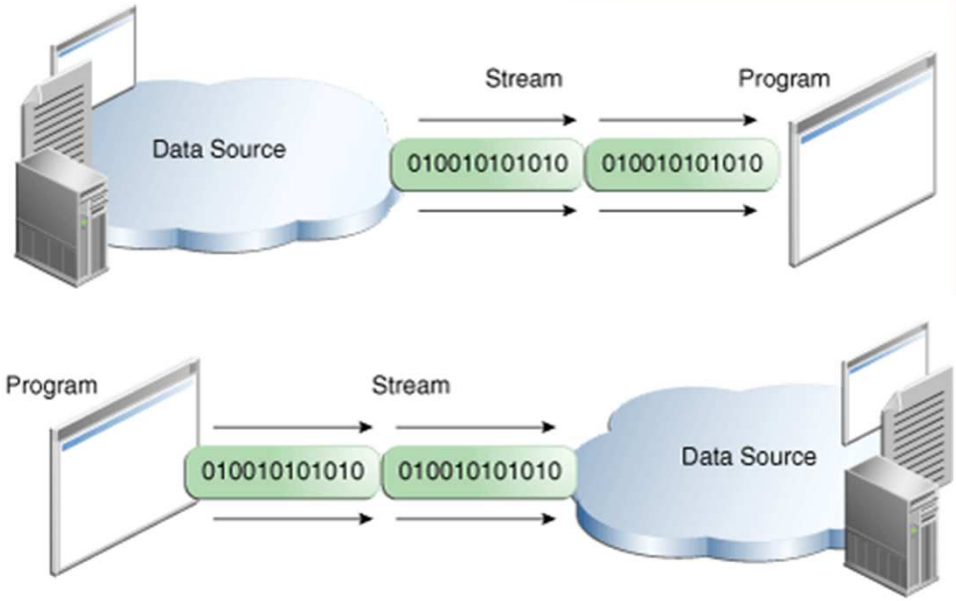
Programació

Lectura i escriptura d'informació.

Fitxers (Streams o fluxos).

Flux (Stream)

- Un flux de dades és una seqüència de dades. Un programa utilitzarà un flux d'entrada, *input stream*, per a llegir dades d'un origen, i un flux de sortida, *output stream*, per a escriure dades a un destí.
- Les operacions tant amb un com amb l'altre es realitzen amb un element a la vegada.



- L'origen o el destí de les dades poden ser de diversos tipus: fitxers de disc, dispositius, altres programes, arrays de memòria, ... qualsevol cosa que pugui mantenir, generar o consumir dades.
- Els fluxos poden manejar qualsevol tipus de dades, des de byte, fins objectes passant per tipus primitius.

Byte streams

- Els nostres programes utilitzaran els fluxos de bytes quan volguem accedir a recursos a baix nivell. No els hauríem d'utilitzar a no ser que sigui imprescindible, ja que hi ha classes molt més especialitzades.
- Els veurem perquè són la base, la superclasse, de tots els altres fluxos. Utilitzarem FileInputStream per a llegir un fitxer de text i mostrar-lo per pantalla. El constructor espera la ruta del fitxer com a paràmetre:

```
in = new FileInputStream(origen);
```

- Per a llegir utilitzarem *read*. Torna un *int* i no un *byte* perquè així pot tornar el valor -1 per indicar que ha acabat de llegir el fitxer. Per tant per llegir tots els bytes del fitxer:

```
while ((c = in.read()) != -1)
```

- Els streams sempre s'han de tancar, millor dins un bloc finally per assegurar-nos. Deixar un recurs obert, com ara un fitxer, pot provocar problemes, per exemple per extreure un USB:

```
 } finally {
      if (in != null) {
            in.close();
       }
```

- Per escriure un fluxe de bytes podem utilitzar *FileOutputStream*. El constructor també espera un *String* amb la ruta del fitxer on ha d'escriure les dades.
- El mètode a utilitzar és *write*. Té com a paràmetre un int.

```
out.write(b);
```

- Aquest mètode està sobrecarregat, per exemple amb un array de bytes com a paràmetre.
- Les classes que defineixen els streams de bytes són abstractes, InputStream i OutputStream. Algunes subclasses són:

▪FileInputStream i FileOutputStream: Per a treballar amb fitxers.

▪ByteArrayInputStream i ByteArrayOutputStream: per a treballar amb arrays com a font o destinació de dades.

▪AudioInputStream: Especialitzada en fonts d'audio.

▪ObjectInputStream i ObjectOutputStream: Per treballar amb objectes.

Parèntesi: try-with-resources

- Aquesta variant del *try* declara recursos, objectes que el programa ha de tancar quan ja no els necessiti més. S'assegura que els objectes declarats es tanquin, tant si el bloc de codi del *try* llença excepcions com si no.
- Es poden utilitzar com a recursos qualsevol objecte que implementi la interfície *AutoCloseable*. Disponible a partir del Java 7.

```
try (FileInputStream in = new FileInputStream(origen)) {
    int c;
    while ((c = in.read()) != -1) {
        System.out.print(c);
    }
}
```

- Evidentment, hi poden haver tants blocs *catch* com ens facin falta. Abans d'executar-se, però, es tancaran els recursos declarats al *try*.

Character streams

- Java emmagatzema els caràcters en Unicode. Els fluxos de caràcters automàticament els transformen a i des de el conjunt de caràcters locals del sistema. Eviten els problemes que apareixen amb els fluxos de bytes quan trobam lletres accentuades o caràcters especials.
- Totes les classes que implementen fluxos de caràcters descendeixen de *Reader* i *Writer*. Per llegir fitxers de caràcters disposam de *FileReader* i *FileWriter*. Els constructors accepten la ruta del fitxer que llegiran o on escriuran.
- El mètode *read* de *FileReader* llegeix un caràcter de l'stream. Si no n'hi ha cap de disponible bloca el programa. Podem comprovar l'estat del fluxe amb el mètode *ready*.
- El mètode *write* de *FileWriter* escriu un caràcter al fitxer.
- Tant un com l'altre utilitzen *int* per manejar les dades llegides o escrites al fitxer.

Buffered streams

- Els fluxos que hem vist fins ara no disposen de cap *buffer*. Això vol dir que cada petició de lectura o escriptura s'envia directament al SO, amb la conseqüent pèrdua d'eficiència, ja que normalment impliquen operacions de disc o xarxa que són molt costoses en temps. Per millorar el rendiment, Java implementa fluxos amb *buffer*, amb una memòria intermèdia.
- Durant la lectura, aquesta memòria intermèdia s'omple de cop i l'stream accedeix a les dades des de la memòria; en ser buida es torna a omplir. Durant l'escriptura, les dades s'escriuen a aquesta memòria i en ser plena s'envien al disc o a la xarxa.
- Disposam de 4 buffered streams: : *BufferedInputStream* i *BufferedOutputStream* per byte streams, i *BufferedReader* i *BufferedWriter* per streams de caràcter.
- Per a crear un buffered stream, al seu constructor li hem de passar un flux sense buffer:
  - `inputStream = new BufferedReader(new FileReader("xanadu.txt"));`
  - `outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));`
- Si volem escriure les dades al disc directament sense esperar a omplir el buffer podem executar el mètode *flush*.
- *BufferedReader* i *BufferedWriter* ens proporcionen operacions sobre línies de text. Per treballar amb fitxers de text són més eficients que *FileReader* i *FileWriter*.

