CLASE TEÓRICA 3 / 12

ÍNDICE DE CONTENIDO

3.1 EXCEPCIONES

Un punto trascendental en la elaboración y ejecución de programas es el manejo de errores, los cuales pueden ser detectados en dos fases:

- En la compilación, que serían los errores de sintaxis y de estos brinda mucha ayuda el compilador, ya que indica el número de la línea donde se detecto el error, así como el mensaje de error.
- En la ejecución, donde la mayoría de esos errores serían los de semántica o lógicos, como por ejemplo la división por cero, que el compilador en muchos de los casos no lo detecta, ya que puede ser algún dato que lea el programa el que se utilice, para realizar esa división y que en algún momento pueda llegar a ser cero. Si no se detectan a tiempo estos errores, puede hacer que el programa termine abortando su ejecución, dando resultados que no son los deseados, si es que llega a darlos, y en algunos casos, se puede llegar a perder inclusive información.

Por lo que es conveniente, que varios de estos errores sean manejados en el momento de la ejecución del programa, para que esta no se vea afectada.

Es necesario aislar la posible causa de error y manejarla, de tal forma que no cause problemas, ya sea indicando haga “algo” y que el programa aborte su ejecución o que prosiga su ejecución.

Cuando sucede una excepción impide la continuación de la ejecución de una parte del programa y no puede continuar, porque no se tiene la información suficiente para solucionar ese “problema o situación inesperada”.

JAVA permite seleccionar la parte del código donde se contemple que pueden suceder situaciones “inesperadas”, así como cuales podrían ser y en cada una de ellas se indica que acciones se van a seguir.

Ya que los errores son inevitables, las excepciones nos proporcionan una estructura de control que permite implementar los “casos normales” con facilidad y tratar separadamente los “casos excepcionales”

Además, las excepciones nos permitirán mantener información acerca de lo que falló (tipo de error, detalles relevantes y lugar en el que se produjo el error) y enviar esta información al método que queramos que se encargue de tratar el problema, todo esto sin interferir con el funcionamiento normal del programa (Ej. sentencias return).

Las excepciones en JAVA son objetos y tienen su propio árbol de jerarquía. La clase raíz de ellas es **Throwable**, que es una subclase de **Object** (la clase madre en JAVA). Los métodos que se definen para estos objetos serán los que manden los mensajes de error que estén relacionados con cada uno de los diferentes tipos de excepciones.

Los errores y las excepciones son subclases de Throwable, por lo que heredan sus métodos:

- `toString()` muestra el nombre de una excepción junto con el mensaje que devuelve `getMessage()`.
- `getMessage()` se utiliza para obtener un mensaje de error asociado con una excepción.
- `printStackTrace()` permite imprimir el registro del stack donde se inició la excepción.

Con el manejo de excepciones, un programa puede seguir su ejecución (en vez de terminar) después

de arreglar un problema en tiempo de ejecución. Esto colabora a que las aplicaciones sean robustas.

3.2 CLÁUSULA TRY - CATCH

Una excepción indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema no ocurre con frecuencia; si la "regla" es que una instrucción por lo general se ejecuta en forma correcta, entonces el problema representa la "excepción a la regla". El manejo de excepciones nos permite crear programas tolerantes a fallas que pueden resolver (o manejar) las excepciones. En muchos casos, esto permite a un programa continuar su ejecución como si no hubiera encontrado ningún problema.

Los problemas más severos podrían evitar que un programa continuara su ejecución normal, en vez de requerir que el programa notifique al usuario sobre el problema y luego termine de forma correcta o que intervenga el usuario. Cuando la JVM o un método detecta un problema, como un índice de arreglo inválido o el argumento de un método inválido, lanza una excepción; es decir, ocurre una excepción. Cuando se activa una excepción, se "lanza", alguien debe de capturarla y hacer "algo".

Se utilizan los controladores de excepción para determinar en un momento la excepción que se ha "lanzado". El manejo de excepciones en JAVA permite separar el código del problema a resolver, del código de errores que se puede generar.

Se delimita un bloque de código a través de la palabra clave **`try`** dentro del método, para aislar el código que puede generar una excepción, esto es para especificar el bloque de declaraciones cuyas excepciones serán controladas por medio de una serie de cláusulas **`catch`**, las cuales pude variar en cuanto a su número. Esto quiere decir que un mismo bloque delimitado por **`try`** puede tener varios cachetas, que detecten diferentes causas de excepción y diferentes acciones a seguir, estos son conocidos como manejadores de excepciones.

Cada bloque de código de **`catch`** funciona como un método de un solo parámetro o argumento, siendo éste el que indica que tipo de error o excepción es con el que se activa. La sintaxis general es:

```
try {
  //cuerpo del bloque que puede generar excepciones
}
catch(nombreDelError1 identificador1){
  // Método que maneja la excepción del nombreDelError1
}
catch(nombreDelError2 identificador2){
  // Método que maneja la excepción del nombreDelError2
}
  .
  .
  //Tantos cachetas como sean necesarios
  .
  //(número de excepciones que se puedan generar)
```

También se puede atrapar cualquier tipo de excepción, a través de la clase base **`Exception`** su sintaxis es la siguiente:

```
catch (Exception e)
{
// Bloque que maneja la excepción en general
}
```

Dentro del bloque se pueden manejar algunos de los métodos de la superclase de **`Exception`**, que es **`Throwable`**.

Cuando se lanza una excepción:

1. Se sale inmediatamente del bloque de código actual
2. Si el bloque tiene asociada una cláusula **catch** adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la cláusula **catch**.
3. Si no, se sale inmediatamente del bloque (o método) dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula **catch** apropiada.
4. El proceso continúa hasta llegar al método **main** de la aplicación. Si ahí tampoco existe una cláusula **catch** adecuada, la máquina virtual JAVA finaliza su ejecución con un mensaje de error.

Este modelo en el cual el control del programa no regresa al punto de lanzamiento, sino que sigue después del último bloque **catch**, se denomina modelo de terminación del manejo de excepciones. Otros lenguajes utilizan el modelo de reanudación del manejo de excepciones en el cual reanudan el control justo después del punto de lanzamiento (Ej. Visual Basic)

Al igual que para cualquier bloque de código, cuando un bloque **try** o un bloque **catch** termina, se destruyen todas las variables locales que hayan sido declaradas dentro de este bloque, quedando fuera de alcance.

```
public class EjemploExcepcion {

    public static void main (String args[]){
          int num = 5, den = 0;
          int res = 0;
          try{
                res = num / den; //división por cero
                System.out.println("Resultado = " + res);//Si se genera una
                                                        // excepción, esto no se
                                                        // imprime.
          }
          catch (ArithmeticException e){
                System.out.println("El denominador no puede ser 0");
          }
          System.out.println("Resultado = " + res);
    }
}
```

3.2.1 ORDEN DE LOS BLOQUES CATCH

Se produce un error en compilación si hay más de un bloque **catch** que coincida exactamente con el tipo de la excepción a atrapar.

