
Accés a Dades

Tema 2: Gestió del contingut de fitxers



Objectius



Objectius

L'objectiu d'aquest tema serà accedir al contingut dels fitxers, bé per a llegir únicament o bé per a guardar informació de forma permanent.

Per a açò en Java disposem de les classes de **flux de dades (streams)**. Lamentablement Java disposa d'una quantitat de streams extraordinàriament gran, cosa que suposa una gran complexitat per al programador (ja que ha de recordar moltes classes). Per contra ofereix una potencialitat molt gran.

Donada aquesta varietat de classes i la complexitat inherent, aquest tema començarà el tractament del contingut dels fitxers separant les classes en 2 categories: considerant la informació com un conjunt de bytes, i considerant-la com un conjunt de caràcters.

En el següent tema es continuarà, veient formats especials de fitxers i també diferents maneres d'accedir.

Nota

Per als exemples i exercicis d'aquest tema, heu de crear un projecte nou, **Tema2**.

Haureu de crear 2 paquets, **Exemples** i **Exercicis**, de forma similar a com havíem fet en el Tema 1.

1 - Magatzems i fluxos de dades

En la major part de programes necessitem guardar informació de forma permanent, de manera que perduren, encara que finalitze l'execució del programa, o que serveixen de dades inicials. Els fitxers ens proporcionen la forma més senzilla de guardar informació.

En el tema anterior vam veure com poder accedir tant a un directori com a un fitxer, però no vam accedir al contingut d'aquests. Dels fitxers només podíem veure les seues característiques externes: nom, tipus, permisos, grandària, ... Però en cap moment vam accedir al seu contingut. Serà el que veurem en aquest tema.

En un fitxer, com dèiem, quedarà guardada la informació de forma permanent. Serà una **seqüència de bits**, un darrere de l'altre que representaran les dades guardades, bé siguin caràcters d'un text, dades numèriques, o els bytes d'una imatge, per exemple. Aquesta seqüència de bits ens aporta una visió estàtica de les dades, ja que queden guardades al llarg del temps.

Posant un exemple clàssic, els pantans i dipòsits on s'emmagatzema l'aigua serien comparables als fitxers. Però per a nosaltres aquests pantans i dipòsits queden lluny. I quan parlem d'aigua corrent tenim més tendència a pensar en les canonades i aixetes que ens porten aquesta aigua emmagatzemada fins a nosaltres. Doncs de forma similar, des del punt de vista de l'aplicació, el que realment cobra importància és la transferència de dades, més que el magatzem, que arriben aquestes dades a l'aplicació o que l'aplicació les pugui transferir fins al fitxer. L'eina que ens permet controlar aquestes transferències, de forma similar a les aixetes i canonades, l'anomenem **flux de dades**. És un concepte associat a la transmissió seqüencial d'una sèrie de dades des de l'aplicació al dispositiu d'emmagatzematge o a l'inrevés. Ens dona una visió eminentment dinàmica de la informació.

Java utilitza els **streams** (fluxos de dades) per a poder accedir a la informació. Però els *streams* no limiten la transferència de dades d'un fitxer, sinó ho generalitza per a qualsevol font de dades: memòria, xarxa, fins i tot altres aplicacions. D'aquesta manera es generalitza l'accés a la informació des de qualsevol procedència: si connectem un stream a un fitxer, estarem accedint a un fitxer, però si connectem el stream a un altre programa estarem accedint a les dades proporcionades per un altre programa. Intentarem veure exemples d'accés a diferents fonts a través d'un stream, però ho aplicarem sobretot a l'accés a fitxers, clar.

Fluxos d'entrada i d'eixida

La primera diferenciació que farem en els fluxos de dades és si són d'entrada o d'eixida.

- **Fluxos d'entrada** són aquells que serviran per introduir dades des de l'exterior al programa, és a dir a la zona de memòria controlada pel programa (variables, ...)
- **Fluxos d'eixida** són aquells que serviran per a guardar les dades des de les variables del programa fins a l'exterior, per exemple un fitxer, per a que es guarden de forma permanent.

Fluxos i tipus de dades

Per mig del stream aconseguirem que una dada es guardi en un fitxer (ja hem vist que pot servir per a dur-la a uns altres llocs, però nosaltres ens centrarem sobretot en fitxers), o millor dit una sèrie de dades. Quan guardem moltes dades, es compactaran unes al costat de les altres (el que havíem comentat com a seqüenciació de bits). Si intentem recuperar-les, haurem d'anar amb molt de compte amb la grandària de cadascuna de les dades i el seu tipus. Anem a posar un exemple:

- Suposem que volem guardar una dada numèrica en un enter (**int**). Els enters, en Java, es guarden en 32 bits. Si volem guardar el número **1.213.156.417**, ens quedarà en binari (els hem posat en grups de 8 bits, per facilitar la lectura):

```
01001000 01001111 01001100 01000001
```

- Suposem ara que volem guardar dos números enters, però del tipus **short**, que només ocupa 16 bits. El número

18.511 es representa en binari com 01001000 01001111, i el número 19.521 es representa 01001100 01000001. Si posem una dada darrere de l'altra (com es guardarà en un fitxer), el resultat serà:

```
01001000 01001111 01001100 01000001
```

- Suposem ara que volem guardar la paraula HOLA. Si guardem el codi ASCII de cada lletra tindrem: **H** (01001000), **O** (01001111), **L** (01001100) i **A** (01000001)

```
01001000 01001111 01001100 01000001
```

En resum, les 3 informacions (el número de 32 bits, els 2 números de 16 bits, i la paraula HOLA) es guarden de forma idèntica, com a seqüència de bits.

Per tant, l'única manera de poder recuperar la informació és saber de quin tipus és i la mida, a banda de l'ordre com està guardada, clar. Els fluxos de dades de Java transfereixen les dades de manera transparent al programador. No cal indicar la quantitat de bits que cal transferir, sinó que es dedueix a partir del tipus de dada que la variable representa. Però sempre haurem de tenir present el tipus de dades i l'ordre.

Hi ha, però, una excepció amb el tipus *char*. La multitud d'estàndards de codificació de caràcters existents en l'actualitat i la diversitat de formats utilitzats a l'hora d'implementar les codificacions, usant segons el cas 8, 16, 32 bits o fins i tot una longitud variable en funció del caràcter a representar, fan que siga molt difícil tractar aquest tipus de dades com una simple seqüència de bytes.

Internament, Java representa el tipus caràcter amb una codificació UNICODE de 16 bits (UTF-16) per tal de suportar múltiples alfabetes a banda de l'occidental. Tot i així, és capaç de gestionar fonts de dades (fitxers entre d'altres) de diverses codificacions (ASCII, ISO-8859, UTF-8, UTF-16...). En funció de la codificació triada, el número de bits utilitzats en l'emmagatzematge variarà. Es fa necessari, doncs, un tractament especial a l'hora de manipular aquestes dades. Com veurem, Java disposa d'una jerarquia específica de classes orientades a fluxos de caràcters per tal de fer aquests canvis i transformacions totalment transparents al programador.

2 - Manipulació dels fluxos de dades

En Java no tindrem una única classe per a manipular els fluxos de dades i així arribar al contingut dels fitxers. És una cosa que de vegades se li critica a Java, que hi ha una jerarquia molt extensa de fluxos, i són moltes classes a recordar i utilitzar. Per contra fa que siga molt versàtil.

Aquestes classes es trobaran en dues jerarquies, la dels **fluxos orientats a bytes** i la dels **fluxos orientats a caràcters**.

- Si les nostres dades són numèriques o de qualsevol altre tipus que puguem imaginar (imatges per exemple), ens convindran les primeres.
- Si la informació és de caràcters, haurem d'utilitzar la segona jerarquia.

