EXCEPCIONES

Si una operación no puede completarse debido a un error el programa deberá:

- Volver a un estado estable y permitir otras operaciones.
- Intentar guardar el trabajo y finalizar.

Esto es difícil debido a que generalmente el código que detecta el error no es el que puede realizar dichas tareas. El que detecta el error debe informar al que pueda manejarlo. La solución más habitual son los códigos de error. Inconvenientes:

- No siempre queda sitio para códigos de error (getchar).
- No se garantiza que se vayan a consultar.
- Si se contemplan todos los errores el código crece considerablemente.
- La lógica del programa queda oscurecida.
- No sirven en los constructores.

En definitiva hacen que el tratamiento de errores sea tedioso y se relaje su uso. Esto impide la construcción de programas robustos. Java ofrece otra forma de tratar con los errores: **las excepciones**.

Detección de Errores

- Una condición excepcional es aquella que impide la continuación de una operación.
- No se sabe como manejarla, pero no se puede continuar.
- En Java se lanza una excepción (throw) para que alguien que sepa manejarla la trate en un contexto superior.

```
if (error de CRC)
   throw new IOExcepcion();
```

- La palabra **throw** finaliza el método y lanza un objeto que facilite información sobre el error ocurrido.
- Normalmente se utilizará una clase para cada tipo de error.
- Java obliga a que un método informe de las excepciones (explícitas) que pueda lanzar.
- Un método no solo tiene que decir que devuelve si todo va bien; debe indicar también que puede fallar.
- La especificación de excepciones se realiza mediante la palabra **throws** en la declaración del método:

```
void f() throws EOFException, FileNotFoundException
{
   if ( . . . )
      throw new EOFException();
   if ( . . . )
      throw new FileNotFoundException();
}
```

Manejo de la Excepción

- Una vez detectado el error hace falta indicar quien se encarga de tratarlo.
- Un manejador de excepciones es de la forma:

```
try {
    // Operaciones con posibles excepciones
}
catch (<tipo de Excepción> ref) {
// tratamiento de la excepción
}
```

- El **try** delimita el grupo de operaciones que puede producir excepciones.
- El bloque **catch** es el lugar al que se transfiere el control si alguna de las operaciones produce una excepción.
- Ejemplo

```
try {
   a.abreFichero();
```

```
            a.leeCabecera();
            a.actualizaDatos();
         }
         catch (IOException ref) {
            System.out.println("Error de E/S");
         }
```

- Si alguna de las operaciones del bloque produce una excepción se interrumpe el bloque try y se ejecuta el catch.
- Al finalizar este se continúa normalmente.
- Si no se produce ninguna excepción el bloque catch se ignora.
- Un bloque try puede tener varios catch asociados.

```
         try {
            a.abreFichero();
            a.leeCabecera();
            a.actualizaDatos();
         }
         catch (FileNotFoundException ref) {
            System.out.println("Error de apertura");
         }
         catch (IOException ref) {
            System.out.println("Error de E/S");
         }
```

- ¿Qué ocurre si se produce una excepción que no se corresponde con ningún catch?
- La excepción se propaga hacia atrás en la secuencia de invocaciones hasta encontrar un catch adecuado.
- Ejemplo:

```
void f1(int accion) throws EOFException
{
   try {
         if (accion==1) throw new FileNotFoundException();
         else if (accion==2) throw new EOFException();
   }
   catch (FileNotFoundException e) {
         System.out.println("Error corregido");
   }
   System.out.println("Finalización normal f1");
}

void f2 (int valor)
{
   try {
         f1(valor);
   }
   catch (EOFException e) {
         System.out.println("Error corregido");
   }
   System.out.println("Finalización normal f2");
}
```

- Cuando se invoca a un método que lanza excepciones es obligatorio:
  - Que se maneje el error (catch)
  - Que se indique mediante throws su propagación.
- Es decir, o se trata el error o se avisa de que se propaga.

Ejemplo:

```
void f1() throws IOException { . . . }
```

```
void f2 () {
   f1(); // Incorrecto
}

void f2 () {
   try { // Alternativa 1
     f1();
   }
   catch (IOException e) {
      // Tratamiento
   }
}

void f2() throws IOException {
   f1(); // Alternativa 2
}
```

Finally

Puede haber ocasiones en que se desea realizar alguna operación tanto si se producen excepciones como sino. Dichas operaciones se pueden situar dentro de un bloque finally.

```
try {
   // Operaciones con posibles excepciones
}
catch (<tipo de Excepción> ref) {
   // Tratamiento de la excepción
}
finally {
   // Operaciones comunes
}

class Recurso {
   void reserva() { . . . }
   void libera() { . . . }
}

class Ejemplo {
      void prueba() {
      Recurso recurso = new Recurso();
      try {
         Recurso.reserva = new Recurso();
         // Operaciones con posibles excepciones
         recurso.libera();
      }
      catch (ExceptionA a) {
            // Tratamiento del error
            recurso.libera();
      }
      catch (ExceptionB b) {
            // Tratamiento del error
            recurso.libera();
      }
}
```

- Solución con finally.

```
class Ejemplo {
      void prueba() {
```

```
            Recurso recurso = new Recurso();
            try {
               Recurso.reserva();
               // Operaciones con posibles excepciones
            }
            catch (ExceptionA a) {
               // Tratamiento del error
            }
            catch (ExceptionB b) {
               // Tratamiento del error
            }
            catch (ExceptionC c) {
               // Tratamiento del error
            }
            finally {
              recurso.libera();
            }
         }
}
```

- Si no hay ninguna excepción se ejecutará el bloque try y el finally.
- Si se produce alguna excepción
  - Si es atrapada por un catch del mismo try se ejecuta este y luego el finally. La ejecución continúa después del finally normalmente.
  - Si no es atrapada se ejecuta el finally y la excepción se propaga al contexto anterior.