Pero puede haber distintos **catch** para las excepciones de una jerarquía de clases.

Ej. si en un programa agregamos un catch (Exception e), que es la superclase de ArithmeticException, antes del catch (ArithmeticException ae), como vemos a continuación

```
try {
}
catch (Exception e){
}
catch (ArithmeticException ae){
}
```

Si se produce un error de división por cero dentro del bloque **try**, va a ser atrapado por el primer **catch**, o sea el **catch** (**Exception** e) pues **ArithmeticException** es del tipo **Exception**. Por lo tanto si la lista de bloques **catch** empiezan por la superclase, un error del tipo de una subclase nunca será atrapado por su tipo específico. Por lo tanto debemos ordenar los **catch** desde las subclases hasta

las superclases, para lograr que un error pueda sea atrapado por su tipo específico.

```
try {
}
catch (ArithmeticException ae){
}
catch (Exception e){
}
```

3.3 CLÁUSULA FINALLY

En ocasiones, nos interesa ejecutar un fragmento de código independientemente de si se produce o no una excepción (por ejemplo, cerrar un archivo que estemos manipulando).

Luego de un bloque **try** debe haber a continuación por lo menos un bloque **catch** o un bloque **finally**. Bloques **catch**, puede haber varios, pero bloque **finally**, uno solo o ninguno.

Es obligatorio que **haya** un bloque **finally** si no hay bloques **catch**, pero si hay por lo menos un bloque **catch** es optativo el bloque **finally**.

```
try {
    <código>
}
catch (<tipo_excepción> <nombre_parámetro>) {
    <código>
}
.
.
.
finally {
    <código>
}
```

Como un bloque finally casi siempre se ejecuta, se lo utiliza para liberar recursos, como ser:

- cerrar archivos
- cerrar conexiones a bases de datos
- cerrar conexiones de red
- etc.

3.4 CLÁUSULA THROWS

Los programas escritos en JAVA también pueden lanzar excepciones explícitamente mediante la instrucción **throw**, lo que facilita la devolución de un “código de error” al método que invocó el método que causó el error.

Al llamar a métodos, ocurre un problema con las excepciones. El problema es, si el método da lugar a una excepción, ¿quién la maneja? ¿El propio método?¿O el código que hizo la llamada al método?

Con lo visto hasta ahora, sería el propio método quien se encargara de sus excepciones, pero esto complica el código. Por eso otra posibilidad es hacer que la excepción la maneje el código que hizo la llamada.

Si ocurre una excepción en el método, el código abandona ese método y regresa al código desde el que se llamó al método. Allí se posará en el **catch** apropiado para esa excepción.

Cuando en la declaración de un método después de la lista de parámetros del método y antes del

cuerpo del mismo, aparece una cláusula **throws** seguida de una lista de excepciones (en este caso aparece solo una excepción, **ArithmeticException**, si son varias, estarán separadas por coma) que especifica las excepciones que este método podrá lanzar o alguna de sus subclases. Estas excepciones pueden lanzarse mediante instrucciones o por medio de llamadas a otros métodos en el cuerpo del método.

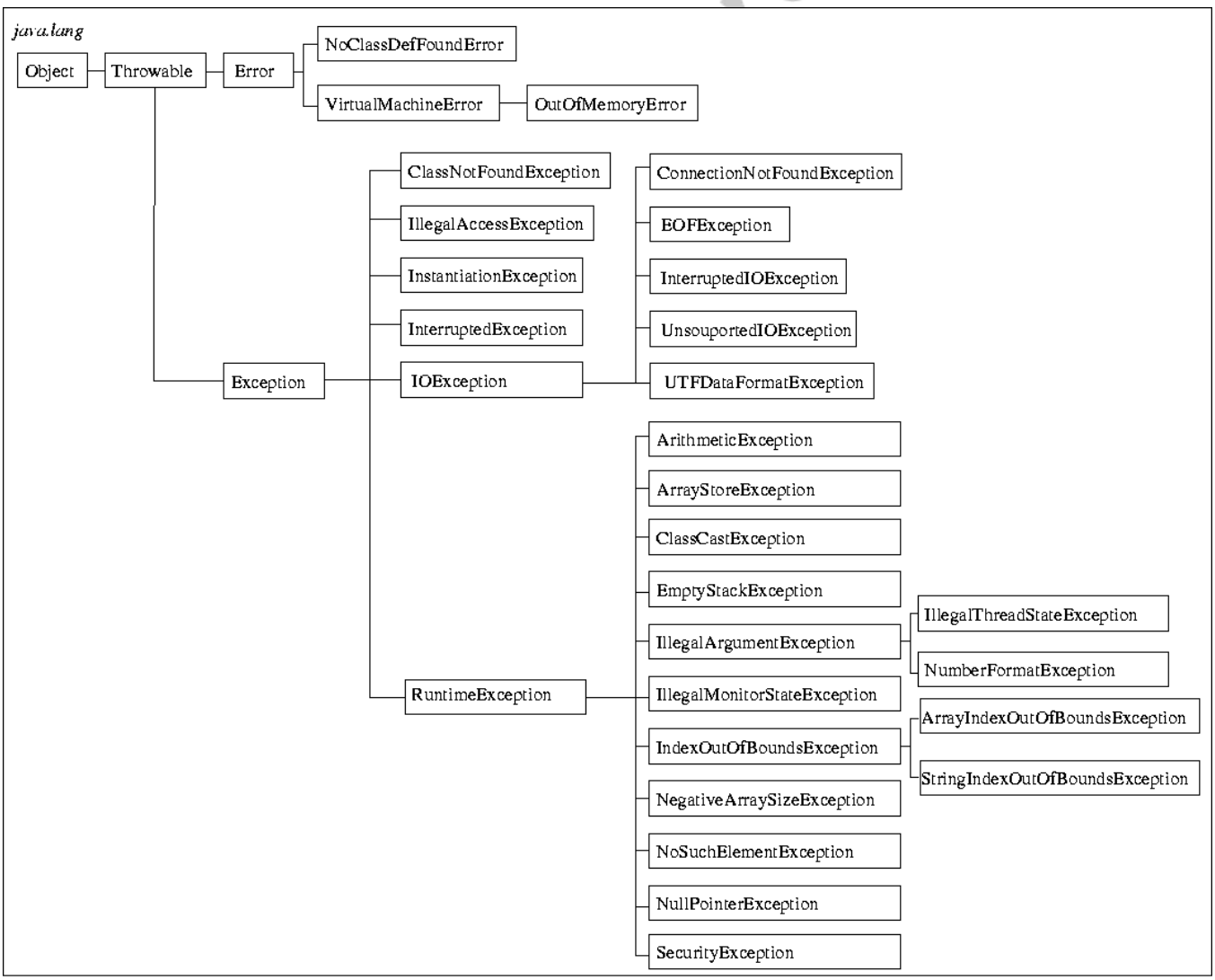
Para lanzar la excepción es necesario crear un objeto de tipo **Exception** o alguna de sus subclase (Ej. **ArithmeticException**) y lanzarlo mediante la instrucción **throw**.

```
public static void dividir(int num, int den) throws ArithmeticException {
    if (num / den < 1)
          throw new ArithmeticException();
    else
          System.out.println(num / den);
}
```

3.5 TIPOS DE EXCEPCIONES

Jerarquía de clases para el manejo de excepciones en JAVA

> Throwable



Clase base que representa todo lo que se puede “lanzar” en JAVA. Contiene una instantánea del

estado de la pila en el momento en el que se creó el objeto (“stack trace” o “call chain").

