1 Saída

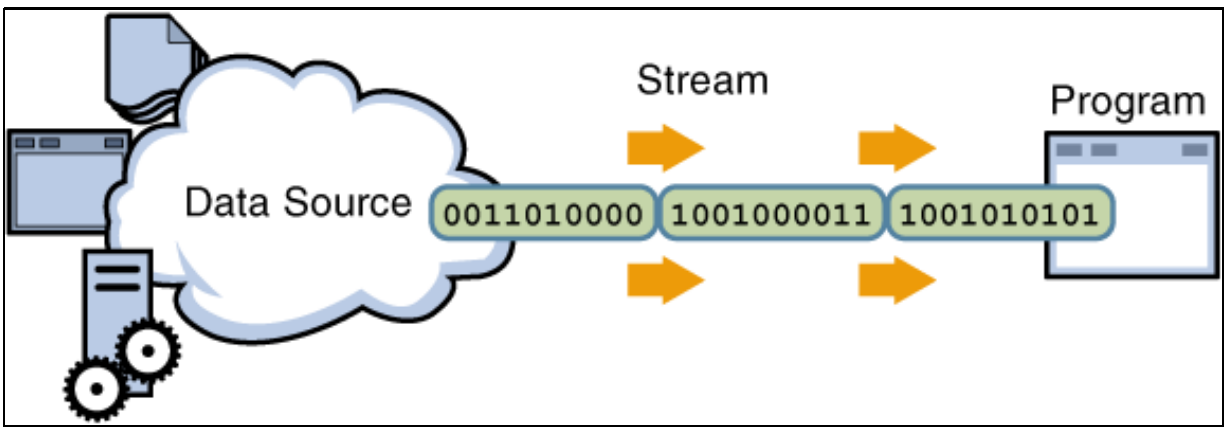
1.1 Sumario

1.2 E/S mediante fluxos

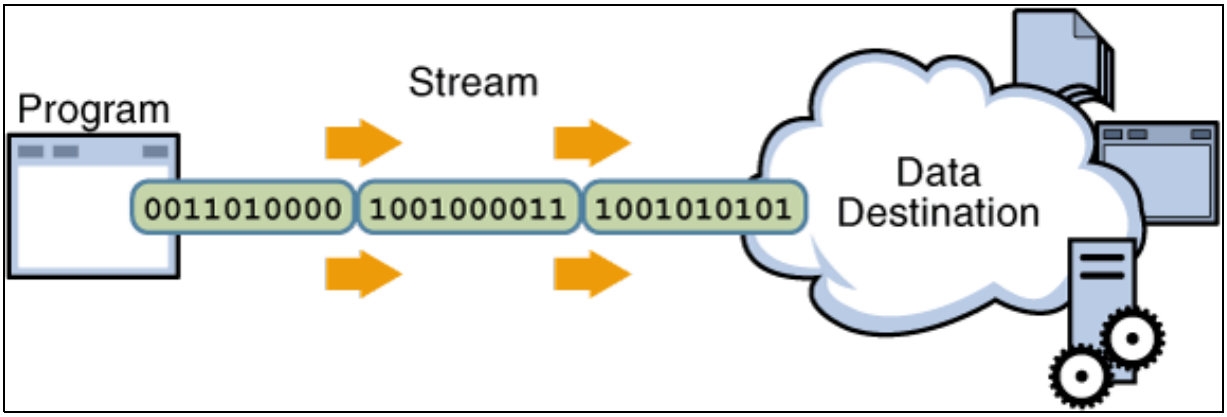
Un fluxo (*Stream* en inglés) E/S representa unha fonte ou un destino de datos. Un fluxo pode proceder de diferentes fontes ou ir a distintos destinos, tales como discos, *devices*, outros programas, vectores de memoria, etc.

Os fluxos soportan diferentes tipos de datos, incluíndo bytes, tipos de datos primitivos, caracteres e obxectos. Algúns fluxos simplemente pasan os datos mentres que outros os manipulan e transforman de distintas maneiras.

Non importa como traballen internamente, todos os fluxos presentan un mesmo modelo aos programas que os utilizan: un fluxo é unha secuencia de datos. Un programa utilizará un **fluxo de entrada** para ler datos dende unha determinada fonte, item por item:



Un programa utiliza un **fluxo de saída** para escribir datos nun determinado destino, item por item:



As fontes e orixes de datos que aparecen nas imaxes anteriores poden ser calquera elemento que almacene, xere ou consuma datos. Obviamente isto incluirá ficheiros; pero unha fonte de datos poderá ser tamén outro programa, un periférico, un socket ou un array.

1.3 Fluxos de bytes

Os fluxos de bytes proporcionan unha maneira eficiente de manexar entradas e saídas de bytes. Este tipo de fluxos son os empregados habitualmente cando se traballa con datos binarios ou con ficheiros.

Todos os fluxos de bytes herdan de dúas clases abstractas chamadas InputStream e OutputStream. A primeira define as características comúns das clases de entrada, mentres que a segunda o fai coas de saída. A partires delas créanse varias subclases ofrecendo unha funcionalidade variable que manexa os detalles de lectura e escritura de diferentes dispositivos.

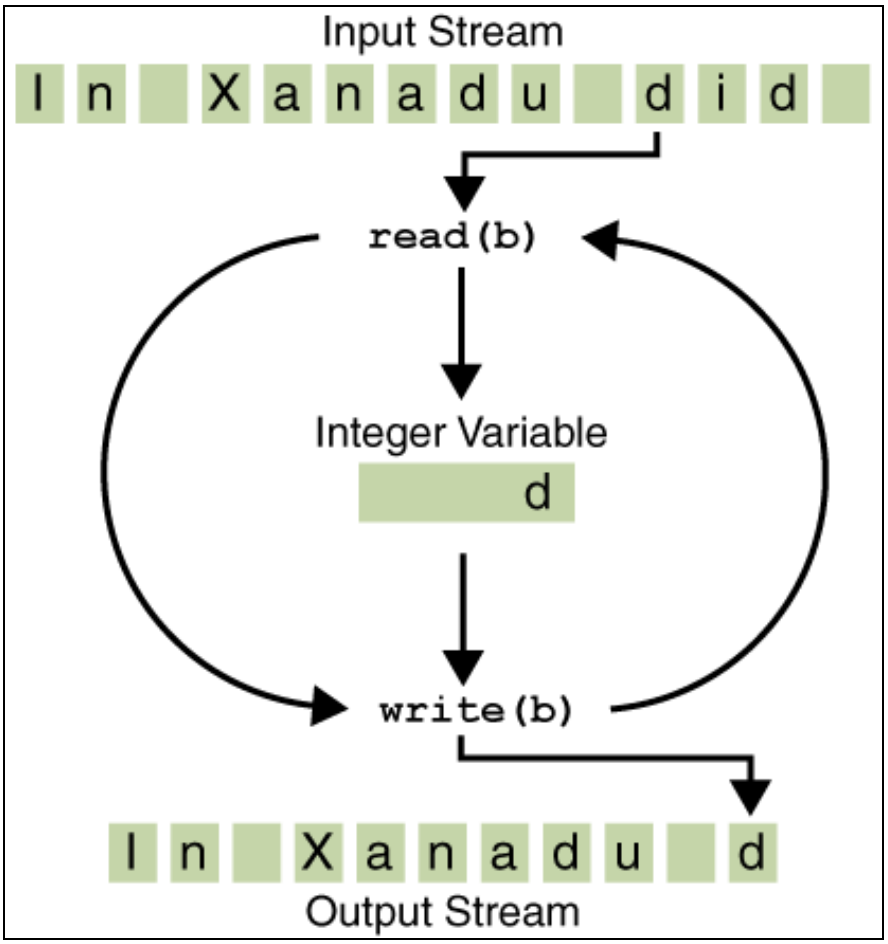
Para demostrar como funciona un fluxo de bytes, examinaremos o caso de lecturas/escrituras en ficheiros, e para iso utilizaremos as subclases FileInputStream e FileOutputStream. Os outros tipos de fluxos de bytes funcionarán basicamente do mesmo xeito, diferenciándose destes na maneira de crealos.

1.3.1 Utilizar fluxos de bytes

Ilustraremos a utilización de FileInputStream e FileOutputStream mediante o programa *CopiarBytes* que copiará o ficheiro *xanadu.txt*, byte por byte:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopiarBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream entrada = null;
        FileOutputStream saida = null;
        try {
            entrada = new FileInputStream("xanadu.txt");
            saida = new FileOutputStream("copiaxanadu.txt");
            int c;
            while ((c = entrada.read()) != -1) {
                saida.write(c);
            }
        } finally {
            if (entrada != null) {
                entrada.close();
            }
            if (saida != null) {
                saida.close();
            }
        }
    }
}
```

Este programa pasará a meirande parte do seu tempo dentro do bucle que lee o fluxo de entrada e escrebe no fluxo de saída, un byte de cada vez, tal e como mostra a seguinte figura:



Observa que *read()* devolve un valor *int*. Se a entrada é un fluxo de bytes, por que *read()* non devolve entón un tipo *byte*?. É sinxelo, un *int* permite o valor -1 para indicar que se alcanzou o final do fluxo.

1.3.2 Pechar un fluxo

É moi importante pechar os fluxos unha vez que xa non son necesarios. Esta é a razón de que o anterior exemplo utilice un bloque finally para garantir que ambos fluxos se pecharán (independentemente de se se produce ou non un erro).

Un posible erro sería que *CopiarBytes* fose incapaz de abrir un ou os dous ficheiros. Se isto ocorrese a variable que referencia o fluxo de entrada nunca cambiaría o seu valor dende *null*. Este é o motivo de preguntar se o fluxo non é nulo antes de pechalo.

1.3.3 Cando non usar fluxos de bytes

CopiarBytes parece un programa bastante común, sen embargo representa un tipo de E/S de baixo nivel que deberíamos evitar sempre. Xa que *xanadu.txt* contén datos de tipo carácter, a mellor aproximación sería utilizar fluxos de carácter. Existen tamén outros tipos de fluxos para utilizar con tipos de datos máis complicados. Os fluxos de bytes de E/S deberían utilizarse só coas operacións máis primitivas.

Todos os outros tipos de fluxos constrúense a partires dos de bytes.

1.4 Fluxos de caracteres

A plataforma Java almacena os caracteres coa codificación Unicode. Un fluxo de caracteres de E/S traduce automaticamente este formato interno a ou dende o xogo de caracteres local. Con un local occidental, o xogo de caracteres é habitualmente un superconxunto de ASCII con 8 bits.

A utilización de fluxos de caracteres é moi similar á utilización de fluxos de bytes por parte dunha aplicación. A entrada e a saída faise mediante clases de fluxo que automaticamente trasladan dente e ata o xogo de caracteres local. Un programa que utilice fluxos de caracteres en lugar de fluxos de bytes adapta automaticamente este ao xogo de caracteres local e está listo para un cambio de local sen ningún esforzo extra pola parte do programador.

1.4.1 Usar fluxos de caracteres

Todos os fluxos de caracteres estenden as clases abstractas Reader e Writer. Como ocorría cos fluxos de bytes, tamén existen clases especializadas no tratamento de ficheiros (FileReader e FileWriter). A continuación ilustramos a súa utilización mediante o exemplo *CopiarCaracteres*.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CopiarCaracteres {
    public static void main(String[] args) throws IOException {
        FileReader fluxoEntrada = null;
        FileWriter fluxoSaida = null;
        try {
            fluxoEntrada = new FileReader("xanadu.txt");
            fluxoSaida = new FileWriter("saidacaracter.txt");
            int c;
            while ((c = fluxoEntrada.read()) != -1) {
                fluxoSaida.write(c);
            }
        } finally {
            if (fluxoEntrada != null) {
                fluxoEntrada.close();
            }
            if (fluxoSaida != null) {
                fluxoSaida.close();
            }
        }
    }
}
```

CopiarCaracteres é moi parecido a *CopiarBytes*. A principal diferencia descansa na utilización de clases diferentes para facer a entrada e a saída (*FileReader* e *FileWriter* frente a *FileInputStream* e *FileOutputStream*). Observa que tanto *CopiarCaracteres* como *CopiarBytes* utilizan un varible int para ler e escribir. Sen embargo neste caso o carácter almacenarase utilizando os 16 bits en tanto que o byte do caso anterior almacenase utilizando unicamente os últimos 8 bits.

1.4.2 Fluxos de caracteres utilizando fluxos de bytes

Os fluxos de caracteres son habitualmente "wrappers"(envoltorios) para os fluxos de bytes. O fluxo de caracteres utiliza un fluxo de bytes para realizar a operación de E/S desexada, mentres que o fluxo de caracteres encargase unicamente da tradución entre caracteres e bytes. Por exemplo, *FileReader* usa *FileInputStream*, mentres que *FileWriter* utiliza *FileOutputStream*.

Existen dous fluxos "ponte" entre bytes e caracteres: InputStreamReader e OutputStreamReader. Utilizarémolas cando non existan fluxos de caracteres apropiados ás nosas necesidades.

1.4.3 E/S en modo liña

As operacións de E/S con caracteres ocorren normalmente en unidades maiores que caracteres simples. Así, unha unidade común de intercambio é a liña (unha cadea de caracteres con un terminador de liña ó final). O terminador de liña pode ser o retorno de carro máis o cambio de liña ("\r\n") ou o retorno de carro ("\r") ou o cambio de liña("\n").

Para ilustrar a súa utilización imos a modificar o exemplo *CopiarCaracteres* para usar operacións de E/S en modo liña. Para iso usaremos as clases BufferedReader e PrintWriter:

Este exemplo, que chamaremos *CopiarLiñas* invoca os métodos `BufferedReader.readLine` e `PrintWriter.println` para realizar a E/S liña por liña.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;
public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader fluxoEntrada= null;
        PrintWriter fluxoSaida= null;
        try {
```

```
                fluxoEntrada=
                    new BufferedReader(new FileReader("xanadu.txt"));
                fluxoSaida=
                    new PrintWriter(new FileWriter("saidacaracteres.txt"));
                String l;
                while ((l = fluxoEntrada.readLine()) != null) {
                    fluxoSaida.println(l);
                }
            } finally {
                if (fluxoEntrada!= null) {
                    fluxoEntrada.close();
                }
                if (fluxoSaida!= null) {
                    fluxoSaida.close();
                }
            }
        }
    }
```

A chamada a *readLine* devolve unha liña de texto lida dende o ficheiro de entrada, en tanto que *println* enviará á saída un bloque de caracteres aos que previamente lles engadiu o indicador de final de liña (depende do SO).

1.4.4 Fluxos de buffer

Os exemplos vistos ata agora realizan operacións de E/S sen buffer. Isto significará que cada operación de lectura ou escritura será manexada directamente polo SO, o que fará o programa moito máis lento xa que cada operación implicará acceso a disco, acceso á rede e algunhas outras operacións que polo xeral son bastante custosas.

Para reducir este inconveniente podemos utilizar fluxos de E/S con buffer. Así as lecturas realizaranse dende unha memoria intermedia chamada buffer (a chamada á lectura nativa realizase unicamente cando o buffer esté baleiro). De xeito similar, os fluxos de saída escribirán os datos nun buffer ou memoria intermedia dende o que se fará a escritura nativa unicamente cando este estea cheo.

Un programa pode converter un fluxo sen buffer nun con buffer usando clases "envolventes"("wrapper" en inglés) ás que se lle pasarán no construtor o nome da clase co fluxo sen buffer. A continuación móstrase os cambios a realizar no exemplo *CopiarLiñas* para utilizar fluxos con buffer:

```
fluxoEntrada = new BufferedReader(new FileReader("xanadu.txt"));
fluxoSaida =  new BufferedWriter(new FileWriter("saidacaracteres.txt"));
```

Existen 4 clases que permiten "envolver" os fluxos sen buffer (tanto de bytes como de caracteres) para convertelos en fluxos con buffer. BufferedInputStream e BufferedOutputStream para crear fluxos de bytes con buffer, mentres que BufferedReader e BufferedWriter crean fluxos de caracteres con buffer. Lembra que a cada unha destas clases se lles pasará como parámetro no seu construtor o nome do fluxo sen buffer.

1.4.5 Facer un "flush" dun fluxo con buffer.

Ás veces faise necesario escribir o contido do buffer en momentos determinados, sen ter que esperar a que estea cheo. Esta operación chámase facer un "flush" do buffer.

Algunhas clases de saída con buffer soportan "autoflush", o que virá determinado mediante un parámetro extra no seu construtor. Cando se activa o opción de "autoflush", certos eventos fan que o buffer se limpe. Por exemplo un "autoflush", no caso dun obxecto *PrintWriter* prodúcese cada vez que invocamos os métodos *println* ou *format*.

Fara facer un "flush" dun xeito manual, invocaremos directamente o método `flush`. Este método pode ser invocado dende calquera fluxo de saída, pero so terá algún efecto se o fluxo ten buffer.

1.5 Escaneado e formatado

A programación E/S implica normalmente mostrar os datos nun formato amigable para o usuario final. Para axudarnos con este tarefa, o API Java proporciona certas clases para o escaneado ou ruptura dos fluxos de entrada en unidades máis pequenas e outras para o formatado ou configuración dos fluxos para mostrar nun formato máis humano.

1.5.1 Escanear

Os obxectos do tipo Scanner utilízanse para dividir entradas con formato en unidades máis pequenas ou "tokens" e traducilos en función do seu tipo de datos.

1.5.1.1 Dividir un fluxo entrada en "tokens"

Un *scanner* por defecto utiliza os espazos en branco para dividir un fluxo (isto inclúe o espazo en branco, un tab, un terminador de liña. Ver Character.isWhitespace para ver a lista completa). Para ilustrar o funcionamento do escaneado usaremos o programa de exemplo *XanScan* para ler cada unha das palabras do ficheiro *xanadu.txt* e imprimilas por pantalla, unha por liña.

```
import java.io.*;
import java.util.Scanner;
public class XanScan {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Observa que 'XanScan' invoca ao final o método `close` mediante o propio obxecto *Scanner*. Aínda que este non é propiamente un fluxo, é preciso pechalo para indicar que xa non se utilizará máis o fluxo a partires do que se construe.

Para utilizar separadores diferentes ao espacio en branco usaremos o método `useDelimiter()`, e unha expresións regular. Por exemplo, supoñamos que queremos que o carácter separador sexa a coma, seguida opcionalmente de un espazo en branco: s.useDelimiter(",\\s*");

1.5.1.2 Traducir un "token"

XanScan trata todos os "tokens" de entrada como tipo *String*. Sen embargo Scanner soporta "tokens" para calquera dos tipos primitivos Java, excepto char, así como para as clases BigInteger e BigDecimal. Os valores numéricos poderán usar separadores de miles, por exemplo en España (un local ES), o *Scanner* lerá correctamente o *String* 32.700 como un valor enteiro. Mencionamos o local, xa que os separadores dos miles e os decimais dependen do país. Polo tanto o exemplo anterior non funcionará correctamente se o local esta posto, por exemplo, a US. Normalmente non debemos preocuparnos destes asuntos xa que os ficheiros de entrada están configurados co mesmo local no que traballa o programa. O exemplo *SumScan* le unha lista de valores double e súmaos todos xuntos:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;
public class ScanSum {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;
        try {
            s = new Scanner(
                    new BufferedReader(new FileReader("numerosformatousa.txt")));
            s.useLocale(Locale.US);
            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                        sum += s.nextDouble();
                    } else {
                        s.next();
                    }
            }
        } finally {
            s.close();
        }
        System.out.println(sum);
```

```
    }
}
```

Aquí temos o ficheiro *numerosformatousa.txt*

```
8.5
32,767
3.14159
1,000,000.1
```

A saída será un String co valor "1032778.74159". O separador decimal será diferente dependendo do local, xa que o *System.out* é un obxecto *PrintStream* no que non hai ningún xeito de sobreescribir o local. Sen embargo podemos sobreescribir o local de todo o programa ou ben usar un conxunto de clases da API para dar formato.

1.5.2 Formato

Os obxectos que implementan fluxos con formato serán instancias de PrintWriter para os fluxos de caracteres e PrintStream para os fluxos de bytes. Como todos os obxectos de fluxo de bytes e caracteres, as instancias de *PrintWriter* e de *PrintStream* implementarán un conxunto estándar de métodos *write* para saídas simples de byte ou de carácter. Además, ambos implementan o mesmo conxunto de métodos de conversión para xerar saídas con formato. Existen dous niveis de formatado:

- **print** e **println** dan formato a valores individuais dun xeito estándar
- **format** formatea case calquerá tipo de valor baseándose nun cadea contendo o formato desexado, con múltiples opcións.

1.5.2.1 Os métodos *print* e *println*

print e **println** enviarán ao dispositivo de saída un valor simple despois de convertelo usando o método *toString* apropiado.

```
public class RaizCadrada {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
        System.out.print("A raiz cadrada de ");
        System.out.print(i);
        System.out.print(" é ");
        System.out.print(r);
        System.out.println(".");
        i = 5;
        r = Math.sqrt(i);
        System.out.println("A raiz cadrada de " + i + " é " + r + ".");
    }
}
```

Aquí mostrase a saída de *RaizCadrada*

```
A raíz cadrada de 2 é 1.4142135623730951.
A raíz cadrada de 5 é 2.23606797749979.
```

As variables *i* e *r* son formatadas dúas veces: a primeira mediante a sobrecarga oportuna do método *print*, a segunda por unha conversión automática xerada internamente polo compilador, que tamén utilizará o método *toString()*. Podemos darlle formato a calquera valor deste xeito, pero non teremos demasiado control sobre os resultados.

1.5.2.2 O método *format*

Este método dará formato a múltiples argumentos baseándose nunha cadea co formato desexado na que poderemos especificar opcións bastante complexas, aínda que neste apartado só veremos exemplos sinxelos. Para un exame máis detallado ver a sintaxe das cadeas de formato na API.

O exemplo *RaizCadrada2* da formato a dous números con unha soa chamada ao método *format*.

```
public class RaizCadrada2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
        System.out.format("A raíz cadrada de %d é %f.%n", i, r);
    }
}
```

A saída será

```
A raíz cadrada de 2 é 1.414214.
```

Tal e como se ve no exemplo anterior, todas as cadeas de formato empezan con carácter % e rematan con un ou dous caracteres de conversión que indicarán que tipo de saída imos xerar. Os tres tipos utilizados aquí son:

- **d** mostra unha variable enteira como decimal
- **f** mostra unha variable en punto flotante como decimal
- **n** emite un terminador de liña

Aquí temos outros tipos de conversións:

- **x** da un formato hexadecimal a un enteiro
- **s** da un formato *String* a calquera tipo de valor.
- **TB** da formato a un enteiro como un nome de mes (dependerá do local)

Existen moitos outros tipos de conversións (ver API).

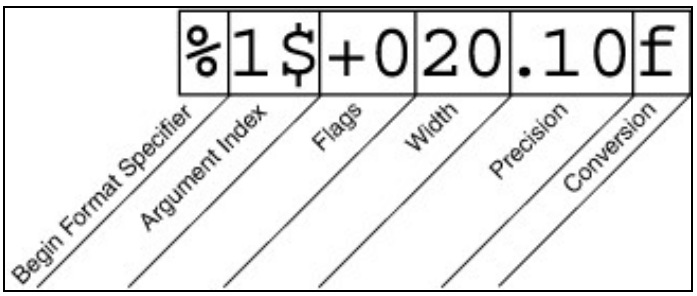
Ademais da conversión, o formato permite outro tipo de parámetros para personalizar a saída. A continuación mostrase un exemplo:

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

A saída será

```
3.141593, +00000003.1415926536
```

Os elementos adicionais son opcionais. A seguinte figura mostra como un especificador de formato máis longo se divide en elementos máis pequenos.



Os elementos mostrados na figura teñen que aparecer na orde mostrada. Dende a dereita, a lista de elementos e:

- **Precisión:** en valores en punto flotante representa a precisión matemática do valor. Coa conversión s(*String*) este parámetro conterá o ancho máximo do valor. Este truncarase pola dereita en caso de ser necesario.
- **Ancho** (*Width*): o ancho mínimo do valor. En caso necesario completarase o valor pola esquerda con espazos en branco (defecto).
- **Bandeiras** (*Flags*): opcións de formato adicionais. O **+** indica que o número sempre debe ir precedido do signo, o **0** especifica que este será o carácter de recheo. **-** indica que se completará ata o tamaño mínimo pola dereita. A **,** indica que debe aparecer o separador de miles que o local indique.
- **Índice do argumento** (*Argument index*): un número enteiro positivo indicando a posición do argumento na lista de argumentos. A lista empeza en 1, non en 0, para a primeira posición.

1.6 E/S dende a liña de comandos

Algunhas veces un programa é executado dende a liña de comando. Java soporta este tipo interacción de dúas maneiras: con fluxos estándar ou con consola.

1.6.1 Fluxos estándar

Os fluxos estándar están presentes na meirande parte dos SO. A entrada por defecto é o teclado, mentres que a saída por defecto é o monitor. Tamén

é posible definir fluxos E/S sobre ficheiros, ou entre programas, aínda que esta característica se controla a través do intérprete da liña de comando, non co programa en si.

Java soporta tres fluxos estándar:

1. **A entrada estándar**, á que se accede con *System.in*
2. **A saída estándar**, á que se accede con *System.out*
3. **O erro estándar** ao que se accede mediante *System.err*.

Estes obxectos están dispoñibles sempre sen necesidade de abrilos explicitamente. Para obter máis información consultar os fluxos estándar.

Os fluxos estándar, por razóns históricas, son fluxos de bytes. *System.out* e *System.err* defínense como obxectos da clase PrintStream, que aínda que é un fluxo de bytes, utiliza un obxecto de fluxo de caracteres interno para proporcionar moitas das características deste tipo de fluxos. Sen embargo, *System.in* é un fluxo de bytes, con ningunha característica dos fluxos de caracteres. Para usar a entrada estándar como un fluxo de caracteres, hai que envolver (*wrap*) o *System.in* nun obxecto *InputStreamReader*.

```
InputStreamReader cin = new InputStreamReader(System.in);
```

Para ler do teclado liña por liña usando os fluxos estándar crearemos un *InputStreamReader* a partires do *System.in* (como no exemplo anterior) e pasaremos este *InputStreamReader* ao construtor dun *BufferedReader*, deste xeito as lecturas que fagamos sobre o *BufferedReader* son realidade realizadas sobre o *System.in* coa vantaxe de que se pode ler unha liña completa. A continuación móstrase un exemplo de como facelo. Ten en conta que seguinte fragmento de código pode arroxar excepcións.

```
String liña;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
liña = br.readLine();
System.out.println(liña);
```

1.6.2 A consola

Unha alternativa máis avanzada aos fluxos estándar é a utilización da consola(clase Console), que proporcionará todas as características destes máis algunhas adicionais. A consola é particularmente útil para a introdución segura de contrasinais. A consola tamén proporcionará a entrada e a saída como fluxos de caracteres a través dos métodos *reader* e *writer*.

Antes de que un programa poida usar a consola, debe obter un obxecto Console chamando a *System.console()*. Se a consola está dispoñible o anterior método devolveraa. Se devolve null a consola non estará dispoñible, ben porque o SO non a soporta, ben porque o programa foi lanzado nun entorno non interactivo.

A consola permite a introdución segura de contrasinais a través do método *readPassword*, que suprimirá o echo a pantalla dos caracteres tecleados e devolvendo este nun array de chars, non nun *String*, de xeito que se poida sobreescribir o contrasinal unha vez que xa non sexa necesario, borrando así da memoria.

O seguinte exemplo permite cambio dun contrasinal usando a consola.

```
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;
public class Contrasinal {
    public static void main (String args[]) throws IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println("A consola non está dispoñible.");
            System.exit(1);
        }
        String usuario = c.readLine("Usuario: ");
        char [] contrasinalVello = c.readPassword("Contrasinal vello: ");
        if (comprobarContrasinal(usuario, contrasinalVello)) {
            boolean sonDistintos;
            do {
                char [] novoContrasinal1 =
                    c.readPassword("Novo contrasinal: ");
                char [] novoContrasinal2 =
                    c.readPassword("Novo contrasinal outra vez: ");
                sonDistintos = ! Arrays.equals(novoContrasinal1, novoContrasinal1);
                if (sonDistintos) {
```

```
                    c.format("Os contrasinais non son iguais, tenteo de novo.%n");
                } else {
                    cambiarContrasinal(usuario, novoContrasinal1);
                    c.format("Contrasinal de %s cambiado.%n", login);
                }
                Arrays.fill(novoContrasinal1, ' ');
                Arrays.fill(novoContrasinal2, ' ');
            } while (sonDistintos);
        }
        Arrays.fill(contrasinalVello, ' ');
    }
    // Método baleiro, aquí verificaríase se o contrasinal se corresponde co usuario
    static boolean comprobarContrasinal(String usuario, char[] contrasinal) {
        return true;
    }
    // Método baleiro. Aquí iría o código para cambiar o contrasinal
    static void cambiarContrasinal(String usuario, char[] contrasinal) {}
}
```

O seguinte exemplo segue o seguintes pasos:

1. Tenta obter un obxecto *Console*, se non está dispoñible aborta o programa.
2. Chama o método *Console.readLine* para ler o nome do usuario.
3. Chama o método *Console.readPassword* para ler o contrasinal do usuario.
4. Chama o método *comprobarContrasinal* para comprobar se ese usuario está autorizado a cambiar o contrasinal ou non.
5. Repite os seguintes pasos ata que o usuario introduza o mesmo contrasinal dúas veces.
    - Chama ao método *Console.readPassword* dúas veces para obter o novo contrasinal
    - Se o novo contrasinal se teclea igual dúas veces, entón cambia o contrasinal vello polo novo usando o método *cambiarContrasinal*
    - Sobreescribe os dous novos contrasinais tecleados con brancos
6. Sobreescribe o contrasinal vello con brancos.

1.7 Fluxos de datos

Os fluxos de datos soportan E/S binarias cos tipos primitivos(*boolean, char, int, long, float e double*) así como os *Strings*.

Todos os fluxos de datos implementan a interface DataInput ou DataOutput. Esta sección centrarase nas implementacións maís comúns dos anteriores interfaces: DataInputStream e DataOutputStream.

O exemplo *FluxoDatos* ilustrará os uso dos fluxos de datos escribindo e lendo un conxunto dos mesmos.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.EOFException;
public class DataStreams {
    static final String ficheiroDatos= "factura";
    static final double[] prezos= { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] unidades= { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Camiseta Java",
        "Tazón Java",
        "Boneca Java",
        "Pin Java",
        "Cadea para chaves Java" };
    public static void main(String[] args) throws IOException {
        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new
                    BufferedOutputStream(new FileOutputStream(ficheiroDatos)));
            for (int i = 0; i < prezos.length; i ++) {
                out.writeDouble(prezos[i]);
                out.writeInt(unidades[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
```

```
        }

        DataInputStream in = null;
        double total = 0.0;
        try {
            in = new DataInputStream(new
                    BufferedInputStream(new FileInputStream(ficheiroDatos)));
            double prezo;
            int unidade;
            String desc;
            try {
                while (true) {
                    prezo= in.readDouble();
                    unidade= in.readInt();
                    desc = in.readUTF();
                    System.out.format("Mercáronse %d unidades de %s a $%.2f%n",
                         unidade, desc, prezo);
                    total += unidade * prezo;
                }
            } catch (EOFException e) { }
            System.out.format("Por un total de: $%.2f%n", total);
        }
        finally {
            in.close();
        }
    }
}
```

Cada rexistro estará formado por tres valores de tres tipos diferentes presentes nunha factura, tal e como mostra a seguinte táboa.

Orde no rexistro	Tipo datos	Descripción	Método saída	Método entrada	Valor exemplo
1	double	Prezo	DataOutputStream.writeDouble	DataInputStream.readDouble	19.99
2	int	Unidades	DataOutputStream.writeInt	DataInputStream.readInt	12
3	String	Descripción	DataOutputStream.writeUTF	DataInputStream.readUTF	"Camiseta Java"

Examinemos as seccións cruciais de código dentro do exemplo. Primeiro, o programa define algunhas constantes contendo o nome do ficheiro de datos e os datos que se escribirán no mesmo.

```
static final String ficheiroDatos = "factura";
static final double[] prezos = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] unidades = { 12, 8, 13, 29, 50 };
static final String[] descs = { "Camiseta Java",
        "Tazón Java",
        "Boneca Java",
        "Pin Java",
        "Cadea para chaves Java" };
```

Logo ábrese un fluxo de saída. Xa que un obxecto *DataOutputStream* pode crearse unicamente como "wrapper"(envoltura) para un obxecto xa existente de fluxo de datos, neste caso proporcionase un fluxo de saída de bytes cara un ficheiro con buffer.

```
out = new DataOutputStream(new
            BufferedOutputStream(new FileOutputStream(ficheiroDatos)));
```

Nel escribiranse cada un dos rexistros e logo pecharase o fluxo de saída.

O método *writeUTF* escribe Strings no formato UTF-8.

Agora *FluxosDatos* lerá de novo os datos escritos. Primeiro debemos crear un fluxo de entrada, e as variables que conterán os datos de entrada. Como no caso dos *DataOutputStream*, *DataInputStream* debe construírse como "wrapper" dun fluxo de bytes tal e como se mostra a continuación:

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(ficheiroDatos)));
double prezo;
int unidade;
String desc;
```

```
double total = 0.0;
```

Agora pódese ler cada un dos rexistros dentro do fluxo, mostrando cada un dos datos lidos.

```
try {
    while (true) {
        prezo = in.readDouble();
        unidade = in.readInt();
        desc = in.readUTF();
        System.out.format("Mercáronse %d unidades de %s a $%.2f%n",
                unidade, desc, prezo);
        total += unidade * prezo;
    }
} catch (EOFException e) {
}
```

Observa que no anterior exemplo detéctase a condición de final de ficheiro capturando a excepción EOFException, en lugar de comprobar os valores de retorno inválidos. Todas as implementacións dos métodos *DataInput* utilizarán *EOFException* en lugar de valores de retorno.

Observa tamén cada un dos *write* dentro do fluxo de datos correspóndese exactamente con un *read*. Será tarefa do programador comprobar que os tipos escritos se corresponden exactamente cos tipos lidos ou viceversa. Os fluxos de entrada consistirán en fluxos de datos binarios, sen indicación algunha do tipo de cada valor, ou de onde comeza e remata o fluxo.

O exemplo *FluxosDatos* utiliza unha mala técnica de programación que consiste no uso de de números en punto flotante para representar os prezos. En xeral, o punto flotante non resulta axeitado para valores precisos, sendo particularmente pouco axeitado para fraccións decimais, xa que valores comúns tales como (0.1) non teñen unha representación binaria.

O tipo axeitado para usar neste exemplo sería java.math.BigDecimal. Desafortunadamente, *BigDecimal* non é un tipo primitivo senón un obxecto e por tanto non funcionará con fluxos de datos. Para utilizar este obxecto debemos facer uso dos fluxos de obxectos.

1.8 Fluxos de obxectos

Así como os fluxos de datos soportan E/S dos tipos primitivos, os fluxos de obxectos soportarán E/S de obxectos. A meirande parte das clases estándar Java, aínda que non todas, soportan a **serialización** dos seus obxectos. Para que un obxecto poda almacenarse en disco é necesario que a clase á que pertence sexa serializable. Serán clases serializables aquelas que implementen o interface `java.io.Serializable`. O interface serializable non contén ningún método, basta con que unha clase o implemente para que os seus obxectos podan ser serializados pola máquina virtual e, xa que logo, almacenados no disco.

As clases que permiten traballar con fluxos de obxectos son ObjectInputStream e ObjectOutputStream. Estas clases implementan ObjectInput e ObjectOutput que son subinterfaces de *DataInput* e *DataOutput* respectivamente. Isto significa que todos os métodos de E/S con datos primitivos vistos na sección anterior están tamén presentes nos fluxos de obxectos. Deste xeito un fluxo de obxecto pode conter un mistura de datos primitivos e valores de obxecto. O seguinte exemplo ilustrará isto, facendo o mesmo que no caso anterior aínda que esta vez usando fluxos de obxectos. Hai, con todo, algunhas diferencias co exemplo da sección anterior (*FluxoDatos*). Primeiro, os prezos agora veñen representados mediante obxectos *BigDecimal*. Segundo, escribirase un obxecto Calendar no ficheiro de datos, indicando a data da factura.

Se o método *readObject()* non devolve o tipo de obxecto esperado, podemos obter a excepción ClassNotFoundException. No exemplo seguinte isto nunca sucederá, así que non se captura esta excepción. Aínda así dicímoslle ao compilador que somos conscientes desa posibiliade engadindo un throws *ClassNotFoundException* no método principal.

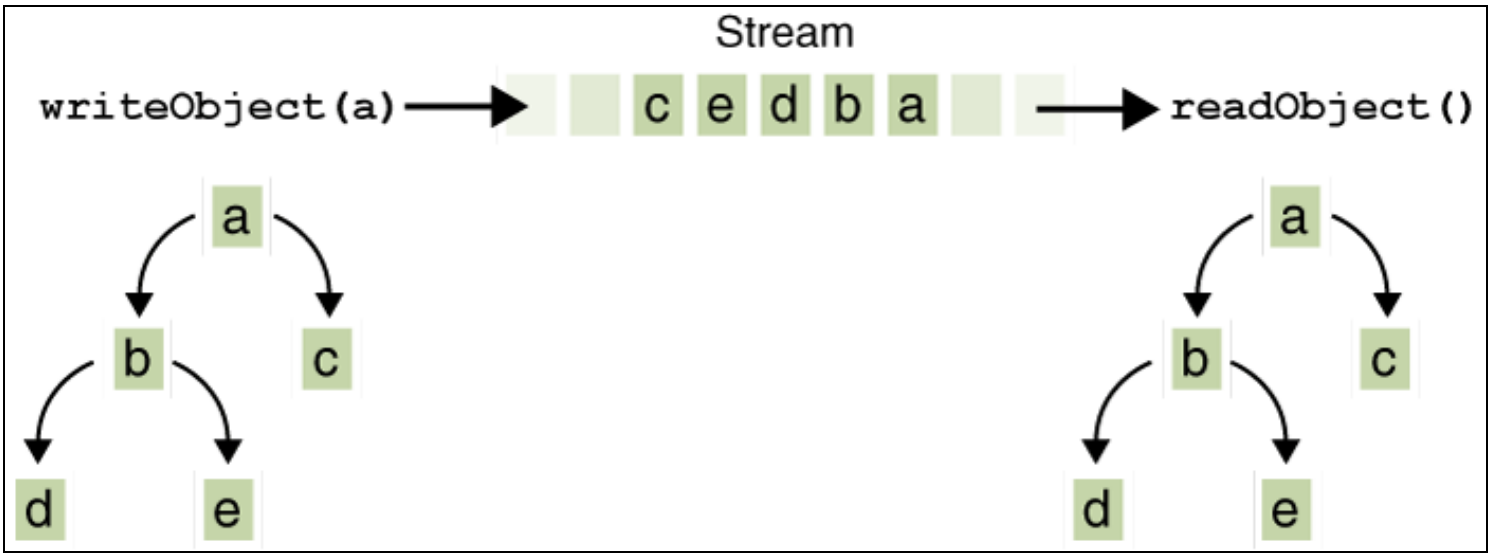
```
import java.io.*;
import java.math.BigDecimal;
import java.util.Calendar;
public class ObjectStreams {
    static final String ficheiroDatos = "factura";
    static final BigDecimal[] prezos = {
        new BigDecimal("19.99"),
        new BigDecimal("9.99"),
        new BigDecimal("15.99"),
        new BigDecimal("3.99"),
        new BigDecimal("4.99") };
    static final int[] unidades = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Camiseta Java",
            "Tazón Java",
            "Boneca Java",
            "Pin Java",
            "Cadea para chaves Java" };
```

```
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(new
                    BufferedOutputStream(new FileOutputStream(ficheiroDatos)));
            out.writeObject(Calendar.getInstance());
            for (int i = 0; i < prezos.length; i ++) {
                out.writeObject(prezos[i]);
                out.writeInt(unidades[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }
        ObjectInputStream in = null;
        try {
            in = new ObjectInputStream(new
                    BufferedInputStream(new FileInputStream(ficheiroDatos)));
            Calendar data = null;
            BigDecimal prezo;
            int unidade;
            String desc;
            BigDecimal total = new BigDecimal(0);
            data = (Calendar) in.readObject();
            System.out.format ("No %tA, %<tB %<te, %<tY:%n", data);
            try {
                while (true) {
                    prezo = (BigDecimal) in.readObject();
                    unidade = in.readInt();
                    desc = in.readUTF();
                    System.out.format("Mercaches %d unidades de %s a $%.2f%n",
                            unidade, desc, prezo);
                    total = total.add(prezo.multiply(new BigDecimal(unidade)));
                }
            } catch (EOFException e) {}
            System.out.format("Por un total de: $%.2f%n", total);
        } finally {
            in.close();
        }
    }
}
```

1.8.1 Entrada e saída de obxectos complexos

Os métodos *readObject* e *writeObject* son sinxelos de usar, sen embargo dentro conteñen algunha lóxica de control bastante complicada. Isto non resulta demasiado importante para clases como *Calendar* que o único que fai é encapsular tipos primitivos. Sen embargo algúns obxectos conteñen referencias a outros. Se o método *readObject* ten que construír un obxecto a partires dun fluxo, ten que ser capaz de construír tamén todos os obxectos aos que o obxecto orixinal fai referencia. Estes obxectos adicionais poden tamén ter a súas propias referencias, e así sucesivamente. Nesta situación *writeObject* percorrerá toda a rede de obxectos referenciados polo obxecto orixinal para incluílos tamén dentro do fluxo. Deste xeito unha soa invocación ao método *writeObject* pode causar que un gran número de obxectos sexan escritos no fluxo.

Isto demóstrase na seguinte figura, onde se invoca *writeObject* para escribir un obxecto chamado **a**, que á súa vez contén referencias aos obxectos **b** e **c**, mentres que **b** contén referencias a **d** e mais **e**. Chamar a *writeObject(a)*, escribe non só **a**, senón todos obxectos precisos para reconstruír **a**(neste caso **b**,**c**,**d**,**e**). Cando se le a mediante *readObject*, os outros 4 obxectos tamén se len e, por tanto, as referencias orixinais mantéñense.



Podémonos preguntar que ocorre se dous obxectos dentro do mesmo fluxo conteñen unha referencia a un mesmo obxecto. Un fluxo pode conter unha única copia dun obxecto aínda que haxa moitas referencias ao mesmo. Desta maneira se escribimos explicitamente un obxecto dentro dun fluxo dúas veces, estaremos escribindo realmente un único obxecto e dúas referencias. Exemplo:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Cada *writeObject* ten que ter o seu correspondente *readObject*, así que o código para ler o anterior fluxo de obxectos será:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

Obteremos dúas variables *ob1* e *ob2*, que referencian un único obxecto.

Sen embargo, cando escribimos un obxecto a dous fluxos diferentes, o obxecto estarase duplicando realmente. Así un programa que lea os dous fluxos obterá dous obxectos diferentes.

1.9 Obxectos File

A clase File simplifica a escritura de código multiplataforma para manipular ficheiros. O nome desta clase é un pouco enganoso xa que **as instancias da clase *File* representan os nomes dos ficheiros e non os ficheiros en si mesmos**.

Ás veces podemos crear obxectos *File* para ficheiros que non existen realmente, isto ten sentido cando queremos "parsear" o nome do ficheiro para ver se está ben formado ou non. Un ficheiro tamén pode crearse pasando un obxecto *File* ao construtor dalgunhas clases, tales como *FileWriter*.

Se o ficheiro existe, o programa pode examinar os seus atributos e realizar distintas operacións sobre o mesmo, tales como renomeado, borrado, ou modificación dos seus permisos.

A clase File non se utiliza para transferir información entre a aplicación e o disco (para iso xa están os fluxos), senón para obter información sobre os ficheiros e directorios de este.

1.9.1 Un ficheiro con moitos nomes

Un obxecto *File* contén un *String* co seu nome, utilizado para construílo. O *String* nunca cambia ó longo da vida do obxecto. O programa pode usar o obxecto *File* para obter outras versións do nome do ficheiro, sendo estas iguais, ou non, ás do *String* pasado ao construtor.

Supoñamos que creamos un obxecto *File* do seguinte xeito:

```
File a = new File("xanadu.txt");
```

O programa chamará un número determinado de métodos para obter versións diferentes do nome do ficheiro. A continuación móstrase unha táboa coas diferentes versións dos nomes do ficheiro dependendo da plataforma.

Método chamado	Valor devolto en Windows	Valor devolto en Solaris / Linux
a.toString()	xanadu.txt	xanadu.txt
a.getName()	xanadu.txt	xanadu.txt
a.getParent()	NULL	NULL
a.getAbsolutePath()	c:\java\exemplos\xanadu.txt	/home/cafe/java/exemplos/xanadu.txt
a.getCanonicalPath()	c:\java\examplos\xanadu.txt	/home/cafe/java/exemplos/xanadu.txt

Tamén se pode crear un obxecto *File* a partires dun nome máis complicado, usando File.separator para especificar o nome do ficheiro dun xeito non dependente da plataforma.

```
File b = new File(".." + File.separator + "exemplos" + File.separator + "xanadu.txt");
```

Aínda que **b** se refire ao mesmo ficheiro que **a**, os métodos devolven valores lixeiramente diferentes.

Método chamado	Valor devolto en Windows	Valor devolto en Solaris / Linux
b.toString()	..\exemplos\xanadu.txt	../exemplos/xanadu.txt
b.getName()	xanadu.txt	xanadu.txt
b.getParent()	..\exemplos	../exemplos
b.getAbsolutePath()	c:\java\exemplos\..\exemplos\xanadu.txt	/home/cafe/java/exemplos/../exemplos/xanadu.txt
a.getCanonicalPath()	c:\java\exemplos\xanadu.txt	/home/cafe/java/exemplos/xanadu.txt

O programa devolverá os mesmos resultados nunha plataforma Linux que nunha plataforma Solaris. Podería ser que o método File.compareTo() non considerase **a** e **b** como o mesmo ficheiro. Aínda que ambos os dous se refiren ao mesmo, os nomes usados para construílo son diferentes.

O exemplo seguinte (*Ficheiros*) crea obxectos *File* a partires de nomes pasados dende a liña de comandos, e exercita diversos métodos sobre eles. Executa o exemplo usando diferentes nomes de ficheiros. Inclúe nomes de ficheiros ou de directorios que non existan e comproba o resultado. Proba a pasarlle tamén nomes de ficheiros con rutas relativas e absolutas.

```
import java.io.File;
import java.io.IOException;
import static java.lang.System.out;
public class Ficheiros {
    public static void main(String args[]) throws IOException {
        out.print("Raíz do sistema de ficheiros: ");
        for (File root : File.listRoots()) {
            out.format("%s ", root);
        }
        out.println();
        for (String fileName : args) {
            out.format("%n------%nnew File(%s)%n", fileName);
            File f = new File(fileName);
            out.format("toString(): %s%n", f);
            out.format("exists(): %b%n", f.exists());
            out.format("lastModified(): %tc%n", f.lastModified());
            out.format("isFile(): %b%n", f.isFile());
            out.format("isDirectory(): %b%n", f.isDirectory());
            out.format("isHidden(): %b%n", f.isHidden());
            out.format("canRead(): %b%n", f.canRead());
            out.format("canWrite(): %b%n", f.canWrite());
            out.format("canExecute(): %b%n", f.canExecute());
            out.format("isAbsolute(): %b%n", f.isAbsolute());
            out.format("length(): %d%n", f.length());
            out.format("getName(): %s%n", f.getName());
            out.format("getPath(): %s%n", f.getPath());
            out.format("getAbsolutePath(): %s%n", f.getAbsolutePath());
            out.format("getCanonicalPath(): %s%n", f.getCanonicalPath());
            out.format("getParent(): %s%n", f.getParent());
```

```
                out.format("toURI: %s%n", f.toURI());
            }
        }
    }
```

1.9.2 Manipular ficheiros

Se un obxecto *File* se refire a un ficheiro que realmente existe no sistema de ficheiros, un programa poderá usalo para realizar unha serie de operacións sobre o mesmo, incluíndo pasar o obxecto *File* ó construtor dun fluxo para abrilo para lectura ou escritura.

O método **delete** borra o ficheiro inmediatamente, mentres que **deleteOnExit** borra o ficheiro cando finaliza a máquina virtual.

setLasModified establece a data de modificación do ficheiro. Por exemplo para modificar a data de *xanadu.txt* un programa podería facer o seguinte:

```
new File("xanadu.txt").setLastModified(new Date().getTime());
```

O método **renameTo()** renomea un ficheiro. Observa sen embargo que o *String* co nome do ficheiro permanecerá inmutable, e por tanto o obxecto *File* xa non se referirá o ficheiro renomeado.

1.9.3 Traballar con directorios

O método **mkdir** crea un novo directorio.

Os métodos **list** e **listFiles** devolven o contido dun directorio. O método *list* devolve un array de *Strings* cos nomes dos ficheiros, mentres que *listFiles* devolve un array de *Files*.

1.9.4 Métodos estáticos

O método **createTempFile** crea un novo ficheiro cun nome único e devolve un obxecto *File* apuntado a el.

O método **listRoots** devolve unha lista dos nomes dos directorios raíz. En Windows será *c:*\, *d:*\, etc. mentres que en linux será */*.

1.10 Ficheiros de acceso aleatorio

Os ficheiros de acceso aleatorio permiten accesos non secuenciais, ou aleatorios, aos seus contidos.

Consideremos un ficheiro ZIP contendo outros ficheiros comprimidos. Tamén conterá unha entrada directorio ó final cos nomes dos distintos ficheiros comprimidos, así como o lugar onde empezan.

Supoñamos agora que queremos extraer un ficheiro concreto do ZIP, se utilizamos un acceso secuencial teríamos que seguir os seguintes pasos:

1. Abrir o ficheiro ZIP.
2. Buscar a través de todo o ZIP ata atopar o ficheiro que queremos extraer.
3. Extraer o ficheiro.
4. Pechar o ficheiro ZIP.

Usando o anterior procedemento, de media teríamos que ler polo menos a metade do ficheiro ZIP antes de atopar o ficheiro desexado. Sen embargo podemos extraer o mesmo ficheiro dun ZIP utilizando un método máis eficiente a través da facilidade de búsqueda dentro dos ficheiros de acceso aleatorio.

1. Abrir o ficheiro ZIP.
2. Buscar o directorio dentro do ZIP e localizar a entrada buscada.
3. Ir a posición indicada polo directorio.
4. Extraer o ficheiro desexado.
5. Pechar o ficheiro ZIP.

Este último algoritmo é máis eficiente, xa que só le o directorio e o ficheiro que queremos extraer.

A clase java.io.RandomAccessFile implementa os interfaces *DataInput* e *DataOutput* e por tanto pode ser usada tanto para lectura como para escritura. *RandomAccessFile* é similar a *FileInputStream* e *FileOutputStream* en que hai que especificar un ficheiro para abrir dentro do sistema nativo de ficheiros.

Cando creamos un obxecto *RandomAccessFile*, debemos indicar se accederemos a ese ficheiro en modo lectura ou en modo escritura. O seguinte exemplo crea un obxecto *RandomAccessFile* para ler o ficheiro *xanadu.txt*

```
new RandomAccessFile("xanadu.txt", "r");
```

Este outro exemplo abre o mesmo ficheiro para lectura e escritura:

```
new RandomAccessFile("xanadu.txt", "rw");
```

Unha vez que se abre un ficheiro podemos usar os métodos *write* e *read* definidos nos interfaces *DataInput* e *DataOutput* para realizar as operacións E/S desexadas sobre o ficheiro.

Os obxectos *RandomAccessFile* soportan apuntadores de ficheiros. Estes apuntadores indican a posición actual dentro do ficheiro. Cando se crea o ficheiro este apuntador establécese a 0, indicando así o comezo do ficheiro. As chamadas aos métodos *read* e *write* modificarán este apuntador en función do número de bytes escritos ou lidos.

Ademais dos métodos de E/S sobre o ficheiro que implicitamente modifican o apuntador, a clase *RandomAccessFile* contén tres métodos que manipulan explicitamente o devandito apuntador.

- *int skipBytes(int)* : Move o apuntador cara diante o número de bytes indicados.
- *void seek(long)*: Posiciona o apuntador diante do byte indicado.
- *long getFilePointer()*: Devolve a posición actual do apuntador.