuerda que ese método devolvía la representación de la fecha asociada al fichero como el número de milisegundos transcurridos desde las cero horas del 1 de Enero de 1970 hasta esa fecha.

Este ejemplo también requiere ser ejecutado desde la línea de comandos. Si se ejecuta sin argumentos, nos escribe un texto de ayuda en el que explica los usos o posibles parámetros que pueden darse en la llamada y para qué se emplean. También puede ejecutarse introduciendo los argumentos para la aplicación desde el entorno NetBeans.

Descarga el archivo CrearDirectorios.java

Te mostramos cómo ejecutar el programa suministrándole una lista de parámetros equivalente a la de la línea de comandos desde el entorno NetBeans en una demostración.

DEMO: Visualiza una de las maneras de crear un proyecto nuevo a partir de código ya creado

3.6. FileInputStream y BufferedInputStream

FileInputStream y BufferedInputStream

Hasta ahora la clase `File` nos ha permitido manipular los ficheros y directorios de forma similar a como lo haríamos desde el Sistema Operativo, consultando su información, creándolos, borrándolos, renombrándolos, etc. Pero aún no hemos guardado información en ningún fichero. ¿No se suponía que los ficheros sirven para eso? ¿Qué flujos debo usar para escribir o leer algo en un fichero?

La clase `FileInputStream` es la clase básica **para leer datos desde un fichero**. Sirve para leer todo tipo de datos de todo tipo de ficheros. Esta clase trabaja con bytes, es decir, se leen las informaciones por bytes desde el flujo, que estará asociado a un fichero.

En el apartado 3.3, en la tabla con la jerarquía de clases, decíamos que `FileInputStream` define un

origen de datos, que es un fichero, pero que **`BufferedInputStream`** lo único que hace es modificar la forma de trabajar con ese flujo. En concreto, lo que hace es añadirle un buffer al flujo **`FileInputStream.`** ¿Qué significa eso?

Imagina que vivieras en una casa perdida en el campo, sin agua corriente, y que la fuente más cercana se encuentra a varios kilómetros de distancia. ¿Irías a beber a la fuente cada vez que tuvieras necesidad de beber? Es una posibilidad, pero seguro que te pasarías gran parte de tu tiempo en el camino. Ir a la fuente es una tarea lenta, que requiere tiempo. Lo más práctico sería aprovechar el viaje, y traerte de la fuente toda el agua que pudieras en cada viaje que tuvieras que dar, por ejemplo, una garrafa de 10 litros. Cada vez que sintieras sed, irías a la garrafa y beberías cómodamente, mientras la garrafa tuviera agua. Y si cuando vas a beber la garrafa está vacía, habrá que ir a la fuente de nuevo, pero te traerás otra garrafa llena, con la que seguramente tendrás para beber varios días. Así habrás logrado ir a la fuente una vez cada 3 ó cuatro días, en vez de 3 ó 4 veces cada día. Con todos los viajes que te estás ahorrando, seguro que dispones de mucho más tiempo para hacer otras tareas, como limpiar la casa, labrar el huerto, ordeñar las vacas o chatear con tus amigos en Internet.

Esa misma es la filosofía de un buffer, que es lo que proporciona **`BufferedInputStream.`** **Cuando una aplicación debe leer datos de un fichero, tiene que estar esperando a que el disco en el que está el fichero le suministre la información. Cualquier dispositivo de memoria masiva, por rápido que sea, es infinitamente más lento que la CPU del ordenador**. Si nuestra aplicación accede con mucha frecuencia al dispositivo, estará obligando a la CPU a permanecer parada, a la espera de que le llegue desde el fichero el dato solicitado. ¡Y el tiempo de CPU es costoso, ya que siempre tiene muchas otras tareas que atender! Por tanto, **resulta útil minimizar el número de accesos al fichero a fin de mejorar la eficiencia de la aplicación, y para ello se asocia al fichero una memoria intermedia (hace de intermediaria entre el fichero y la aplicación), de forma que cuando se necesita leer un byte del archivo, en realidad se traen hasta el buffer asociado al flujo (asociado a su vez al fichero) tanta información como le quepa. Mientras quede información en el buffer, nuestra aplicación leerá los datos directamente de la memoria principal, donde se encuentra el buffer, que es mucho más rápida que cualquier dispositivo de memoria masiva**. Así, reducimos el número de accesos al fichero, y mejoramos la eficiencia de nuestra aplicación, que tendrá distraída a la CPU el menor tiempo posible.

En la analogía, la memoria masiva es la fuente, el flujo será el camino que tiene que recorrer el agua para llegar a la casa desde la fuente, y el buffer sería la garrafa. La aplicación que consume los datos serías tú, que eres el que se bebe el agua. Con la ventaja adicional de que cuando la CPU no tenga otra cosa que hacer, puede "acercarse a la fuente y traer más agua para llenar la garrafa".

DEMO: Visualiza un ejemplo del funcionamiento del buffer de una aplicación

A continuación tienes el enlace a un ejemplo que muestra cómo crear y usar un flujo **`FileInputStream.`** También verás que se usa un **`BufferedInputStream`** para añadirle un buffer al flujo. Si en el ejemplo se elimina el objeto **`BufferedInputStream`** y se trabaja directamente con el objeto **`FileInputStream,`** todo funciona aparentemente igual. La diferencia es que si el fichero fuera realmente grande, y si estuviéramos leyéndolo entero constantemente (en vez de leer sólo dos insignificantes bloques de 50 bytes cada uno), la versión con **`BufferedInputStream`** sería mucho más eficiente.

Lee con atención las explicaciones que aparecen en los comentarios del código, y observa las formas alternativas de construir en una única sentencia tanto el objeto **`File,`** como el flujo **`FileInputStream`** y el **`BufferedInputStream.`**

Descarga el archivo EjemploFileInputStream.java

3.7. FileOutputStream y BufferedOutputStream

FileOutputStream y BufferedOutputStream

En el apartado anterior hemos visto cómo usar la clase **`FileInputStream`** para leer información desde un fichero. Evidentemente, también es posible escribir información desde nuestra aplicación en un fichero, pero para ello hay que usar un flujo de salida. Recuerda que los flujos son unidireccionales, y no puede usarse un flujo de entrada para escribir, ni un flujo en salida para leer.

La clase básica que nos permite usar un fichero para salida o escritura de bytes en él, es la clase `FileOutputStream.` La filosofía es la misma que para la lectura de datos, pero ahora el flujo es en dirección contraria, desde la aplicación que hace de fuente de datos hasta el fichero, que los "consume".

Un flujo de tipo **FileOutputStream,** puede construirse a partir de un objeto **File** o directamente a partir de un **String** que especifique un path. No obstante, también en este último caso se crea el objeto **File,** pero de forma implícita por el compilador, que es el que lo utiliza.

Al igual que antes, para mejorar la eficiencia de la aplicación reduciendo el número de accesos a los dispositivos de salida en los que se almacena el fichero, podemos montar un buffer asociado al flujo de tipo **FileOutputStream.** De eso se **encarga la clase BufferedOutputStream, que permite que la aplicación pueda escribir bytes en el flujo sin que necesariamente haya que llamar al sistema operativo subyacente para cada byte escrito.**

Puedes ver el funcionamiento de estas clases en el ejemplo siguiente:

Descarga el archivo EjemploFileOutputStream.java

3.8. Clases FileWriter, FileReader, BufferedWriter y BufferedReader

Clases FileWriter, FileReader, BufferedWriter y BufferedReader

En los apartados anteriores hemos usado las clases **FileInputStream** y **FileOutputStream** para leer y escribir en ficheros, que realmente contenían texto. Pero recuerda que en el apartado 3.3 veíamos que además de las clases base para entrada y salida orientada a byte, que son **InputStream** y **OutputStream,** existen otras dos clases base para entrada y salida orientada a caracteres y texto, y que es recomendable usar cuando los datos que se manipulan son de tipo texto. Esas clases son **Reader** para entrada o lectura de caracteres y **Writer** para salida o escritura de caracteres. Estas clases están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por dos bytes.

Las subclases de **Writer** y **Reader** que permiten trabajar con ficheros de texto son:

- **FileReader, para lectura o entrada desde un fichero de texto. Crea un flujo de entrada que trabaja con caracteres en vez de con bytes.**
- **FileWriter, para escritura o salida hacia un fichero de texto. Crea un flujo de salida que trabaja con caracteres en vez de con bytes.**

Estas clases sólo se podrán usar para manejar ficheros de texto, y además es recomendable usarlas en este caso, porque son más eficientes.

Por lo demás, la forma de trabajar con ellas es muy similar a la forma de trabajar con las clases vistas hasta ahora, aunque evidentemente disponen de sus propios métodos.

También se puede montar un buffer sobre cualquiera de los flujos que definen estas clases:

- **BufferedWriter se usa para montar un buffer sobre un flujo de salida de tipo FileWriter.**
- **BufferedReader se usa para montar un buffer sobre un flujo de entrada de tipo FileReader.**

Te proponemos un ejemplo de un rudimentario procesador de texto, que usa las clases **FileReader** y **FileWriter** para manipular ficheros de texto. De hecho, a la hora de guardar la información en el fichero, el programa nos da a elegir entre usar la clase **FileWriter** o **FileInputStream.** Igualmente, para abir el fichero y leer el texto que contiene, nos da a elegir entre usar la clase **FileReader** o la clase **FileInputStream.** También sirve de ejemplo de cómo usar las clases **BufferedWriter** y **BufferedReader.**

Adicionalmente se hacen una serie de comprobaciones, como por ejemplo, avisar si se intenta salir del programa sin haber guardado los cambios hechos sobre el texto en memoria, o avisar si se intenta abrir un nuevo fichero sin haber guardado el que tenemos en memoria, o confirmar antes de sobrescribir el fichero, si es que este ya existe, etc.

Descarga el archivo FicherosDeTexto.java

3.9. Serialización: Interface Serializable y clases ObjectInputStream y ObjectOutputStream

Serialización: Interface Serializable y clases ObjectInputStream y ObjectOutputStream

Hemos visto como crear ficheros de texto, y hemos mencionado que **`FileInputStream`** y **`FileOutputStream`** sirven realmente para almacenar cualquier otro tipo de dato u objeto. Pero para eso tendríamos que descomponer ese objeto en una cadena de bytes, y escribirlo en el flujo byte a byte. Y para leerlo, tendríamos que ir recibiendo uno a uno los bytes que formaban el objeto desde el flujo para a partir de ellos recomponer el objeto original. ¡Qué lío!

Sería estupendo que Java nos permitiera escribir y leer directamente objetos en los flujos asociados a los ficheros. Además, si para Java todo son objetos, no parece muy lógico que no se puedan escribir objetos en los flujos, ¿verdad?

Efectivamente, no sería muy lógico. Y por eso **Java proporciona un mecanismo para poder escribir y leer directamente objetos en los flujos asociados a los ficheros. Ese mecanismo recibe el nombre de Serialización.**

¿Qué debemos tener en cuenta para usar la serialización?

- **La Serialización consiste en la descomposición automática por parte de la máquina virtual de un objeto en una secuencia (serie) de bytes para poder escribirlos en un flujo asociado a un fichero**.
- Evidentemente también es posible hacer el proceso contrario de "deserialización", que lee del flujo asociado al fichero la cadena de bytes, y a partir de ellos reconstruye el objeto original en memoria.
- Para que los objetos de una clase puedan ser serializados, **esa clase debe implementar el interface `Serializable.`**
- El interface **`Serializable`** no requiere implementar ningún método. Es sólo una "etiqueta" que debe colocársele a nuestra clase para que el compilador sepa que debe proporcionar para ella el mecanismo de "descomposición en secuencias de bytes" de sus objetos.
- **Al serializar un objeto, también se serializan todos los objetos que lo compongan, siempre y cuando también pertenezcan a clases serializables.** Si al serializar un objeto la máquina virtual se encuentra que uno de sus campos es una referencia de un tipo serializable, seguirá esa referencia serializando también el objeto al que apunta.
- Por ejemplo, al serializar un objeto Persona, se serializa también el nombre de la persona, que es de tipo **`String,`** ya que la clase **`String`** también es serializable. Esto es una característica poderosa. Para grabar en fichero toda una lista enlazada de objetos **`Persona,`** por ejemplo, no sólo no tenemos que ir guardando de forma individual cada uno de los datos que componen a cada persona de la lista, es que ni siquiera tenemos que ir guardando las personas una a una. Basta con que escribamos en el flujo la referencia a la primera persona de la lista, y toda la lista entera será serializada y guardada correctamente en el fichero. Para leer los datos de esas personas del fichero, bastará con asignar el resultado de la lectura a la referencia que apunta a la primera persona de la lista para que tengamos reconstruida en memoria la lista completa. En realidad, para Java la referencia que apunta a la primera persona es como si no apuntara a la primera persona, si no a un único objeto "lista enlazada de personas" que es el que se serializa.
- **Para serializar debemos usar flujos `ObjectOutputStream` y `ObjectInputStream`**, sobre los que podremos escribir y leer objetos directamente. Realmente estos flujos modifican la forma de trabajar con los flujos sobre los que se montan, para que se puedan escribir objetos en ellos en vez de bytes.
- Como la serialización realmente transforma los objetos en cadenas de bytes, que son lo que se envía al fichero, **los flujos `ObjectOutputStream` deben montarse sobre un flujo `FileOutputStream` y los flujos `ObjectInputStream` deben montarse sobre un flujo `FileInputStream`**. No es posible usar Serialización con flujos de las jerarquías **`Writer`** y **`Reader`**.
- **Los flujos `ObjectOutputStream` y `ObjectInputStream` proporcionan métodos `writeObject()` y `readObject()` para la ecritura y lectura de objetos de los flujos, pero además proporcionan toda una serie de métodos para trabajar con datos de tipo primitivo**. Así tendremos un método write y un método read para cada tipo primitivo. Por ejemplo, **`writeInt()`** y **`readInt()`** permitirán escribir y leer valores int de los flujos.
- **Al "deserializar" (leer los objetos del flujo) debemos aplicar un casting explícito a la clase del objeto que queremos leer**. A fin de cuentas, lo que llega a la aplicación desde el flujo no es más que una serie de bytes, que pueden representar cualquier cosa. Al hacer el casting, realmente le estamos proporcionando a la máquina virtual la información que necesita sobre qué es lo que debe intentar ver en esa cadena de bytes, de forma que si debe buscar un objeto **`Persona`**, buscará un nombre (**`String`**), una edad (**`int`**) y un sexo (**`char`**) en esa cadena de bytes, y en ese orden. El casting le proporciona el nombre de la clase en la que "buscar los planos" para construir el objeto a partir de la