Almacena un mensaje (variable de instancia de tipo String) que podemos utilizar para detallar qué error se produjo.

Puede tener una causa, también de tipo Throwable, que permite representar el error que causó este error.

> Error

Subclase de **Throwable** que indica problemas graves que una aplicación no debería intentar solucionar (documentación de JAVA).

Ejemplos: Memoria agotada, error interno de la JVM...

> Exception

Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.

Los dos tipos principales de excepciones son:

RuntimeException (errores del programador, como una división por cero o el acceso fuera de los límites de un array) **throws**

IOException (errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa).

```
public void f() {
    // Fragmento de código libre de excepciones
    try {
          // Fragmento de código que puede
          // lanzar una excepción de tipo IOException
          // (Ej. Acceso a un archivo en el disco)
    }
    catch (IOException error) {
          // Tratamiento de la excepción
    }
    finally {
          // Liberar recursos (siempre se hace, haya o no error)
    }
}
```

La subclase **Exception** es el tipo básico que puede generarse desde cualquiera de los métodos de la biblioteca standard de JAVA.

JAVA divide las excepciones en dos categorías:

1) Las excepciones verificadas son todas las subclases de la clase **Exception**, menos las subclases de **RuntimeException**.

El compilador requiere que éstas excepciones sean atrapadas o declaradas.

Para satisfacer la parte de atrapadas, el código que genera la excepción debe estar encerrado en un bloque **try**, seguido de un bloque **catch** que tenga como parámetro este tipo de excepción verificada o alguna superclase de ésta.

Para satisfacer la parte de declaradas, en la declaración del método que contiene el código que genera la excepción, después de la lista de parámetros y antes de su cuerpo, debe escribirse la cláusula **throws** seguido del tipo de excepción verificada (puede haber más de una).

Da error en tiempo de compilación si, una excepción verificada, no está atrapada ni declarada.

2) Las excepciones no verificadas son todas las subclases de la clase **Error** y de la clase

RuntimeException. Las excepciones de la clase **Error** y sus subclases (Ej. **OutOfMemoryError**) representan situaciones anormales que pueden ocurrir en la JVM. Los errores del tipo **Error** ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones. En general, no es posible que una aplicación se recupere de un error del tipo **Error**.

Alguna de las subclases de **RuntimeException** son: **InputMismatchException**, **ArithmeticException**, **ArrayIndexOutOfBoundsException**, **ClassCastException**. Todas éstas subclases son no verificadas, por ende si no se las declara y/o atrapa, no hay error en tiempo de compilación.

No es necesario declarar las excepciones no verificadas en la cláusula **throws** de un método, y si se las declara, no es obligatorio que la aplicación atrape dichas excepciones.

Las clases de excepciones se definen verificables cuando son lo suficientemente importantes para atraparlas o declararlas.

3.5.1 Creación de nuevos tipos de excepciones

Un nuevo tipo de excepción puede crearse fácilmente: basta con definir una subclase de un tipo de excepción ya existente.

```
public class DivisionPorCero extends ArithmeticException {

    public DivisionPorCero(String message) {
          super(message);
    }
}
```

Una excepción de este tipo puede entonces lanzarse como cualquier otra excepción:

```
public double dividir(int num, int den) throws DivisionPorCero {
    if (den == 0)
          throw new DivisionPorCero("Error, el denominador no puede ser 0!");
    return ((double) num / (double) den);
}
```

Una nueva clase de excepción debe extender de una clase de excepción existente, para poder asegurar que dicha clase funcione con el mecanismo de excepciones. Una nueva clase de excepción puede contener atributos y métodos, pero en general va a contener sólo dos constructores:

- uno sin argumentos que le pasa un mensaje de excepción predeterminado al constructor de la superclase
- otro con un argumento, que recibe un mensaje de excepción personalizado y se lo pasa al constructor de la superclase

NOTA: Las aplicaciones suelen definir sus propias subclases de la clase **Exception** para representar situaciones excepcionales específicas de cada aplicación.

3.6 Vectores de referencia a objetos

En la Clase 1 se explicó y definió vectores de elementos de tipos primitivos.

En JAVA también podemos definir vectores cuyos elementos son de tipo referencia, como objetos.

Supongamos que queremos almacenar en un vector los datos de varias personas, podríamos definir un vector de personas, como sigue:

```
Persona vectorPersonas[] = new Persona [10];
```

donde la clase Persona podría estar definida de la siguiente forma:

```
public class Persona {
    private String nombre;
    private long DNI;
    private int edad;

    public Persona(String n, long dni, int ed){
          this.nombre = n;
          this.DNI = dni;
          this.edad = ed;
    }
}
```

Para inicializar cada elemento del vector con una referencia a una persona escribimos:

```
Persona vectorPersonas[] = new Persona[10];
String nom;
long dni;
int edad;
for (int i = 0; i < vectorPersonas.length; i++) {
    nom = JOptionPane.showInputDialog("Ingrese nombre");
    dni =  Long.parseLong(JOptionPane.showInputDialog("Ingrese su DNI"));
    edad = Integer.parseInt(JOptionPane.showInputDialog("Ingrese su edad"));
    vectorPersonas[i] = new Persona(nom, dni, edad);
}
```

3.6.1 Clase Arrays del paquete java.util

Dentro del paquete ***java.util*** se encuentra la clase **Arrays** que provee métodos **static** para trabajar con vectores de cualquier tipo.

