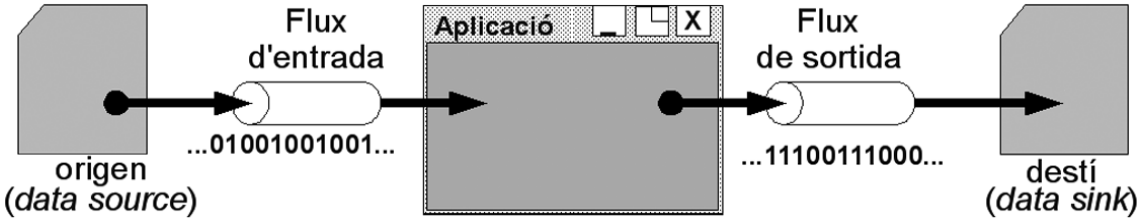
UD10. ENTRADA I SORTIDA DE DADES. FITXERS

10.1 Definició de flux de dades

Per poder dur a terme operacions d'entrada i de sortida, de manera que sigui possible llegir o escriure dades, Java disposa d'un mecanisme unificat, independent de l'origen o la destinació de les dades: els fluxos (stream).

Un flux (**stream**) és el terme abstracte usat per referir-se al mecanisme que permet a un conjunt de dades seqüencials transmetre's des d'un origen de dades (**data source**) a una destinació de dades (**data sink**).

Els **fluxos d'entrada** serveixen per llegir dades des d'un origen (per exemple, seria el cas de llegir un fitxer), per tal de ser processades, mentre que els **fluxos de sortida** són els responsables d'enviar les dades a una destinació (per a un fitxer, seria el cas d'escriure-hi).



Dins Java, hi ha dos tipus de flux. D'una banda, hi ha els **fluxos orientats a dades**, que són aquells en què les dades que es transmeten són purament **binàries**, amb una interpretació totalment dependent de l'aplicació. D'altra banda, hi ha els **fluxos orientats a caràcter**, en què les dades processades sempre es poden interpretar com a **text**. Els primers operen sempre amb el tipus primitiu **byte** i els segons ho fan amb el tipus **char**.

Totes les classes vinculades a l'entrada/sortida en Java es troben definides dins el package **java.io**. Dins aquest mateix paquet també es defineix el tipus d'**excepció** vinculada a errors sorgits durant el procés de lectura i escriptura de dades: IOException.

10.2 Gestió de fitxers

Dins la biblioteca java.io, la classe que representa un fitxer a Java és **File**. Aquesta permet al desenvolupador manipular qualsevol aspecte vinculat al sistema de fitxers. Es pot usar tant per manipular fitxers de dades com directoris.

La classe **File** indica, més concretament, **una ruta** dins el sistema de fitxers.

El constructor més emprat es el següent:

```
public File (String ruta)
```

A l'hora de definir la cadena que indicarà la ruta hem de tenir en compte que el caràcter emprat pels sistemes operatius per separar directoris pot ser diferent: en Linux es la barra (/) i en Windows la contrabarra (\ ). De fet, en sistemes Windows cal ser especialment acurat amb aquest fet, ja que la contrabarra no és un caràcter permès dins una cadena de text, en servir per declarar valors especials d'escapament (\n salt de línia, \t tabulador, etc.).

L'element final de la ruta pot existir realment o no, però això no impedeix de cap manera poder instanciar File. Si s'intenten llegir dades des d'una ruta que en realitat no existeix, es produeix un error, i es llança una FileNotFoundException.

La classe File ofereix tot un seguit de mètodes que permeten realitzar operacions amb la ruta especificada. Alguns dels més significatius per entendre'n les funcionalitats són:

- **public boolean exists()**. Indica si la ruta especificada realment existeix en el sistema de fitxers.
- **public boolean isFile()/isDirectory()**. Aquests dos mètodes serveixen per identificar si la ruta correspon a un fitxer, o bé a un directori.

Els mètodes següents només es poden cridar sobre rutes que especifiquen fitxers o, en cas contrari, no faran res:

- **public long length()**. Retorna la mida del fitxer.
- **public boolean createNewFile()**. Crea un nou fitxer buit en aquesta ruta, si encara no existeix. Retorna si l'operació ha tingut èxit.

Amb la classe File en realitat podem crear nous fitxers o directoris, com veurem en l'exemple següent on es crearà un nou fitxer si encara no existeix:

```
File file = new File(ruta);
if (!file.exists())
      file.createNewFile();
```

En contraposició, els mètodes següents només es poden cridar sobre rutes que especifiquen directoris:

- **public boolean mkdir()**. Crea el directori, si no encara existeix. Retorna si l'operació ha tingut èxit.
- **public String[ ] list()**. En retorna el contingut en forma d'array de cadenes de text.
- **public String[ ] list(FilenameFilter filter)**. Mitjançant el paràmetre addicional filter, és possible filtrar el resultat, de manera que només es retorna el conjunt de fitxers i directoris

que compleixen certs criteris. FilenameFilter és una interface, per la qual cosa és responsabilitat del desenvolupador proporcionar la implementació adequada d'acord amb les condicions en les quals es vol llistar el contingut del directori. Només té un mètode:

- **boolean accept(File dir, String name)**. Cada cop que es crida el mètode list(), aquest crida internament accept per cada fitxer o directori contingut. El paràmetre dir indica el directori en què està ubicat el fitxer o directori processat, mentre que name n'indica el nom. Retorna cert o fals segons si es vol, o no, que sigui inclòs en la llista retornada per la crida al mètode list().

Vegem un exemple d'utilització del filtre per exemple per llistar només els arxius .png d'un directori:

```
file.list( new FilenameFilter() {
      public boolean accept(File f, String name) {
            if (name.endsWith(".png"))
                  return true;
            else
                  return false;
      }
});
```

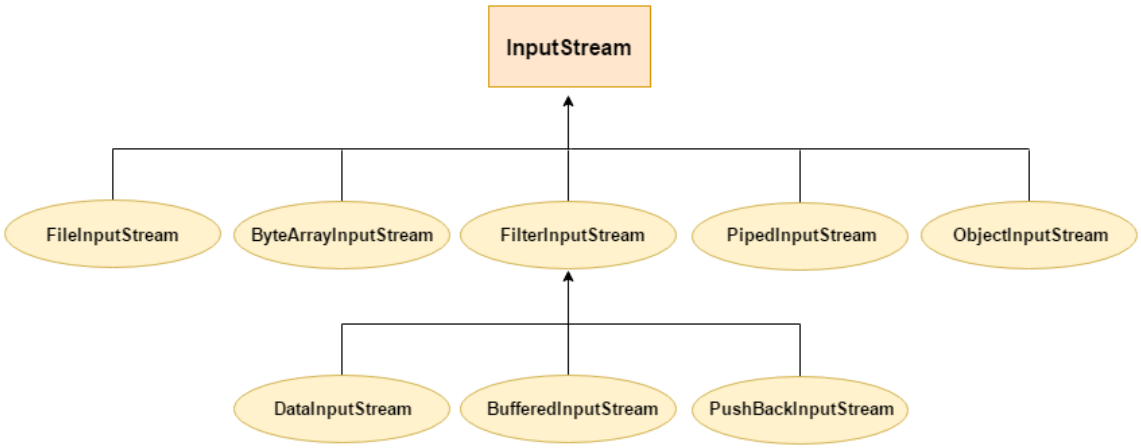
Exercici 10.1: Llistar tot els contingut d'un directori del vostre disc dur.