Data streams

- Els fluxos de dades soporten valors de tipus de dades primitius (boolean, char, byte, short, int, long, float, and double) i String. Tots els fluxos de dades implementen o bé la interfície *DataInput* o bé la *DataOutput*. Com a exemple veurem els dos més utilitzats, *DataInputStream* i *DataOutputStream*. La primera proporciona mètodes per a llegir dades del tipus *readBoolean*, *readChar*, *readInt*, *readUTF*(per String), ... i la segona *writeBoolean*, ...
  - out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(dataFile)));
  - in = new DataInputStream(new BufferedInputStream(new FileInputStream(dataFile)));
- *DataOutputStream* guarda les dades com una successió de bytes. *DataInputStream* llegeix aquests bytes i els transforma a un determinat tipus segons el mètode que hem utilitzat. Per tant, **per llegir correctament un fitxer hem de saber en quin ordre s'han escrit les dades.** L'única manera que tenim de saber quan acaba un fitxer és recollir l'excepció *EOFException*.

Escriptura	Lectura

for (int i = 0; i < prices.length; i ++) {
  out.writeDouble(prices[i]);
  out.writeInt(units[i]);
  out.writeUTF(descs[i]);
}
	try {
	  while (true) {
	  price = in.readDouble();
	  unit = in.readInt();
	  desc = in.readUTF();
	  System.out.format("You ordered %d" + " units of %s at $%.2fn", unit, desc, price);
	  total += unit * price;
	  }
	} catch (EOFException e) {
	}

Object streams

- Els fluxos d'objectes soporten objectes, és a dir en lloc de guardar a disc un *float* o un *String*, podem guardar directament un *Alumne* o un *Banc*.
- Per poder guardar un objecte a un stream, la seva classe ha d'implementar la interfície *Serializable*. Es tracta d'una *interfície marcadora*, és a dir, sense cap mètode, que només serveix per indicar que volem permetre que els objectes de la classe s'enviïn a un stream.
- Utilitzarem els streams *ObjectInputStream* i *ObjectOutputStream* per a llegir i escriure objectes al flux.

```
out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(dataFile)));
in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(dataFile)));
```

- Per enviar un objecte al flux: *out.writeObject(alumne);*
- Per recuperar un objecte del flux: *alumne=(Alumne) in.readObject();*
- Si l'objecte recuperat no és del tipus esperat es generarà una excepció del tipus *ClassNotFoundException*.
- Aquest streams no donen cap senyal d'haver arribat al final de fitxe, *EOF*. El més habitual és posar le *while* que llegeix a *true* i capturar *EOFException*. Una altra possibilitat és guardar qualque tipus de metadata, un objecte inicial que indiqui quans n'hi ha o situar un objecte al final que poguem reconèixer com a final de fitxer.

- **Escriptura i lectura d'objectes complexos**
- Quan tots els atributs d'un objecte són tipus primitius és realment simple escriure aquest objecte a un stream. Però moltes vegades ens trobem que un objecte té un atribut que és un altre objecte. I aquest, en pot contenir un altre, i... Per exemple, un alumne pot incloure una llista amb les seves qualificacions.
- En aquest cas, en recuperar l'objecte amb readObject, també haurem de recuperar tots aquests altres objectes que conté. En recuperar l'alumne, també haurà de recuperar la llista amb totes les seves qualificacions. Això vol dir que el mètode *writeObject* ha de ser capaç de recórrer tota aquesta estructura i guardar-la de manera que el *readObject* la pugui reconstruir.
- Si dos objectes fan referència a un altre, per exemple dos alumnes fan referència al mateix mòdul, en reconstruir-los es seguiran mantenint aquestes referències, no es crearan dos mòduls diferents.
- Si escrivim dues vegades el mateix objecte al mateix stream estam guardant l'objecte una vegada. El que repetim són les referències, de forma que en tornar a recuperar les dades de l'stream, seguirem tenint un sol objecte amb dues referències. En canvi, si escrivim el mateix objecte a dos streams diferents, i recuperem l'objecte dels dos streams, tindrem dos objectes diferents.

- Exemples d'ús de fitxers de referencia:

https://jarroba.com/lectura-escritura-ficheros-java-ejemplos/

http://chuwiki.chuidiang.org/index.php?title=Lectura_y_Escritura_de_Ficheros_en_Java