Dichos métodos están sobrecargados para elementos **Object** y para todos los tipos primitivos. Algunos de los métodos definidos en Arrays son:

Métodos de Arrays	Significado
equals(v1,v2)	Devuelve true si v1[i]=v2[i] para todos los i dentro del rango
fill(v,x)	Llena el vector v con valores x
sort(v)	Ordena los elem del vector v en forma ascendente. Los elem del vector v deben ser de tipo primitivo o el tipo debe implementar la interfaz Comparable
binarySearch(v,x)	Devuelve el índice de la posición donde se encuentra x en v o un nro negativo si no está. El vector previamente debe estar ordenado y el tipo de cada elemento del vector debe ser primitivo o el tipo debe implementar la interfaz Comparable
deepToString(Object[] v)	Devuelve una representación string del contenido profundo del arreglo v
deepEquals(Object[] v1, Object[] v2)	Devuelve true si los dos arreglos son iguales en todos sus niveles
copyOfRange(Object[]v1, int pos, cant)	Devuelve un vector cuyas componentes son la copia de la cantidad **cant** de elementos desde la posición **pos** del v1

Ejemplo de uso de la clase Arrays

```
public class PruebaArrays {

    public static void main(String[] args) {
          char[] miArray1 = { 'b', 'e', 'd', 'h', 'j', 'a', 'c', 'f', 'g', 'i' };
          char[] miArray2 = { 'b', 'e', 'd', 'h', 'j', 'a', 'c', 'f', 'g', 'i' };
```

```
        // Compara 2 arrays
        if (Arrays.equals(miArray1, miArray2)) {
            System.out.println("Los dos arrays son iguales!");
        } else {
            System.out.println("Los dos arrays NO son iguales!");
        }

        miArray1[8] = 'k';

        // Imprimir array
        System.out.println("Array sin ordenar...");
        System.out.println(Arrays.toString(miArray1));

        // Ordenar el array
        Arrays.sort(miArray1);

        // imprime el array ordenado
        System.out.println("Array ordenado ...");
        System.out.println(Arrays.toString(miArray1));

        // Devuelve el índice de un valor particular
        int index = Arrays.binarySearch(miArray1, 'k');
        System.out.println("k esta en la posición " + index);

        String[][] taTeTi = { { "X", "O", "O" }, { "O", "X", "X" }, { "X", "O",
"X" } };

        System.out.println(Arrays.deepToString(taTeTi));

        // llena un vector entero con el valor 8
        int[] vecInt = new int[5];
        Arrays.fill(vecInt, 8);

        System.out.println(Arrays.toString(vecInt));
        System.out.println();

        // Copiar un vector a otro vector
        char[] vecA = { 'p', 'r', 'o', 'g', 'r', 'a', 'm', 'a', 'e', 'n', 'J',
'A', 'V', 'A' };

        char[] vecB = new char[7];
        // el método de la clase Arrays
        // devuelve un arreglo nuevo que se copia en vecC
        char[] vecC = Arrays.copyOfRange(vecA, 2, 9);
        // Copia con el método de la clase System
        System.arraycopy(vecA, 3, vecB, 0, 4);
        // crea un String con los caracteres del vector
        // e imprime el String
        System.out.println(new String(vecB));
        System.out.println(new String(vecC));
        // imprime el vector
        System.out.println(Arrays.toString(vecB));
        System.out.println(Arrays.toString(vecC));
    }
}
```

La salida del programa es la que sigue:

```
Los dos arrays son iguales!
Array sin ordenar...
[b, e, d, h, j, a, c, f, k, i]
Array ordenado ...
[a, b, c, d, e, f, h, i, j, k]
k esta en la posición 9
[[X, O, O], [O, X, X], [X, O, X]]
[8, 8, 8, 8, 8]

gram
ogramae
[g, r, a, m,
[o, g, r, a, m, a, e]
```

3.6.2 CLASE ARRAYLIST DEL PAQUETE *JAVA.UTIL*

Un **`ArrayList`** es un array dinámico. No tiene restricciones de capacidad. Su tamaño se ajusta de forma dinámica.

El Constructor por defecto es **`new`** *`ArrayList`*`()` que crea una lista vacía con una capacidad inicial de 10 elementos.

Los elementos dentro de un **`ArrayList`** son Objetos, no pueden ser de tipo simple. Para agregar un elemento utilizamos el método *`add`* y para obtener el elemento que está en una posición determinada usamos el método *`get`*.

```
import java.util.ArrayList;

public class PruebaArrayList {
    public static void main(String[] args) {
          ArrayList unArrayList = new ArrayList();
          unArrayList.add("hola");
          unArrayList.add(" , como");
          unArrayList.add("te");
          unArrayList.add("va");
          unArrayList.add("?");
          for (int i = 0; i < unArrayList.size(); i++)
                System.out.print(unArrayList.get(i) + " ");
    }
}
```

La salida del programa es:

```
hola , como te va ?
```

Métodos de ArrayList	Significado
`add``(Elemento e)`	Agrega al final de la lista el Elemento e especificado
`add``(int i, Elemento e)`	Inserta en la posición i el Elemento e
`clear``()`	Elimina todos los elementos de la lista
`contains``(Elemento e)`	Devuelve true si el Elemento e esta dentro de la lista
`get``(int i)`	Devuelve el Elemento que se encuentra en la posición i
`indexOf``(Elemento e)`	Devuelve la posición de la primera ocurrencia en que aparece el Elemento e. Si no está en la lista devuelve -1
`isEmpty()`	Devuelve true si la lista está vacía
`remove``(int i)`	Elimina el elemento que está en la posición i
`remove``(Elemento e)`	Elimina la primera ocurrencia del Elemento e, si existe
`size``()`	Devuelve el número de elementos que tiene la lista

3.7 FLUJOS DE DATOS

La comunicación entre el programa y el origen o el destino de cierta información, se realiza mediante un flujo de datos (maestrea) que no es mas que un objeto que hace de intermediario entre el programa y el origen o destino de la información. Esto es, el programa leerá o escribirá en el flujo sin importarle desde donde viene la información o a donde va y tampoco importa el tipo de los datos.

El paquete *java.io* contiene todas las clases relacionadas con las funciones de entrada (input) y salida (output). Se habla de E/S (o de I/O) refiriéndose a la entrada y salida de datos hacia o desde un

programa.

Los datos fluyen en serie, byte a byte de forma secuencial. Se habla entonces de un ***Stream*** (flujo de datos, o mejor dicho, flujo de bytes). Hay otro stream que lanza caracteres (tipo char Unicode, de dos bytes), se habla entonces de un ***Reader*** (si es de lectura) o un ***Writer*** (escritura).

Los problemas de entrada / salida suelen causar excepciones de tipo **`IOException`** o de sus derivadas. Con lo que la mayoría de operaciones deben ir inmersas en un **`try`**.