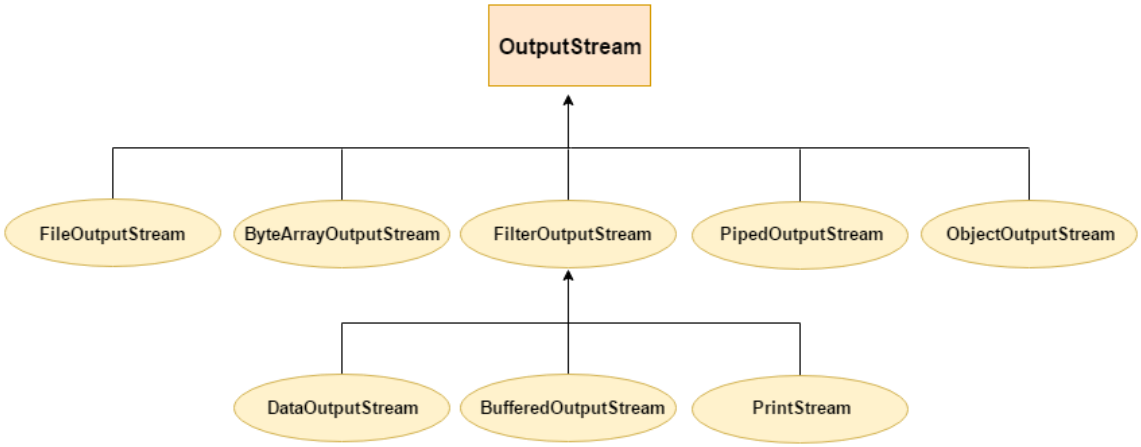
Exercici 10.2: Seleccionar un directori del vostre disc dur que contengui imatges i llistar només les imatges que siguin en format .jpg o .png.

10.3 Fluxos orientats a dades. Les superclasses InputStream i OutputStream

Java ofereix un sistema d'accés homogeni al mecanisme de fluxos orientats a dades mitjançat una jerarquia de classes.

Les superclasses **InputStream** i **OutputStream** especifiquen els mètodes relatius al comportament comú a qualsevol flux, i cada subclasse s'encarrega llavors de sobreescriure'ls, o afegir-ne de nous, segons les seves particularitats. Tots els fluxos a Java hereten d'alguna d'aquestes dues classes.





El fet de que tots els fluxos a Java hereten alguna de les superclasses InputStream i OutputStream té sentit, ja que, per exemple, independentment del format de l'origen de les dades, sempre hi ha l'opció de llegir, però les tasques que cal fer internament per a això són totalment diferents segons l'origen de dades concret. No és el mateix llegir d'un fitxer que d'un buffer de memòria. Hi ha una **subclasse** per cada **tipus d'origen o destinació** de dades.

La classe **InputStream** ofereix els mètodes descrits a continuació per llegir dades des de l'origen mitjançant un **flux d'entrada**. Un aspecte a destacar és que en una operació de lectura **mai no es pot garantir quants bytes es llegiran realment**, independentment que es conegui per endavant el nombre de bytes disponibles a l'origen (i, per tant, a priori, es pugui suposar aquesta garantia). Sempre cal crear algoritmes que tinguin en compte el **valor de retorn** dels diferents mètodes:

- **int available()**. Retorna tots els bytes que hi ha en el flux pendents de ser llegits. En un fitxer, seria el nombre de bytes que ocupa i encara no s'han processat.
- **int read()**. Llegeix exactament un byte. Aquest mètode retorna un enter, ja que fa ús del valor de retorn -1 per indicar que ja no queden més dades per llegir en l'origen. Per obtenir realment el byte llegit cal fer un cast del valor retornat sobre una variable de tipus byte.
- **int read(byte[ ] b)**. Intenta llegir tants bytes com la longitud de l'array passat com a paràmetre, on els emmagatzema a partir de l'índex 0. Retorna el nombre de bytes llegits realment. Cal tenir molt present que si s'ha llegit un nombre de bytes N, inferior a b.length, les dades emmagatzemades entre N i b.length - 1 no són vàlides, ja que no corresponen a la lectura. Retorna -1 si no queden més dades per llegir en l'origen.
- **int read (byte[ ] b, int offset, int len)**. Intenta llegir len bytes, que emmagatzema dins de l'array b a partir de l'índex indicat pel valor offset. Com en el cas anterior, retorna el nombre real de bytes llegits i -1 si no queden més dades per llegir en l'origen.

Al mateix temps, la classe **OutputStream** ofereix els mètodes complementaris per escriure dades cap a la destinació mitjançant un **flux de sortida**:

- **void write(int b)**. Escriu exactament un byte.
- **void write(byte[ ] b)**. Escriu tots els bytes emmagatzemats a b, de manera ordenada des de l'índex 0 a b.length - 1.
- **void write (byte[ ] b, int offset, int len)**. Escriu tots els bytes emmagatzemats a b, de manera ordenada des de l'índex offset a offset + length - 1.

Quan les operacions de lectura o escriptura sobre un flux han finalitzat, és imprescindible tancar-lo. Per tancar qualsevol flux de dades es disposa del mètode **close()**. Només en tancar un flux es pot garantir que absolutament qualsevol dada escrita ja es troba realment a la destinació.

10.3.1 Origen i destinació en fitxers

Dins la jerarquia de fluxos orientats a dades, les classes responsables de crear fluxos **vinculats a fitxers**, aquestes són les classes **FileInputStream** (per lectura, origen) i **FileOutputStream** (per escriptura, destinació). Ambdues disposen de constructors que tenen com a paràmetre d'entrada o bé una instància de **File**, o directament una **cadena de text** amb la ruta del fitxer:

```
FileInputStream(File fitxer)
FileInputStream(String ruta)
FileOutputStream(File fitxer)
FileOutputStream(String ruta)
FileOutputStream(File fitxer, boolean append)
FileOutputStream(String ruta, boolean append)
```

Sempre que es genera un flux de sortida sobre un fitxer ja existent, aquest **se sobreescriu**, de manera que es perden absolutament totes les dades emmagatzemades anteriorment. L'única excepció d'aquest comportament són els constructors amb el paràmetre **append**. Aquest permet indicar, si es crida amb el valor true, que es volen concatenar les dades tot just a partir del final del fitxer actual.