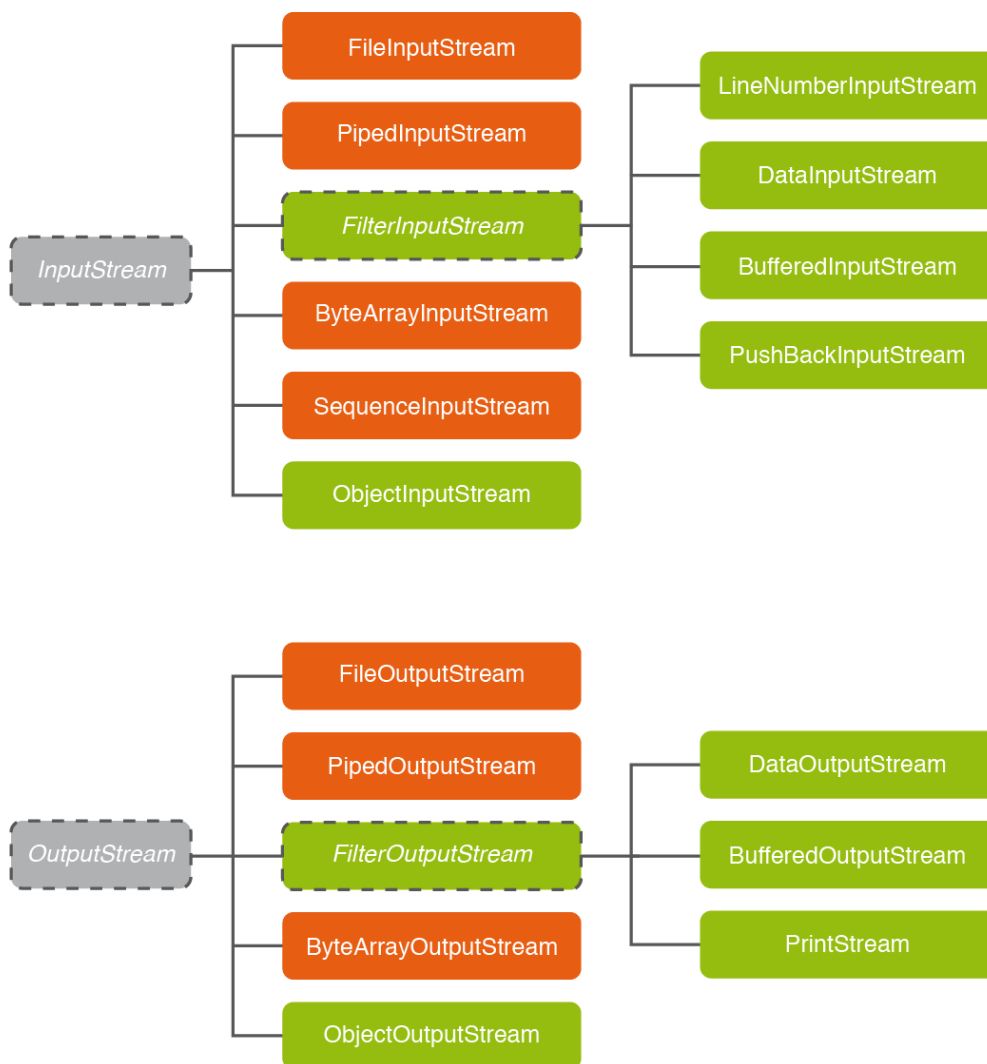
La raó de que existesca aquesta segona jerarquia orientada a caràcters és la multitud de sistemes de codificació existents. Com havíem comentat en la pregunta anterior, Java utilitza internament codificació UNICODE de 16 bits (UTF-16), on cada caràcter ocupa 16 bits i així poder suportar tots els llenguatges com el grec, àrab, ciríl·lic, xinès, Però UTF-8 està molt estés, i en aquesta codificació de vegades un caràcter ocupa 8 bits, i de vegades 16. I no podem oblidar altres sistemes de codificació, com ASCII, ISO-8859, ... La jerarquia de classes orientades a caràcter suportarà totes les codificacions.

2.1 - Fluxos orientats a bytes

L'arrel, la base de tota la jerarquia són **InputStream** i **OutputStream**, respectivament per a fluxos d'entrada i d'eixida. Comentarem els fluxos d'entrada, i els d'eixida són totalment paral·lels.

La super-classe **InputStream** servirà per a fer l'entrada des de qualsevol dispositiu: fitxer, array de bytes, una tuberia (per a dur dades des d'una altra aplicació)... Totes les classes d'entrada heretaran d'ella, i serviran per especificar exactament d'on (per exemple un fitxer: **FileInputStream**) o per a donar alguna altra funcionalitat, com anirem veient a poc a poc. D'aquesta manera, els mètodes que es defineixen s'hauran d'implementar per les classes que hereten de la super-classe i assegura una uniformitat, siga quina siga la font.

Fem una ullada ràpida a la jerarquia de classes en la següent imatge:



De moment mirem únicament les que estan en color **taronja**, que especificaran quina serà la font de dades:

Classe	Explicació
FileInputStream	Per a llegir informació d'un fitxer
PipedInputStream	Per a llegir des d'una tuberia (és a dir informació que ve d'un altre programa)
ByteArrayInputStream	L'entrada serà un array de bytes

SequenceInputStream	Servirà per enllaçar dues entrades en una de sola, seqüencialment
----------------------------	---

Evidentment ens centrarem en la primera, que és la que més ens interessa per a la permanència de les dades, però posarem algun exemple de les altres (concretament **ByteArrayInputStream**).

Els fluxos d'eixida són molt molt pareguts, tots ells heretaran de **OutputStream**:

Classe	Explicació
FileOutputStream	Per a guardar informació en un fitxer
PipedOutputStream	Per a traure cap a una tuberia (és a dir informació que anirà a un altre programa)
ByteArrayOutputStream	L'eixida serà un array de bytes

Constructors de FileInputStream

Com hem comentat, qui més ens interessa de tots els InputStream és el **FileInputStream**, per a poder accedir a la informació d'un fitxer. Dos són els constructors de FileInputStream:

- **FileInputStream (File f)**: en el paràmetre se li passa un File (dels vistos en el tema anterior), que ha de ser una referència al fitxer.
- **FileInputStream (String nom_f)**: en el paràmetre se li passa un String amb el nom (i la possible ruta) del fitxer. Ens permetrà fer referència al fitxer de forma més ràpida, sense haver de passar per un File.

Constructors del FileOutputStream

Canviaran lleugerament respecte als d'entrada, ja que a més de fer referència al fitxer, opcionalment podrem d'especificar la manera d'escriure en el fitxer en cas que aquest ja existesca: bé afegint al final, o bé destruint la informació anterior. Aquests són els constructors:

- **FileOutputStream (File f)**: en el paràmetre se li passa un File. Si no existia, el crearà; si ja existia esborrarà el contingut. En ambdós casos l'obrirà en mode escriptura.
- **FileOutputStream (String nom_f)**: igual que en l'anterior, però en el paràmetre se li passa un String amb el nom (i la possible ruta) del fitxer.
- **FileOutputStream (File f, boolean afegir)**: és com el primer, però si en el segon paràmetre se li passa **true**, en cas que ja existira el fitxer, la informació s'afegirà al final, en compte de substituir el que ja hi havia. Si en aquest paràmetre se li passa **false** s'esborrarà el contingut anterior (com en el primer cas).
- **FileOutputStream (String nom_f, boolean afegir)**: igual que en l'anterior, però en el primer paràmetre se li passa un String amb el nom (i la possible ruta) del fitxer.

2.1.1 - Mètodes del InputStream

Nota

Per a fer els primers exemples, i únicament per comoditat, utilitzarem fitxers de text, encara que estiguem en fluxos orientats a byte. Açò no és l'adequat, ja que per a fitxer de text hauríem d'utilitzar fluxos orientats a caràcter. Però com dic és per comoditat, perquè serà molt fàcil crear fitxers des de qualsevol editor de textos, i que després utilitzarem des de Java. L'inconvenient serà que no tots els caràcters eixiran de forma correcta, justament per utilitzar els fluxos de dades orientats a byte.

El primer mètode que hem de veure del **InputStream** és aquell que ens permet una lectura senzilla:

- **int read():** llig el següent byte del flux d'entrada i el retorna com un **enter**. Si no hi ha cap byte disponible perquè s'ha arribat al final de la seqüència de bytes, es retornarà -1. Si no es pot llegir el següent byte per alguna causa (per exemple si després d'arribar al final intentem llegir un altre byte, o perquè es produeix un error en llegir l'entrada) es llançarà una excepció del tipus **IOException**. Es tracta d'un mètode abstracte, que les classes específiques sobreescriran adaptant-lo a una font de dades concreta (un fitxer, un array de bytes, ...). I observeu com es tracta d'una lectura seqüencial. Comencem pel primer byte del fitxer, i a cada **read** llig el següent byte fins arribar al final. Els tractaments que veurem en aquest tema seran sempre seqüencials.

Abans de veure altres mètodes, mirem un exemple. Per a aquest exemple fa falta un fitxer anomenat **f1.txt**, que pot ser un fitxer de text creat amb qualsevol editor senzillet, com per exemple **gedit** o el **Bloc de notes**. Ha d'estar en el directori del projecte (el projecte **Tema2**), i així no caldrà posar la ruta. Per exemple podríem posar el següent contingut:

Hola, què tal?

El que farà el programa és traure per pantalla caràcter a caràcter (en línies diferents). Recordeu copiar l'exemple en un paquet anomenat **Exemples** en el projecte del Tema 2:

```
import java.io.FileInputStream;
import java.io.IOException;

public class Exemple_2_01 {
    public static void main(String[] args) throws IOException {

        FileInputStream f_in = new FileInputStream("f1.txt");
        int c = f_in.read();
        while (c!=-1) {
            System.out.println((char)c);
            c=f_in.read();
        }
        f_in.close();
    }
}
```

El resultat en Ubuntu serà aquest:

```
H
o
l
a
,
q
u
è
t
a
l
?
```

Potser en Windows si que apareguen bé tots els caràcters, ja que utilitza per defecte una altra codificació. Però no li donarem ara importància al fet que no apareguen bé els caràcters especials. Observeu com estem utilitzant un

InputStream, concretament un **FileInputStream**, per a llegir un fitxer de text. Açò no és el més apropiat, com ja havíem comentat abans, sinó que hauríem d'utilitzar algun flux orientat a caràcters, i no orientat a bytes. El programa funcionarà si utilitzem codificació ASCII (o ISO-8859) ja que cada caràcter es guarda en un byte. Si ens despistem i el fitxer el guardem en UTF-8, no eixiran bé els caràcters com ç, ñ o vocals accentuades (que es guarden en 2 bytes). I si el guardem en UTF-16, encara eixirà pitjor.

Hem utilitzat el constructor que accepta un String com a paràmetre. Queda més curt, però seria totalment equivalent substituir la construcció anterior per aquestes dues línies

```
File f = new File("f1.txt");
FileInputStream f_in = new FileInputStream(f);
```

Observeu també com obligatòriament hem de controlar l'excepció **IOException**, en aquest cas a través de **throws** (per a no gestionar-la). El **read** obté un enter, que després l'intentem convertir en caràcter. Finalitzem quan l'enter és -1.

Aquest segon exemple té l'entrada no des d'un fitxer, sinó des d'un **ByteArrayInputStream**. A banda de que l'hem d'inicialitzar diferent, podem observar com el tractament posterior és idèntic:

```
import java.io.ByteArrayInputStream;
import java.io.IOException;

public class Exemple_2_02 {
    public static void main(String[] args) throws IOException {

        String ent_1 = "Aquest és un byte array";
        byte[] ent = ent_1.getBytes();
        ByteArrayInputStream f_in = new ByteArrayInputStream(ent);
        int c = f_in.read();
        while (c != -1) {
            System.out.println((char) c);
            c = f_in.read();
        }
        f_in.close();
    }
}
```

Una altra vegada els caràcters especial eixiran malament, ja que en compte de un **InputStream** (en aquest cas **ByteArrayInputStream**) el més adequat seria un flux orientat a caràcters, però com a exemple sí que ens val.

Mirem un tercer exemple, per veure el **SequenceInputStream**, on es poden enganxar de forma seqüencial diferents **InputStream**. Després d'aquest exemple ja ens centrarem en els fitxers, que és el que ens interessa.

```
import java.io.ByteArrayInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.SequenceInputStream;

public class Exemple_2_03 {
    public static void main(String[] args) throws IOException {

        FileInputStream f1 = new FileInputStream("f1.txt");
        String ent_1 = "Aquest és un byte array";
        ByteArrayInputStream f2 = new ByteArrayInputStream(ent_1.getBytes());
        SequenceInputStream f_in = new SequenceInputStream(f1, f2);
        int c = f_in.read();
        while (c != -1) {
            System.out.println((char) c);
            c = f_in.read();
        }
        f_in.close();
        f1.close();
        f2.close();
    }
}
```

Altres mètodes del **InputStream** són:

- **int read(byte[] buffer)**: llig un número determinat de bytes de l'entrada, guardant-los en el paràmetre (que actuarà com un buffer). El número de bytes llegits serà com a màxim la grandària del buffer, encara que podria ser menor (si no hi ha prou bytes, per exemple). El mètode tornarà el número de bytes que realment s'han llegit com un enter. Si no hi haguera cap byte disponible, es retornarà -1.

- `int available()`: indica quants bytes hi ha disponibles per a la lectura. Sobretot serviria com a condició de final de bucle (si hi ha 0 bytes disponibles, és que ja hem acabat), encara que hi ha altres maneres de fer la condició de final de bucle.
- `long skip(long desp)`: salta, despreciant-los, tant bytes com indica el paràmetre. Podria ser que no poguera saltar el número de bytes especificat per diferents raons. Torna el número de bytes realment saltats.
- `int close()`: tanca el flux de dades.

Mirem un altre exemple, utilitzant ara el buffer com a paràmetre del **read**. Fa falta que existesca un fitxer anomenat **f2.txt** en l'arrel del projecte, com es comenta després:

```
import java.io.FileInputStream;
import java.io.IOException;

public class Exemple_2_04 {
    public static void main(String[] args) throws IOException {

        FileInputStream f_in = new FileInputStream("f2.txt");
        byte[] buffer = new byte[30];
        int n = f_in.read(buffer);
        while (n != -1) {
            for (int i=0; i<n; i++)
                System.out.print((char)buffer[i]);
            System.out.println("");
            n = f_in.read(buffer);
        }
        f_in.close();
    }
}
```

Es llegiran els caràcters de 30 en 30, ja que el buffer és d'aquesta grandària. Com que es guarda en un buffer de bytes (bytes, no caràcters), haurem de recórrer aquest buffer (fins el número de caràcters llegits, que és **n**) convertint cada byte en caràcter. Hem suposat que en el fitxer **f2.txt** tenim un text prou llarg com per a veure el funcionament. Si per exemple el contingut de **f2.txt** és aquest:

```
Hola. Aquest és un text més llarg, per veure com gestiona els bytes amb un
buffer de 30 caràcters.
Com que ho llegim des d'un InputStream, els caràcters especials potser no
isquen bé.
```

Aquesta seria l'eixida:

```
Hola. Aquest   s un text m  s
llarg, per veure com gestiona
els bytes amb un buffer de 30
car  cters.
Com que ho llegim
des d'un InputStre  m, els car  
cters especials potser no isq
uen b   .
```

Recordeu que estem llegint un fitxer de text des d'un `InputStream`, cosa gens convenient ja que els caràcters com `ç`, `ñ`, o vocals accentuades difícilment podrem fer que apareguen bé. Ho arreglarem amb els fluxos orientats a caràcter.

2.1.2 - Mètodes del OutputStream

Comencem també pel més senzill i primordial, el mètode que escriu un byte (recordeu que estem en fluxos orientats a bytes).

- void **write(int byte)** : escriu el byte passat com a paràmetre en el flux d'eixida. Encara que el paràmetre és de tipus int, només s'escriurà un byte. Si no es poguera fer l'escriptura per qualsevol motiu (per exemple, disc ple), es llançarà una excepció de tipus **IOException**.

Igual que en l'apartat anterior, anem a veure un exemple senzill d'utilització, en què guardarem en un fitxer el contingut d'una cadena (encara que ja sabem que no és el més apropiat utilitzar fluxos orientats a bytes per a informació de caràcters).

En aquest primer exemple del **OutputStream** treballarem sobre un fitxer inexistent. Es podrà comprovar que el resultat serà la creació del fitxer amb el contingut. Hem de fer constar que si no es tanca el fitxer (millor dit el flux d'eixida) podria ser que no es guardara res en el fitxer. Per tant és una operació ben important que no hem d'oblidar.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class Exemple_2_11 {
    public static void main(String[] args) throws IOException {

        String text = "Contingut per al fitxer.";
        FileOutputStream f_out = new FileOutputStream("f3.txt");
        for (int i=0;i<text.length();i++) {
            f_out.write(text.charAt(i));
        }
        f_out.close();
    }
}
```

En el constructor del OutputStream no hem indicat el segon paràmetre, aquell que indicava si era per a afegir o no, i per tant si no existia el fitxer el crearà, però si ja existia el fitxer, destruirà el seu contingut i el substituirà pel nou contingut. Per això si tornem a executar el programa, tindrem el mateix resultat.

Contingut per al fitxer.

La codificació del fitxer haurà seguit la que tinga per defecte el Sistema Operatiu, que en el cas d'Ubuntu és UTF-8, i en el cas de Windows és ISO-8859.

Anem a provar a substituir el constructor, posant ara

```
FileOutputStream f_out = new FileOutputStream("f3.txt",true);
```

Si l'executem una altra vegada, veurem que afegirà al final, sense destruir el que ja hi havia.

Contingut per al fitxer.Contingut per al fitxer.

Altres mètodes del OutputStream són:

- void **write(byte[] buffer)** : escriu el contingut de l'array de bytes al fitxer. Cal que buffer no siga nul, o provocarem un error.
- void **write(byte[] buffer, int pos, int llarg)** : escriu al fitxer el contingut de l'array que està a partir de la posició **pos** i tants bytes com assenya **llarg**.
- void **flush()** : Guardar les dades en un fitxer és una operació relativament lenta, ja que és accedir a un dispositiu lent (millor dit, no tan ràpid com la memòria). És habitual que s'utilitze una memòria intermèdia per a que les coses no vagen tan lentes (com si fóra una caché). Però potser que les dades no estiguen guardades encara en el fitxer, sinó que encara estiguen en aquesta caché. El mètode **flush** obliga a escriure els bytes que queden encara a la caché físicament al fitxer d'eixida.
- void **close()** : tanca el flux d'eixida, alliberant els recursos. Si quedava alguna cosa en la caché, es guardarà al fitxer i es tancarà el flux.

En aquest exemple es copia el contingut del fitxer **f2.txt** en el fitxer **f4.txt**, però en compte d'anar byte a byte, anirem de 30 en 30, amb un buffer de 30 posicions. Podríem cometre l'error que està marcat en roig, de escriure sempre els 30 caràcters:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Exemple_2_12 {

    public static void main(String[] args) throws IOException {
        FileInputStream f_in = new FileInputStream("f2.txt");
        FileOutputStream f_out = new FileOutputStream("f4.txt");

        byte[] buffer = new byte[30];
        int num = f_in.read(buffer);
        while (num != -1) {
            f_out.write(buffer);
            num = f_in.read(buffer);
        }
        f_in.close();
        f_out.close();
    }
}
```

D'aquesta manera, l'última vegada que és llig és molt possible que no hi hagen exactament 30 caràcters. Si hi ha menys de 30 caràcters, només es llegiran els que queden al principi del buffer, i en la resta del buffer hi ha la informació anterior, la de la penúltima lectura. En definitiva, tenim "basura", i si no ho controlem el resultat no serà el correcte. Aquest serà el contingut de **f4.txt**:

```
Hola. Aquest és un text més llarg, per veure com gestiona els bytes amb un
buffer de 30 caràcters.
Com que ho llegim des d'un InputStream, els caràcters especials potser no
isquen bé.
pecials potser no isq
```

Ha eixit d'aquesta manera perquè l'última vegada només s'han llegit 9 bytes. Els 21 restants tenen la informació encara de la penúltima lectura.

Per a fer-lo de forma correcta, ens aprofitem de que **read(buffer)** torna el número de bytes realment llegits, per escriure exactament aquest número. Per tant substituïrem la línia marcada en roig per:

```
f_out.write(buffer,0,num);
```

Ara el contingut de **f4.txt** serà idèntic al de **f2.txt**

Nota important

Per a assegurar-nos que realment escrivim en el fitxer i no es queda res en la memòria intermèdia, **hem de tancar sempre els fluxos d'eixida**. Si ens oblidem de tancar-los, és molt fàcil que no s'acabe d'escriure físicament en el fitxer.

2.2 - Fluxos orientats a caràcters

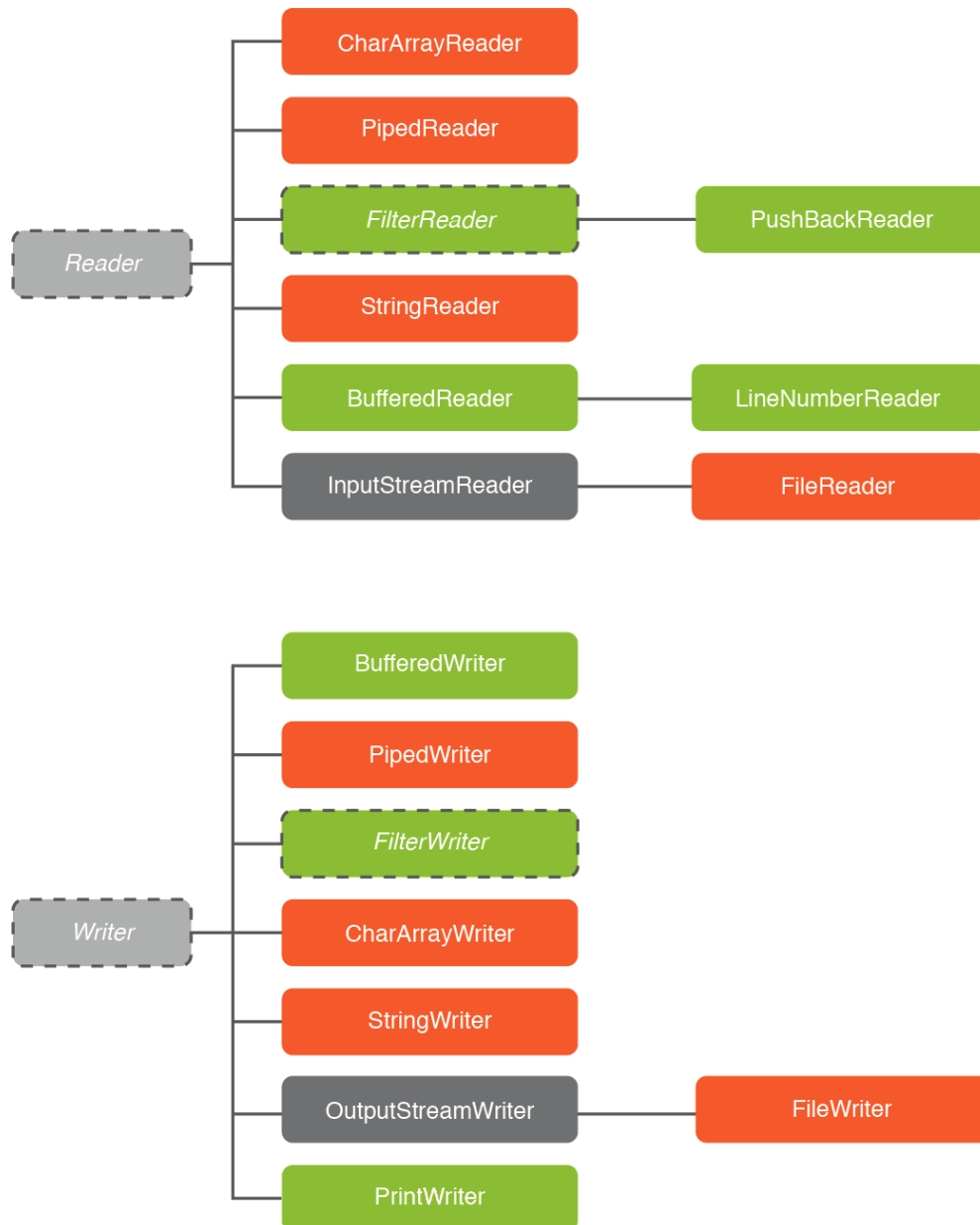
Treballar amb caràcters implica una dificultat apreciable, a causa sobretot de la diversitat de codificacions existents.

Per poder solucionar-ho, Java disposa de dues jerarquies, una d'entrada i una d'eixida, diferents de les que ja hem vist per a bytes (que eren `InputStream` i `OutputStream`). Aquestes jerarquies per a caràcters seran molt semblants a les de bytes, però sempre orientades a caràcters.

Igual que en els casos anteriors, tindrem unes classes abstractes, **Reader** i **Writer**, que no es poden instanciar directament (no podem crear un objecte d'aquestes classes). Serviran per a homogeneïtzar tots els fluxos d'entrada i d'eixida orientats a caràcter.

La super-classe **Reader** servirà per a fer l'entrada des de qualsevol dispositiu: fitxer, array de caràcters, una tuberia (per a dur dades des d'una altra aplicació). Totes les classes d'entrada heretaran d'ella, i serviran per especificar exactament d'on (per exemple un fitxer: **FileReader**) o per a donar alguna altra funcionalitat, com anirem veient a poc a poc. D'aquesta manera, els mètodes que es defineixen s'hauran d'implementar per les classes que hereten d'ella i assegura una uniformitat, siga quina siga la font.

Fem una ullada ràpida a la jerarquia de classes en la següent imatge:



De moment mirem únicament les que estan en color **taronja**, que especificaran quina serà la font de dades:

Classe	Explicació
FileReader	Per a llegir caràcters d'un fitxer
PipedReader	Per a llegir des d'una tuberia (és a dir informació que ve d'un altre programa)
CharArrayReader	L'entrada serà un array de caràcters
StringReader	L'entrada serà un string

Evidentment ens centrarem en la primera, que és la que més ens interessa per a la permanència de les dades.

Els fluxos d'eixida són molt molt pareguts, tots ells heretaran de **Writer**:

Classe	Explicació
FileWriter	Per a guardar caràcters en un fitxer

PipedWriter	Per a traure cap a una tuberia (és a dir informació que anirà a un altre programa)
CharArrayWriter	L'eixida serà un array de caràcters
StringWriter	L'eixida serà un string

Hem de fer constar que les classes d'emmagatzematge intern, com ara **CharArrayReader**, **CharArrayWriter**, **StringReader**, **StringWriter**, **PipedReader**, **PipedWriter** utilitzen sempre la codificació pròpia de Java (unicode de 16 bits: **UTF-16**), ja que guarden les dades a la memòria basant-se en els tipus dades de tractament de caràcters de Java (*char* i *String*).

En canvi les classes **FileReader** o **FileWriter** agafen la codificació **per defecte del sistema operatiu amfitrió**. L'usuari no pot seleccionar diferents sistemes de codificació en crear les instàncies. Així, una màquina virtual Java sobre Windows utilitzarà, per defecte, la codificació ISO-8859-1, però si corre sobre Linux, la codificació serà UTF-8. De tota manera veurem que sí que podrem arribar a especificar quin és el joc de caràcters que volem utilitzar en la pregunta 3.3. Intentarem veure exemples de tot.

Constructors de FileReader

De forma totalment paral·lela als fluxos orientats a byte, el **FileReader** té dos constructors, acceptant com a paràmetre un *File* o un *String* (amb el nom del fitxer). La diferència ara és que la unitat de transferència serà el caràcter (en compte d'un byte):

- **FileReader (File f)**: en el paràmetre se li passa un *File* (dels vistos en el tema anterior), que ha de ser una referència al fitxer.
- **FileReader (String nom_f)**: en el paràmetre se li passa un *String* amb el nom (i la possible ruta) del fitxer. Ens permetrà fer referència al fitxer de forma més ràpida, sense haver de passar per un *File*.

Constructors del FileWriter

També totalment paral·lel al *FileOutputStream*. Canviaran lleugerament respecte als d'entrada, ja que a més de fer referència al fitxer, opcionalment podrem d'especificar la manera d'escriure en el fitxer en cas que aquest ja existesca: bé afegint al final, o bé destruint la informació anterior. Aquestos són els constructors:

- **FileWriter (File f)**: en el paràmetre se li passa un *File*. Si no existia, el crearà; si ja existia esborrarà el contingut. En ambdós casos l'obrirà en mode escriptura.
- **FileWriter (String nom_f)**: igual que en l'anterior, però en el paràmetre se li passa un *String* amb el nom (i la possible ruta) del fitxer.
- **FileWriter (File f, boolean afegir)**: és com el primer, però si en el segon paràmetre se li passa **true** en compte de substituir el que ja hi havia, la informació s'afegirà al final. Si en aquest paràmetre se li passa **false** s'esborrarà el contingut anterior (com en el primer cas).
- **FileWriter (String nom_f, boolean afegir)**: igual que en l'anterior, però en el primer paràmetre se li passa un *String* amb el nom (i la possible ruta) del fitxer.

2.2.1 - Mètodes del Reader

Els mètodes del **Reader** són absolutament similars als del **InputStream**. La diferència és que ara llegirà sempre un caràcter. I no ens haurem de preocupar pel format en què està guardat, i de i ocupa un o dos bytes. Sempre el llegirà bé, siga quina siga la codificació utilitzada, com ja havíem comentat abans:

- `int read()`: llig el següent caràcter del flux d'entrada i el retorna com un enter. Si no hi ha cap caràcter disponible perquè s'ha assolit el final de la seqüència, es retornarà -1. Si no es pot llegir el següent caràcter per alguna causa (per exemple si després d'arribar al final intentem llegir un altre caràcter, o perquè es produeix un error en llegir l'entrada) es llançarà una excepció del tipus **IOException**. Es tracta d'un mètode abstracte, que les classes específiques sobreescriran adaptant-lo a una font de dades concreta (un fitxer, un array de caràcters, ...).

Abans de veure altres mètodes, mirem un exemple que és idèntic al primer exemple del **InputStream**, però canviant **FileInputStream** per **FileReader**. Llegirà el mateix fitxer anomenat **f1.txt**, utilitzat en aquell moment, però ara segurament llegirà tots els caràcters bé. El que farà és traure per pantalla caràcter a caràcter (en línies diferents).

```
import java.io.FileReader;
import java.io.IOException;

public class Exemple_2_21 {
    public static void main(String[] args) throws IOException {

        FileReader f_in = new FileReader("f1.txt");
        int c = f_in.read();
        while (c!=-1) {
            System.out.println((char)c);
            c=f_in.read();
        }
        f_in.close();
    }
}
```

Ara segurament sí que haurà llegit bé tots els caràcters, incloent ñ, ç, vocals accentuades, etc. Si encara tenim el mateix contingut en **f1.txt**, el resultat serà ara:

```
H
o
l
a
,

q
u
è

t
a
l
?
```

El més normal és que en crear el fitxer **f1.txt** amb algun editor, el guardem amb la codificació per defecte, que en cas de Windows és ASCII (o ISO-8859) i en el cas de Linux és UTF-8. I després des de Java, el **FileReader** utilitzarà la codificació per defecte del Sistema Operatiu. És a dir que en Linux el fitxer ha d'estar guardat en UTF-8 per a que el puga llegir bé, i en Windows en ASCII.

Mirem també l'exemple equivalent al segon. Allà utilitzàvem un **ByteArrayInputStream** com a entrada. Ara podríem utilitzar un **CharArrayReader**, però ho farem amb un **StringReader**, i quedarà més curt. A banda de que l'hem d'inicialitzar diferent, podem observar com el tractament posterior és idèntic:

```
import java.io.StringReader;
import java.io.IOException;

public class Exemple_2_22 {
    public static void main(String[] args) throws IOException {

        String ent = "Hola. Aquest és un String normal i corrent";
        StringReader f_in = new StringReader(ent);
        int c = f_in.read();
        while (c != -1) {
```



```

        System.out.println((char) c);
        c = f_in.read();
    }
    f_in.close();
}

```

Altres mètodes del **Reader** són:

- **int read(char[] buffer)**: llig un número determinat de caràcters de l'entrada, guardant-los en el paràmetre (que actuarà com un buffer). El número de caràcters llegits serà com a màxim la grandària del buffer, encara que podria ser menor (si no hi ha prou caràcters, per exemple). El mètode tornarà el número de caràcters que realment s'han llegit com un enter. Si no hi haguera cap caràcter disponible, es retornaria -1.
- **int available()**: indica quants caràcters hi ha disponibles per a la lectura. Sobretot serviria com a condició de final de bucle: si hi ha 0 caràcters disponibles, és que ja hem acabat. Tot i això, hi ha altres maneres de fer la condició de final de bucle.
- **long skip(long despl)**: salta, despreciant-los, tants caràcters com indica el paràmetre. Podria ser que no poguera saltar el número de caràcters especificat per diferents raons. Torna el número de caràcters realment saltats.
- **int close()**: tanca el flux de dades.

Mirem un altre exemple, utilitzant ara el buffer com a paràmetre del **read**. És idèntic al de l'apartat del **InputStream**. La diferència és que ara s'haurien de llegir bé tots els caràcters.

```

import java.io.FileReader;
import java.io.IOException;

public class Exemple_2_23 {
    public static void main(String[] args) throws IOException {

        FileReader f_in = new FileReader("f2.txt");
        char[] buffer = new char[30];
        int n = f_in.read(buffer);
        while (n != -1) {
            for (int i=0; i<n; i++)
                System.out.print(buffer[i]);
            System.out.println("");
            n = f_in.read(buffer);
        }
        f_in.close();
    }
}

```

Es llegiran els caràcters de 30 en 30, ja que el buffer és d'aquesta grandària. Com que ara es guarda en un buffer de caràcters, haurem de recórrer aquest buffer (fins el número de caràcters llegits, que és **n**). Hem suposat que en el fitxer **f2.txt** tenim un text prou llarg com per a veure el funcionament.

Aquesta seria l'eixida:

```

Hola. Aquest és un text més ll
arg, per veure com gestiona el
s bytes amb un buffer de 30 ca
ràcters.
Com que ho llegim des
d'un InputStream, els caràcte
rs especials potser no isquen
bé.

```

Efectivament, s'han llegit tots els caràcters perfectament.

2.2.2 - Mètodes del Writer

Comencem també pel més senzill i primordial, el mètode que escriu un caràcter.

- void **write(int car)** : escriu el caràcter passat com a paràmetre en el flux d'eixida. En cas que siga un `FileWriter`, escriurà el caràcter amb la codificació per defecte del S.O. : en Windows ISO-8839 i en Linux UTF-8. Si no es poguera fer l'escriptura per qualsevol motiu (per exemple, disc ple), es llançarà una excepció de tipus **IOException**.

Igual que en l'apartat anterior, anem a veure un exemple senzill d'utilització, en el qual guardarem en un fitxer el contingut d'una cadena, ara ja sense por als caràcters estranys.

En aquest primer exemple del **Writer** treballarem sobre un fitxer inexistent. Es podrà comprovar que el resultat serà la creació del fitxer amb el contingut.

Nota

Hem de fer constar que si no es tanca el fitxer (millor dit el flux d'eixida) podria ser que no es guardara res en el fitxer. Per tant és una operació ben important que no hem d'oblidar.

```
import java.io.FileWriter;
import java.io.IOException;

public class Exemple_2_31 {
    public static void main(String[] args) throws IOException {

        String text = "Contingut per al fitxer. Ara ja sense por a caràcters
especials: ç, à, ú, ...";
        FileWriter f_out = new FileWriter("f5.txt");
        for (int i=0;i<text.length();i++) {
            f_out.write(text.charAt(i));
        }
        f_out.close();
    }
}
```

En el constructor del **Writer** no hem indicat el segon paràmetre, aquell que indicava si era per a afegir o no, i per tant si no existia el fitxer el crearà, però si ja existia el fitxer, destruirà el seu contingut i el substituirà pel nou contingut. Per això si tornem a executar el programa, tindrem el mateix resultat en **f5.txt**

```
Contingut per al fitxer. Ara ja sense por a caràcters especials: ç, à, ú, ...
```

Anem a provar a substituir el constructor, posant ara

```
FileWriter f_out = new FileWriter("f3.txt",true);
```

Si l'executem, veurem que afegirà al final, sense destruir el que ja hi havia.

```
Contingut per al fitxer. Ara ja sense por a caràcters especials: ç, à, ú,
...Contingut per al fitxer. Ara ja sense por a caràcters especials: ç, à, ú,
...
```

Altres mètodes del **Writer** són:

- void **write(char[] buffer)** : escriu el contingut de l'array de caràcters al fitxer. Cal que buffer no siga nul, o provocarem un error.
- void **write(char[] buffer, int pos, int llarg)** : escriu al fitxer el contingut de l'array que està a partir de la posició **pos** i tants caràcters com assenyalen **llarg**.
- void **flush()** : Guardar les dades en un fitxer és una operació relativament lenta, ja que és accedir a un dispositiu lent (millor dit, no tan ràpid com la memòria). És habitual que s'utilitze una memòria intermèdia per a que les coses no vagen tan lentes (com si fóra una caché). Però potser que les dades no estiguen guardades encara en el fitxer, sinó que encara estiguen en aquesta caché. El mètode **flush** obliga a escriure els caràcters que queden

encara a la caché físicament al fitxer d'eixida.

- void **close()** : tanca el flux d'eixida, alliberant els recursos. Si quedava alguna cosa en la caché, es guardarà al fitxer i es tancarà el flux.

Aquestos mètodes són totalment similars als del `OutputStream`. A banda d'aquestos, el **Writer** té un altre, que pot ser especialment útil per a caràcters:

- void **write(String text)** : escriu tot el contingut del `String` en el fitxer.

3 - Fluxos decoradors

Anomenem classes "**decoradores**" a aquelles que hereten d'una classe determinada i serveixen per a dotar d'una funcionalitat extra que no tenia la classe original.

En els fluxos, en els d'entrada i en els d'eixida, veurem uns quants "decoradors" que ens permetran una funcionalitat extra: llegir o escriure una línia sencera (en compte de byte a byte, o caràcter a caràcter), o guardar amb determinat format de dades, ... En el cas de caràcters també ens permetran triar la codificació (ISO-8859-1, UTF-8, UTF-16, ...).

Anirem veient-los poc a poc, classificats per la classe arrel, és a dir, per una banda els decoradors del InputStream i OutputStream (orientats a byte), i per una altra banda els de Reader i Writer (orientats a caràcter)

3.1 - Decoradors de InputStream i OutputStream

Com hem comentat ens serviran per a donar una funcionalitat extra. Són els que estan de verd en la següent imatge:



Fixem-nos primers en els decoradors de **InputStream**:

Classe	Explicació
FilterInputStream	No és instanciable, únicament està per a que les altres depenguen d'ella (no la veurem)
LineNumberInputStream	Afegeix el número de línia de cada línia del InputStream (no la veurem)
DataInputStream	Permet llegir dades de qualsevol tipus de dades: enter, real, booleà, ...
BufferedInputStream	Munta un buffer d'entrada (no la veurem)
PushBackInputStream	Permet retrocedir un byte en la lectura, i per tant permet anar cap arrere (no la veurem)
ObjectInputStream	Permet llegir tot un objecte

I de forma quasi paral·lela tenim els decoradors de **OutputStream**:

Classe	Explicació
--------	------------

FilterOutputStream	No és instanciable, únicament està per a que les altres depenguen d'ella (no la veurem)
DataOutputStream	Permet guardar al flux de dades d'eixida dades de qualsevol tipus: enter, real, booleà, ...
BufferedOutputStream	Munta un buffer d'eixida (no la veurem)
PrintStream	Permet escriure dades de diferents tipus, i té també els mètodes <i>printf</i> i <i>println</i>
ObjectOutputStream	Permet escriure (serialitzar) tot un objecte

Comentem-los un poquet més.

BufferedInputStream i **BufferedOutputStream** ens ofereixen un buffer d'entrada i d'eixida respectivament, per a fer la transferència més efectiva. En la pràctica ens oferirà poques funcionalitats útils (a banda de l'eficiència en la transferència, clar). Quan anem als decoradors de fluxos orientats a caràcters, sí que trobarem utilitats als decoradors semblants a aquests, com per exemple llegir o escriure una línia sencera de caràcters. Però aquests orientats a bytes, no els veurem.

DataInputStream i **DataOutputStream** ens oferiran la possibilitat de llegir o escriure còmodament dades de diferents tipus: enter, real, booleà, strings, ... Els veurem en detall en el proper **Tema 3**

ObjectInputStream i **ObjectOutputStream** (que curiosament són els únics que no depenen de **FilterInputStream** i **FilterOutputStream**) ens permetran guardar o recuperar de cop tot un objecte, és a dir totes les seues propietats (les dades de l'objecte). No ens haurem de preocupar ni de l'ordre ni del tipus de les propietats de l'objecte: quan escrivim l'objecte, es guardaran totes les dades de forma compacta; i quan llegim es recuperaran de forma correcta. És per tant una parella de classes d'extrema utilitat per a guardar objectes, que en definitiva són l'essència de la programació en Java. Els veurem en detall en el proper **Tema 3**.

PrintStream

L'únic que ens queda és el que veurem ara amb un poquet més de detall, el **PrintStream**. Ens permetrà bàsicament 3 coses:

- Escriure dades de més d'un tipus de dades. Per exemple **print(5.25)** escriu un número real, i **print("Hola")** escriu tot un string.
- Donar un determinat format a l'eixida, amb tota la funcionalitat a què estem acostumats amb **printf**
- Escriure tota una línia amb **println**, és a dir, acabar una dada amb el retorn de carro, per a baixar de línia.

Mirem un exemple que ens pot donar idea de la seua funcionalitat.

```
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Exemple_2_41 {
    public static void main(String[] args) throws IOException {

        PrintStream f_out = new PrintStream(new FileOutputStream("f6.txt"));

        float a = (float) 5.25;
        String b = "Hola.";
        f_out.print(b);
        f_out.println("Què tal?");
        f_out.println(a+3);
        f_out.printf("El número %d en hexadecimal és %x",27,27);

        f_out.close();
    }
}
```

Es crearà el fitxer **f6.txt** (si ja existia esborrarà el contingut anterior) amb el següent contingut:

```
Hola.Què tal?
8.25
El número 27 en hexadecimal és 1b
```

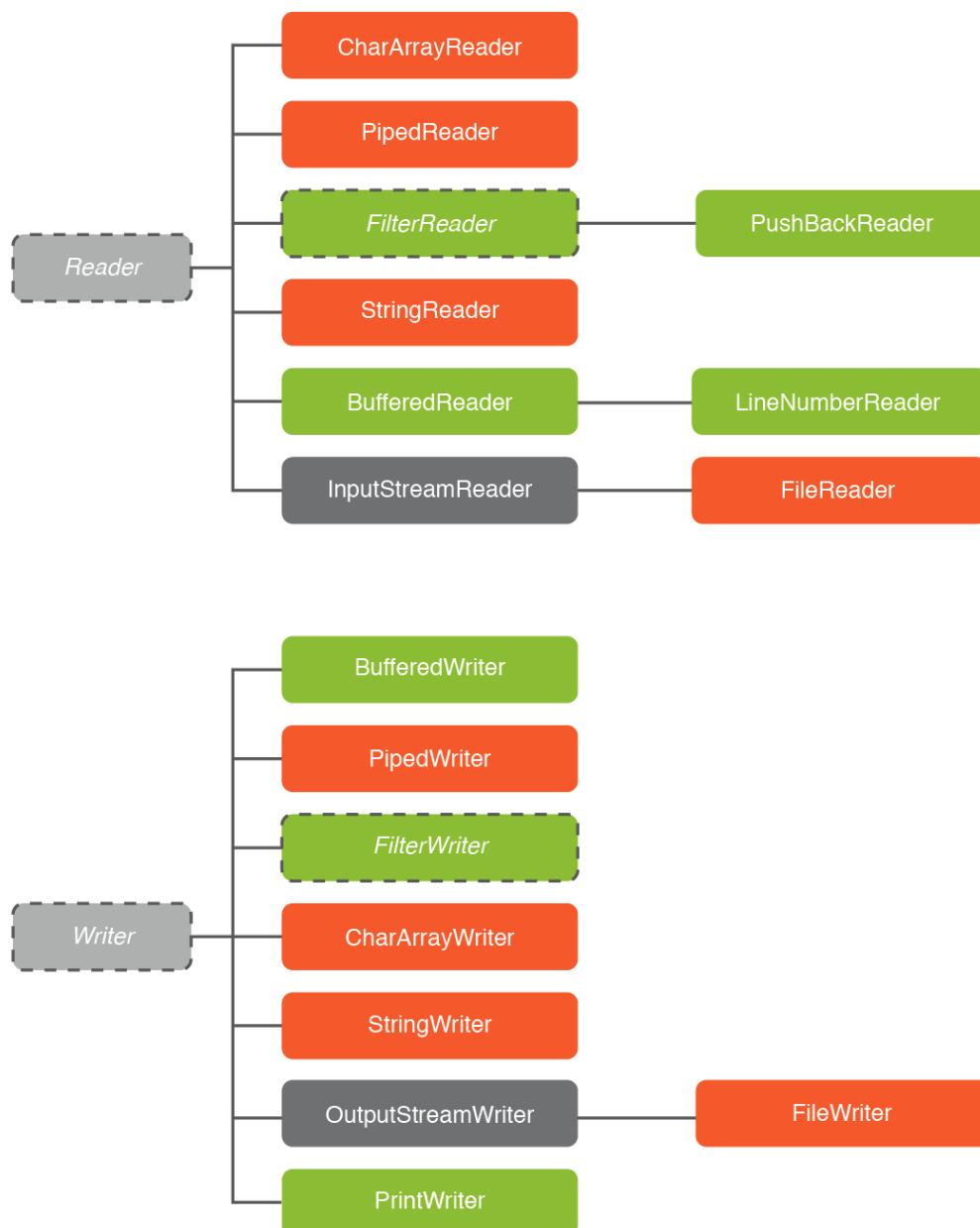
En realitat el **PrintStream**, a banda del constructor que accepta un **OutputStream**, també té un altre que accepta un

File i fins i tot un altre que accepta un **String** amb el nom del fitxer. Per tant, la següent sentència també ens funcionaria:

```
PrintStream f_out = new PrintStream("f6.txt");
```

3.2 - Decoradors de Reader i Writer

Mirem ara els decoradors de la jerarquia **Reader** i **Writer** . Tornen a ser els de color verd. Els de color gris **InputStreamReader** i **OutputStreamWriter** són conversors que permeten passar un **InputStream** a un **Reader** i un **OutputStream** a **Writer**. Els veurem en la següent pregunta.



Fixem-nos primers en els decoradors de **Reader**:

Classe	Explicació
FilterReader	No és instanciable, únicament està per a que les altres depenguen d'ella (no la veurem)
PushBackInputStream	Permet retrocedir un caràcter en la lectura, i per tant permet anar cap arrere (no la veurem)
BufferedReader	Munta un buffer d'entrada, i permet entre altres coses llegir una línia sencera
LineNumberReader	Afegeix el número de línia de cada línia del fitxer (no la veurem)

I de forma quasi paral·lela tenim els decoradors de **Writer**:

Classe	Explicació
FilterWriter	No és instanciable, únicament està per a que les altres depenguen d'ella (no la veurem)
BufferedWriter	Munta un buffer d'eixida, i permet entre altres coses escriure una línia sencera
PrintWriter	Permet escriure dades de diferents tipus, i té també els mètodes <i>printf</i> i <i>println</i>

El **PrintWriter** funciona quasi exactament igual que el **PrintStream**, i per a caràcters és més útil que l'altre (per ser **Writer**), per tant és el candidat a recordar.

El **BufferedReader** sí que ens oferirà facilitats interessants, com llegir una línia sencera. En canvi el **BufferedWriter** no ens ofereix tantes facilitats com el **PrintWriter**, és un poc més incòmode.

BufferedReader i BufferedWriter. PrintWriter

BufferedReader i BufferedWriter munten un buffer (d'entrada i d'eixida respectivament) de caràcters per a fer més eficient la transferència. A banda d'això tindran uns mètodes que ens seran molt útils.

BufferedReader

- mètode **readLine()** que ens permet llegir una línia sencera del fitxer (fins al final de línia). Açò és de molta utilitat en els fitxers de text.

BufferedWriter

- mètode **newLine()** que permet introduir el caràcter de baixada de línia
- mètode **write(String cad, int com, int llarg)** que permet escriure tot un string, o una part d'ell, especificant on comença el que volem escriure i la llargària

Com veieu el **BufferedReader** sí que ens ofereix la possibilitat de llegir una línia sencera, però en canvi el **BufferedWriter** es queda un poc curt. Per això preferirem el **PrintWriter**.

PrintWriter

- mètodes **print(qualsevol_tipus)**, que permeten imprimir una dada de qualsevol tipus: booleà, char, tots els numèrics, string, ... Serà segurament el que més utilitzarem.
- mètodes **println(qualsevol_tipus)**, a banda de tot el de **print**, baixen de línia
- mètode **printf()**, que permet donar un format

Veiem un senzill exemple per a copiar el contingut d'un fitxer de text i modificar-lo lleugerament. El més còmode serà anar línia a línia. Per tant utilitzarem el **BufferedReader** per a llegir línies, i el **PrintWriter** per a escriure línies. La lleugera modificació consistirà en posar el número de línia davant

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Exemple_2_51 {

    public static void main(String[] args) throws IOException {
        BufferedReader f_ent = new BufferedReader(new
            FileReader("f7_ent.txt"));
        PrintWriter f_eix = new PrintWriter(new FileWriter("f7_eix.txt"));
        String cad=f_ent.readLine();
        int i=0;
        while (cad != null){
            i++;
            f_eix.println(i + ".- " + cad);
            cad=f_ent.readLine();
        }
    }
}
```

```
        f_eix.close();f_ent.close();  
    }  
}
```

Si en el fitxer d'entrada (**f7_ent.txt**) tenim guardada la següent informació (introduïda amb el notepad o gedit):

```
Primera  
Segona  
Tercera
```

En el fitxer d'eixida (**f7_eix.txt**) tindrem:

```
1.- Primera  
2.- Segona  
3.- Tercera
```

3.3 - Conversors: InputStreamReader i OutputStreamWriter

Una vegada vistes les jerarquies de les classes **InputStream-OutputStream** per una banda, i **Reader-Writer** per una altra, veurem ara unes classes que serviran per a passar d'una jeraquia a una altra. És a dir, poder passar un **InputStream** a **Reader**, o el que és el mateix, un flux orientat a bytes en un flux orientat a caràcters. I el mateix amb el **OutputStream** i el **Writer**.

- **InputStreamReader**: passa un **InputStream** a **Reader**. Accepta com a paràmetre el **InputStream** i dóna com a resultat un **Reader**.
- **OutputStreamWriter**: passa un **OutputStream** a **Writer**. Accepta com a paràmetre el **OutputStream** i dóna com a resultat un **Writer**.

A més en el constructor dels dos, **InputStreamReader** i **OutputStreamWriter**, tenim la possibilitat d'especificar el tipus de codificació, a més del **InputStream** o **OutputStream**. Açò ens serà molt útil, perquè fins el moment no podíem triar el tipus de codificació d'un **FileReader** o **FileWriter** que era UTF-8 en el cas de Linux, i ASCII (millor dit la seua extensió ISO-8859-1) en el cas de Windows.

Mirem aquest exemple, en el qual transformem el mateix fitxer d'una configuració a una altra. Aprofitem algun dels fitxers que ja disposem (per exemple f5.txt, que tenia caràcters especials com vocals accentuades). En l'exemple el tindrem en codificació UTF-8, ja que està provat en Linux. El transformarem a ISO-8859-1.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

public class Exemple_2_61 {

    public static void main(String[] args) throws IOException {
        InputStreamReader f_ent = new InputStreamReader(new
            FileInputStream("f5.txt"), "UTF-8");
        OutputStreamWriter f_eix = new OutputStreamWriter(new
            FileOutputStream("f5_ISO.txt"), "ISO-8859-1");

        int car = f_ent.read();
        while(car!=-1) {
            f_eix.write(car);
            car = f_ent.read();
        }
        f_eix.close();
        f_ent.close();
    }
}
```

Hem posat l'entrada explícitament que siga de UTF-8. En realitat no faria falta, ja que si treballem en Linux, aquesta serà la codificació per defecte, i per tant seria la que utilitzaria un **FileReader**.

```
FileReader f_ent = new FileReader("f5.txt");
```

Anem a fer una altra versió del mateix programa. A banda de no especificar la codificació del fitxer d'entrada, utilitzarem els decoradors **BufferedReader** i **PrintWriter** per a poder anar còmodament línia a línia.

```
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class Exemple_2_62 {

    public static void main(String[] args) throws IOException {
        BufferedReader f_ent = new BufferedReader (new FileReader("f5.txt"));
        PrintWriter f_eix = new PrintWriter(new OutputStreamWriter(new
            FileOutputStream("f5_ISO.txt"), "ISO-8859-1"));
    }
}
```

```
String cad = f_ent.readLine();
while(cad!=null) {
    f_eix.println(cad);
    cad = f_ent.readLine();
}
f_eix.close();
f_ent.close();
}
```

Exercicis



Exercici 2_1

Aquest primer exercici és per a tractament de bytes, ja que es tractarà de modificar una imatge. Per a poder provar-lo podeu utilitzar la imatge **Penyagolosa.bmp** que se us proporciona en l'aula virtual i l'heu de copiar al directori arrel del projecte **Tema2** per a un funcionament més còmode.

No es pretén construir un editor d'imatges. Tan sols pretenem agafar la informació del fitxer byte a byte, reslitzar alguna transformació en els bytes i guardar-la en un altre fitxer.

El format d'un fitxer **bmp**, aproximadament és el següent:

- En els 54 primers bytes es guarda informació diversa, com la grandària de la imatge, paleta de colors, ...
- A partir d'ahí es guarda **cada punt** de la imatge com **3 bytes**, un per al roig (R), un per al verd (G) i un per al blau (B), anant d'esquerra a dreta i de dalt a baix.

Copia't i modifica la classe **fitxerImatge**, creant els mètodes oportuns seguint aquestes pautes:

- El constructor **fitxerImatge(File fEnt)** ha d'inicialitzar la propietat **f (File)** **si i només si** existeix el fitxer i l'extensió del fitxer és **.bmp** (ho controlarem senzillament perquè el nom del fitxer acaba així). En cas contrari, traure els missatges d'error oportuns per l'eixida estàndar.
- Els mètodes de transformació (**transformaNegatiu**, **transformaObscur** i el voluntari **transformaBlancNegre**) han de crear un nou fitxer que contindrà la imatge transformada com veurem més avant. El nom del nou fitxer s'ha de formar a partir del nom del fitxer d'entrada, el que hem guardat en el constructor. Serà sempre posant abans del .bmp un guió baix i un identificatiu de la transformació realitzada: **_n** per al negatiu, **_o** per a l'obscur i **_bn** per al blanc i negre (part voluntària). És a dir, si el fitxer d'entrada fóra **imatge1.bmp**, el d'eixida haurà de ser:
 - **imatge1_n.bmp** per al mètode **transformaNegatiu**
 - **imatge1_o.bmp** per al mètode **transformaObscur**
 - **imatge1_bn.bmp** per al mètode voluntari **transformaBlancNegre**
- En cada transformació, els primers 54 bytes s'han de copiar sense modificar: s'han d'escriure en el fitxer de destí tal i com s'han llegit del fitxer d'entrada
- A partir del 54, cada vegada que es llegirà un byte, s'haurà de transformar abans d'escriure'l en el destí. La transformació és d'aquesta manera:
 - Per al **negatiu (transformaNegatiu)**, cada byte de color (RGB) de cada punt, s'ha de transformar en el complementari. Com estem parlant de bytes però que en llegir els guardem en enters, senzillament serà calcular **255 - b** (si b és el byte llegit).
 - Per a l'**obscur (transformaObscur)**, cada byte de color (RGB) de cada punt, s'ha de baixar d'intensitat a la meitat. Senzillament serà calcular **b / 2** (si b és el byte llegit).
 - Per al **blanc i negre (transformaBlancNegre)**, que és el voluntari, hem de donar el mateix valor per al roig, el blau i el verd (RGB) de cada punt, i així aconseguirem un gris d'intensitat adequada. Una bona manera serà **llegir els tres bytes** de cada punt (no s'aconsella utilitzar una lectura amb un array de 3 posicions; millor fer tres lectures guardades en tres variables diferents), **calcular la mitjana** d'aquests 3 valors, i **escriure el resultat 3 vegades** en el fitxer de destí.

A mode orientatiu del que es vol fer, us adjunte la classe **fitxerImatge** a la qual heu de modificar el constructor i els tres mètodes de transformació (l'últim és voluntari). Recordeu que ha d'anar al paquet **Exercicis**.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class fitxerImatge {
    File f = null;    // en f és on es guardarà la referència del
    fitxer

    public fitxerImatge(File fEnt) {
```

```

        // Control d'existència del fitxer i control de l'extensió .bmp
        (traure missatges d'error)
        // En cas que tot siga correcte, inicialitzar f amb fEnt
    }

    public void transformaNegatiu() throws IOException{
        // Transformar a negatiu i guardar en _n.bmp
    }

    public void transformaObscur() throws IOException{
        // Transformar a una imatge més fosca i guardar en _o.bmp
    }

    /* Part voluntària
    public void transformaBlancNegre() throws IOException{
        // Transformar a una imatge en blanc i negre i guardar en
        bn.bmp
    }
    */
}

```

Aquest seria un exemple de programa principal, que podeu utilitzar si voleu. Recordeu que ha d'anar al paquet **Exercicis**.

```

import java.io.File;
import java.io.FileNotFoundException;


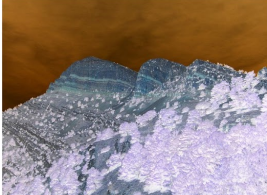


public class Exercici_2_1 {

    public static void main(String[] args) throws IOException {
        File f = new File("Penyagolosa.bmp");

        fitxerImatge fi = new fitxerImatge(f);
        fi.transformaNegatiu();
        fi.transformaObscur();
        // voluntari
        // fi.transformaBlancNegre();
    }
}

```

Per a la imatge que se us proporciona (i que està en la construcció del File del programa principal), que és la de l'esquerra, haurien d'eixir les de la dreta:

Imatge inicial	negatiu	obscur	blanc i negre (voluntari)
			
Penyagolosa.bmp	Penyagolosa_n.bmp	Penyagolosa_o.bmp	Penyagolosa_bn.bmp



Exercici 2_2

Aquest programa serà el primer que utilitzarà components gràfics.

Per a entendre els components gràfics de la llibreria **Swing** i els contenidors de la llibreria **Awt**, que són els que utilitzarem, us aconselle que us mireu l'annex **Gràfics en Java: llibreries AWT i SWING** que teniu en la secció d'annexos, al final del curs de Moodle. De tota manera, us proporcione "l'esquelet" del programa, i només us deman

Anem a fer un senzill editor de text amb el següent aspecte:



En el `TextField` posarem el nom (i ruta) del fitxer.

- Quan apremem al botó **Obrir** ha de bolcar el contingut del fitxer al `JTextArea` (controlant prèviament que existeix el fitxer).
- Quan apremem a **Guardar**, ha de bolcar el contingut del `JTextArea` en el fitxer (el nom del qual tenim en el `TextField`).

L'esquelet del programa és el que trobareu a continuació. He utilitzat la filosofia de l'annex, que consisteix a tenir un programa principal (el main) que l'únic que fa és utilitzar un objecte de la classe que hereta de `JFrame`, que és realment la finestra.

Una vegada copiats en el paquet **Exercicis** del projecte **Tema2**, només heu de completar el que teniu al final de tot, al mètode `actionPerformed`

```
import java.io.IOException;

public class Exercici_2_2 {

    public static void main(String[] args) throws IOException {
        final Exercici_2_2_Pantalla finestra = new
Exercici_2_2_Pantalla();
        finestra.iniciar();
    }

}
```

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Exercici_2_2_Pantalla extends JFrame implements
ActionListener {

    private static final long serialVersionUID = 1L;
    JLabel et_f = new JLabel("Fitxer:");
    JTextField fitxer = new JTextField(25);
    JButton obrir = new JButton("Obrir");
    JButton guardar = new JButton("Guardar");
    JLabel et_a = new JLabel("Contingut:");
    JTextArea area = new JTextArea(10, 50);
    JScrollPane scrollPane = new JScrollPane(area);

    // en iniciar posem un contenidor per als elements anteriors
    public void iniciar() {
        getContentPane().setLayout(new GridLayout(2, 1));
        setTitle("Editor de text");

        JPanel panell1 = new JPanel(new GridLayout(0, 1));
```

```

        JPanel panell1_1 = new JPanel(new FlowLayout());
        panell1.add(panell1_1);
        panell1_1.add(et_f);
        panell1_1.add(fitxer);

        JPanel panell1_2 = new JPanel(new FlowLayout());
        panell1.add(panell1_2);
        panell1_2.add(obrir);
        panell1_2.add(guardar);
        JPanel panell2 = new JPanel(new GridLayout(0, 1));
        panell2.add(scrollPane);
        area.setEditable(true);

        getContentPane().add(panell1);
        getContentPane().add(panell2);
        setVisible(true);
        pack();
        obrir.addActionListener(this);
        guardar.addActionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == obrir) {
            // Instruccions per a bolcar el contingut del fitxer en el
JTextArea
        }
        if (e.getSource() == guardar) {
            // Instruccions per a guardar el contingut del JTextArea al
fitxer. No us oblideu de tancar el flux d'eixida.
        }
    }
}

```

Nota

En l'operació de guardar, no us oblideu de tancar el flux d'eixida per assegurar-vos que s'escriu en el fitxer

**Exercici 2_3. Voluntari**

Com a exercici voluntari us propose una altra versió del Editor de Text de l'anterior exercici.

Ara serà únicament un JTextArea, i les opcions les tindrem en menú. Utilitzeu el component **JFileChooser** per a buscar fitxers i per a guardar-los que ens proporciona **Swing**. També heu d'implementar l'opció d'eixir.



Si teniu temps i ganes, afegiu un component baix de tot per a triar la codificació entre **UTF-8** i **ISO-8859-15**

Aquest seria l'esquelet del programa principal i la classe que implementa JFrame:

```
import java.io.IOException;

public class Exercici_2_3 {

    public static void main(String[] args) throws IOException {
        final Exercici_2_3_Pantalla finestra = new Exercici_2_3_Pan
        finestra.iniciar();
    }
}
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JSeparator;

public class Exercici_2_3_Pantalla extends JFrame implements Action
    JMenuBar menu_p = new JMenuBar();

    JMenu menu_arxiu = new JMenu("Arxiu");
    JMenu menu_ajuda = new JMenu("Ajuda");

    JMenuItem obrir = new JMenuItem("Obrir");
    JMenuItem guardar = new JMenuItem("Guardar");
    JMenuItem guardarCom = new JMenuItem("Guardar com ...");
    JMenuItem eixir = new JMenuItem("Eixir");

    JMenuItem quantA = new JMenuItem("Quant a Editor");

    JFileChooser fCh = new JFileChooser();

    public void iniciar(){
        this.setSize(400, 300);
        this.setJMenuBar(menu_p);
        menu_p.add(menu_arxiu);
        menu_p.add(menu_ajuda);

        menu_arxiu.add(obrir);
        menu_arxiu.add(guardar);
        menu_arxiu.add(guardarCom);
        menu_arxiu.add(new JSeparator());
        menu_arxiu.add(eixir);
    }
}
```

```
        menu_ajuda.add(quantA);

        this.setVisible(true);

        obrir.addActionListener(this);
        guardar.addActionListener(this);
        guardarCom.addActionListener(this);
        eixir.addActionListener(this);

        quantA.addActionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if ((e.getSource() == eixir))
            System.exit(0);
    }
}
```

Llicenciat sota la [Llicència Creative Commons Reconeixement NoComercial CompartirIgual 2.5](#)