Clases de streams de caracteres	Clases de streams de bytes equivalentes
Reader **BufferedReader** **LineNumberReader** **CharArrayReader** **InputStreamReader** **FileReader** **FilterReader** **PushbackReader** **PipedReader** **StringReader**	**InputStream** **BufferedInputStrem** **LineNumberInputStream** **ByteArrayInputStream** (ninguna) **FileInputStream** **FilterInputStream** **PushbackInputStream** **PipedInputStream** **StringBufferInputStream**
Writer **BufferedWriter** **CharArrayWriter** **FilterWriter** **OutputStreamWriter** **FileWriter** **PrintWriter** **PipedWriter** **StringWriter**	**OutputStream** **BufferedOutputStream** **ByteArrayOutputStream** **FilterOutputStream** (ninguna) **FileOutputStream** **PrintStream** **PipedOutputStream** (ninguna)

3.7.1 Las clases in y out

`java.lang.`**`System`** es una clase que poseen multitud de pequeñas clases relacionadas con la configuración del sistema. Entre ellas están la clase **`in`** que es un **`InputStream`** que representa la entrada estándar (normalmente el teclado) y **`out`** que es un **`OutputStream`** que representa a la salida estándar (normalmente la pantalla). El uso podría ser:

```
InputStream ingreso = System.in;
OutputStream salida = System.out;
```

El método *`read()`* permite leer un byte. Este método puede lanzar excepciones del tipo **`IOException`** por lo que debe ser capturada dicha excepción.

```
int valor = 0;
try {
    valor = System.in.read();
}
catch (IOException e) {
    ...
}
System.out.println(valor);
```

No tiene sentido el código anterior, ya que *`read()`* lee un byte de la entrada estándar, y en esta entrada se suelen enviar caracteres, por lo que el método *`read`* no es el apropiado. El método *`read()`* puede poseer un argumento que es un array de bytes que almacenará cada carácter leído y devolverá el número de caracteres leído, pero demanda una gran complejidad de codificación.

Otra opción posible es utilizando un **`StringBuffer`**

```
public static void main(String arg[]) {
    StringBuffer salida = new StringBuffer();
    char caracter;
    System.out.println("\nEscriba un texto: ");
    try {
          do {
                caracter = (char) System.in.read();
                // Es necesaria una conversión de int a char
                salida.append(caracter);
```

```
            } while (caracter != '\n');
        }
        catch (Exception e) {
            System.err.println(e);
        }
        System.out.println("\nLa línea escrita es: ");
        System.out.println(salida);
    }
```

3.7.2 Reader

El hecho de que las clases **InputStream** y **OutputStream** usen el tipo byte para la lectura, complica mucho su uso. Desde que se impuso Unicode y con él las clases **Reader** y **Writer**, hubo que resolver el problema de tener que usar las dos anteriores.

La solución fueron dos clases: **InputStreamReader** y **OutputStreamWriter**. Se utilizan para convertir secuencias de byte en secuencias de caracteres según una determinada configuración regional. Permiten construir objetos de este tipo a partir de objetos **InputStream** u **OutputStream**. Puesto que son clases derivadas de **Reader** y **Writer** el problema está solucionado.

El constructor de la clase **InputStreamReader** requiere un objeto **InputStream** y, opcionalmente, una cadena que indique el código que se utilizará para mostrar caracteres (por ejemplo “ISO-8914-1” es el código Latín 1, el utilizado en la configuración regional). Sin usar este segundo parámetro se construye según la codificación actual (es lo normal).

Lo que hemos creado de esa forma es un objeto convertidor. De esa forma podemos utilizar la función *read* orientada a caracteres Unicode que permite leer caracteres extendidos. Está función posee una versión que acepta arrays de caracteres, con lo que la versión *writer* del código anterior sería:

```
InputStreamReader stdin = new InputStreamReader(System.in);
char caracter[] = new char[1024];
int numero = -1;
try {
    numero = stdin.read(caracter);
} catch (IOException e) {
    System.out.println("Error en la lectura");
}
for (int i = 0; i < numero; i++)
    System.out.print(caracter[i]);
```

3.7.3 Lectura con readLine

El uso del método *read* con un array de caracteres sigue siendo un poco enrevesado. Por ello para leer cadenas de caracteres se suele utilizar la clase **BufferedReader**. La razón es que esta clase posee el método *ReadLine()* que permite leer una línea de texto en forma de String, que es más fácil de manipular. Esta clase usa un constructor que acepta objetos ***Reader*** (y por lo tanto **InputStreamReader**, ya que desciende de ésta) y, opcionalmente, el número de caracteres a leer.

Hay que tener en cuenta que el método *ReadLine* (como todos los métodos de lectura) puede provocar excepciones de tipo **IOException** por lo que, como ocurría con las otras lecturas, habrá que capturar dicha lectura.

```
String texto = "";
try {
    // Obtención del objeto Reader
    InputStreamReader conv = new InputStreamReader(System.in);
    // Obtención del BufferedReader
    BufferedReader entrada = new BufferedReader(conv);
    System.out.println("Ingrese un texto");
    texto = entrada.readLine();
} catch (IOException e) {
    System.out.println("Error");
}
System.out.println(texto);
```

3.8 Archivos

Un archivo es una colección de datos relacionada lógicamente, que se almacena en algún soporte físico (memoria, disco rígido…) para poder manipularla en cualquier momento. Son transmitidos como flujos de datos por el sistema.

En el paquete *java.io* se encuentra la clase **File** pensada para poder realizar operaciones de información sobre archivos. No proporciona métodos de acceso a los archivos, sino operaciones a nivel de sistema de archivos (listado de archivos, crear carpetas, borrar archivos, cambiar nombre,...).

La dificultad de este tipo de operaciones está en que los sistemas de archivos (*File System*) son distintos en cada sistema operativo y aunque JAVA intentar aislar la configuración específica, no consigue evitarlo del todo.

Para construir una referencia a un archivo se utiliza como único argumento una cadena que representa una ruta en el sistema de archivo. También puede recibir, opcionalmente, un segundo parámetro con una ruta segunda que se define a partir de la posición de la primera.

```
File archivo1 = new File("/datos/archivo.txt");
File archivo2 = new File("archivo.txt");
```

El primer formato utiliza una ruta absoluta y el segundo una ruta relativa. La ruta absoluta se realiza desde la raíz de la unidad de disco en la que se está trabajando y la relativa cuenta desde la carpeta actual de trabajo.

Otra posibilidad de construcción es utilizar como primer parámetro un objeto **File** ya hecho. A esto se añade un segundo parámetro que es una ruta que cuenta desde la posición actual.

```
File carpeta1 = new File("c:/datos");// ó c\\datos
File archivo3 = new File(carpeta1, "archivo.txt");
```

Si el archivo o carpeta que se intenta examinar no existe, la clase **File** no devuelve una excepción. Habrá que utilizar el método *exists*. Este método recibe true si la carpeta o archivo es válido (puede provocar excepciones **SecurityException**).

También se puede construir un objeto **File** a partir de un objeto URL.

Cuando se crean programas en JAVA hay que tener muy presente que no siempre sabremos qué sistema operativo utilizará el usuario del programa. Esto provoca que la realización de rutas sea problemática porque la forma de denominar y recorrer rutas es distinta en cada sistema operativo.