Exemple: Còpia d'un fitxer.

Com exemple del funcionament dels fluxos orientats a dades vinculats a fitxers, a continuació es mostra el fragment de codi que realitzaria una còpia del fitxer ubicat a ruta, escrivint-lo a novaRuta exactament igual. En l'exemple, les dades es llegeixen en blocs de 100 bytes consecutius, si bé cal tenir molt present que mai es pot donar per garantit el nombre de bytes llegits realment en una crida del mètode read. Per aquest motiu, és imprescindible controlar que només s'escrigui a la destinació exactament el mateix nombre de bytes que s'ha llegit.

```
InputStream in = new FileInputStream(ruta);
OutputStream out = new FileOutputStream(novaRuta);
copiaDades(in, out);
...
public void copiaDades(InputStream in, OutputStream out) {
      try {
            byte[] dades = new byte[100];
            int llegits = 0;
            while (-1 != (llegits = in.read(dades))) {
                  out.write(dades,0,llegits);
            }
            out.close();
            in.close();
      } catch (IOException) { ... }
}
```

En operar amb fluxos, és imprescindible capturar les possibles excepcions en el procés d'entrada/sortida (IOException).

Exercici 10.3: Fer un programa que faci una còpia d'un fitxer en format binari, per exemple un document .pdf y mostri al final una estadística amb la mida del fitxer original, la mida del fitxer còpia i el número total de bytes copiats.

10.3.2 Origen i destinació en buffers de memòria

Un altre parell de classes molt útils quan es processen dades mitjançant fluxos són les relatives a orígens i destinacions de dades vinculats a buffers de memòria dinàmics: **ByteArrayInputStream** i **ByteArrayOutputStream**.

En el cas del flux d'entrada, permet llegir dades des d'un array de bytes (byte[ ]) **seqüencialment**, en lloc d'haver d'accedir per índex. Per aquest motiu, en el seu constructor cal indicar quin és l'array origen de les dades:

- **ByteArrayInputStream(byte[ ] buf)**. En el cas del flux de sortida, les dades escrites s'emmagatzemen en un bloc indeterminat de la memòria del programa, que augmenta la mida dinàmicament a mesura que s'escriuen noves dades. No hi ha cap límit excepte la memòria física de l'ordinador, si bé es recomana no usar-los per a quantitats molt grans de dades. Un cop ha finalitzat l'escriptura, és possible obtenir totes les dades emmagatzemades mitjançant el mètode específic:
    - **byte[ ] toByteArray()**. En la posició 0 de l'array hi ha el primer byte escrit en el flux, i així successivament fins a trobar el darrer escrit en la posició length - 1.

Exemple: Canviar la destinació de les dades en l'exemple anterior.

Per observar els avantatges que ofereix l'abstracció mitjançant fluxos, suposem que es vol canviar la destinació de les dades del tros de codi que serveix per copiar un fitxer i, en lloc d'un fitxer, es vol escriure sobre un **buffer de memòria dinàmic**. En aquest cas, l'única modificació en el fragment de codi seria, en crear el flux de sortida a la segona línia, simplement instanciar **ByteArrayOutputStream** en lloc de FileOutputStream. La resta del codi queda exactament igual. Passa el mateix si es canvia l'origen.

El codi següent escriuria un array de bytes en un fitxer:

```
byte[] array = ...

InputStream in = new ByteArrayInputStream(array);
OutputStream out = new FileOutputStream(novaRuta);
copiaDades(in, out);
```

El codi següent llegiria el contingut en un array de bytes:

```
InputStream in = new FileInputStream(ruta);
OutputStream out = new ByteArrayOutputStream();
copiaDades(in, out);
byte[] array = out.toByteArray();
```

Per tant, el codi del mètode **copiaDades** es manté exactament igual. Amb aquest sistema, el processament de les dades és absolutament transparent en el seu origen o destinació real. I un cop més, tot plegat gràcies a l'aplicació correcta del polimorfisme.

10.4 Fluxos orientats a caràcter

La particularitat principal dels fluxos orientats a caràcter, en contraposició amb els orientats a dades, és que els mètodes de les seves classes operen amb el tipus primitiu **char** en lloc de **byte**.

En menor mesura, al contrari que amb els bytes, hi ha caràcters amb un cert significat especial. Saber que les dades que s'estan transmetent són caràcters permet processar-les correctament i poder detectar ubicacions concretes dins els text. El cas més clar d'aquest fet és el salt de línia, que permet distingir entre línies diferents dins un text.

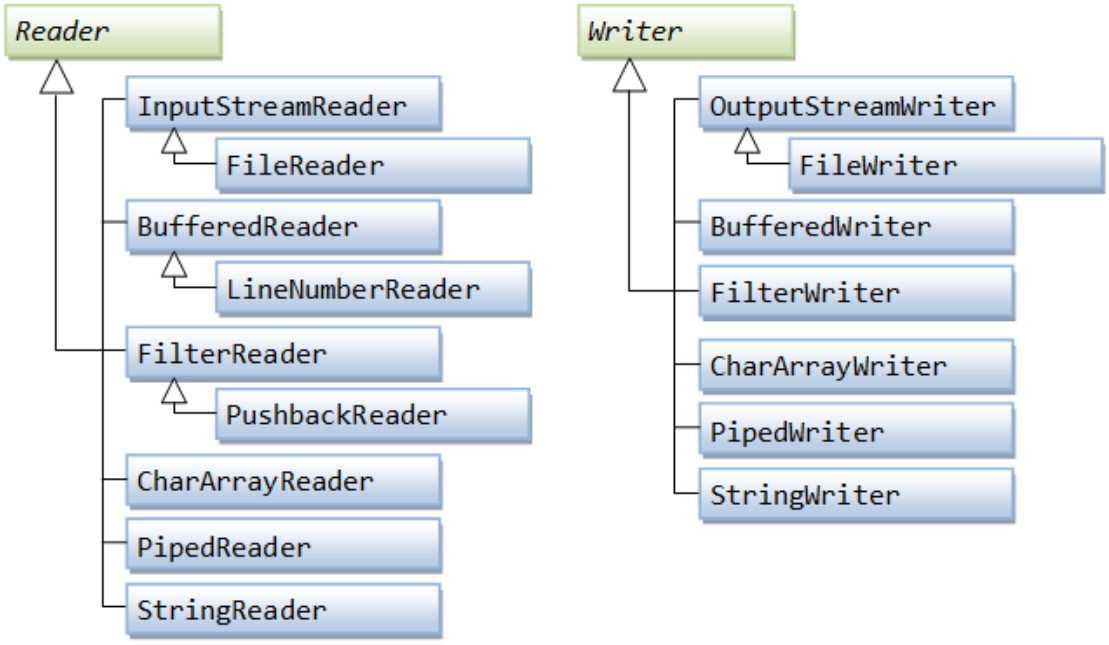
En major mesura, la diferenciació entre bytes i caràcters permet la internacionalització d'aplicacions. Tradicionalment, el sistema per codificar caràcters ha estat, i en moltes aplicacions encara ho és, el sistema **ASCII**. A pesar de l'enorme acceptació, aquest sistema té un problema molt important: es basa totalment en l'alfabet llatí i, encara més concretament, en el llenguatge anglès. Per tant, qualsevol llenguatge no representable en aquest alfabet no es pot representar en ASCII: grec, rus, pràcticament totes les llengües orientals, etc. Per resoldre aquest problema es va crear la codificació **Unicode** al final de la dècada dels vuitanta. Aquesta codificació es basa en 16 bits i és capaç d'englobar una gran quantitat d'alfabets.

Java es basa totalment en l'Unicode, els tipus primitius char ocupen 2 bytes, cosa que permet que una aplicació Java s'executi sobre qualsevol plataforma, independentment de l'idioma.

Les classes **Reader** i **Writer** representen les superclasses associades a **fluxos orientats a caràcter**. Sempre que es treballa amb caràcters cal usar la seva jerarquia de classes.

Java **Reader** és una classe abstracta per llegir fluxos de caràcters. Els únics mètodes que ha d'implementar una subclasse són read(char[ ], int, int) i close(). La majoria de subclasses, però, sobreescriuen alguns dels mètodes per proporcionar una major eficiència, funcionalitat addicional o ambdues.

Java **Writer** és una classe abstracta per escriure en fluxos de caràcters. Els mètodes que ha d'implementar una subclasse són write(char[ ], int, int), flush() i close(). La majoria de subclasses sobreescriuen alguns dels mètodes definits aquí per proporcionar una major eficiència, funcionalitat o ambdues.



La filosofia dels fluxos orientats a caràcter és exactament igual que en els fluxos orientats a dades, i només canvia tota ocurrència de byte a **char**.

Per cada origen o destinació de dades també hi ha una classe concreta, **FileReader** i **FileWriter** per processar fitxers. Fins i tot el nombre, nom i format dels mètodes són idèntics (**write**, **read**).

Exemple: Còpia de fitxers de text.

Les diferències de la còpia de fitxers de text són mínimes respecte a la manera d'operar dels fluxos orientats a dades.

```
Reader in = new Filereader(ruta);
Writer out = new FileWriter(novaRuta);
copiaDades(in, out);
...
public void copiaDades(Reader in, Writer out) {
      try {
            char[] dades = new char[100];
            int llegits = 0;
            while (-1 != (llegits = in.read(dades)))
                  out.write(dades,0,llegits);
            out.close();
            in.close();
      } catch (IOException) { ... }
}
```

De manera homònima als fluxos relatius a buffers de memòria, també hi ha classes per gestionar **buffers de caràcters**: **CharArrayReader** i **CharArrayWriter**. En aquest cas, el constructor del flux d'entrada té com a paràmetre una variable de tipus **char[ ]** i el de sortida permet obtenir les dades emmagatzemades usant el mètode **toCharArray()**.

Exercici 10.4: Implementar l'exemple de còpia d'un **fitxer de text** dins d'un altre fitxer.

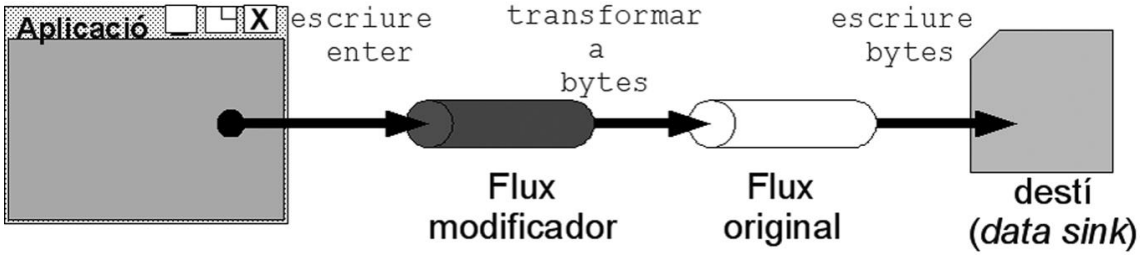
Exercici 10.5: Modificar el programa de l'exercici anterior per a que copiï el fitxer dins una ubicació dins de la memòria. Recuperar-lo després i posar-lo dins un array amb el mètode adequat i imprimir-lo per la consola.

10.5 Modificadors de fluxos

Hi ha situacions en les quals haver de processar qualsevol informació binària directament a un nivell tan baix com de byte o de caràcter pot ser una feina pesada per al desenvolupador.

Una **classe modificadora** d'un flux altera el seu funcionament per defecte, i proporciona mètodes addicionals que permeten el preprocés de dades complexes abans d'escriure o llegir-les del flux. Aquest preprocés el realitza de manera transparent el desenvolupador.

La figura mostra un esquema del comportament d'aquestes classes partint de la problemàtica exposada a l'inici. El que vol el desenvolupador és simplement poder disposar d'un mecanisme per escriure enters en el flux, i deixar-lo que s'encarregui de totes les transformacions necessàries per convertir-lo en una cadena de bytes.



Les classes modificadores més significatives són els fluxos de tipus de dades, els fluxos amb buffer intermedi, la sortida amb format, la compressió de dades, la transformació del flux orientat a caràcter de dades i la lectura per línies. En tots els casos, el seus constructors tenen com a paràmetre el flux que es vol modificar.

10.5.1 Fluxos de tipus de dades

Les classes **DataInputStream** i **DataOutputStream** proporcionen un seguit de mètodes addicionals que permeten **escriure directament tipus primitius** sense que el desenvolupador s'hagi de preocupar de com cal codificar-los en bytes:

- void writeInt(int i)
- int readInt()
- void writeBoolean(boolean b)
- boolean readBoolean()
- void writeDouble(double d)
- double readDouble()
- etc.

Recordem que la quantitat de bits amb la que Java codifica cada tipus de dada no és constant:

Tipus	Paraula clau Java	Mida (bits)	Rang
byte	byte	8	-128 a 127
enter curt	short	16	-32768 a 32767
enter simple	int	32	-2147483648 a 2147483648
enter llarg	long	64	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,808
real de simple precisió	float	32	-3.40292347*10^38 a 3.40292347*10^38
real de doble precisió	double	64	-4.94065645841246544*10^24 a 1.79769313486231570*10^308

L'objecte DataOutputStream modificarà un flux de dades preexistent. Per tant, el constructor rebrà com a paràmetre un altre objecte tipus FileOutputStream que serà el flux de dades que emmagatzemarà les dades modificades dins el fitxer al qual apunti.

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream(ruta));
```

Exemple:
Un exemple d'utilització d'aquest modificador, en què s'escriu directament un valor enter, seria:

```
DataOutputStream dos =
      new DataOutputStream(new FileOutputStream(ruta));
dos.writeInt(enter);
```

```
dos.writeBoolean(boolea1);
dos.writeBoolean(boolea2);
dos.writeDouble(doble);
dos.close();
```

En usar aquest tipus de flux cal anar amb molt de compte de llegir dades en exactament l'ordre invers en què s'han escrit, ja que en cas contrari el programa serà erroni. Així, doncs, per llegir el fitxer anterior correctament cal fer:

```
DataInputStream dis = new DataInputStream(new FileInputStream(ruta));
double d = dis.readDouble();
boolean b1 = dis.readBoolean();
boolean b2 = dis.readBoolean();
int i = dis.readInt();
dis.close();
```

10.5.2 Fluxos amb buffer intermedi

La classe BufferedInputStream proporciona la capacitat de disposar d'un buffer de memòria intermedi entre l'aplicació i un flux d'entrada orientat a dades. A efectes pràctics, això significa que permet tornar enrera en qualsevol moment de la lectura o l'escriptura, al contrari del que normalment es permet. Per assolir aquesta fita, disposa de mètodes addicionals:

- **void mark(int limit)**. Marca una posició del flux. La posició marcada es conserva sempre que no es llegeixin més de limit bytes (que correspondria a la mida del buffer intermedi). En cas que això succeeixi, la marca es perd. Si bé aquest mètode es pot cridar diverses vegades al llarg de la vida del flux, com a màxim hi pot haver una única marca vàlida. Aquesta sempre serà la corresponent a la darrera crida d'aquest mètode.
- **void reset()**. El flux retrocedeix el processament de dades de nou fins a la posició marcada. En les lectures següents, es tornaran a obtenir exactament les mateixes dades que es van obtenir tot just després de cridar mark.

Exemple: Retrocedint en la lectura d'un flux

Un tros de codi mostrant el seu funcionament seria el següent. En aquest es llegeixen els primers 200 bytes i llavors, tot seguit, es tornen a llegir. El paràmetre del mètode mark indica quina és la mida del buffer i, per tant, el límit de bytes que es poden llegir fins que la marca deixa de ser vàlida i es perd:

```
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(ruta));
byte[] dades = new int[100];
bis.mark(500); //Marca. El buffer serà de 500 bytes
bis.read(dades); //Llegim els bytes 0-99 del fitxer
bis.read(dades); //Llegim els bytes 100-199 del fitxer
bis.reset(); //Tornem a la marca
bis.read(dades); //Llegim els bytes 0-99 del fitxer
bis.read(dades); //Llegim els bytes 100-199 del fitxer
bis.read(dades); //Llegim els bytes 200-299 del fitxer
bis.read(dades); //Llegim els bytes 300-399 del fitxer
bis.read(dades); //Llegim els bytes 400-499 del fitxer
bis.read(dades); //La marca es perd. Ja no es pot fer reset()
...
bis.close();
```

També existeix una classe BufferedOutputStream, tot i que aquesta té un comportament absolutament diferent. Simplement serveix per optimitzar alguns dels processos d'escriptura de dades del sistema operatiu. A part d'això, no aporta cap altre funcionalitat en forma de nous mètodes.

10.5.3 Sortida amb format

La classe **PrintStream** és imprescindible dins de qualsevol aplicació que ha d'escriure cadenes de text dins d'un flux, ja que es tracta d'un modificador de fluxos de sortida que proporciona dos mètodes, sobrecarregats per poder tractar paràmetres de qualsevol tipus primitiu o objecte:

- **void print(…)**. Escriu la representació en forma de cadena de text del paràmetre d'entrada. Per exemple, si el paràmetre és un booleà a cert, escriu la cadena de text "true", si és el número 24, escriu la cadena de text "24", etc.
- **void println(…)**. Escriu la representació en forma de cadena de text del paràmetre d'entrada i al final fa un salt de línia.

Així, doncs, aquests mètodes transformen qualsevol cosa en una cadena de bytes d'acord amb la seva representació com a cadena de text (un objecte String). En cas que el paràmetre sigui un objecte, la transformació en cadena de text es realitza mitjançant la crida interna del mètode toString(). Atès que aquest mètode està definit en la mateixa classeObject Object, sempre es pot garantir que és possible cridar-lo.

System.out és un PrintStream. Per això per escriure línies de text per pantalla s'usa: System.out.println(…).

Tot i ser una mica estrany, en tractar-se d'un flux que transforma dades en cadenes de text, es considera orientat a dades i no a caràcter. PrintStream també és un cas especial en el fet que disposa de **constructors addicionals** en què es poden especificar directament alguns tipus de destinacions de dades.

- PrintStream(File fitxer)
- PrintStream(OutputStream out)
- PrintStream(String nomFitxer)

Exemple: Mostrant enters

En el fragment de codi que es mostra tot seguit es veu un exemple senzill de les funcionalitats de PrintStream, mitjançant el qual és possible imprimir línies de text amb un format complex:

```
PrintStream ps = new PrintStream(ruta);
for (int i = 0; i < 10; i++)
      ps.println( i + ". i val 5? " + (i == 5));
...
ps.close();
```

10.5.4 Compressió de dades

El Java incorpora dins la biblioteca de fluxos la possibilitat de transmetre i llegir dades comprimides de manera transparent. La manera més simple de fer-ho és mitjançant les classes **GzipInputStream** i **GzipOutputStream**, que usen l'algorisme de compressió GZIP. Cap de les dues classes inclou nous mètodes fora dels definits en les superclasses InputStream i OutputStream. A mesura que s'escriuen o es llegeixen dades amb els mètodes write o read, aquestes es comprimeixen automàticament sense que sigui necessari fer cap altra tasca addicional.

Al contrari que la resta de classes, aquestes pertanyen al paquet java.util.zip.

Exemple:

Tot seguit es mostra un exemple senzill dels mecanismes de compressió de dades. Com es pot veure, tot és igual a escriure o llegir dades d'un flux qualsevol.

```
try {
      GZIPOutputStream out = new GZIPOutputStream(new
FileOutputStream(ruta));
      FileInputStream in = new FileInputStream(inFilename);
      byte[] dades = new byte[1024];
      int llegits = 0;
      while (-1 != (llegits = in.read(dades)))
            out.write(dades,0,llegits);
      in.close();
      out.close();
} catch (IOException) { ... }
```

Exercici 10.6: Comprimir un fitxer qualsevol i comprovar que el resultat es pot descomprimir de la manera usual de descomprimir un fitxer .gz (amb 7zip o WinRar per exemple).

10.5.5 Traducció de flux orientat a caràcter a dades

Dins el conjunt de classes modificadores, també hi ha dues classes que permeten traduir un flux orientat a dades a un orientat a caràcter, de manera que es pot operar a nivell de char en lloc de byte. Es tracta de les classes **InputStreamReader** i **OutputStreamWriter**.

Totes aquestes classes modificadores, de fet, són subclasses de InputStream i OutputStream, ja que també es consideren fluxos per si mateixes.

Exemple: D'InputStream a Reader

A continuació es mostra com un flux orientat a dades amb origen en un fitxer es pot processar com un flux orientat a caràcter, sempre que se sàpiga a priori que el fitxer conté text.

```
FileInputStream fis = new FileInputStream (ruta);
FileOutputStream fos = new FileOutputStream (novaRuta);
...
public void copiaText(InputStream in, OutputStream out) {
      try {
            Reader rd = new InputStreamReader(in);
            Writer wr = new OutputStreamWriter(out);
            char[] dades = new char[100];
            int llegits = 0;
            while (-1 != (llegits = rd.read(dades)))
                  wr.write(dades,0,llegits);
            wr.close();
            rd.close();
      } catch (IOException) { ... }
}
```

10.5.6 Lectura per línies

La classe més útil entre els modificadors de fluxos orientats a caràcter és **BufferedReader**, que permet la lectura de línies completes de text mitjançant el mètode readLine(), que retorna directament un String. Ella sola s'encarrega de llegir automàticament tots els caràcters necessaris fins a trobar un salt de línia. Aquesta classe també es considera un Reader, perquè és una subclasse seva.

Tot seguit es mostra com es pot usar amb unes línies de codi d'exemple, en què es mostra per pantalla el contingut d'un fitxer de text, llegint-lo línia a línia:

```
BufferedReader br = new BufferedReader (new FileReader(ruta));
String linia = null;
//Es mostra tot el contingut per pantalla
while (null != (linia = br.readLine()))
      System.out.println(linia);
br.close();
```

Exercici 10.7: Llegir un fitxer de text línia a línia i imprimir-lo per consola posant-li una numeració davant. Per exemple:

Línea 1: primera línia llegida del fitxer...

Línea 2: segona línia llegida del fitxer...

...

10.6 Seriació d'objectes

Hi ha situacions en què el desenvolupador pot decidir que no vol haver de pensar quines dades concretes cal emmagatzemar dins un fitxer, o haver d'especificar quin format han de tenir i processar-les en format binari o de text. Simplement, el que vol és agafar el mapa d'objectes del Model exactament **tal com està representat en la memòria** i fer un abocament directe. Java ofereix la possibilitat de fer aquesta acció mitjançant el mecanisme de seriació d'objectes.

S'anomena **seriació d'objectes** el procés d'escriptura d'un objecte sobre una destinació de dades **en forma de cadena de bits** a partir de la seva representació en memòria, de manera que a partir de les dades resultants posteriorment es pugui **restaurar** en exactament el mateix estat.

Perquè un objecte es pugui seriar, cal que la seva classe implementi la interface Java **java.io.Serializable**. Aquesta és una interface molt especial, ja que no obliga a implementar absolutament cap mètode, només serveix per indicar que les instàncies d'una classe són seriables.

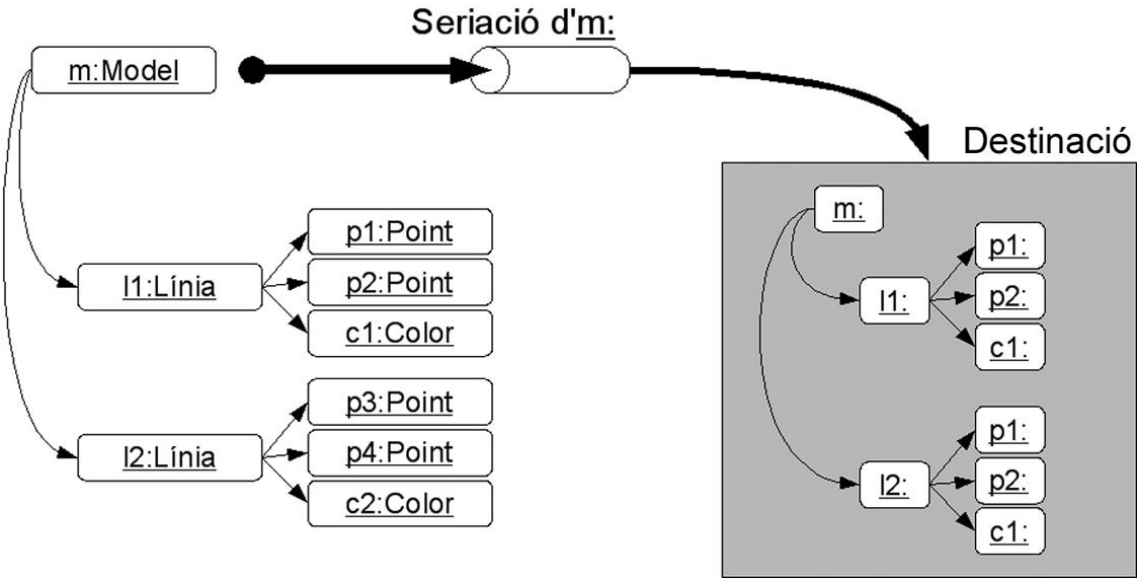
Per tant, si tenim la classe:

```
public class Model {...}
```

Per seriar-ne els objectes, l'única modificació que cal fer és:

```
import java.io.Serializable;
public class Model implements Serializable {...}
```

Una de les propietats que fa especialment potent el mecanisme de seriació d'objectes del Java és el fet que, en seriar un objecte, se segueixen totes les seves referències a altres objectes, els quals, al mateix temps, també se serien. Aquest procés es produeix iterativament en els nous objectes seriats fins que ja no es troben més referències. Per tant, és possible seriar un mapa d'objectes complet simplement indicant que cal seriar l'objecte de nivell més alt. En restaurar l'objecte seriat, amb ell és recupera el mapa d'objectes complet.



Si un mapa d'objectes es compon d'instàncies de diferents classes, com serà el cas freqüentment, cal que totes implementin Serializable. En cas contrari, en intentar seriar una instància d'una classe que no la implementa, es produirà una excepció tipus java.io.NotSerializableException.

Els tipus de fluxos de dades associats a la seriació d'objectes són **java.io.ObjectOutputStream** i **java.io.ObjectInputStream**, per llegir i escriure respectivament. Ambdues classes ofereixen un ventall de mètodes per escriure de manera seqüencial tota mena de tipus de dades (en lloc de només bytes o caràcters). Per seriar objectes, els mètodes que cal usar són:

- **public void writeObject(Object o)**. Escriu un objecte en un flux de dades ObjectOutputStream.
- **public Object readObject()**. Llegeix un objecte d'un flux de dades ObjectInputStream.

A continuació es mostra un exemple de seriació d'objectes a fitxer. La classe Model pot ser tan complexa com calgui i referenciar als seus atributs instàncies de qualsevol altra classe. Mentre aquestes també implementin Serializable, l'estructura d'objectes completa s'escriurà a disc:

```
Model m = new Model();
File f = new File(ruta);
...
ObjectOutputStream oos = new ObjectOutputStream (new
FileOutputStream(f));
oos.writeObject(m);
oos.close();
```

Tot seguit també es mostra l'exemple associat a la recuperació de l'objecte seriat anteriorment. En recuperar l'objecte m:Model serialitzat, també s'ha recuperat tot el seu mapa d'objectes subjacent:

```
File f = new File(ruta);
```

```
...
ObjectInputStream ois = new ObjectInputStream (new
FileInputStream(f));
Model m = (Model)ois.readObject();
ois.close();
```

Fixeu-vos que, com que el mètode readObject retorna un Object, cal fer un cast per assignar la instància retornada a una referència de tipus Model. Això implica que **el desenvolupador ha de saber exactament quin tipus d'objecte es va emmagatzemar**, o en cas contrari es produeix un error en fer aquest cast, una ClassCastException. Evidentment, també s'ha de complir que totes les classes seriades, els seus fitxers .class, estiguin instal·lades en l'ordinador en què s'executa aquest codi, ja que en cas contrari es produeix una ClassNotFoundException. Per tant, cal anar amb compte quan hi ha un intercanvi d'objectes seriats entre diferents màquines.

L'única excepció dins dels mecanismes de seriació per defecte de Java són els atributs estàtics (static), que no se serien.

Com es pot apreciar pels exemples, el mecanisme de seriació d'objectes és relativament fàcil d'usar, ja que Java s'encarrega automàticament de tots els detalls interns sobre la representació dels objectes seriats i la seva instanciació a memòria un cop restaurats.

Exercici 10.8: Recuperar l'exercici 6.5 on fèiem ús de persones (alumnes i professors). Crear uns quants alumnes i professors i serialitzar l'ArrayList que els conté dins uns fitxer. Recuperar-los després amb els fluxos i mètodes adients.

10.7 Accés aleatori

Una de les característiques essencials de la gestió de dades emmagatzemades dins un fitxer mitjançant fluxos és el caràcter seqüencial en les operacions tant de lectura com d'escriptura. En tractar-se d'un mecanisme genèric, es va definir el denominador comú a qualsevol operació d'entrada sortida. Tot i així, per al cas concret d'un fitxer, es pot garantir que totes les dades són en una ubicació concreta, de manera que s'hi pot accedir de manera aleatòria.

Accés aleatori es la capacitat de llegir dades en qualsevol ubicació dins una seqüència, sense haver de processar prèviament les dades anteriors. Per poder accedir de manera aleatòria a un fitxer, el Java ofereix la classe **RandomAccessFile**.

Els seus constructors són:

- RandomAccessFile(File fitxer, String mode)
- RandomAccessFile(String ruta, String mode)

Novament, el constructor està sobrecarregat per acceptar tant un objecte File com directament la ruta del fitxer per mitjà dels paràmetres **fitxer** o **ruta**. El paràmetre **mode** indica en quin mode es vol obrir el fitxer. Els diferents modes possibles són:

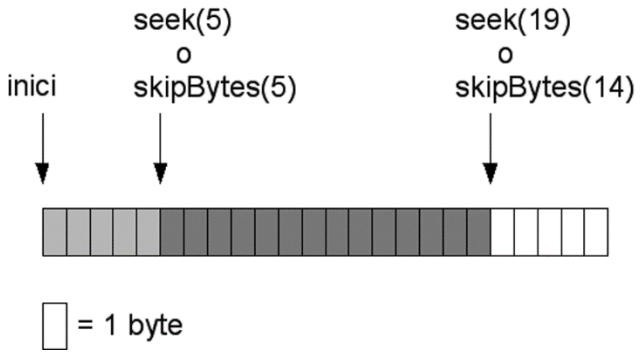
- r : Mode escriptura. Qualsevol intent d'escriure en el fitxer, incloent-hi el fet que no existeixi, causarà una excepció.
- rw: Mode escriptura-lectura. Si el fitxer no existeix, se'n crearà un de nou, buit.
- rws: Igual que el cas rw, però, addicionalment, es força l'actualització al sistema de fitxers cada cop que es fa una modificació en les dades del fitxer o les seves metadades. Aquest comportament és semblant a fer un flush cada cop que es fa una escriptura en el fitxer.

- rwd:Igual que el cas anterior, però només es força l'actualització per al cas de dades, i no metadades.

L'accés a un RandomAccessFile es basa en un apuntador intern que és possible desplaçar arbitràriament a qualsevol posició, partint del fet que la posició 0 correspon al **primer byte** del fitxer. Tots els increments en la posició d'aquest apuntador són en nombre de bytes. Per gestionar la posició d'aquest apuntador, la classe defineix amb els mètodes específics següents:

- **void seek(long pos)**. Ubica l'apuntador exactament en la posició especificada pel paràmetre pos, **en bytes** de manera que qualsevol accés a les dades serà sobre aquest byte. No hi ha cap restricció en el valor d'aquest paràmetre, i és possible ubicar l'apuntador molt més enllà del final real del fitxer. En aquest cas, la mida del fitxer es veurà incrementada fins a pos bytes en el moment en què es faci alguna escriptura.
- **long getFilePointer()**. Retorna la posició exacta de l'apuntador, en nombre de bytes, des de l'inici del fitxer.
- **int skipBytes(int n)**. Salta n bytes a partir de la posició actual de l'apuntador, de manera que aquest passa a valer (apuntador + n). Retorna el nombre real de bytes saltats, ja que si s'arriba al final del fitxer, el desplaçament de l'apuntador s'atura.
- **void setLength(long len)**. Assigna una nova longitud al fitxer. Si la nova longitud és menor que l'actual, el fitxer es trunca.

La figura mostra un esquema de posicionament de l'apuntador del fitxer d'acord amb crides successives als mètodes seek (posicionament absolut) o skipBytes (posicionament relatiu respecte al darrer valor de l'apuntador).



Un cop ubicats en una posició concreta dins el fitxer, és possible llegir o escriure dades utilitzant tot un seguit de mètodes de lectura i escriptura definits, havent-n'hi una per cada tipus primitiu (read/writeBoolean, read/writeInt, etc.). En aquest aspecte, RandomAccessFile es comporta com les classes DataInputStream i DataOutputStream i totes les consideracions esmentades per a aquestes classes també s'apliquen en el cas d'accés aleatori. El nombre de bytes escrits dependrà de la mida associada al tipus primitiu a Java.

Cada cop que es fa una operació de lectura o escriptura, l'apuntador es desplaça el mateix nombre de bytes que el nombre al qual s'ha accedit.

Exemple: Accés aleatori en un fitxer d'enters i reals

En aquest exemple es mostra com es gestiona un fitxer relativament senzill en què un cert nombre de valors són de tipus enter (valorsInt), i tot seguit hi ha un altre conjunt de valors de tipus real (valorsDouble).

```
int[] valorsInt = ...;
double[] valorsDouble = ...;
```

Per generar un fitxer amb aquests valors, n'hi ha prou d'ubicar l'apuntador en la posició inicial del fitxer i anar escrivint els valors usant el mètode writeXXX adequat.

```
RandomAccessFile file = new RandomAccessFile(ruta, "rw");
for(int i = 0; i < valorsInt.length; i++) file.writeInt(valors[i]);
for(int i = 0; i < valorsDouble.length; i++)
file.writeDouble(valors[i]);
file.close();
```

Per modificar un valor qualsevol, cal ubicar l'apuntador fins a l'inici del valor adequat. Ara bé, per a això s'ha de calcular el desplaçament correcte d'acord amb el nombre de bytes que ocupa cadascun dels valors emmagatzemats. Un int en Java ocupa 4 bytes mentre que un double n'ocupa 8.

Aquest exemple modifica el tercer real emmagatzemat en el fitxer. Per fer-ho, cal saltar els bytes associats a tots els enters i els dos primers reals.

```
double nouValorDouble = ...;
RandomAccessFile file = new RandomAccessFile(ruta, "rw");
file.seek(4*valorsInt.length + 8*2);
file.writeDouble(nouValorDouble);
file.close();
```

Per llegir valors, cal ubicar l'apuntador a l'inici de cada un i anar fent crides al mètode readXXX associat al tipus primitiu esperat. Novament, si es volen llegir posicions no consecutives, cal anar recalculant els desplaçaments correctes dins el fitxer. En aquest tros de codi es mostren els primers quatre valors per cada cas.

Tal com es desprèn dels exemples, un dels aspectes amb què cal anar amb més cura en usar l'accés aleatori és el fet que el posicionament de l'apuntador dins el fitxer es realitza comptant en nombre de bytes, però totes les escriptures i lectures es realitzen directament en tipus primitius. Això implica que el desenvolupador que està generant codi, per accedir a un fitxer ha de saber exactament la seva estructura interna i recordar la mida exacta de cada tipus primitiu de Java, per poder fer els salts a les posicions exactes en què comença cada dada emmagatzemada. En cas contrari, si es comet un error es llegiran o se sobreescriuran parcialment dades incorrectes.

Exercici 10.9: Crear un nou fitxer i escriviu unes quantes línies dins ell. Després recuperau el caràcter ubicat en una posició qualsevol (per exemple a la posició 50)

10.8 Lectura d'un arxiu d'un servidor d'Internet

Per llegir un fitxer d'internet es pot emprar com a flux d'entrada un objecte de la classe URL. La resta es pot tractar com ja hem vist.

```
URL url = new URL("http://www.iesjoanramis.org");
try {
      // Volcam el rebut al buffer
      in = new BufferedReader(new
InputStreamReader(url.openStream()));
} catch(Throwable t){}
      // Mentre no acabi de llegir la página s'adjunta a un String
while ((inputLine = in.readLine()) != null) {
      inputText = inputText + inputLine;
}
```

Exercici 10.10: Recuperau una pàgina web qualsevol i emmagatzemar-la dins un fitxer del vostre disc dur. Comprovar després que s'ha emmagatzemat bé i que la podeu veure utilitzant un navegador qualsevol.

Exercici 10.11: Crear un programa que generi un array amb 1000 números aleatoris entre 1 i 1000. A partir d'aquest array, el programa haurà de llegir un a un els números de l'array i separar-los en dos fitxers de text, perquè es puguin veure amb un editor de text normal: un contindrà tots els números parells de l'array i l'altre tots els números imparells.

Exercici 10.12: Modificar el programa Mastermind per a que emmagatzemi les partides i totes les seves dades serialitzades dins un fitxer. Aquest fitxer s'ha de llegir cada vegada que arranca el programa i recuperar totes les partides jugades els dies anteriors.

Exercicis extra de repàs i consolidació de conceptes:

Exercici E1: Fer un programa que creï un directori buit i copiï diversos documents dins ell.

Exercici E2: Fer un programa que llegeixi el contingut d'un directori concret i ens torni la següent informació:

- Número de subdirectoris que hi ha.
- Quants de fitxers de text hi ha.
- Quants fitxers executables hi ha.
- Quina mida ocupa el directori.

Exercici E3: Fer un programa que compti el número de dígits que hi conté un fitxer concret proporcionat per l'usuari.

Exercici E4: Fer un programa que analitzi el percentatge de cada caràcter que apareix dins un fitxer de text donat

Exercici E5: Pràctica per examen: Donat el fitxer de temperatures "temperatures.txt", on trobam, en format text, una línia amb els noms dels mesos i una quantitat indeterminada de files amb les temperatures mitjanes de cada mes separat per comes (","), llegir i tractar totes les dades i emmagatzemar-les en una estructura de tipus HashMap<String, Double> on la clau serà el nom del mes i el valor la temperatura mitjana de cada mes calculada tenint en compte tots els anys (suma del mateix mes de cada fila i dividit per la quantitat de files).

Al final del programa imprimir el nom de cada mes i devora la temperatura mitjana.

Pista: Per separar la informació de cada fila podeu emprar el mètode .split, de manera que cada temperatura de cada mes quedi separada per exemple dins un array.

```
Enero,Febrero,Marzo,Abril,Mayo,Junio,Julio,Agosto,Septiembre,Octubre,Noviembre,Diciembre

2.5,3.5,8,12.5,18,22,25.5,25,20.5,15,9,4

2.7,3.8,8.2,12.8,18.3,22.3,25.7,25.2,20.7,15.2,9.2,4.2

2.6,3.6,8.1,12.6,18.1,22.1,25.6,25.1,20.6,15.1,9.1,4.1

2.8,3.9,8.3,12.9,18.4,22.4,25.8,25.3,20.8,15.3,9.3,4.3

2.7,3.7,8.2,12.7,18.2,22.2,25.7,25.2,20.7,15.2,9.2,4.2

...
```

Enero: 2,8
Febrero: 3,8
Marzo: 8,3
Abril: 12,8
Mayo: 18,3
Junio: 22,3
Julio: 25,8
Agosto: 25,3
Septiembre: 20,8
Octubre: 15,3
Noviembre: 9,3
Diciembre: 4,3