secuencia de bytes que recibe.

- Ya que al deserializar es necesario usar la clase del objeto para el casting explícito**, esta clase debe estar accesible, y si no lo está se producirá una** `ClassNotFoundException`, que deberá ser capturada convenientemente mediante un bloque `try-catch`.
- **Sólo son serializables los objetos. Si necesitamos guardar en el flujo como parte importante de la información alguna variable de tipo static, debe guardarse de forma independiente, aunque en el mismo flujo.**
- La serialización resulta de utilidad en algunos aspectos avanzados de Java, que se escapan a los propósitos de este módulo profesional, tales como RMI (invocación remota de métodos) y JavaBeans (creación de componentes reutilizables), así como para proporcionar una característica denominada "persistencia ligera" de objetos. Por tanto, debes saber que existen consideraciones adicionales sobre Serialización, más allá de lo que aquí explicamos. Nos limitamos a darte una herramienta para facilitar el almacenamiento y recuperación de datos desde ficheros, cuando esos datos están constituidos por objetos de clases que se han definido como serializables.

PARA SABER MÁS:
Como casi siempre, buscar información completa en la documentación que proporciona la Web de Sun es una buena idea. El siguiente enlace te lleva a la parte del tutorial de Java que habla de todo lo relativo a la Serialización
Serialización **[Versión en caché]**

Va siendo hora de ver un ejemplo de uso de la serialización. En el último apartado de la unidad 11, proponíamos un ejemplo para crear un array de Personas, que se creaba mediante la introducción de los datos de cada persona manualmente desde teclado. Una vez que teníamos creado ese array de Personas, cuando cerrábamos el programa, perdíamos todos los datos.

Lo lógico sería tener la posibilidad de guardar todos esos datos en un fichero y recuperarlos la próxima vez que abriéramos el programa, sin tener que empezar desde el principio. De hecho, si los datos de esas personas los utilizamos para algo, no sólo sería lógico. ¡Sería imprescindible!

En el ejemplo que proponemos a continuación nos vamos a encargar justamente de guardar los datos de todas esas personas en ficheros, y ya que una persona es un objeto de tipo Persona, la modificaremos para que sea Serializable, y guardaremos los datos usando serialización. Evidentemente, los recuperaremos usando también serialización.

Como siempre, en el código del programa destacamos con comentarios aquellos aspectos que debes tener más en cuenta.

Descarga el archivo PersonaSerializable.java

Descarga el archivo VectorPersonasSerializable.java

Autoevaluación