Por ejemplo en MS Windows se puede utilizar la barra / o la doble barra invertida \\ como separador de carpetas, en muchos sistemas Unix sólo es posible la primera opción. En general es mejor usar las clases **Swing** (como **JFileDialog**) para especificar rutas, ya que son clases en las que la ruta de elige desde un cuadro y, sobre todo, son independientes de la plataforma.

También se pueden utilizar las variables estáticas que posee File. Estas son:

Propiedad	Uso
`char separatorChar`	El carácter separador de nombres de archivo y carpetas. En Linux/ Unix es "/" y en MS Windows es "\", que se debe escribir como \\, ya que el carácter \ permite colocar caracteres de control, de ahí que haya que usar la doble barra.
`String separator`	Como el anterior pero en forma de **`String`**
`char pathSeparatorChar`	El carácter separador de rutas de archivo que permite poner más de un archivo en una ruta. En Linux/Unix suele ser ":", en MS Windows es ";"
`String pathSeparator`	Como el anterior pero en forma de **`String`**

Para poder garantizar que el separador usado es el del sistema en uso:

```
String ruta = "documentos/manuales/java.pdf";
ruta = ruta.replace('/', File.separatorChar);
```

Método	Uso
`String toString()`	Para obtener la cadena descriptiva del objeto
`boolean exists()`	Devuelve **`true`** si existe la carpeta o archivo
`boolean canRead()`	Devuelve **`true`** si el archivo se puede leer
`boolean canWrite()`	Devuelve **`true`** si el archivo se puede escribir
`boolean isHidden()`	Devuelve **`true`** si el objeto **`File`** es oculto
`boolean isAbsolute()`	Devuelve **`true`** si la ruta indicada en el objeto **`File`** es absoluta
`boolean equals(File f2)`	Compara f2 con el objeto File y devuelve **`true`** si son iguales.
`String getAbsolutePath()`	Devuelve una cadena con la ruta absoluta al objeto **`File`**.
`File getAbsoluteFile()`	Como la anterior pero el resultado es un objeto **`File`**
`String getName()`	Devuelve el nombre del objeto **`File`**.
`String getParent()`	Devuelve el nombre de su carpeta superior si la hay y si no **`null`**
`File getParentFile()`	Como la anterior pero la respuesta se obtiene en forma de objeto **`File`**.
`boolean setReadOnly()`	Activa el atributo de sólo lectura en la carpeta o archivo.
`URL toURL()` throws **`MalformedURLException`**	Convierte el archivo a su notación URL correspondiente
`URI toURI()`	Convierte el archivo a su notación URI correspondiente
`boolean isDirectory()`	Devuelve true si el objeto **`File`** es una carpeta y false si es un archivo o si no existe.
`boolean mkdir()`	Intenta crear una carpeta y devuelve true si fue posible hacerlo
`boolean mkdirs()`	Usa el objeto para crear una carpeta con la ruta creada para el objeto y si hace falta crea toda la estructura de carpetas necesaria para crearla.
`boolean delete()`	Borra la carpeta y devuelve true si puedo hacerlo
`String[] list()`	Devuelve la lista de archivos de la carpeta representada en el

	objeto **File**.
`static File[] listRoots()`	Devuelve un array de objetos **File**, donde cada objeto del array representa la carpeta raíz de una unidad de disco.
`File[] listfiles()`	Igual que la anterior, pero el resultado es un array de objetos **File**.
`boolean isFile()`	Devuelve true si el objeto **File** es un archivo y false si es carpeta o si no existe.
`boolean renameTo(File f2)`	Cambia el nombre del archivo por el que posee el archivo pasado como argumento. Devuelve true si se pudo completar la operación.
`boolean delete()`	Borra el archivo y devuelve true si puedo hacerlo
`long length()`	Devuelve el tamaño del archivo en bytes
`boolean createNewFile() Throws IOException`	Crea un nuevo archivo basado en la ruta dada al objeto **File**. Hay que capturar la excepción **IOException** que ocurriría si hubo error crítico al crear el archivo. Devuelve true si se hizo la creación del archivo vacío y false si ya había otro archivo con ese nombre.
`static File createTempFile(String prefijo, String sufijo)`	Crea un objeto **File** de tipo archivo temporal con el prefijo y sufijo indicados. Se creará en la carpeta de archivos temporales por defecto del sistema. El prefijo y el sufijo deben de tener al menos tres caracteres (el sufijo suele ser la extensión), de otro modo se produce una excepción del tipo **IllegalArgumentsException** Requiere capturar la excepción **IOException** que se produce ante cualquier fallo en la creación del archivo
`static File createTempFile(String prefijo, String sufijo, File directorio)`	Igual que el anterior, pero utiliza el directorio indicado.
`void deleteOnExit()`	Borra el archivo cuando finaliza la ejecución del programa

3.8.1 Lectura y escritura byte a byte

Los archivos binarios guardan una representación de los datos en el archivo, es decir, cuando guardamos texto no guardan el texto en si, sino que guardan su representación en un código llamado UTF-8.

Para leer y escribir datos a archivos, JAVA utiliza dos clases especializadas que leen y escriben orientando a byte, son **FileInputStream** (para la lectura) y **FileOutputStream** (para la escritura).

Se crean objetos de este tipo construyendo con un parámetro que puede ser una ruta o un objeto **File**:

```
FileInputStream flujo1 = new FileInputStream(objetoFile);
FileInputStream flujo2 = new FileInputStream("/textos/texto25.txt");
```

La construcción de objetos **FileOutputStream** se hace igual, pero además se puede indicar un segundo parámetro booleano que con valor **true** permite añadir más datos al archivo (normalmente al escribir se borra el contenido del archivo, valor **false**).

Estos constructores intentan abrir el archivo, generando una excepción del tipo **FileNotFoundException** si el archivo no existiera u ocurriera un error en la apertura.

Los métodos de lectura y escritura de estas clases son los heredados de las clases **`InputStream`** y **`OutputStream`**. Los métodos *`read`* y *`write`* son los que permiten leer y escribir. El método *`read`* devuelve -1 en caso de llegar al final del archivo.

Otra posibilidad, más interesante, es utilizar las clases **`DataInputStream`** y **`DataOutputStream`**. Estas clases están mucho más preparadas para escribir datos de todo tipo.

El proceso sería:

1. Crear un objeto **`FileOutputStream`** a partir de un objeto **`File`** que posee la ruta al archivo que se desea escribir.
2. Crear un objeto **`DataOutputStream`** asociado al objeto anterior. Esto se realiza en la construcción de este objeto.
3. Usar el objeto del punto 2 para escribir los datos mediante los métodos *`writeTipo`* donde tipo es el tipo de datos a escribir (Int, Double, ...). A este método se le pasa como único argumento los datos a escribir.
4. Se cierra el archivo mediante el método close del objeto **`DataOutputStream`**.