Jerarquía de Excepciones

- Toda excepción debe ser una instancia de una clase derivada de **Throwable.**
- De ella hereda los siguientes métodos:
  - getMessage
  - toString
  - printStackTrace
  - fillInStackTrace
- Hay situaciones en las que interesa relanzar una excepción ya atrapada.

```
catch (Exception e) {
   // Salvar datos pero propagar
   throw e;
}
```

- La excepción pasaría a buscarse entre los catch de un contexto anterior (los del mismo nivel se ignoran).

Resumen

- Antes

```
class Fichero {
   int open(String name) { } // -1 si error
   int leer() { } // -1 si EOF, -2 si error
}

class Ejemplo {
   public static void main(String args [ ] ) {
      Fichero f = new Fichero();
      If (f.open("z.txt") == -1)
         Error()
      Else {
         Int dato f.leer();
```

```
            While (dato != -1 && dato != -2) {
               System.out.println(dato);
               Dato = f.leer();
            }
            if (dato == -2)
               error();
            }
         }
      }
```

- En la práctica:

```
class Ejemplo {
   Public static void main(String args [] ) {
      int dato;
      Fichero f = new Fichero();

      f.open("z.txt");
      while (dato = f.leer() != -1)
         System.out.println(dato);
   }
}
```

- Con excepciones

```
class Fichero {
   Fichero(String name) throws FileNotFoundException
   { }
   int leer() throws IOException { }
   . . .
}
```

- Con excepciones

```
class Ejemplo {
   public static void main(String args[]) {
      int dato;
      Try {
            Fichero f=new Fichero("z.txt");
            while (dato = f.leer() != -1)
                  System.out.println(dato);
      } catch (Exception e) {
           System.out.println("Error de E/S");
      }
   }
}
```

Jerarquías de Excepciones

- La base de la jerarquía de excepciones en Java es
  - Throwable
    - Error
    - Exception
      RuntimeException

- Las clases derivadas de Error describen errores internos de la JVM.
- No deben ser lanzados por las clases de usuario.
- Estas excepciones rara vez ocurren y cuando así sea lo único que se puede hacer es intentar cerrar el programa sin perder datos.

- Ejemplo: OutOfMemoryError, StackOverflow, etc.
- Los programas en Java trabajarán con las excepciones de la rama Exception.
- Este grupo se divide a su vez en:
  - Las clases que derivan directamente de Exception (explícitas).
  - Las que derivan de RuntimeException (implícitas).
- Se utilizan las RuntimeException para indicar un error de programación.
- Si se produce una excepción de este tipo hay que arreglar el código.
- Ejemplos:
  - Un cast incorrecto.
  - Acceso a un array fuera de rango.
  - Uso de un puntero null.
- El resto de las Exception indican que ha ocurrido algún error debido a alguna causa ajena al programa (está correcto).
- Ejemplos:
  - Un error de E/S.
  - Error de conexión.
- Los métodos deben declarar sólo las excepciones explícitas.
- Las implícitas no deben declararse (aunque pueden producirse igualmente).
  void f() throws IOException { . . . }
- Por tanto cuando un método declara una excepción avisa de que puede producirse dicho error además de cualquier error implícito.

Creación de Excepciones

- Si se necesita notificar de algún error no contemplado en Java se puede crear una nueva clase de Excepción.
- La única condición es que la clase derive de Throwable o derivado.
- En la práctica generalmente derivará_
  - de RuntimeException si se desea notificar un error de programación.
  - de Exception en cualquier otro caso.
- En un método que tienen como parámetro un entero entre 1 y 12 se recibe un 14 ¿Qué tipo de excepción se crearía para notificarlo?
- Si un método para listar Clientes por impresora se encuentra con que deja de responder ¿Qué tipo de excepción debería crear?

```
class PrinterException extends Exception { }

class Ejemplo {
   void printClients(Client[ ] clients) throws PrinterException
   {
        for (int i = 0; i<clients.length; i++) {
           // imprimir Cliente[i]
           if (error impresión)
              throw new PrinterException();
      }
   }
}


class InvalidMonthException extends RuntimeException
{}

class Ejemplo {
   void setMonth(int month) {
```

```
            if (month < 1 || month > 12)
                  throw new InvalidMonthException();
      }
   }
```

- Una excepción, como cualquier otra clase, puede tener operaciones que permitan obtener mas información sobre el error.

```
class PrinterException extends Exception
{
    PrinterException(int printed) {
          this.printed = printed;
    }

    int getPrintedCount() { return printed; }

    private int printed;
}
```

Ejemplo:

```
class Ejemplo
{
    void printClients(Client [ ] clients) throws PrinterException
    {
       for (int pagina=0; pagina<clients.length; i++)
       {
        // Imprimir cliente [pagina]
         if (error impresión)
           throw new PrinterException(pagina);
       }
     }
    }
    void Informe()
    {
       try {   printClients(clients);  }
       catch (PrinterException e)
       {
    System.out.println("Solo se han imprimido"+ e.getPrinterCount());
       }
    }
}

class InvalidMonthException extends RuntimeException
{
      InvalidMonthException(int month);
      {
            this.month = month;
      }
      getInvalidMonth()
      {
            return month;
      }
      public String toString()
      {
            return "Mes invalido: "+month;
      }
      private int month;
```

```
}

class Fecha
{
      void setMonth(int month)
      {
            if (month < 1 || month>12)
                  throws new InvalidMonthException(month);
      }
}
```

Ejemplo:

```
class Ejemplo
      {
      public static void main (String[ ] args)
      {
            Fecha fecha= new Fecha()
            fecha.setMonth(14);
      }
}
```

- Java obliga a atrapar las excepciones explícitas.¿Qué pasa si no se atrapa una implícita?.

Resumen

- Excepciones explícitas:
  - Derivan de exception (no de RuntimeException).
  - Indican un error externo a la aplicación.
  - Si se lanzan es obligación declararlas.
  - Si se invoca un método que las lanza es obligatorio atraparlas o declararlas.

- Excepciones implícitas:
  - Derivan de RunTimeException.
  - Indican un error de programación.
  - No se declaran: se corrigen!!.
  - Se pueden atrapar. En caso contrario finaliza la aplicación.

Excepciones y Herencia

- Al redefinir un método se está dando otra implementación para un mismo mensaje.
- El nuevo método solo podrá lanzar excepciones declaradas en el método original.

```
class A {
      void f() throws EOFException {  }
}

class B extends A
{
      // Incorrecto
      void f() throws EOFException, FileNotFoundException {  }
}
```

Un método puede declarar excepciones que no lance realmente.

- Así un método base puede permitir que las redefiniciones puedan producir excepciones.
- Los constructores, al no heredarse, pueden lanzar nuevas excepciones.

Normas en el uso de excepciones

Uso de Excepciones

- **Norma 1.**
  - Si una excepción se puede manejar no debe programarse.
  - Es más cómodo usar métodos que no produzcan errores.

- **Norma 2.**
  - No utilizar las excepciones para evitar una consulta. No abusar de ellas.

```
try {   stack.pop();   }
catch  (EmptyStackExecution e) {
   .....
}
while (!stack.empty())
       stack.pop();
```

- **Norma 3.**

- Separar el tratamiento de errores de la lógica.

```
for ( int i=0; i< len; i++) {
   try
   {
      obj1.mesg1();
   }
   catch (ExcA a)
   {
      //tratamiento
   }
   try
   {
      obj2.mesg2();
   }
   catch (ExcB  b)
   {
      //tratamiento
   }
 }


try
  {
     for (int i = 0; i<len; i++)
     {
        obj1.mesg1();
        obj2.mesg2();
     }
  }
  catch (ExcA a) {
     // Tratamiento
  }
  catch (ExcB b) {
     // Tratamiento
  }
}
```

- **Norma 4.**
  - No ignorar una excepción.

```
try
{
     // operaciones;
}
catch {  EmptyStackExecution e)
{   } // Para que calle el compilador
```

PERSISTENCIA

Introducción

El almacenamiento de datos en variables y arreglos es temporal; los datos se pierden cuando una variable local "sale de su alcance" o cuando el programa termina, por ejemplo. Se usan archivos para conservar a largo plazo grandes cantidades de datos, incluso después de que el programa que creó los datos ha terminado. Los datos mantenidos en archivos se conocen como persistentes. Las computadoras guardan los archivos en dispositivos de almacenamiento secundario como discos magnéticos, discos ópticos y cintas magnéticas. En este capítulo explicaremos como se crean, actualizan y procesan archivos mediante Java. Veremos los archivos tanto de "acceso secuencial" como de "acceso aleatorio" e indicaremos los tipos de aplicaciones para las que es apropiado cada tipo.

El procesamiento de archivos es una de las capacidades más importantes que un lenguaje debe tener para apoyar aplicaciones comerciales que suelen procesar enormes cantidades de datos persistentes.

La Jerarquía de datos

En última instancia, todos los datos procesados por una computadora se reducen a combinaciones de ceros y unos. Esto es así porque resulta sencillo y económico construir dispositivos electrónicos capaces de asumir dos estados estables; un estado representa 0 y el otro 1. Resulta asombroso que las impresionantes funciones desempeñadas por las computadoras sólo impliquen manipulaciones más fundamentales de unos y ceros.

El elemento de información más pequeño que una computadora puede asumir es el valor 0 o el valor 1. Tal elemento se denomina bit (abreviatura de "binary digit", dígito binario; es decir, un dígito que sólo puede asumir dos valores). Los circuitos de las computadoras realizan varias manipulaciones sencillas de bits, como examinar el valor de un bit, establecer el valor de un bit e invertir un bit (de 0 a 1 y de 1 a 0).

Para los programadores no es fácil trabajar con los datos en forma de más bajo nivel (como bits). En vez de ello, los programadores prefieren trabajar con los datos en forma de dígitos decimales (es decir, 0,1,2,3,4,5,6,7,8 y 9), letras (es decir, A a Z y a a z) y símbolos especiales (como, $, @, %, &, *, (,), +, -, :, /, ¿, ?, y muchos más). Los dígitos, letras y símbolos especiales se denominan caracteres. La colección de todos los caracteres empleados para escribir programas y representar elementos de información en una computadora en particular es el conjunto de caracteres de esa computadora.

Puesto que las computadoras sólo pueden procesar ceros y unos, cada carácter del conjunto de caracteres se representa como un patrón de ceros y unos(los caracteres en Java son caracteres Unicode formados por 2 bytes). Los bytes por lo regular se componen de ocho bits. Los programadores crean programas y datos con caracteres; las computadoras manipulan y procesan esos caracteres como patrones de bits.

Así como los caracteres se componen de bits, los campos se componen de caracteres. Un campo es un grupo de caracteres que comunica un significado. Por ejemplo, un campo formado exclusivamente por letras mayúsculas y minúsculas puede servir para representar el nombre de una persona.

Los elementos de información procesados por las computadoras forman una jerarquía de datos en el que los elementos de información se hacen más grandes y con una estructura más compleja conforme avanzamos de bits a caracteres (bytes), a campos, etc.

Un registro (una clase en Java) se compone de varios campos (atributos en Java). En un sistema de nómina, por ejemplo, un registro para un empleado específico podría consistir en los siguientes campos:

1. Número de identificación del empleado
2. Nombre
3. Dirección
4. Sueldo por hora, etc.

Así, un registro es un grupo de campos relacionados entre sí. En el ejemplo anterior, todos los campos pertenecen al mismo empleado. Desde luego, una compañía puede tener muchos empleado, y tendrá un registro de nomina para cada empleado. Un archivo es un grupo de registros relacionados entre sí. El archivo de nómina de una compañía normalmente contiene un registro para cada empleado. Por ejemplo, el archivo de nómina de una

compañía pequeña podría contener sólo 22 registros, en tanto que un archivo de nómina para una compañía grande podría contener 100000 registros.

No es inusual que una compañía tenga muchos archivos, cada uno con millones de caracteres de información.

Para facilitar la recuperación de registros específicos de un archivo, se escoge por lo menos un campo de cada registro. Una clave de registro identifica un registro como perteneciente a una persona o a una entidad en particular y distinto de todos los demás registros del archivo. En el registro de nómina antes descrito, el número de identificación del empleado normalmente se escogería como clave de registro.

Hay muchas formas de organizar los registros de un archivo. La forma de organización más común se denomina archivo secuencial, y en él los registros por lo regular se almacenan en orden según el campo clave del registro. En un archivo de nómina, los registros normalmente se colocan en orden según el número de identificación del empleado. El primer registro de empleado del archivo contiene el número de identificación de empleado más bajo, y los registros subsecuentes contienen números de identificación de empleado cada vez más altos.

La mayor parte de las empresas emplean muchos archivos distintos para almacenar datos. Por ejemplo, las compañías podrían tener archivos de nómina, de cuentas por cobrar (listando el dinero que los clientes deben a la compañía), de cuentas por pagar (listando el dinero que la compañía debe a sus proveedores), de inventario (listando información acerca de todos los artículos que la empresa maneja) y muchos otros tipos de archivos. A veces, un grupo de archivos relacionados entre sí se denomina base de datos. Una colección de programas diseñados para crear y manejar bases de datos en un sistema de administración de bases de datos (DBMS, database management system).

Archivos y Flujos (stream)

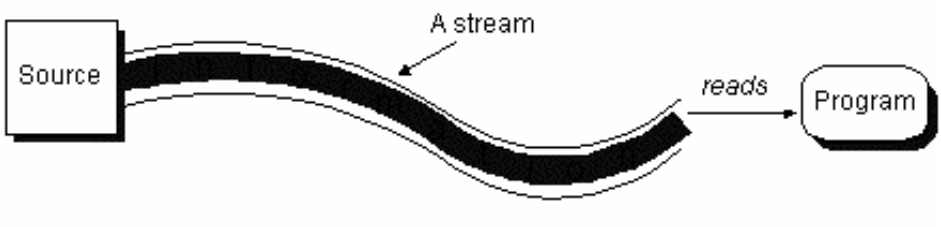
¿Qué es un stream?
En ocasiones los programas necesitan obtener o enviar información a una fuente externa. Esta fuente externa puede encontrarse en...

+ un archivo + un disco + en la red
+ un puerto + en memoria + en otro programa

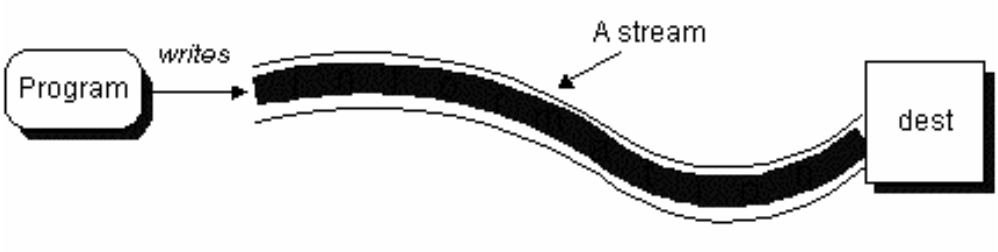
La información que el programa envía o recibe puede ser...

+ objetos + caracteres + imágenes + sonidos + etc...

Para obtener la información, el programa abre un conducto (stream) de entrada a la fuente de información (archivo, memoria, red...).



Para enviar información, el programa abre un conducto (stream) de salida al destino externo de información (archivo, red, memoria...)



Stream: Conducto por el cual fluye información de manera secuencial

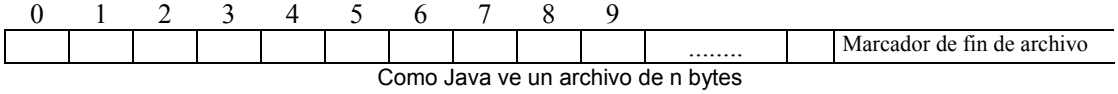
entrada: fuente externa -> programa
salida: programa -> destino externo

Algoritmos

Para leer	Para escribir
open a stream	open a stream
while more information	while more information
read information	write information
close the stream	close the stream

La librería java.io.* contiene una colección de clases stream que permiten implementar las operaciones de lectura y escritura.

Java considera los archivos simplemente como flujos secuenciales de bytes. Cada archivo termina con un marcador de fin de archivo o bien en un número de byte específico registrado en una estructura de datos administrativa mantenida por el sistema. Cuando se abre un archivo, se crea un objeto y se asocia un flujo (streams) a dicho objeto. Cuando comenzamos a ejecutar una aplicación o una applet de Java, se crean automáticamente tres objetos de flujo: System.in, System.out y System.err. Los flujos asociados a estos objetos proporcionan canales de comunicación entre un programa y un archivo o dispositivo en particular. Por ejemplo, el objeto System.in (objeto de flujo de entrada estándar) permite a un programa introducir bytes desde el teclado, el objeto System.out (objeto de salida estándar) permite a un programa enviar datos a la pantalla, y el objeto System.err (objeto de flujo de error estándar) permite a un programa enviar mensajes de error a la pantalla.



Como Java ve un archivo de n bytes

Si queremos procesar archivos en Java necesitamos importar el paquete **java.io**. Este paquete incluye las definiciones de las clases de flujos como FileInputStream (para el flujo de entrada de un archivo) y FileOutputStream(para el flujo de salida de un archivo). Los archivos se abren creando objetos de estas clases de flujos que se derivan de (es decir, heredan la funcionalidad de) las clases InputStream (flujo de entrada) y OutputStream(flujo de salida), respectivamente. Así, todos los métodos de éstas clases de flujos se pueden aplicar también a los flujos de archivos. Para realizar operaciones de entrada y salida de tipos de datos primitivos, se usarán objetos de las clases DataInputStream (flujo de entrada de datos) y DataOutputStream (flujo de salida de datos) junto con las clases de flujos de los archivos. Las relaciones de herencia de las clases E/S de Java se resumen a continuación:

Jerarquía de clases del paquete java.io

Object
- File
- FileDescriptor
- StreamTokenizer
- InputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - PipedInputStream
  - FileInputStream
  - FilterInputStream
    - DataInputStream
    - BufferedInputStream
    - PushBackInputStream
    - LineNumberInputStream
- OutputStream
  - ByteArrayOutputStream
  - PipedOutputStream
  - FileOutputStream
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PrintStream
- RandomAccessFile

Java ofrece muchas de las clases para realizar operaciones de entrada/salida. En esta sección haremos una breve reseña de cada una y explicaremos cómo se relacionan entre sí. En el resto del capítulo pondremos a trabajar varias clases de flujos importantes al implementar diversos programas de procesamiento de archivos que crean, manipulan y destruyen archivos de acceso secuencial y archivos de acceso aleatorio.

InputStream (una subclase de Object) y OutputStream (una subclase de Object) son clases abstractas que definen métodos para realizar operaciones de entrada y salida, respectivamente: sus clases derivadas supeditan estos métodos.

La entrada/salida de archivos se efectúa con FileInputStream (una subclase de InputStream) y FileOutputStream (una subclase de OutputStream).

Usamos un PrintStream (flujo de impresión, una subclase de FilterOutputStream) para enviar salidas a la pantalla (o en la "salida estandar" definida por su sistema operativo local). De hecho, hemos estado usando salidas PrintStream en todo el texto; System.out es un PrintStream (lo mismo que System.err).

Un FilterInputStream filtra un InputStream y un FilterOutputStream; el término filtrar simplemente significa que el flujo tiene una funcionalidad adicional, como almacenar datos temporalmente, llevar la cuenta de los números de línea o combinar los bytes de datos en unidades significativas de tipos de datos primitivos.Leer los datos como bytes en bruto es rápido pero burdo. Por lo regular los programas desean leer los datos como agregados de bytes que forman un int, un float, un double, etc. Para lograr usamos un DataInputStream (una subclase de FilterInputStream).

Las clases DataInputStream y la clase RandomAccessFile (archivo de acceso aleatorio, de la que hablaremos un poco más adelante) implementan la interfaz DataInput (entrada de datos) pues la necesitan para leer datos primitivos de un flujo. Los DataInputStream permiten a un programa leer los datos binarios de un InputStream.

Por lo regular, encadenaremos un DataInputStream a un FileInputStream. La interfaz DataInput incluye los métodos read (leer, para arreglos de bytes), readBoolean, readByte, readChar, readDouble, readFloat, readFully (ller plenamente, para arreglos de bytes), readInt, readLine (ller línea), readLong, readShort, readUnsignedByte (leer byte sin signo), readUnsignedShort (leer entero corto sin signo), readUTF (para cadenas en formato Unicode) y skipBytes (saltar bytes).

Las clases DataOutputStream (una subclase de FilterOutputStream) y la clase RandomAccessFile implementan la interfaz DataOutput, pues la necesitan para escribir tipos de datos primitivos en un OutputStream. Los DataOutputStream permiten a un programa escribir datos binarios en un OutputStream. Por lo regular, encadenaremos un DataOutputStream a un FileOutputStream. La interfaz DataOutput incluye método flush (vaciar), size(tamaño), write (escribir, para un byte), write (para un arreglo de bytes), writeBoolean, writeByte, writeBytes, writeChar, writeChars (para objetos String de Unicode), writeDouble, writeFloat, writeInt, writeLong, writeShort y writeUTF.

Un RandomAccessFile (archivo de acceso aleatorio, una subclase de Object) es útil para aplicaciones de acceso directo, como las aplicaciones de procesamiento de transacciones de las cuales los sistemas de reservaciones de las líneas aéreas y los sistemas de punto de venta son ejemplos. Con un archivo de acceso secuencial, cada solicitud de entrada/salida sucesiva se lee o escribe el siguiente conjunto de datos consecutivo del archivo. Con un archivo de acceso aleatorio, cada solicitud de entrada/salida sucesiva puede dirigirse a cualquier parte del archivo, tal vez muy distante de la parte del archivo a la que se hizo referencia en la solicitud anterior. Las aplicaciones de acceso directo ofrecen acceso rápido a elementos de información específicos de archivos grandes; tales aplicaciones se utilizan con frecuencia cuando hay personas que están esperando una respuesta. Dichas respuestas deben obtenerse rápidamente, pues de lo contrario las personas pueden impacientarse y "llevar a cabo su negocio en otro lado".

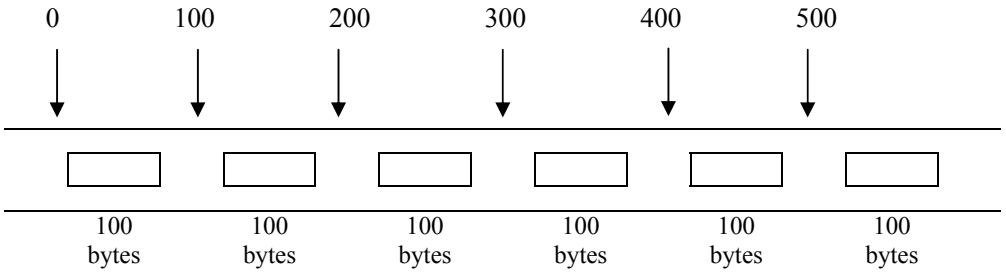
Archivos de acceso aleatorio

Hasta ahora, hemos visto cómo crear archivos de acceso secuencial y cómo leerlos para encontrar información específica. Los archivos de acceso secuencial no son apropiados para las llamadas aplicaciones de "acceso instantáneo" en las que es preciso localizar de inmediato un registro de información en particular. Como ejemplos de aplicaciones de acceso instantáneo populares podemos citar los sistemas de reservaciones de líneas aéreas, los sistemas bancarios, los sistemas de puntos de venta, los cajeros automáticos y otros tipos de sistemas de procesamiento de transacciones que requieren acceso rápido a datos específicos. El banco en el tenemos una cuenta de ahorro o corriente podría tener cientos de miles o incluso millones de otros clientes, pero cuando usamos un cajero automático sólo se necesitan unos segundos para examinar nuestra cuenta y determinar si tenemos suficientes fondos. Este tipo de acceso instantáneo es posible con archivos de acceso aleatorio. Podemos acceder de forma directa (y rápida) a los registros de un archivo de acceso aleatorio sin tener que examinar otros registros.

Como ya dijimos, Java no impone estructura alguna a los archivos, *así que la aplicación que desee usar archivos de acceso aleatorio deberá crearlos*. Podemos usar diversas técnicas para crear archivos de acceso aleatorio. Tal vez la más sencilla sea exigir que **todos los registros del archivo tengan la misma longitud fija.**

El empleo de registros de longitud fija permite a un programa calcular con facilidad (en función del tamaño de registro y la clave de registro) la posición exacta de cualquier registro relativa al principio del archivo. Pronto veremos cómo esto facilita el acceso inmediato a registros específicos, incluso en archivos grandes.

En la siguiente figura se ilustra cómo Java ve un archivo de acceso aleatorio compuesto por registros de tamaño fijo (cada registro tiene 100 bytes de longitud). Un archivo de acceso aleatorio es como un ferrocarril con muchos vagones, algunos vacíos y otros con contenido.



Supongamos que nuestro problema consiste en realizar un programa que permita almacenar los siguientes datos de un cliente: Número, Nombre y Saldo. Por ejemplo:

```
Número: 73
Nombre: Oscar Chávez
Saldo: 450.45

Número: 38
Nombre: Agustín Lara
Saldo: 984.26
```

Antes de comenzar analizar como estos datos han de conformar un "registro".
Cuando se escriben los datos sean del tipo que sean en un archivo aleatorio estos deben ser convertidos a su contraparte en bytes, lo que implica que si se abre algún archivo aleatorio con el Bloc de Notas por ejemplo, no se verán los datos como se guardaron sino que se verán unos símbolos especiales.

Registro 1	Registro 2	Registro 3
Número	Número	Número
Nombre	Nombre	Nombre
Saldo	Saldo	Saldo

Y así sucesivamente hasta llegar al registro n. Aunque por motivos didácticos lo representamos de la siguiente manera:

Registro 1
Número
Nombre
Saldo
Registro 2
Número
Nombre
Saldo
Registro 3...
Número
Nombre
Saldo

Y así sucesivamente. Recuerden que pretendemos visualizar un archivo de acceso aleatorio como un grupo de registros, cada registro tiene 3 campos: un entero, una cadena y double.

A continuación debemos saber el tamaño exacto del registro. En este caso es de **27 bytes**.
int = **4 bytes** (para el numero de cuenta)
string = **15 bytes** (para el nombre)
double = **8 bytes** (para el saldo)
Estos 27 bytes determinan el comienzo y fin de cada registro en el archivo a:

byte 0-26	registro 1
byte 27-53	registro 2
byte 54-71	registro 3
- -	- -
Byte (27 * n-1)-()	Registro n

Esto significa, que si nos desplazamos directamente al byte 27 podremos acceder directamente al 2do. registro del archivo. Y a partir de ahí podremos leer los datos de ese cliente, numero, nombre y saldo. Pero si hemos realizado mal el cálculo y nos movemos al byte 28 o 29, el dato que leeremos sera basura.

Otro aspecto fundamental es entonces que, los primeros 4 bytes de nuestro registro son para el numero, a continuación están los 15 bytes para el nombre y los últimos 8 son para el saldo.

Cuenta	Nombre	Saldo
Byte 0-1-2-3	Byte 4-5-6-7-8-9-10-11-12-13-14-15-16-17-18	Byte 19-20-21-22-23-24-25-26-27

Esta forma en que hemos estructurado el registro, debemos respetarla "siempre", para obtener los datos (leer) en la misma forma en la que los hemos almacenado (escribir).

Los datos del registro se leen y escriben en el archivo SIEMPRE en el mismo orden.

De esta manera, podemos agregar datos en un archivo de acceso aleatorio sin destruir los demás datos del archivo. Los datos almacenados previamente también pueden actualizarse o eliminarse sin tener que reescribir todo el archivo. En las siguientes secciones explicaremos la forma de crear un archivo de acceso aleatorio, introducir datos en él, leer los datos de forma tanto secuencial como aleatoria, actualizar los datos y eliminar datos que ya no se necesitan.

Creación de un archivo de acceso aleatorio

Los objetos RandomAccessFile (archivo de acceso aleatorio) tienen todas las capacidades de los objetos. DataInputStream y DataOutputStream que vimos en las secciones anteriores. Cuando se asocia un flujo RandomAccessFile a un archivo, los datos se leen o escriben a partir del punto en el archivo especificado por el apuntador de posición en el archivo, y todos los datos se leen o escriben como tipos de datos primitivos. Al escribir un valor int, se envían cuatro bytes al archivo. Al leer un valor double, se introducen ocho bytes del archivo. El tamaño de los diversos tipos de datos está garantizado porque Java tiene tamaños fijos para todos los tipos de datos primitivos, sea cual sea la plataforma de computadora.

Tamaño de datos en bits y bytes

Tipo de dato	Tamaño en bits	Tamaño en bytes
boolean	1	1
char	16	2
int	32	4
long	64	8
float	32	4
double	64	8

Nota: Ocho bits corresponden a un solo y único byte, (8bits = 1byte).

A continuación presentaremos las técnicas necesarias para procesar clientes. Proponemos codificar una clase que "gestione" o "maneje" el archivo de clientes. La clase ArchivoClientes permitirá abrir el archivo, agregar, borrar, modificar y buscar clientes como operaciones básicas.

Se pasan dos argumentos al constructor RandomAccessFile: el archivo y el modo de apertura del archivo. El modo de apertura para un RandomAccessFile es **"r"** si se desea abrir el archivo para lectura o bien **"rw"** si se desea abrir el archivo para lectura y escritura. Si ocurre un IOException durante el proceso de apertura, el programa terminará. Si el archivo se abre correctamente,

```
import java.io.*;

class ArchivoClientes {

    RandomAccessFile archivo;

  ArchivoClientes(String name)
```

```
  {
    try
    {
       archivo = new RandomAccessFile(name,"rw");
    }
    catch (IOException e){ System.out.println ("ErrorXX E/S"); }
  }
```

El efecto de construir un objeto RandomAccessFile puede ser diferente, si el archivo no existe físicamente, se crea y el puntero se posiciona en el byte 0, en cambio, si el archivo existe, lo abre y posiciona el puntero en byte 0.

Agregar un registro

El método altaCliente(Cliente o) permite agregar registros de cliente al final del archivo. Para lo cual es necesario mover el puntero al final del archivo. Los métodos de RandomAccessFile que permiten realizar esta operación es: *seek(n)* mueve el archivo al byte n y *length()* que proporciona la cantidad de byte que tiene el archivo (tamaño).
El método write (escribir) envía a la salida un registro de información del objeto RandomAccessFile que se le pasa como argumento. Este método usa el método writeInt para escribir el número de cuenta, el método write para escribir el apellido y el nombre como arreglos de bytes (15 bytes cada uno) y el método writeDouble para escribir el saldo.

```
  void altaCliente(Cliente o) // agrega al final del archivo
   {
    try
        {
       if (archivo.length()>0) // verifica si es el 1er. registro a dar de alta o si hay otros
              archivo.seek(archivo.length());    // corre el puntero al final del archivo
       archivo.writeInt(o.getCuenta());
       archivo.write(paso(o.getNombre()));
       archivo.writeDouble(o.getSaldo());
      }
      catch (IOException e){ System.out.println ("ErrorXX E/S"); }
    }
```

Al escribir cadenas (strings) en nuestro archivo debemos ser cuidadosos, ya que una cadena consta de una dimensión "variable". Si los registros tienen nombre de diferentes tamaños, ya nuestro archivo no consta de registros de tamaño fijo y el manejo que realizaremos del puntero no funcionará correctamente. Por este motivo, antes de escribir la cadena, usamos el método paso:

```
byte[] paso(String s)
  {
    byte [] b = new byte[15];
    if (s != null)
       s.getBytes(0,s.length(),b,0);
    return b;
  }
```

Este método convierte cualquier String en un array de 15 bytes. En caso que las cadenas a guardar sean mas largas, podemos trabajar con arreglos de bytes más grandes.

Buscar un registro

Esta operación implica mover el puntero, registro por registro, comparando un valor determinado, con el valor de alguno de los campos almacenados, hasta hallarlo o hasta que ya no hayan mas datos en el archivo.
En este caso buscamos una determinada cuenta y el método retorna el número de byte donde el registro correspondiente se encuentra. Se posiciona el puntero al comienzo del archivo con el método seek y se utiliza un bucle while para recorrer el archivo. La condición de fin de bucle usa el método *getFilePointer()* que devuelve el numero de byte en el cual esta posicionado en puntero.
La primera vez que entra al bucle devuelve 0. Si el archivo esta vació no entra al bucle porque 23 no es menor/igual que el length() del archivo. Caso contrario, el valor de cuenta se compara con lo que devuelve readInt

(numero de cuenta almacenada). Si es verdadero, en aux se almacena la posición del puntero, que ya no esta al comienzo del registro porque ha avanzado 4 bytes al realizar la lectura, por eso se debe decrementar en 4. Si es falso, se avanza al comienzo del próximo registro, lo cual se hace con el método *skipBytes(n)*, como ya se han leído los primeros 4 bytes del registro, lo que resta para el próximo registro son 23 bytes. Si se avanza 27, el puntero no quedara al comienzo del próximo registro.

```
long buscarCliente(int cuenta)
{  long aux = 0;
   try
      {
       boolean enc = false;
       archivo.seek(0);              // empieza al principio del archivo
       while (archivo.getFilePointer()+23 <= archivo.length() && !enc)
       {
         if (cuenta = = archivo.readInt())   //  leer la cuenta
            { aux = archivo.getFilePointer() - 4; // posiciona el puntero al inicio del reg.
              enc = true;}
            else
              archivo.skipBytes(23);  // avanza el tamano de nombre+saldo
       }

      } catch (IOException e){ System.out.println ("ErrorXX E/S"); }

    return aux ;
 }
```

El método retorna 0, si el numero de cuenta no existe el archivo.
Este mismo algoritmo puede utilizarse para buscar una cuenta pero por nombre, pero deberán modificarse los valores que corresponden al movimiento del puntero.

Consulta de un registro
Para consultar un determinado registro, primero debemos saber donde esta. Lo cual puede hacerse con el buscarCliente, antes de invocar al método buscar o desde este método de consulta.
Se usan los métodos readInt y readDouble para leer el número de cuenta y el saldo, respectivamente. Se emplea el método readFully para leer el apellido y el nombre y colocarlos en arreglos de 15 bytes cada uno. Estos arreglos se utilizan como valores iniciales de los objetos String asignados a los objetos apellido y nombre, respectivamente.

```
void ConsultaCliente(long pos)
   {        // falta testear pos
       try
          { archivo.seek(pos);  // avanza al byte donde se encuentra el registro a consultar
            byte b[]=new byte[15];
             System.out.println("Cuenta: "+archivo.readInt());
            archivo.readFully(b);
            System.out.println("Nombre: "+ new String(b,0));
             System.out.println("Saldo: "+archivo.readDouble());
          }
        catch (IOException e){ System.out.println ("ErrorXX E/S"); }
    }
```

Este método se puede sobrecargar para que devuelva un objeto Cliente, correspondiente al cliente consultado.

Modificar un registro
Para modificar un determinado registro, primero debemos saber donde esta. Lo cual puede hacerse con el método buscarCliente, antes de invocar al método buscar o desde este método.

```
void modificaCliente(long pos, Cliente o)
```

```
{   // falta testear pos
  try
     {
      archivo.seek(pos);  // avanza al byte donde se encuentra el registro a modificar
      archivo.writeInt(o.getCuenta());
       archivo.write(paso(o.getNombre()));
       archivo.writeDouble(o.getSaldo());
     }
      catch (IOException e){ System.out.println ("ErrorXX E/S"); }
}
```

Cerrar el archivo

Antes que finalice el programa debemos cerrar el archivo, para que todas las modificaciones se realicen físicamente.

```
void cerrar ()
 {    try
      {
        archivo.close();
      } catch (IOException e){ System.out.println ("ErrorXX E/S"); }
 }
```

Otros métodos:

Las operaciones para borrar y listar clientes quedan a implementar por el alumno.

El siguiente listado muestra algunos métodos de la clase RandomAccessFile:

void close()
Cierra el archivo de acceso aleatorio y libera todos los recursos del sistema asociados con este archivo
long getFilePointer()
Regresa la posición actual del apuntador (lapicito)
long length()
Regresa la longitud del archivo
boolean readBoolean()
Lee un dato bolean de este archivo
char readChar()
Lee un caracter unicode de este archivo (una letra)
double readDouble()
Lee un double de ese archivo
int readInt()
Lee un entero de este archivo
String readUTF()
Lee una cadena de este archivo
void seek(long pos)
Fija el apuntador en la posición "pos" partiendo del principio del archivo para colocarlo en donde se piensa leer o escribir
void writeBoolean(boolean v)
Escribe un valor bolean
void writeChar(int v)
Escribe un caracter
void writeChars(String s)
Escribe una serie de caracteres
void writeDouble(double v)
Escribe un double
void writeInt(int v)

Escribe un int
void writeUTF(String str)
Escribe una cadena en formato variable

A continuación se escribe la clase Cliente que se ha utilizado en los algoritmos previos.

```
class Cliente
{
  String nombre; // 15 bytes
  int cuenta;    // 4 bytes
  double saldo;  // 8 bytes

 Cliente(String nombre, int cuenta, double saldo)
 {
  this.nombre = nombre;
  this.cuenta = cuenta;
  this.saldo = saldo;
 }

 void setNombre(String nombre)
  { this.nombre =nombre;}
 String getNombre()
  {return nombre;}
 void setCuenta(int cuenta)
  { this.cuenta =cuenta;}
 int getCuenta()
  { return cuenta;}
 void setSaldo(double saldo)
  { this.saldo =saldo;}
 double getSaldo()
  { return saldo;}
}
```

Clases ObjectOutputStream y ObjectInputStream

¿¿¿Podemos guardar directamente objetos en un archivo y posteriormente recuperarlos en forma directa???

Si, DEBEMOS USAR LAS CLASES ObjectOutputStream y ObjectInputStream.