El proceso de lectura es análogo. Sólo que hay que tener en cuenta que al leer se puede alcanzar el final del archivo. Al llegar al final del archivo, se produce una excepción del tipo **`EOFException`** (que es subclase de **`IOException`**), por lo que habrá que controlarla.

```
public static void main(String args[]) {
    File f = new File("/home/profmatiasgarcia/Documentos/prueba.out");
    Random r = new Random(); // un objeto Random
    boolean finArchivo = false;// Para bucle infinito
    double d = 1.00;

    //Escritura
    try {
          FileOutputStream fos = new FileOutputStream(f);
          DataOutputStream dos = new DataOutputStream(fos);
          for (int i = 0; i < 256; i++) { // Se repite 256 veces
                dos.writeDouble(r.nextDouble() * 10);// Aleatorios entre 0 y 10
          }
          dos.close();
    }
    catch (FileNotFoundException e) {
          System.out.println("No se encontró el archivo");
    }
    catch (IOException e) {
          System.out.println("Error al escribir");
    }
    System.out.println("<< Se escribió en el archivo >>");

    //Lectura del archivo
    try {
          FileInputStream gis = new FileInputStream(f);
          DataInputStream dis = new DataInputStream(fis);
          while (!finArchivo) {
          d = dis.readDouble();
                System.out.println(d);
          }
          dis.close();
    }
    catch (EOFException e) {
          finArchivo = true; //la excepción corta el bucle
    }
    catch (FileNotFoundException e) {
          System.out.println("No se encontró el archivo");
    }
    catch (IOException e) {
          System.out.println("Error al leer");
    }
}
```

3.8.2 LECTURA Y ESCRITURA MEDIANTE CARACTERES

Como ocurría con la entrada estándar, se puede convertir un objeto **FileInputStream** o **FileOutputStream** a forma de Reader o Writer mediante las clases **InputStreamReader** y **OutputStreamWriter**.

Existen además dos clases que manejan caracteres en lugar de bytes (lo que hace más cómodo su manejo), son **FileWriter** y **FileReader**.

La construcción de objetos del tipo **FileReader** se hace con un parámetro que puede ser un objeto **File** o un **String** que representarán a un determinado archivo.

La construcción de objetos **FileWriter** se hace igual sólo que se puede añadir un segundo parámetro booleano que, en caso de valer **true**, indica que se abre el archivo para añadir datos; en caso contrario se abriría para grabar desde cero (se borraría su contenido).

Para escribir se utiliza *write* que es un método **void** que recibe como parámetro lo que se desea escribir en formato **int**, **String** o array de caracteres. Para leer se utiliza el método *read* que devuelve un **int** y que puede recibir un array de caracteres en el que se almacenaría lo que se desea leer. Ambos métodos pueden provocar excepciones de tipo **IOException**.

```
public static void main(String args[]) {
    File f = new File("archivo.txt");
    int x = 77;
    try {
          FileWriter fw = new FileWriter(f);
          fw.write(x); //escribe M (ASCII 77) en el archivo
          fw.close();
    }
    catch (IOException e) {
          System.out.println("error");
          return;
    }
    // Lectura de los datos
    try {
          FileReader fr = new FileReader(f);
          x = fr.read(); //Lee el entero que corresponde al ASCII de M
          fr.close();
    }
    catch (FileNotFoundException e) {
          System.out.println("Error al abrir el archivo");
    }
    catch (IOException e) {
          System.out.println("Error al leer");
    }
    System.out.println(x);
}
```

Otra forma de escribir datos (imprescindible en el caso de escribir texto) es utilizar las clases **BufferedReader** y **BufferedWriter** vistas en el tema anterior. Su uso sería:

```
public static void main(String[] args) {
    File f = new File("archivo.txt");

    FileWriter fw = null;
    BufferedWriter bw = null;
    FileReader fr = null;
    BufferedReader br = null;

    //Escribir en el archivo
    try {
          fw = new FileWriter(f);
          bw = new BufferedWriter(fw);
          String s = "Prof Matias Garcia";
          for (int i=0; i < 10; i++){
                bw.write(s);
                bw.write("\n");
          }
    }
```

```
        catch (FileNotFoundException e) {
                System.out.println("Error al abrir el archivo");
        }
        catch (IOException e) {
                System.out.println("Error al escribir");
        }
        finally {
                try {
                        if (bw != null)
                                bw.close();

                        if (fw != null)
                                fw.close();
                }
                catch (IOException ex) {
                        ex.printStackTrace();
                }
        }

        //Leer el archivo
        try {
                fr = new FileReader(f);
                br = new BufferedReader(fr);
                String s;
                do {
                        s = br.readLine();
                        System.out.println(s);
                } while (s != null);
        }
        catch (FileNotFoundException e) {
                System.out.println("Error al abrir el archivo");
        }
        catch (IOException e) {
                System.out.println("Error al leer");
        }
        finally {
                try {
                        if (br != null)
                                br.close();

                        if (fr != null)
                                fr.close();
                }
                catch (IOException ex) {
                        ex.printStackTrace();
                }
        }
}
```

3.8.3 ARCHIVOS DE ACCESO ALEATORIO

La clase **RandomAccessFile** permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento, situando el puntero del archivo en una posición determinada sin tener que recorrer previamente el contenido . Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último. No permite seriar objetos.

Para poder situarnos en diferentes posiciones del archivo se utiliza un puntero de Lectura/Escritura.

Esta es una clase primitiva que implementa los interfaces **DataInput** y **DataOutput** y sirve para leer y escribir datos.

La construcción requiere de una cadena que contenga una ruta válida a un archivo o de un archivo **File**. Hay un segundo parámetro obligatorio que se llama modo. El modo es una cadena que puede contener una r (lectura), w (escritura) o ambas, rw.

Como ocurría en las clases anteriores, hay que capturar la excepción **FileNotFound** cuando se ejecuta el constructor.

```
File f = new File("archivo.out");
RandomAccessFile archivo = new RandomAccessFile( f, "rw");
```

Método	Uso
`seek(long pos)`	Permite colocarse en una posición concreta, contada en bytes, en el archivo. Lo que se coloca es el puntero de acceso que es la señal que marca la posición a leer o escribir.
`skipBytes(int n)`	Posicionamiento relativo saltando n bytes
`long getFilePointer()`	Posición actual del puntero de acceso
`long length()`	Devuelve el tamaño del archivo

`readBoolean()`
`readByte()`
`readChar()`
`readInt()`
`readDouble()`
`readFloat()`
`readUTF()`
`readLine()`
	Funciones de lectura.
	Leen un dato del tipo indicado.
	En el caso de readUTF lee una cadena en formato Unicode.

`writeBoolean()`
`writeByte()`
`writeChar()`
`writeInt()`
`writeDouble()`
`writeFloat()`
`writeUTF()`
`writeLine()`
	Funciones de escritura.
	Todas reciben como parámetro, el dato a escribir. Escribe encima de lo ya escrito.
	Para escribir al final hay que colocar el puntero de acceso al final del archivo.

```
public static void main(String[] args) {
    BufferedReader teclado;
    InputStreamReader inStream;
    PrintWriter pantalla;
    String nombrearchivo, posición, b;
    RandomAccessFile archivo;
    inStream = new InputStreamReader(System.in);
    teclado = new BufferedReader(inStream);
    File f;
    long punteroF;
    try {
          // PrintWriter con autoflush a true
          pantalla = new PrintWriter(System.out, true);
          pantalla.println("Nombre del archivo:");
          nombrearchivo = teclado.readLine();
          f = new File(nombrearchivo);
          // apertura del archivo en modo lectura/escritura.
          archivo = new RandomAccessFile(f, "rw");
          pantalla.println("Para terminar CTRL+d en Linux o CTRL+z en MS
Windows");
          pantalla.println("byte a examinar:");
          while ((posición = teclado.readLine()) != null) {
                try {
                      // buscar la posición en el archivo.
                      // Integer.parseInt() obtiene int a partir
                      // de un String
                      archivo.seek(Integer.parseInt(posición));
                      // obtener el puntero del archivo
                      punteroF = archivo.getFilePointer();
                      // leer y mostrar el byte en la posición
                      pantalla.println("Valor del byte = " + archivo.readByte());
                      pantalla.println("Nuevo valor del byte:");
                      b = teclado.readLine();
                      // posicionar el puntero del archivo en
                      // el lugar guardado anteriormente
                      archivo.seek(punteroF);
                      // escribir el nuevo valor del byte
                      archivo.writeByte(Integer.parseInt(b));
                }
                catch (IOException e) {
                      System.err.println("No existe ese byte.");
                }
                catch (NumberFormatException e) {
                      System.err.println("error num.");
                }
                pantalla.println("byte a examinar:");
          }
          teclado.close();
          pantalla.close();
          archivo.close();
    }
    catch (IOException e) {
```

```
            System.err.println("Error de E/S");
        }
    }
```

3.8.4 SERIALIZACIÓN

Normalmente la operación de enviar una serie de objetos a un archivo en disco para hacerlos persistentes recibe el nombre de seriación, y la operación de leer o recuperar su estado (valores de atributos, etc) del archivo para reconstruirlos en memoria recibe el nombre de deseriación.

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz **`Serializable`** que es la que permite esta operación. Si un objeto ejecuta esta interfaz puede ser guardado y restaurado mediante una secuencia.

Mediante este mecanismo pueden almacenarse objetos en ficheros o incluso en bases de datos como BLOBs (Binary Large OBject), o se pueden enviar a través de sockets, en aplicaciones cliente/servidor de un equipo a otro, por ejemplo.

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción **`implements Serializable`** (además de importar la clase *java.io.Serializable*) en la cabecera de clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que el objeto sea serializable.

La clase **`ObjectInputStream`** y la clase **`ObjectOutputStream`** se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de **`InputStream`** y **`OutputStream`**, de hecho son casi iguales a **`DataInput`**/**`OutputStream`** sólo que incorporan los métodos *`readObject`* y *`writeObject`* que son muy poderosos. Ej.:

```
import java.io.Serializable;

public class Persona implements Serializable {
      // Esta clase debe ser Serializable para
      // poder ser escrita en un stream de objetos
      private String nombre;
      private int edad;

      public Persona(String s, int i) {
            this.nombre = s;
            this.edad = i;
      }

      public String toString() {
            return nombre + ":" + edad;
      }
}
```

```
import java.io.*;
import java.util.Date;

public class PruebaSerializable {
    public static void main(String args[]) {
            try {
                  // Un OutputStream sobre el que escribir los bytes
                  FileOutputStream f = new FileOutputStream("prueba.dat");
                  // El objeto serializador
                  ObjectOutputStream ost = new ObjectOutputStream(f);
                  Persona persona1 = new Persona("Matias", 35);
                  Date fecha = new Date();
                  ost.writeObject(persona1);
                  ost.writeObject(fecha);
                  ost.writeInt(48);
                  System.out.println("Se serializo el objeto en el archivo");
                  ost.flush(); // vaciar el buffer
                  ost.close();
            } catch (IOException e) {
                  System.err.println(e);
            }

            try {
```

```
                // Un InputStream subyacente del cual leer los bytes
                FileInputStream f = new FileInputStream("prueba.dat");
                // El objeto deserializador
                ObjectInputStream ist = new ObjectInputStream(f);
                Persona personaN = (Persona) ist.readObject();
                Date fecha = (Date) ist.readObject();
                int entero = ist.readInt();
                System.out.println(personaN);
                System.out.println(fecha);
                System.out.println(entero);
                System.out.println("Se deserializo el objeto");
                ist.close();
            } catch (IOException e) {
                System.err.println(e);
            } catch (ClassNotFoundException e) {
                System.err.println(e);
            }

    }
}
```

3.8.5 CUADRO DE DIÁLOGO JFILECHOOSER

Para seleccionar el archivo que se quiere abrir de forma visual, se puede utilizar la clase JFileChooser que contiene el método showOpenDialog() que muestra un cuadro de diálogo, para que el usuario pueda seleccionar archivos o directorios y el método getSelectedFile() para obtener el archivo/directorio seleccionado por el usuario.

```
import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

public class PruebaArchivosGUI {

    public static void main(String[] args) {

        JFileChooser v = new JFileChooser();

        v.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);

        int result = v.showOpenDialog(null);
        if (result == JFileChooser.CANCEL_OPTION)
            System.exit(1);

        File nom = v.getSelectedFile();

        if (nom == null || nom.getName().equals(""))

            JOptionPane.showMessageDialog(null, "nombre de archivo inválido",
"Error", JOptionPane.ERROR_MESSAGE);
        else
            JOptionPane.showMessageDialog(null,
                    (nom.isDirectory() ? "El Directorio " : "el archivo")
+ " seleccionado es: " + nom.getName());
    }
}
```

BIBLIOGRAFÍA

- Berzal Galiano, Fernando, "Apuntes de programación orientada a objetos en Java: Fundamentos de programación y principios de diseño" (2006)
- Ceballos, Fco. Javier, “JAVA 2 Curso de programación” 4ta Ed. (Ra-Ma 2010)
- Deitel, Paul y Deitel, Harvey, “JAVA Cómo programar” 9na Ed. (Pearson 2012)
- Eckel, Bruce, “Piensa en JAVA” 4ta Ed. (Prentice-Hall 2007)
- Kuhn, Mónica, “Apuntes de Programación II” INSPT/UTN (2014)
- Otero, Abraham, “Tutorial básico de JAVA” 3ra Ed. (javahispano.org 2007)
- Pérez, Gustavo Guillermo, “Aprendiendo JAVA y Programación Orientada a Objetos” (2008)
- Sánchez, Jorge, “JAVA 2” (2004)

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar