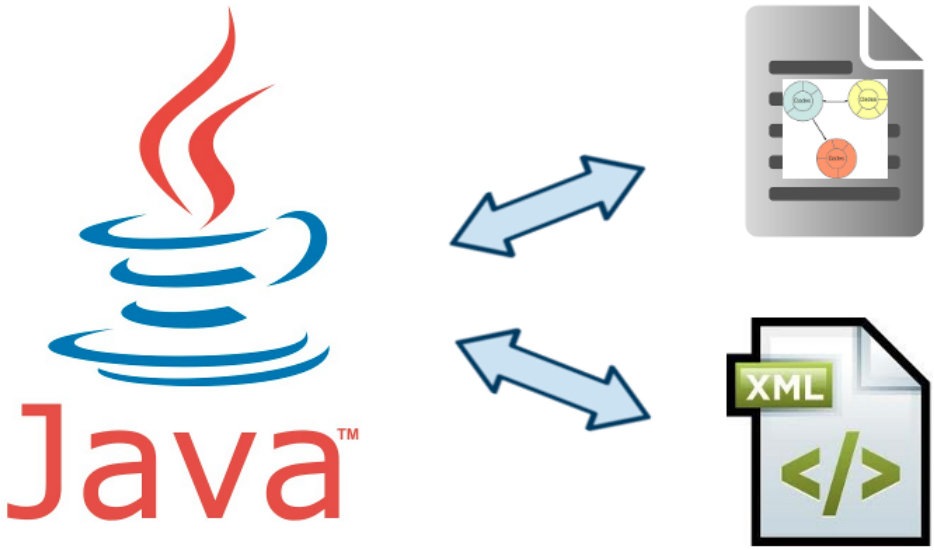
Accés a Dades

Tema 3: Fitxers de diferents formats



Objectius

Objectius

Aquest tema servirà per a finalitzar el contacte amb els fitxers. Una vegada hem vist com accedir als fitxers i al seu contingut, ara és el moment de plantejar-nos com guardar dades de diferents tipus, com guardar objectes, ... , farem menció especial als fitxers XML i al format JSON

- Per una banda es gestionen fitxers amb dades de diferents tipus, ben al contrari que en els fitxers de text, o els exemples d'agafar un fitxer i tractar tots els bytes de forma igual.
- Per una altra els fitxers d'accés directe (també anomenats d'accés relatiu o aleatori).
- També s'introdueix el tema de la seriació d'objectes, és a dir, intentar guardar objectes directament en fitxers. Una tècnica senzilla però que té inconvenients.
- Després es tracten els documents XML específicament, no com un fitxer de text (que ho és), sinó amb un tractament específic per poder accedir a la informació jerarquitzada d'un document XML.
- I finalment farem el mateix amb el format JSON

1.- Fitxers binaris amb formats específics

DataInputStream i DataOutputStream

Ja hem vist com utilitzar els fitxers de caràcters i també de bytes. Però en aquest últim cas sempre ha estat per a llegir o escriure byte a byte, fins al final de fitxer.

Ens plantegem ara com utilitzar els fitxer per a guardar dades estructurades de tipus bàsics **diferents**. De moment no seran complicades però enseguida veurem que ens fa falta alguna cosa per a poder treballar còmodament.

Suposem un exemple d'una empresa que vol guardar dades dels seus empleats. Concretament vol guardar el número d'empleat, el nom, el departament al qual pertany, l'edat i el sou.

Número	Nom	Depart	Edat	Sou
1	Andreu	10	32	1000.00
2	Bernat	20	28	1200.00
3	Claudia	10	26	1100.00
4	Damià	10	40	1500.00

Ja es veu que les dades són de difrents tipus. Si tot fóra de text no hi hauria problema. Però si considerem les dades com a numèriques enteres o reals no ens serveixen els Stream de caràcters (Reader i Writer). Per tant hem d'anar a InputStream i OutputStream, però seria molt dur treballar directament amb bytes. Hauríem de saber exactament quants bytes ocupa cada dada: **int** utilitza 4 bytes; **short** utilitza 2 bytes, **byte** utilitza 1 byte, i **long** utilitza 8 bytes; en el cas dels reals, **float** utilitza 4 bytes i **double** utilitza 8 bytes. Massa varietat i massa fiena recordar-los tots. Ens falta doncs una ajuda per a poder guardar i recuperar dades d'aquestos diferents tipus.

Aquesta funcionalitat ens la proporciona la parella **DataInputStream** i **DataOutputStream**, que són decoradors dels Stream i que disposen de mètodes per a guardar o recuperar dades de diferents tipus, sense haver de saber el format intern de cadascun ni quants bytes ocupen. En la següent taula tenim uns quants mètodes d'aquestos:

DataInputStream	Explicació	**DataOutputStream**
byte **readByte**()	Un byte	void **writeByte**(int)
short **readShort**()	Un enter xicotet (2 bytes)	void **writeShort**(short)
int **readInt**()	Un enter (4 bytes)	void **writeInt**(int)
long **readLong**()	Un enter llarg (8 bytes)	void **writeLong**(long)
float **readFloat**()	Un real en simple precisió	void **writeFloat**(flaot)
double **readDouble**()	Un real en doble precisió	void **writeDouble**(double)
char **readChar**()	Un caràcter Unicode (16 bits)	void **writeChar**(int)
String **readUTF**()	Una cadena de caràcters UTF-8 i la converteix en String (16 bits)	void **writeUTF**(String)

Anem a veure l'exemple, en el qual guardarem en un fitxer anomenat **Empleats.dat** les dades dels 4 empleats. Per comoditat ens els definirem en arrays de 4 elements: un array per als noms, un altre per als departaments, etc.

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


public class CrearEmpleats {

    public static void main(String[] args) throws IOException {
        DataOutputStream f = new DataOutputStream(new
        FileOutputStream("Empleats.dat"));

        String[] noms = {"Andreu","Bernat","Clàudia","Damià"};
        int[] departaments = {10,20,10,10};
        int[] edats = {32,28,26,40};
        double[] sous = {1000.0,1200.0,1100.0,1500.0};
```

```
        for (int i=0;i<4;i++){
            f.writeInt(i+1);
            f.writeUTF(noms[i]);
            f.writeInt(departaments[i]);
            f.writeInt(edats[i]);
            f.writeDouble(sous[i]);
        }
        f.close();
    }
}
```

I aquesta seria la recuperació de les dades:

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

public class ConsultarEmpleats {

    public static void main(String[] args) throws IOException {
        DataInputStream f = new DataInputStream(new
        FileInputStream("Empleats.dat"));

        while (f.available()>0){
            System.out.println("Número: " + f.readInt());
            System.out.println("Nom: " + f.readUTF());
            System.out.println("Depart: " + f.readInt());
            System.out.println("Edat: " + f.readInt());
            System.out.println("Sou: " + f.readDouble());
            System.out.println();
        }
        f.close();
    }
}
```

2.- Accés directe a fitxers

Els exemples de l'apartat anterior ens poden fer reflexionar sobre un altre tipus d'accés als fitxers. De moment tots els accessos que hem fet als fitxers, tant binaris com de caràcter, ha estat **seqüencials**. Això vol dir que sempre comencem pel principi del fitxer fins que arribem a la informació que volem, o en la major part dels casos fins el final de fitxer.

Però, i si volem únicament una determinada informació? En l'exemple de l'apartat anterior, què hauríem de fer si volem només la informació de l'empleada 3 (Clàudia)? Doncs hauríem de passar primer per tots els anteriors. Com només hi ha 2 davant, no sembla molta feina, però és fàcil de veure la dificultat (o millor dit el cost) si el fitxer constara de centenars o milers d'empleats. Suposem un fitxer de 10.000 empleats. Si volem accedir a l'empleat 9.500 hauríem de passar pels 9.499 empleats anteriors, ja que l'accés seqüencial obliga a començar pel principi i anar passant fins que trobem la informació. I encara pitjor: i si després de consultar l'empleat 9.500 ara volem consultar el 9.000? Doncs hauríem de començar des del principi, perquè ja ens l'havíem passat.

Afortunadament i ha una altra manera d'accedir, un altre tipus d'accés. S'anomena **accés directe** perquè permetrà anar directament a una posició determinada del fitxer. Moltes vegades també es diu **accés relatiu** o **accés aleatori**, però el funcionament sempre és el mateix. I mireu que estem parlant d'accés. Per tant el que canviarà no és la classe **File** sinó qui permet accedir al contingut, és a dir les classes de **flux d'informació** (els Streams).

Les classes **InputStream-OutputStream** i **Reader-Writer** només permeten l'accés seqüencial. Per tant per a l'accés directe disposarem d'una altra classe que ens permetrà fer totes les operacions, tant de lectura com d'escriptura. Té l'avantatge que disposa de molts mètodes per a poder accedir a la informació. No ens fara falta, per tant, les classes "decoradores" que afegeixen funcionalitats. Amb aquesta classe tindrem suficient.

RandomAccessFile

La classe **RandomAccessFile** ens permetrà accedir de forma directa a un fitxer. No ens farà falta, en principi, cap altra classe més. Ens proporcionarà tota la funcionalitat necessària.

En els constructors aniran 2 paràmetres. El primer farà referència al fitxer. El segon al mode d'accés: només lectura (**r**) o lectura-escriptura (**rw**).

```
RandomAccessFile(File fitxer, String mode) throws FileNotFoundException
```

```
RandomAccessFile(String fitxer, String mode) throws FileNotFoundException
```

En el primer cas li especifiquem un **File** en el primer paràmetre. En el segon un **String** que correspondrà amb el nom del fitxer

En ambdós casos, el segon paràmetre indicarà el mode:

- **"r"** indica només lectura
- **"rw"** indica lectura escriptura

A pesar de ser una classe completament diferent de la jerarquia de **InputStream-OutpuStream** (o **Readre-Writer**) implementa mètodes que es diuen exactament igual que els d'aquelles classes, cosa que fa molt més còmoda la utilització. Els mètodes més importants són:

Mètode	Explicació	Mètode
int **read()**	llig (escriu) un byte (encara així torna o se li passa un enter)	void **write(int)**
int **read(byte[])**	llig (escriu) un sèrie de bytes, tants com la grandària de l'array (si pot)	int **write(byte[])**
byte **readByte()**	llig (escriu) un byte interpretat com número de 8 bits amb signe	void **writeByte(int)**

char **readChar()**	llig (escriu) un caràcter	void **write(char)**
int **readInt()**	llig (escriu) un enter (4 bytes)	void **write(int)**
short **readShort()**	llig (escriu) un enter xicotet (2 bytes)	void **write(short)**
long **readLong()**	llig (escriu) un enter llarg (8 bytes)	void **write(long)**
float **readFloat()**	llig (escriu) un número real en simple precisió (4 bytes)	void **write(float)**
double **readDouble()**	llig (escriu) un número real en doble precisió (8 bytes)	void **write(double)**
String **readUTF()**	llig (escriu) un cadena de caràcters (interpretat com UTF-8)	void **writeUTF()**
void **seek(long)**	situa el punter en la posició, mesurat des del principi del fitxer	
long **length()**	torna la grandària del fitxer	
void **close()**	Tanca el flux de l'accés directe	

En cada lectura, després de llegir el punter que apunta al fitxer estarà situat després de la dada llegida, siga quina siga la grandària.

Anem a veure un exemple utilitzant el fitxer **Empleats.dat** creat en l'apartat anterior. Obrim l'accés directe únicament en mode lectura, i ens situem directament a la posició 56, que és on comença la informació de l'empleada 3 (Clàudia). Posteriorment utilitzem el mètode de lectura apropiat per a cada tipus de dada.

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class EmpleatsAleatori {
    public static void main(String[] args) throws IOException{
        RandomAccessFile f = new
        RandomAccessFile("Empleats.dat","r");
        f.seek(56);
        System.out.println("Núm.: "+f.readInt());
        System.out.println("Nom: "+f.readUTF());
        System.out.println("Depart: "+f.readInt());
        System.out.println("Edat: "+f.readInt());
        System.out.println("Sou: "+f.readDouble());
        f.close();
    }
}
```

Ràpidament observem una cosa: com sabíem que ens havíem de situar en la posició 56? I si els noms del dos primers hagueren sigut més llargs o més curts?

Per a poder solucionar els problemes anteriors, podríem fer que els noms siguen de llargària fixa. Les altres dades no donen problemes. Intentarem ara donar sempre una grandària de 10 caràcters a cada nom (si sospitàrem que no en tenim prou, hauríem de fer-los més grans). Anem a crear el fitxer **Empleats2.dat**, i serà exactament igual al de l'anterior apartat, excepte que en el moment de posar els noms (en un array de strings) posem exactament 10 caràcters, omplint amb blancs si és necessari. Evidentment, aquesta no és l'única manera, però per a les poques dades que tenim, sí la més ràpida.

```
String[] noms = {"Andreu    ","Bernat    ","Clàudia   ","Damià     "};
```

La sentència anterior és l'única diferència respecte al programa de creació d'Empleats.dat de la pregunta anterior, a banda del nom del fitxer, que ara serà **Empleat2.dat**:

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


public class CrearEmpleats2 {

    public static void main(String[] args) throws IOException {
        DataOutputStream f = new DataOutputStream(new
        FileOutputStream("Empleats2.dat"));

        String[] noms = {"Andreu    ","Bernat    ","Clàudia
        ","Damià     "};
        int[] departaments = {10,20,10,10};
        int[] edats = {32,28,26,40};
```

```
        double[] sous = {1000.0,1200.0,1100.0,1500.0};

        for (int i=0;i<4;i++){
            f.writeInt(i+1);
            f.writeUTF(noms[i]);
            f.writeInt(departaments[i]);
            f.writeInt(edats[i]);
            f.writeDouble(sous[i]);
        }
        f.close();
    }
}
```

Ara que sabem la grandària exacta del nom, podem saber que la informació de cada empleat és:

Número d'empleat (enter)	4 bytes
Nom (10 caràcters + 2 bytes)	10 + 2 bytes
Departament (enter)	4 bytes
Edat (enter)	4 bytes
Sou (doble precisió)	8 bytes
Total:	**32 bytes**

Sabent que cada registre (la informació de cada empleat) ocupa 32 bytes, sembla fàcil anar a un determinat empleat. Per a poder provar-lo bé, introduirem el número d'empleat per teclat, fins introduir 0. Observeu que si s'introdueix 1, hem d'anar a pel primer registre, que està a principi de fitxer. Si introduïm 2, anem a pel segon, que només en té un davant, per tant 32 bytes.

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

public class EmpleatsAleatori2 {
    public static void main(String[] args) throws IOException {
        RandomAccessFile f = new
        RandomAccessFile("Empleats2.dat","rw");
        Scanner sc = new Scanner(System.in);
        System.out.println("Quin registre? (-1 per a eixir): ");
        int num = sc.nextInt();
        while (num != -1) {
            f.seek(32*(num-1));
            System.out.println("Núm.: " + f.readInt());
            System.out.println("Nom: " + f.readUTF());
            System.out.println("Depart: " + f.readInt());
            System.out.println("Edat: " + f.readInt());
            System.out.println("Sou: " + f.readDouble());
            System.out.println();
            System.out.println("Quin registre? (-1 per a eixir):
            ");
            num = sc.nextInt();
        }
        f.close();
    }
}
```

El problema ara també sembla obvi. Hem assumit que cada caràcter ocupa un byte. Com es codificarà en UTF-8, mentre siguen caràcters normals així serà. Però què passarà quan hi haja un caràcter accentuat? Que ocuparà 2 caràcters. Així, com Clàudia té un d'aquestos caràcters, la cadena no ocuparà 10+2 = 12 bytes, sinó 13. Aleshores, si intentem anar al quart empleat, ens donarà problemes.

La manera de solucionar-lo serà escriure de manera que tots els caràcters ocupen sempre el mateix. Hi ha un mètode que ens ho permet: **writeChars**. Guardarà cada caràcter amb dos bytes, i no es guardarà la llargària de la cadena. Podem intentar utilitzar-lo per construir el fitxer **Empleats3.dat**. Només haurem de substituir la següent sentència:

```
f.writeChars(noms[i]);
```

a banda del nom del fitxer, clar:

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

public class CrearEmpleats3 {
    public static void main(String[] args) throws IOException {
```

```
        DataOutputStream f = new DataOutputStream(new
FileOutputStream("Empleats3.dat"));
        String[] noms = {"Andreu    ","Bernat    ","Clàudia   ","Damià     "};
        int[] departaments = {10,20,10,10};
        int[] edats = {32,28,26,40};
        double[] sous = {1000.0,1200.0,1100.0,1500.0};
        for (int i=0;i<4;i++){
            f.writeInt(i+1);
            f.writeChars(noms[i]);
            f.writeInt(departaments[i]);
            f.writeInt(edats[i]);
            f.writeDouble(sous[i]);
        }
        f.close();
    }
}
```

Lamentablement la lectura no és tan fàcil. Haurem de llegir exactament 10 caràcters (podríem utilitzar algun altre mètode, però el que es mostra permet identificar clarament que es llegiran 10 caràcters). Els anirem posant en un array de caràcters, i després el convertim a String per poder mostrar-lo. S'utilitza el constructor (**new String(char[])**), perquè el mètode **toString** d'un array de caràcters no va tan fi. Recordeu també que ara el nom ocupa 20 bytes, que sumats als altres 20 de les altres dades fan un total de 40 bytes per registre.

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

public class EmpleatsAleatori3 {
    public static void main(String[] args) throws IOException {
        RandomAccessFile f = new
        RandomAccessFile("Empleats3.dat","rw");
        Scanner sc = new Scanner(System.in);
        System.out.println("Quin registre? (-1 per a eixir): ");
        int num = sc.nextInt();
        char[] nom = new char[20];
        while (num != -1) {
            f.seek(40*(num-1));
            System.out.println("Núm.: " + f.readInt());
            for (int i=0;i<10;i++)
                nom[i]=f.readChar();
            System.out.println("Nom: " + new String(nom));
            System.out.println("Depart: " + f.readInt());
            System.out.println("Edat: " + f.readInt());
            System.out.println("Sou: " + f.readDouble());
            System.out.println();
            System.out.println("Quin registre? (-1 per a eixir):
            ");
            num = sc.nextInt();
        }
        f.close();
    }
}
```

3.- Seriació d'objectes

Seriació d'objectes

La tècnica de la **seriació** és segurament la més senzilla de totes, però també a la vegada la més problemàtica. Java disposa d'un sistema genèric de seriació de qualsevol objecte, un sistema **recursiu** que es repeteix per cada objecte contingut a la instància que s'està seriant. Aquest procés para en arribar als tipus primitius, els quals es guarden com una sèrie de bytes. A banda dels tipus primitius, Java serialitza també molta informació addicional o metadades específiques de cada classe (el nom de les classe, els noms dels atributs i molta més informació addicional). Gràcies a les metadades es fa possible automatitzar la seriació de forma genèrica **amb garanties de recuperar un objecte tal com es va guardar**.

Lamentablement, **aquest és un procediment específic de Java**. És a dir, no és possible recuperar els objectes seriats des de Java utilitzant un altre llenguatge. D'altra banda, el fet de guardar metadades pot arribar a comportar també problemes, encara que utilitzem sempre el llenguatge Java. La modificació d'una classe pot fer variar les seues metadades. Aquestes variacions poden donar problemes de recuperació d'instàncies que hagen estat guardades amb algunes versions anteriors a la modificació, impedint que l'objecte puga ser recuperat.

Aquestes consideracions desestimen aquesta tècnica per guardar objectes de forma més o menys permanent. En canvi, la seua senzillesa la fa una perfecta candidata per a l'emmagatzematge temporal, per exemple dins de la mateixa sessió.

Per a que un objecte puga ser seriat cal que la seua classe i tot el seu contingut implementen la interfície **Serializable**. Es tracta d'una interfície sense mètodes, perquè l'únic objectiu de la interfície és actuar de marcador per indicar a la màquina virtual quines classes es poden seriar i quines no.

Totes les classes equivalents als tipus bàsics ja implementen Serializable. També implementen aquesta interfície la classe String i tots els contenidors i els objectes Array. La seriació de col·leccions depèn en últim terme dels elements continguts. Si aquestos són seriables, la col·lecció també ho serà.

En cas que la classe de l'objecte que s'intente seriar, o les d'algun dels objectes que continga, no implementaren la interfície Serializable, es llançaria una excepció de tipus **NotSerializableException**, impedint l'emmagatzematge.

Els Streams **ObjectInputStream** i **ObjectOutputStream** són decoradors que afegeixen a qualsevol altre Stream la capacitat de seriar qualsevol objecte Serializable. El stream d'eixida disposarà del mètode **writeObject**. i el stream d'entrada, el mètode de lectura **readObject**.

El mètode readObject només permet recuperar instàncies que siguen de la mateixa classe que la que es va guardar. En cas contrari, es llançaria una excepció de tipus **ClassCastExeception**. A més, cal que l'aplicació dispose del codi compilat de la classe; si no fóra així, l'excepció llançada seria **ClassNotFoundException**.

Exemple

Ens recolzarem en un exemple basat en els anteriors, en els empleats. Ara anem a suposar que els empleats són objectes, i intentarem guardar aquestos objectes en un fitxer amb una seriació.

El primer pas serà construir la classe **Empleat**, que contindrà la mateixa informació que en els altres apartats: número d'empleat, nom, departament, edat i sou. A banda de les propietats per a cadascuna de les dades anteriors (que seran del mateix tipus que en les altres ocasions), també farem el constructor que inicialitza l'objecte, així com els mètodes **get** per a cadascuna de les propietat. No ens caldran els mètodes set, ja que en els exemples sempre utilitzarem el constructor. Observeu com la classe ha de ser **serialitzable**.

```
import java.io.Serializable;
```

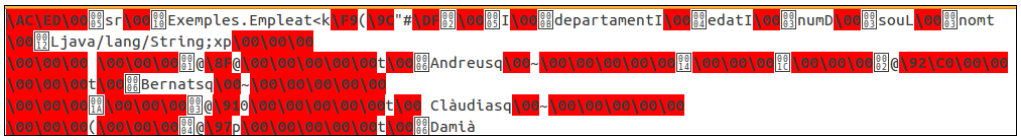
```
public class Empleat implements Serializable {
    private int num=0;
    private String nom=null;
    private int departament=0;
    private int edat=0;
    private double sou=0.0;

    public Empleat(){
    }

    public Empleat(int num, String nom, int dep, int edat, double sou){
        this.num=num;
        this.nom=nom;
        this.departament=dep;
        this.edat=edat;
        this.sou=sou;
    }

    public int getNum(){
        return this.num;
    }

    public String getNom(){
        return this.nom;
    }

    public int getDepartament(){
        return this.departament;
    }

    public int getEdat(){
        return this.edat;
    }

    public double getSou(){
        return this.sou;
    }
}
```

Anem a intentar construir el fitxer. El flux de dades serà un **ObjectOutputStream** per a poder escriure (**writeObject**). I observeu com s'ha de recolzar en un OutputStream, que en aquest cas serà d'un fitxer, és a dir un **FileOutputStream**. A cada iteració del bucle senzillament construirem un objecte de la classe **Empleat** i l'escriurem al fitxer.

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


public class CrearEmpleatsObjecte {

    public static void main(String[] args) throws IOException {
        ObjectOutputStream f = new ObjectOutputStream(new
        FileOutputStream("Empleats.obj"));

        String[] noms = {"Andreu","Bernat","Clàudia","Damià"};
        int[] departaments = {10,20,10,10};
        int[] edats = {32,28,26,40};
        double[] sous = {1000.0,1200.0,1100.0,1500.0};
        Empleat e;

        for (int i=0;i<4;i++){
            e = new
            Empleat(i+1,noms[i],departaments[i],edats[i],sous[i]);
            f.writeObject(e);
        }

        f.close();
    }
}
```

Nota

El fitxer creat, **Empleats.obj**, evidentment no és de text. Tanmateix si l'obrim amb un editor de text podrem veure alguna cosa.



- La primera qüestió és que es guarda el nom de la classe amb el nom del paquet davant. **Exemples.Empleat** és realment el nom de la classe creada.
- Es guarden també els noms dels camps. Tot això són les metadades que havíem comentat, i que permeten la recuperació posterior dels objectes guardats
- I després ja podem veure la informació guardada, on identifiquem els noms dels empleats

Per a llegir el fitxer creat, **Empleats.obj**, utilitzarem el **ObjectInputStream** per a poder fer **readObject**. S'ha de basar en un InputStream, que en aquest cas serà un **FileInputStream**. El tractament de final de fitxer el farem capturant l'excepció (l'error) d'haver arribat al final i intentat llegir encara: **EOFException**. La raó és que **readObject** no torna null, a no ser que s'haja introduït aquest valor. Per tant muntem un bucle infinit, però capturant amb **try ... catch** l'error, que és quan tancarem el Stream.

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

public class ConsultarEmpleatsObjecte {

    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        ObjectInputStream f = new ObjectInputStream(new
        FileInputStream("Empleats.obj"));

        Empleat e;
        try {
            while (true) {
                e = (Empleat) f.readObject();
                System.out.println("Número: " + e.getNum());
                System.out.println("Nom: " + e.getNom());
                System.out.println("Depart: " +
                e.getDepartament());
                System.out.println("Edat: " + e.getEdat());
                System.out.println("Sou: " + e.getSou());
                System.out.println();
            }
        } catch (EOFException eof) {
            f.close();
        }
    }
}
```

4.- Documents XML

Hem vist que la manera més còmoda de guardar objectes és amb la seriació, per mig del **ObjectInputStream** i **ObjectOutputStream**, però que **fora** de Java **no és possible** l'accés a aquestes dades. I també dins de Java podem tenir problemes, perquè el nom de la classe amb el nom del paquet es guarda en el fitxer com a metades, i en un altre programa haurem de tenir la classe creada en un paquet amb el mateix nom, sinó no es podran recuperar les dades.

També hem vist que per a guardar dades individuals de diferents tipus ens van molt bé les classes **DataInputStream** i **DataOutputStream**, però haurem de saber molt bé l'ordre i tipus de dades que estan guardades, sinó, no les podrem recuperar.

I no entrem ja en la possibilitat que diferents Sistemes Operatius representen la informació de forma diferent (per exemple, hi ha Sistemes Operatius que representen els números amb BCD i altres que utilitzen complement a 2).

Per tant, quan vulguem guardar dades que puguen ser llegides per aplicacions fetes en diferents llenguatges i/o executades en diferents plataformes, ens farà falta un format estàndar que tots el puguen entendre i reconéixer, i millor si és autoexplicatiu com és el cas dels **llenguatges de marques**.

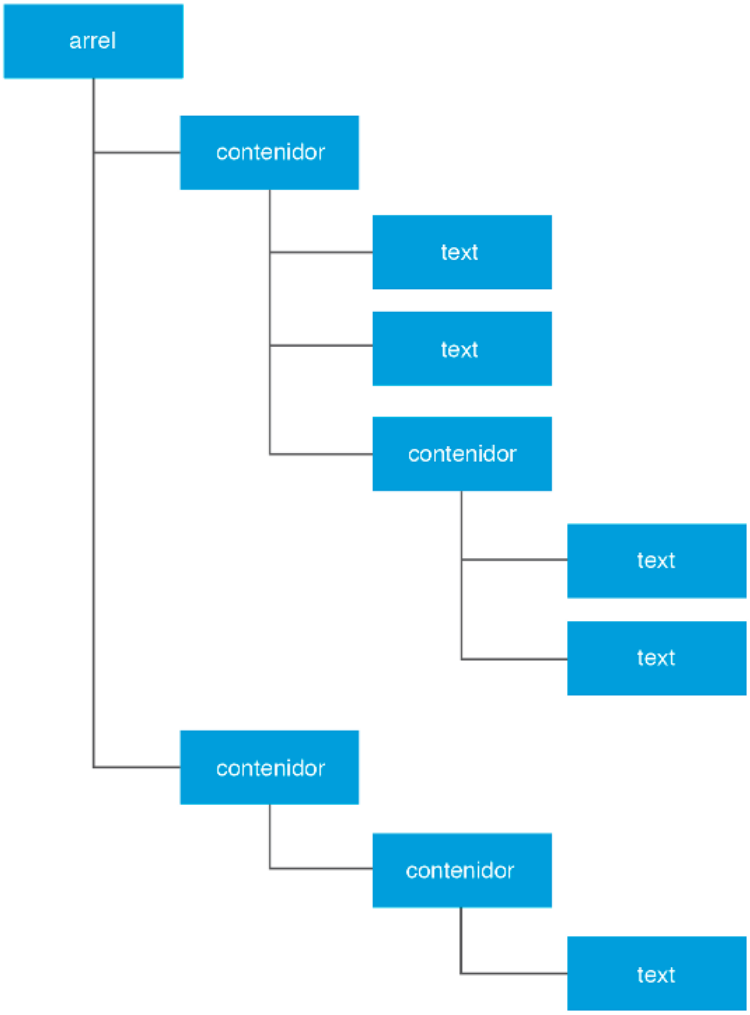
El llenguatge de marques més conegut i més utilitzat és el **XML** (***eXtensible Markup Language***)

Els documents XML aconsegueixen estructurar la informació intercalant unes marques anomenades etiquetes, cada etiqueta amb un principi i un final, i que poden anar unes dins d'unes altres, i també contenir informació de text. D'aquesta manera, es podrà subdividir la informació estructurant-la de forma que puga ser fàcilment interpretada.

Tota la informació serà de text, i per tant no hi haurà el problema mencionat abans de representar les dades de diferent manera. Qualsevol dada, ja siga numèrica, booleana o com siga, es posarà en mode text, i per tant sempre es podrà llegir i interpretar correctament tota la informació continguda en un fitxer XML.

És cert que els caràcters es poden escriure utilitzant diferents sistemes de codificació, però XML ofereix diverses tècniques per evitar que això siga un problema, com per exemple, incloent a la capçalera del fitxer quina codificació s'ha fet servir en el moment de guardar-lo.

Amb les etiquetes, XML aconsegueix estructurar qualsevol tipus d'informació jeràrquica. Es pot establir certa similitud entre la forma com la informació es guarda en els objectes d'una aplicació i la forma com es guardaria en un document XML. La informació, en les aplicacions orientades a objectes, s'estructura, agrupa i jerarquitza en classes, i en els documents XML s'estructura, organitza i jerarquitza en etiquetes contingudes unes dins les altres i atributs de les etiquetes.



Imaginem que volem representar les dades dels empleats com els de l'aparat anterior utilitzant un format XML. No existeix una única solució, però cal que totes respecten la jerarquia del model. Un possible format podria ser el següent:

```
<empresa>
    <empleat>
        <num>1</num>
        <nom>Andreu</nom>
        <departament>10</departament>
        <edat>32</edat>
        <sou>1000.0</sou>
    </empleat>

    <empleat>
        <num>2</num>
        <nom>Bernat</nom>
        <departament>20</departament>
        <edat>28</edat>
        <sou>1200.0</sou>
    </empleat>

    <empleat>
        <num>3</num>
        <nom>Clàudia</nom>
        <departament>10</departament>
        <edat>26</edat>
        <sou>1100.0</sou>
    </empleat>

    <empleat>
```

```
        <num>4</num>
        <nom>Damià</nom>
        <departament>10</departament>
        <edat>40</edat>
        <sou>1500.0</sou>
    </empleat>
</empresa>
```

Però aquesta també podria ser una manera de representar-lo:

```
<empresa>
    <empleat num='1'
    nom='Andreu' departament='10' edat='32' sou='1000.0'/>
    <empleat num='2'
    nom='Bernat' departament='20' edat='28' sou='1200.0'/>
    <empleat num='3'
    nom='Clàudia' departament='10' edat='26' sou='1100.0'/>
    <empleat num='4'
    nom='Damià' departament='10' edat='40' sou='1500.0'/>
</empresa>
```

I podem imaginar moltes altres solucions, combinant considerant com atributs o subetiquetes les diferents característiques dels empleats que volem guardar. Aquest seria el problema principal del XML, les múltiples solucions. Però també és veritat que totes elles són fàcils d'entendre.

4.1 - Parser o analitzador XML

Un ***Parser* XML** és una classe que ens permet analitzar i classificar el contingut d'un arxiu XML extraient la informació continguda en cada una de les etiquetes, i relacionar-la d'acord amb la seua posició dins la jerarquia.

Hi ha dos tipus d'analitzadors depenent de la manera de funcionar.

Analitzadors seqüencials

Els **analitzadors seqüencials** permeten extreure el contingut a mida que es van descobrint les etiquetes d'obertura i tancament. També s'anomenen **analitzadors sintàctics**. Són analitzadors molt ràpids, però presenten el problema que cada vegada que es necessita accedir a una part del contingut, s'ha de rellegir tot el document de dalt a baix.

En Java, l'analitzador sintàctic més popular s'anomena **SAX**, que vol dir **Simple API for XML**. És una analitzador molt utilitzat en diverses biblioteques de tractament de dades XML, però no sol utilitzar-se en aplicacions finals, pel problema abans comentat d'haver de llegir-se tot el document XML a cada consulta. Per aquesta raó no els veurem en aquest curs.

Analitzadors jeràrquics

Generalment, les aplicacions finals que han de treballar amb dades XML solen utilitzar analitzadors jeràrquics.

Els **analitzadors jeràrquics** guarden totes les dades del document XML en memòria dins una estructura jeràrquica, a mida que van analitzant el seu contingut. I per això són ideals per a aplicacions que requereixen una consulta contínua de les dades.

El **format de l'estructura** on es guarda la informació en memòria ha estat especificat per l'organisme internacional W3C (World Wide Web Consortium) i es coneix com a **DOM** (***Document Object Model***). És una estructura que HTML i javascript han popularitzat molt i es tracta d'una especificació que Java materialitza en forma d'interfícies. La principal s'anomena ***Document*** i representa tot un document XML. En tractar-se d'una interfície, pot ser implementada per diverses classes.

L'estàndard W3C defineix la classe **DocumentBuilder** (constructor de documents) per a poder crear estructures DOM a partir d'un XML. Aquesta classe **DocumentBuilder** és una classe abstracta, i per tal que es puga adaptar a les diferents plataformes, pot necessitar fonts de dades o requeriments diversos. Recordeu que les classes abstractes no es poden instanciar de forma directa. Per aquest motiu, el consorci W3 especifica també la classe **DocumentBuilderFactory**, és a dir, el *fabricador* de *DocumentBuilder*.

Les llibreries des d'on importarem les classes comentades són:

- **DocumentBuilderFactory** i **DocumentBuilder** les importarem de la llibreria **javax.xml.parsers.***
- **Document** l'importarem de **org.w3c.dom.***

Hem de cuidar sobretot aquesta última importació, perquè per defecte Java ens ofereix moltes llibreries des d'on importar **Document**. I si no la importem de la llibreria correcta, evidentment després tindrem errors.

Les instruccions necessàries per llegir un fitxer XML i crear un objecte **Document** serien les següents:

```
import java.io.FileInputStream;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

...

    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
```

```
    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

    Document doc = dBuilder.parse(new File("fitxer.xml"));
```

Tornem a insistir en la necessitat d'importar **Document** de la llibreria **org.w3c.dom.***

Anem a basar-nos en un exemple per poder veure a poc a poc la manera d'utilitzar el parser. És un exemple que possiblement es va veure en primer, en el mòdul de Llenguatge de Marques. Suposarem que està en el fitxer **cotxes.xml**, i que està en el directori del projecte on farem les proves.

```
<?xml version="1.0" encoding="UTF-8"?>
<oferta>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="gasolina">duratorc 1.4</motor>
    <matricula>1234AAA</matricula>
    <kilometros>12500</kilometros>
    <precio_inicial>12000</precio_inicial>
    <precio_oferta>10000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="300">llantas</extra>
    <foto>11325.jpg</foto>
    <foto>11326.jpg</foto>
  </vehiculo>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="diesel">duratorc 2.0</motor>
    <matricula>1235AAA</matricula>
    <kilometros>125000</kilometros>
    <precio_inicial>10000</precio_inicial>
    <precio_oferta>9000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="200">spoiler trasero</extra>
    <extra valor="500">climatizador</extra>
    <foto>11327.jpg</foto>
    <foto>11328.jpg</foto>
  </vehiculo>
</oferta>
```

El primer que farem serà intentar connectar amb aquest fitxer, però d'una forma un poc més reduïda que abans, sense definir objectes del **DocumentBuilderFactory** ni **DocumentBuilder**. Tampoc ens caldrà definir-nos el **File** (**FileInputStream**) ja que el mètode **parse** també agafa un **String** com a paràmetre:

```
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

...

    Document doc =
DocumentBuilderFactory.newInstance().newDocumentBuilder().parse("cotxes.xml");
```

Però, i si el procés que necessitem és l'invers? És a dir, i si el que volem és guardar una estructura DOM en un fitxer XML?

En aquest cas el que haurem de fer serà construir un **document buit**, anar posant els elements i atributs (amb els seus valors) d'alguna manera, i posteriorment guardar-lo en un fitxer. Deixem per a un poc més avant com anar construint els nodes del document i centrem-nos en el fet de crear el document buit i guardar-lo en un fitxer. Podem contruir un document nou a partir del **DocumentBuilder**, utilitzant el mètode **newDocument()** :

```
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

...

    Document doc1 =
DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
```

Per a escriure la informació continguda al DOM a un fitxer, es pot fer utilitzantr una altra utilitat de Java anomenada

Transformer. Es tracta d'una utilitat que permet realitzar fàcilment conversions entre diferents representacions d'informació jeràrquica. És capaç, per exemple, de passar la informació continguda en un objecte **Document a un fitxer de text en format XML**. També seria capaç de fer l'operació inversa, però no val la pena perquè el mateix *DocumentBuilder* ja s'encarrega d'això.

Transformer és també una classe abstracta i requereix d'una *factory* per poder ser instanciada. La classe **Transformer** pot treballar amb multitud de contenidors d'informació perquè en realitat treballa amb un parell de tipus adaptadors (classes que fan compatibles jerarquies diferents) que s'anomenen **Source** i **Result**. Les classes que implementen aquestes interfícies s'encarregaran de fer compatible un tipus de contenidor específic al requeriment de la classe **Transformer**. Així, disposem de les classes **DOMSource**, **SAXSource** o **StreamSource** com a adaptadors del contenidor de la font d'informació (DOM, SAX o Stream respectivament). **DOMResult**, **SAXResult** o **StreamResult** són els adaptadors equivalents del contenidor destí. A nosaltres ara, com que volem passar un document DOM a un fitxer, ens convindrà un **DOMSource** i un **StreamResult**

El codi bàsic per realitzar una transformació de DOM a fitxer de text XML seria el següent:

```
Transformer trans = TransformerFactory.newInstance().newTransformer();

DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(file);

trans.transform(source, result);
```

De tota manera, veurem més avant un exemple on ens guardarem una estructura DOM en un fitxer XML.

4.2 - L'estructura DOM

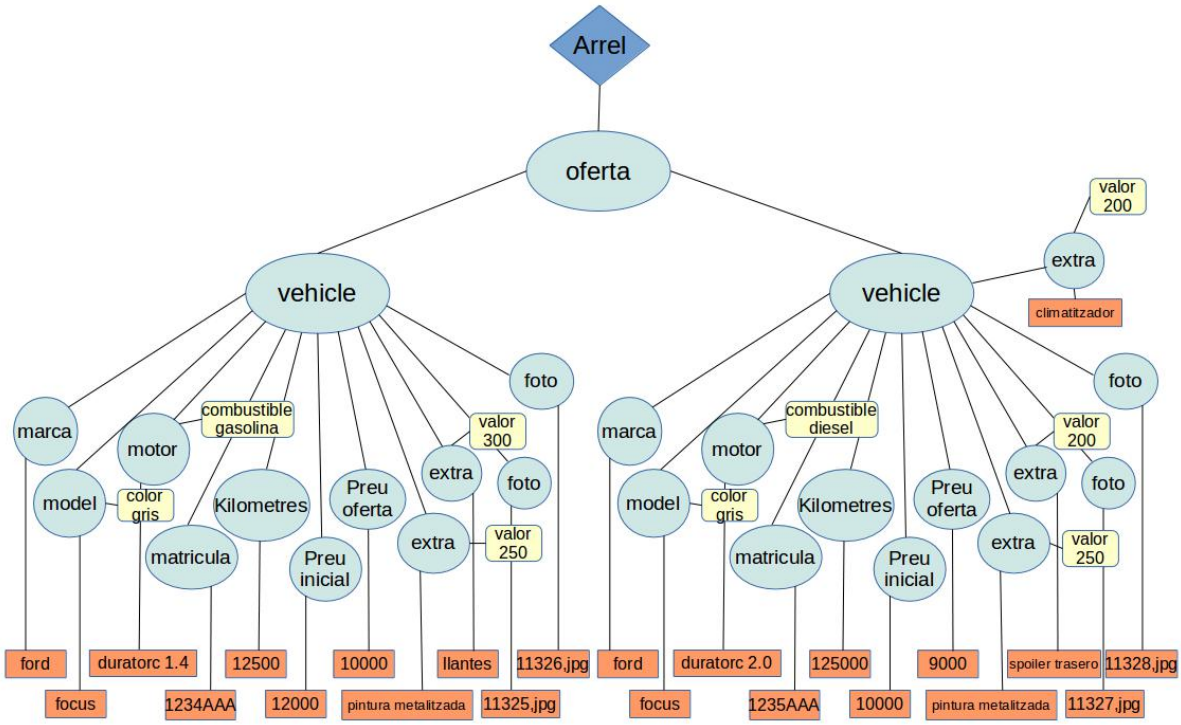
L'estructura DOM pren la forma d'un arbre, on cada part del XML es trobarà representada en forma de **node**. En funció de la posició en el document XML, parlarem de diferents tipus de nodes:

- El node principal que representa tot el XML sencer s'anomena **Document.**
- Les diverses etiquetes, inclosa l'etiqueta arrel, es coneixen com a nodes **Element**.
- El contingut d'una etiqueta de tipus text, serà un node de tipus **TextElement**
- Els atributs seran nodes de tipus **Attribute**.

En l'exemple de la pàgina anterior, el fitxer **cotxes.xml**, que té aquesta estructura:

```
<?xml version="1.0" encoding="UTF-8"?>
<oferta>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="gasolina">duratorc 1.4</motor>
    <matricula>1234AAA</matricula>
    <kilometros>12500</kilometros>
    <precio_inicial>12000</precio_inicial>
    <precio_oferta>10000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="300">llantas</extra>
    <foto>11325.jpg</foto>
    <foto>11326.jpg</foto>
  </vehiculo>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="diesel">duratorc 2.0</motor>
    <matricula>1235AAA</matricula>
    <kilometros>125000</kilometros>
    <precio_inicial>10000</precio_inicial>
    <precio_oferta>9000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="200">spoiler trasero</extra>
    <extra valor="500">climatizador</extra>
    <foto>11327.jpg</foto>
    <foto>11328.jpg</foto>
  </vehiculo>
</oferta>
```

Veient-lo com una estructura jeràrquica ens quedaria així:



On:

- El node **Document** és el rombe
- Els nodes **Element** són els cercles
- Els nodes **TextElement** són els rectàngles de fons taronja.
- Els nodes **Attribute** són els rectàngles arrodonits de color groc

Cada node específic disposa de mètodes per accedir a les seues dades concretes (nom, valor, nodes fills, node pare, etc.). És a dir, que el **node** serveix per a situar-se en una determinada posició (element, atribut, element de text, ...). Tindrà uns mètodes, sobretot per a navegar, encara que també alguns per a traure el contingut. **Element** és un classe derivada de **Node** (per tant hereta tots els seus mètodes), i proporciona algunes coses més, sobretot per a accedir còmodament a les seues parts. Mirem els mètodes més importants, tant de **Node** com de **Element** i **Document**

Mètodes de NODE

Valor tornat	Mètode	Descripció
String	**getNodeName()**	torna el nom d'aquest node
short	**getNodeType()**	torna el tipus d'aquest node (ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, ...
String	**getNodeValue()**	torna el valor del node, si en té
NodeList	**getChildNodes()**	torna una llista amb els nodes fills
Node	**getFirstChild()**	torna el primer fill
Node	**getLastChild()**	torna l'últim fill
NamedNodeMap	**getAttributes()**	torna una llista amb els atributs del node (null si no en té cap)
Node	**getParentNode()**	torna el pare
String	**getTextContent()**	torna el text contingut en l'element i el de tots els seus descendents, si en té
boolean	**hasChildNodes()**	torna cert si el node té algun fill
boolean	**hasAttributes()**	torna cert si el node té algun atribut

Mètodes d' **ELEMENT**

Valor tornat	Mètode	Descripció
String	**getAttribute(**String nom**)**	torna el valor de l'atribut que té aquest nom
NodeList	**getElementsByTagName(**String nom**)**	torna una llista de nodes amb tots els descendents que tenen aquest nom
boolean	**hasAttribute(**String nom**)**	torna cert si l'element té aquest atribut

Mètodes de DOCUMENT

Valor tornat	Mètode	Descripció
Element	**getDocumentElement()**	torna l'element arrel del document
NodeList	**getElementsByTagName(**String nom**)**	torna una llista de nodes amb tots els descendents que tenen aquest nom

Sempre que tinguem una **llista de nodes**, podrem accedir a cadascun dels membres de la llista amb el mètode **item** especificant el número d'ordre. Així, si volem accedir al primer posarem **item(0)**

Posterioment posarem els mètodes que serveixen per anar posant contingut a un document: crear fills, crear atributs, posar contingut, ...

El DOM resultant obtingut des d'un XML acaba sent un còpia exacta del fitxer, però disposat de diferent manera. Tant al XML com al DOM hi haurà informació no visible, com ara els *retorns de carro*, que cal tenir en compte per tal de saber processar correctament el contingut i evitar sorpreses poc comprensibles.

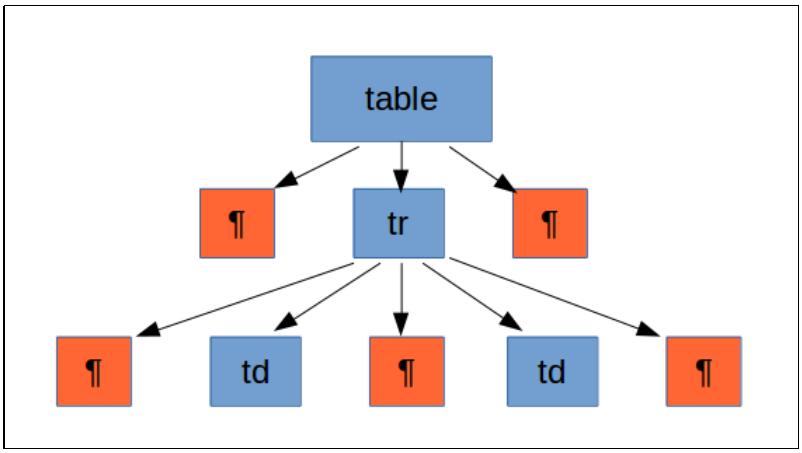
Per a il·lustrar el problema que poden suposar els retorn de carro, imaginem que disposem d'un document XML amb el següent contingut:

```
<table>
   <tr>
      <td> </td>
      <td> </td>
   </tr>
</table>
```

Veurem més clar si representem els retorn de carro en el mateix document:

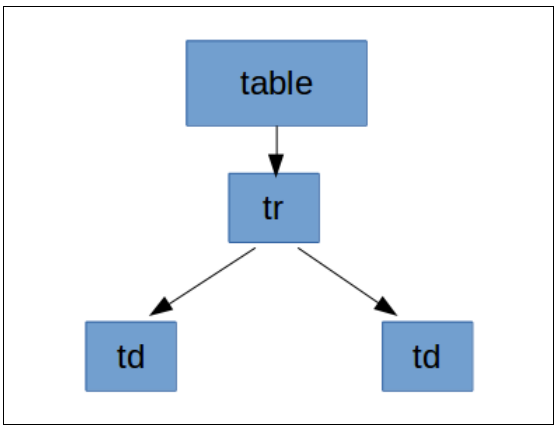
```
<table>¶
   <tr>¶
      <td> </td>¶
      <td> </td>¶
   </tr>¶
</table>
```

A la següent figura es mostra la representació que tindria l'objecte DOM, un vegada estiga ja copiat en memòria. Observeu com l'element ***table*** tindrà tres fills. En un es guardarà el **retorn de carro** que situa l'etiqueta ***<tr>*** a la següent línia, en el segon trobarem l'etiqueta ***<tr>***, i en el tercer el retorn de carro que fa que **</table>** estiga en la línia de baix. El mateix passa amb els fills de ***<tr>***, abans i després de cada node ***<td>*** trobarem un **retorn de carro**.



En canvi, si haguérem partit d'un XML equivalent però sense retorns de carro, el resultat hauria estat diferent:

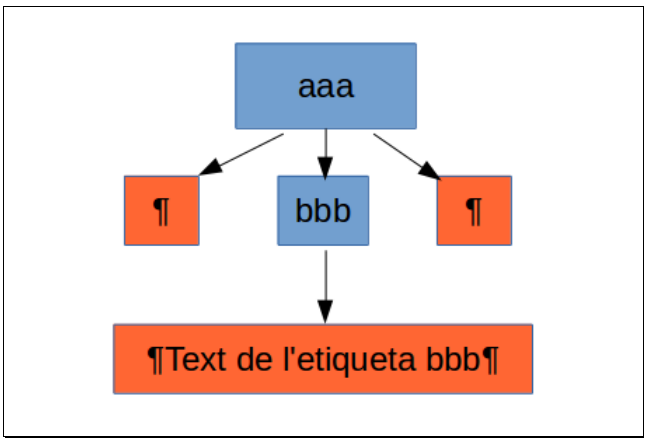
```
<table><tr><td></td><td></td></tr></table>
```



L'absència de retorns de carro en el fitxer implica també l'absència de nodes contenint els retorns de carro en l'estructura DOM.

Un altre aspecte a tenir en compte és que el contingut de les etiquetes es plasma en el DOM com un node fill de l'etiqueta contenidora. És a dir, per obtenir el text d'una etiqueta cal obtenir el primer fill d'aquesta.

```
<aaa>
   <bbb>
      text de l'etiqueta bbb
   </bbb>
</aaa>
```



4.2.1 Lectura

Anem a fer proves per poder comprovar el funcionament. Ens basem en el document **cotxes.xml** esmentat en la pregunta 4.1

> **Nota**
>
> Aneu amb compte, perquè en el document **cotxes.xml** davant de la primera etiqueta no pot haver ni retorn de carro ni un espai en blanc ni res

```
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.SAXException;


public class MirarXML {

   public static void main(String[] args) throws
   ParserConfigurationException, FileNotFoundException, IOException,
   SAXException {
      Document doc =
      DocumentBuilderFactory.newInstance().newDocumentBuilder().parse("cotxes.xml");
      System.out.println(doc.getNodeName());     // torna el nom del
      document. No és l'element arrel. Ens dirà #document
      Element arrel = doc.getDocumentElement();  // apuntarà a l'element
      arrel
      System.out.println(arrel.getNodeName());   // torna el nom de
      l'element. Ens dirà oferta
      System.out.println(arrel.getNodeValue());  // torna el valor de
      l'element. Com és un element que conté altres elements, el valor és
      null
   }
}
```

Tal i com està documentat, aquesta serà l'eixida:

```
#document
oferta
null
```

Anem a comprovar ara que el primer fill de **oferta** no és **vehiculo** sinó el retorn de carro. Els element **vehiculo** són el segon i el quart (índex 1 i 3).

```
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;


public class MirarXML2 {

    public static void main(String[] args) throws
ParserConfigurationException, FileNotFoundException, IOException, SAXException
{
        Document doc =
DocumentBuilderFactory.newInstance().newDocumentBuilder().parse("cotxes.xml");
        Element arrel = doc.getDocumentElement();  // apuntarà a l'element
arrel
        NodeList fills = arrel.getChildNodes();
        System.out.println(fills.item(0).getNodeName());   // el primer fill
és el retorn de carro.
        System.out.println(fills.item(1).getNodeName());   // el segon fill sí
```

```
que és vehiculo
        System.out.println(fills.item(2).getNodeName());   // el tercer fill
és el retorn de carro.
        System.out.println(fills.item(3).getNodeName());   // el quart fill sí
que és vehiculo
        System.out.println(fills.item(4).getNodeName());   // el cinquè fill
és el retorn de carro.
        System.out.println(fills.item(5).getNodeName());   // no existeix el
sisè fill. Donarà error
        }
}
```

Observeu que en l'última sentència estem provocant un error:

```
#text
vehiculo
#text
vehiculo
#text
Exception in thread "main" java.lang.NullPointerException
    at Exemples.MirarXML2.main(MirarXML2.java:25)
```

Per tant, hem d'anar molt en compte amb els ***retorns de carro***.

- Per a poder esquivar els *retorn de carro* podríem mirar el tipus de cada node (**getNodeType()**), menysprear els els de tipus ***TEXT_NODE*** i considerar només els de tipus ***ELEMENT_NODE***.
- Però normalment l'accés que farem serà un poc més directe i més fàcil. Agafarem la llista de tots els element que tinguen un determinat nom amb **getElementsByTagName(*nom*)** . Evidentment en la llista no estaran els retorns de carro i així no tindrem problemes amb ells.

En el següent exemple recorrerem tots els element **vehiculo**. De cadascun agafarem el contingut dels elements **marca** i **matricula**. També agafem el contingut de l'atribut **combustible** de l'element **motor**:

```
public class MirarXmlCotxes {

    public static void main(String[] args) throws
ParserConfigurationException, SAXException, IOException {

        Document doc =
DocumentBuilderFactory.newInstance().newDocumentBuilder()
                .parse(new FileInputStream("cotxes.xml"));
        Element arrel = (Element) doc.getChildNodes().item(0);
        NodeList llista = arrel.getElementsByTagName("vehiculo");

        for (int i = 0; i < llista.getLength(); i++) {
            Element el = (Element) llista.item(i);
            System.out.println(el.getNodeName() + " " + (i + 1));
            System.out.println("Marca: " +
el.getElementsByTagName("marca").item(0).getChildNodes().item(0).getNodeValue());
            System.out.println("Matrícula: " +
el.getElementsByTagName("matricula").item(0).getFirstChild().getNodeValue());
            System.out.println("Motor: " +
el.getElementsByTagName("motor").item(0).getTextContent());
            System.out.println("Combustible: " +
el.getElementsByTagName("motor").item(0).getAttributes().item(0).getNodeValue());
            Element m = (Element) el.getElementsByTagName("motor").item(0);
            System.out.println("Combustible: " +
m.getAttribute("combustible"));
            System.out.println();
        }
        System.out.println(arrel.getTextContent());
    }
}
```

És molt important observar que quan tenim un element que ja té contingut, la informació no és accessible, sinó que hem d'anar al primer fill, que aquest ja és de tips ***TEXT_NODE***, per agafar el seu valor.

En l'exemple:

- Per a **marca** hem agafat de tota la llista de fills el primer, per traure el seu valor.
- En **matricula** en compte d'agafar tota la llista de fills, només hem agafat el primer, i per tant és més ràpid.
- I per a **motor** utilitzem el mètode **getTextContent**, que agafa el congingut de text de l'element i de tots els seus descendents. Com és un node de text ja sabem a priori que ens anirà bé, i per tant **és la forma més ràpida**.

L'atribut **combustible** de l'element **motor** l'hem tret de 2 maneres:

- La primera agafant la llista d'atributs, i després el primer d'aquesta llista.
- En la segona manera s'ha fet més elegant, anat a buscar la propietat en qüestió. Per això hem convertit el node en l'**element m**, per a poder utilitzar **getAttribute**.

Al final fem el **getTextContent()** sobre l'arrel per a comprovar que trau el seu contingut i el de tots els seus fills, per això apareix la informació duplicada

Aquest serà el resultat de l'exemple anterior:

```
vehiculo 1
Marca: ford
Matrícula: 1234AAA
Motor: duratorc 1.4
Combustible: gasolina
Combustible: gasolina

vehiculo 2
Marca: ford
Matrícula: 1235AAA
Motor: duratorc 2.0
Combustible: diesel
Combustible: diesel



    ford
    focus
    duratorc 1.4
    1234AAA
    12500
    12000
    10000
    pintura metalizada
    llantas
    11325.jpg
    11326.jpg


    ford
    focus
    duratorc 2.0
    1235AAA
    125000
    10000
    9000
    pintura metalizada
    spoiler trasero
    climatizador
    11327.jpg
    11328.jpg
```

4.2.2 Escriptura

Anem ara a crear un nou document XML i a guardar-lo en un fitxer. Utilitzarem com a exemple **Empleats**. Al final de tot convertirem el fitxer **Empleats.obj**, generat en la pregunta 3, en el fitxer **Empleats.xml**.

La primera consideració a fer és que partirem d'un document buit. Anirem construint els elements i posant els atributs, i quan tinguem un element creat del tot, l'afegirem a l'estructura, és a dir farem que siga el fill d'un que ja està en l'estructura. Podríem fer-ho també al revés, és a dir, primer penjar-lo de l'estructura i després anar omplint-lo.

Els principals mètodes per anar construint l'estructura són:

Mètodes de DOCUMENT

Valor tornat	Mètode	Descripció
Element	**createElement(**String nom**)**	crea un nou element amb el nom indicat (s'haurà de penjar en l'estructura)
Text	**createTextNode(**String dades**)**	crea un nou element de text (amb contingut)
Node	**appendChild(**Node nou**)**	afegeix el node nou, que serà l'arrel

Mètodes de NODE

Valor tornat	Mètode	Descripció
Node	**appendChild(**Node nou**)**	afegeix el node nou com a l'últim fill fins el moment
void	**removeChild(**Node vell**)**	lleva el node vell com a fill

Mètodes de ELEMENT

Valor tornat	Mètode	Descripció
void	**setAttribute(**String nom,String valor**)**	afegeix un nou atribut a l'element, amb el nom i valor indicats
void	**removeAttribute(**String nom**)**	lleva l'atribut de l'element

Anem a fer directament ja l'exemple dels empleats. Totes les dades seran elements, excepte el número d'empleat, que farem que siga un atribut d'empleat per a practicar. A l'element arrel li direm **empleats.** El resultat ha de ser el fitxer **Empleats.xml**.

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

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;


public class CrearEmpleatsXML {
   public static void main(String[] args) throws IOException,
   ClassNotFoundException, ParserConfigurationException, TransformerException
   {
      ObjectInputStream f = new ObjectInputStream(new
      FileInputStream("Empleats.obj"));
```

```
        Document doc =
        DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element arrel = doc.createElement("empleats");
        doc.appendChild(arrel);

        Empleat e;
        try {
           while (true) {
              e = (Empleat) f.readObject();
              Element emp = doc.createElement("empleat");
              emp.setAttribute("numero", Integer.toString(e.getNum()));

              Element fill = doc.createElement("nom");
              fill.appendChild(doc.createTextNode(e.getNom()));
              emp.appendChild(fill);

              fill = doc.createElement("departament");
              fill.appendChild(doc.createTextNode(Integer.toString(e.getDepartament())));
              emp.appendChild(fill);

              fill = doc.createElement("edat");
              fill.appendChild(doc.createTextNode(Integer.toString(e.getEdat())));
              emp.appendChild(fill);

              fill = doc.createElement("sou");
              fill.appendChild(doc.createTextNode(Double.toString(e.getSou())));
              emp.appendChild(fill);

              arrel.appendChild(emp);
           }

        } catch (EOFException eof) {
              f.close();
        }
        Transformer trans = TransformerFactory.newInstance().newTransformer();

        DOMSource source = new DOMSource(doc);
        StreamResult result = new StreamResult(new
        FileOutputStream("Empleats.xml"));

        trans.transform(source, result);
     }
}
```

En el fitxer XML generat, observareu que no hi ha retorns de carro, tot està en una mateixa línia. Si voleu veure'l bé, el podeu obrir per exemple amb un navegador web, que interpreta bé el format XML.

<?xml version="1.0" encoding="UTF-8" standalone="no"?><empleats><empleat numero="1"><nom>Andreu</nom><departament>10</departament><edat>32</edat><sou>1000.0</sou></empleat><empleat numero="2"><nom>Bernat</nom><departament>20</departament><edat>28</edat><sou>1200.0</sou></empleat><empleat numero="3"><nom>Clàudia</nom><departament>10</departament><edat>26</edat><sou>1100.0</sou></empleat><empleat numero="4"><nom>Damià</nom><departament>10</departament><edat>40</edat><sou>1500.0</sou></empleat></empleats>

Aquest fitxer XML no sembla que tingui cap informació d'estil associada.

```
−<empleats>
 −<empleat numero="1">
    <nom>Andreu</nom>
    <departament>10</departament>
    <edat>32</edat>
    <sou>1000.0</sou>
  </empleat>
 −<empleat numero="2">
    <nom>Bernat</nom>
    <departament>20</departament>
    <edat>28</edat>
    <sou>1200.0</sou>
  </empleat>
 −<empleat numero="3">
    <nom>Clàudia</nom>
    <departament>10</departament>
    <edat>26</edat>
    <sou>1100.0</sou>
  </empleat>
 −<empleat numero="4">
    <nom>Damià</nom>
    <departament>10</departament>
    <edat>40</edat>
    <sou>1500.0</sou>
  </empleat>
</empleats>
```

4.3 - Binding

VOLUNTARI

El ***Binding*** és una tècnica que consisteix en generar (i vincular) **automàticament** classes de Java amb formats específics d'emmagatzematge, que en el cas que ens interessa seran documents XML.

D'aquesta manera, cada etiqueta o atribut de XML es correspondrà amb una propietat d'una determinada classe. Això s'anomena ***mapar***, perquè és fer una espècie de mapa per a indicar que una etiqueta (o atribut) es correspon a una propietat de la classe. Per a poder fer la aquesta correspondència o ***mapatge*** de forma correcta entre les classes de Java i el document XML, farà falta una miqueta d'ajuda o aclaracions, perquè com ja hem vist la mateixa informació es pot guardar de més d'una manera en XML (utilitzant o no atributs, ...).

En Java existeixen diverses biblioteques per gestionar el *binding*, com per exemple *JAXB*, *JiBX*, *XMLBinding*, etc. Des de la versió 6.0 s'ha incorporat en el JDK estàndard **JAXB** (***Java Architecture for XML Binding***), una potent biblioteca.

JAXB utilitza ***Anotacions*** per aconseguir la informació extra necessària per *mapar* el *binding*. Les *Anotacions* són unes classes de Java molt especials. Serveixen per associar informació i funcionalitat als objectes sense interferir en l'estructura del model de dades. Abans d'aparéixer les *Anotacions* era necessari fer servir l'herència per poder afegir funcionalitat a una classe sense haver de codificar-la, però ho complicava molt.

Si, per contra, fem servir *Anotacions*, els objectes disposaran d'informació o de funcionalitat extra sense que el model de dades quede modificat, ja que les *Anotacions* no són visibles des dels objectes. Les *Anotacions* poden associar-se a un paquet, a una classe, a un atribut o fins i tot a un paràmetre. Aquestes classes especials es declaren en el codi de l'aplicació posant el símbol **@** davant del nom de l'*Anotació*. Quan el compilador de Java detecta una *Anotació* crea una instància i la injecta dins l'element afectat (paquet, classe, mètode, atribut, etc.). Això fa que aquestes no apareguen com a atributs o mètodes propis de l'objecte, i per això diem que no interacciona amb el model de dades, però les aplicacions que ho necessiten poden obtenir la instància injectada i fer-la servir.

Nosaltres en aquest curs, únicament aspirarem a veure com realment a partir d'un **esquema XML** que valida una sèrie de documents XML, podem generar còmodament les classes en Java, que permetrien guardar objectes d'aquestes classes. Una vegada construïdes les classes es podria utilitzar JAXB per a transferir informació dels fitxers XML (vàlids per a l'esquema) cap als objectes o a l'inrevès.

La generació de les classes és extremadament senzilla a partir d'una determinada versió d'Eclipse. Com hem comentat la llibreria JAXB està present des de la versió 6.0 del JDK SE (Standard Edition). Però per a la generació còmoda ens fan falta eines (tools) proporcionades en la JDK EE 7.0 (Enterprise Edition). Si tenim aquesta plataforma podrem utilitzar la versió 2.2 de JAXB.

> **Nota**
>
> Lamentablement, a partir de la versió de Java JDK EE 8.0 comença a estar en desús el JAXB (*deprecated*), i a la versió 11 ja no està implementat. Per tant, per a poder provar el que va a continuació **és preferible que utilitzeu una versió antiga de Java** (Java 7 o Java 8). Si no la teniu instal·lada, no valdrà la pena que feu aquest exemple. **Per això aquesta pregunta és voluntària**.
>
> Evidentment, es pot aconseguir que funcione en Java 11, a pesar que no està implementat JAXB, però és a costa d'assenyalar que agafe configuracions anteriors, o important tots els jar necessaris, baixant-los prèviament. No valdrà la pena, perquè no l'utilitzarem

Ens recolzarem en un exemple utilitzat l'any passat. El següent esquema (.xsd) valida documents com el de l'oferta de vehicles, vist en apartats anteriors. Podeu guardar-lo amb el nom **cotxes.xsd**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
<xs:simpleType name="Combust">
  <xs:restriction base="xs:string">
      <xs:enumeration value="gasolina"/>
      <xs:enumeration value="diesel"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="Matr">
  <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{4}[A-Z]{3}"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="oferta">
  <xs:complexType>
    <xs:sequence>
        <xs:element name="vehiculo" maxOccurs="unbounded" >
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="marca" type="xs:string"/>
                    <xs:element name="modelo" >
                        <xs:complexType>
                            <xs:simpleContent>
                                <xs:extension base="xs:string">
                                    <xs:attribute name="color"
type="xs:string" />
                                </xs:extension>
                            </xs:simpleContent>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="motor" >
                        <xs:complexType>
                            <xs:simpleContent>
                                <xs:extension base="xs:string">
                                    <xs:attribute name="combustible"
type="Combust" />
                                </xs:extension>
                            </xs:simpleContent>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="matricula" type="Matr"/>
                    <xs:element name="kilometros" type="xs:integer"/>
                    <xs:element name="precio_inicial"  type="xs:integer"/>
                    <xs:element name="precio_oferta" type="xs:integer"/>
                    <xs:element name="extra" maxOccurs="unbounded" >
                        <xs:complexType>
                            <xs:simpleContent>
                                <xs:extension base="xs:string">
                                    <xs:attribute name="valor"
type="xs:integer" />
                                </xs:extension>
                            </xs:simpleContent>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="foto" maxOccurs="unbounded"
type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Per exemple, el següent document XML **cotxes.xml** (el vist en els apartats anteriors) és vàlid segons l'anterior esquema.

```
<?xml version="1.0" encoding="UTF-8"?>
<oferta  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation ="ejercicio7_1.xsd">
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="gasolina">duratorc 1.4</motor>
    <matricula>1234AAA</matricula>
    <kilometros>12500</kilometros>
    <precio_inicial>12000</precio_inicial>
    <precio_oferta>10000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="300">llantas</extra>
    <foto>11325.jpg</foto>
    <foto>11326.jpg</foto>
```

```
  </vehiculo>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="diesel">duratorc 2.0</motor>
    <matricula>1235AAA</matricula>
    <kilometros>125000</kilometros>
    <precio_inicial>10000</precio_inicial>
    <precio_oferta>9000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="200">spoiler trasero</extra>
    <extra valor="500">climatizador</extra>
    <foto>11327.jpg</foto>
    <foto>11328.jpg</foto>
  </vehiculo>
</oferta>
```

Però per a generar les classes el que ens fa falta és el de l'esquema, **cotxes.xsd**. Si l'incorporem a un projecte **JAXB** serà molt fàcil de generar les classes

En el següent vídeo es veu aquesta generació de les classes a partir de l'esquema. **No cal que ho feu si no voleu, perquè es mostra a nivell únicament il·lustratiu de com es generen les classes, però no ho utilitzarem després**

En el vídeo anterior no ha hagut cap problema, per tenir instal·lat per defecte un JDK de Java antic.

Com s'ha comentat més amunt, si tenim unJDK posterior, com podria ser el JDK 11, segurament tindrem problemes. La raó és perquè a partir de Java 11 ja no s'incorporen les classes de JAXB. Quan anem a generar les classes a partir de l'esquema XML (el fitxer .xsd), donarà l'error que no troba les classes JAXB. Podem solucionar-lo de 3 maneres:

- Baixar les classes necessàries. Té l'inconvenient que són moltes, encara que totes són fàcils de trobar.
- Hi ha documentcaió per Internet que podem configurar les dependències del projecte de manera que vaja a buscar les llibreries de JAXB en versions anteriors.

- I per últim, utilitzar un JDK que encara incorpore aquestes llibreries. L'haurem de tenir instal·lat a l'ordinador.

El següent vídeo mostra com es pot solucionar utilitzant aquesta última possibilitat. Torne a recordar que és voluntari, i si no teniu instal·lat el JDK 8 o anterior, no val la pena que ho feu.

5.- Documents JSON

JSON vol dir **J**ava**S**cript **O**bject **Notation**, és a dir Notació d'Objectes de JavaScript. És una manera de representar objectes inicialment per a JavaScript, però per la seua senzillesa, i com és en text pla, serveix per a qualsevol entorn. Permet representar estructures de dades d'una determinada complexitat, com el XML, però pesa molt menys que aquest, i per això està convertint-se en un estàndar d'intercanvi de dades, sobretot entre un servidor i una aplicació web.

L'extensió d'un fitxer JSON és **.json**

5.1 - Estructura JSON

Amb JSON podrem representar:

- **Valors**, de tipus **caràcter** (entre cometes dobles), **numèric** (sense cometes) , **booleà** (true o false) o **null**.
- **Parelles clau valor**, és a dir un nom simbòlic acompanyat d'un valor associat.. Es representen així: **"nom" : valor**
- **Objectes**, que és una col·lecció de membres, cadascú dels quals pot ser una parella clau valor, o altres objectes (fins i tot arrays): es representen entre claus, i amb els elements separats per comes: **{ "nom1" : "valor1" , "nom2": valor2 , valor 3 , ... }**
- **Arrays**, que són llistes d'elements. Els elements no tenen per què tenir la mateixa estructura, però nosaltres intentarem que sí que la tinguen per coherència. Cada element pot ser un valor , una parella clau valor, un objecte o un array.

Veja'm algun exemples:

```
{ "p1" : 2 , "p2" : 4 , "p3" : 6 , "p4" : 8 , "p5" : 10 }
```

en aquest cas tenim un objecte, l'arrel, que té 5 membres, tots ells parelles clau-valor.

```
{
  "num": 1 ,
  "nom": "Andreu" ,
  "departament": 10 ,
  "edat": 32 ,
  "sou": 1000.0
}
```

ara un objecte, l'arrel, també amb 5 membres que són parelles clau-valor. Observeu com la clau sempre la posem entre cometes, i el valor quan és un string també, però quan és numèric, no.

```
{ "empleat" :
  { "num": 1 ,
    "nom": "Andreu" ,
    "departament": 10 ,
    "edat": 32 ,
    "sou": 1000.0
  }
}
```

en aquest cas tenim un objecte, l'arrel que consta d'un únic objecte, **empleat**, el qual consta de 5 membres clau-valor.

Mirem ara un exemple amb un array:

```
{ "notes" :
  [ 5 , 7 , 8 , 7 ]
}
```

on tenim l'element arrel que consta d'un únic membre, notes, que és un array.

També seria correcte d'aquesta manera, per veure que l'element arrel no té perquè ser un objecte, sinó també un array

```
[ 5 , 7 , 8 , 7 ]
```

I ara un més complet amb la mateixa estructura que el fitxer XML que havíem vist en la pregunta 4. Tindrem un objecte arrel, amb només un objecte, **empresa**, que té un únic element **empleat** que és un array amb 4 elements, cadascun dels empleats:

```
{ "empresa":
   { "empleat":
     [  {
           "num": "1",
           "nom": "Andreu",
           "departament": "10",
           "edat": "32",
           "sou": "1000.0"
```

```
        },
        {
           "num": "2",
           "nom": "Bernat",
           "departament": "20",
           "edat": "28",
           "sou": "1200.0"
        },
        {
           "num": "3",
           "nom": "Clàudia",
           "departament": "10",
           "edat": "26",
           "sou": "1100.0"
        },
        {
           "num": "4",
           "nom": "Damià",
           "departament": "10",
           "edat": "40",
           "sou": "1500.0"
        }
     ]
   }
}
```

Anem a veure un parell de casos més reals. Aquesta és la contestació que fa el WebService de **Bicicas** en sol·licitar l'estat actual de bicicletes en els diferents punts (en el moment de fer els apunts es consulta en l'adreça http://gestiona.bicicas.es/apps/apps.php):

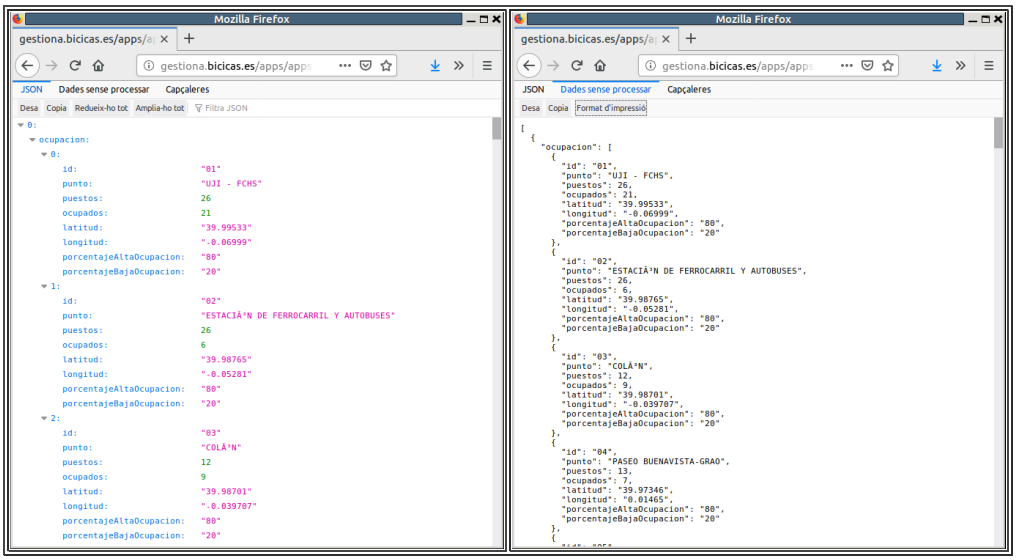
```
[
  {"ocupacion":
    [
      {"id":"01","punto":"UJI -
FCHS","puestos":27,"ocupados":12,"latitud":"39.99533","longitud":"-0.06999",
"porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      {"id":"02","punto":"ESTACIÓN DE FERROCARRIL Y
AUTOBUSES","puestos":24,"ocupados":7,"latitud":"39.98765","longitud":"-
0.05281", "porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      {"id":"03","punto":"PLAZA DE
PESCADERÍA","puestos":28,"ocupados":4,"latitud":"39.98580","longitud":"-
0.03798", "porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      ...
    ]
  }
]
```

Com podeu comprovar, l'arrel no és un objecte, sinó un **Array**. En l'array només ens interessa el primer element que és un objecte amb un únic membre, **ocupacion** (en l'exemple no hi ha més elements, però en poden haver més en un moment determinat, quan volen fer avisos). I **ocupacion és un array**, amb **un objecte per cada estació de bicicas**, amb les parelles clau valor **id**, **punto**, **puestos** (les bicicletes que caben), **ocupados** (quantes bicicletes hi ha col·locades en aquest moment), **latitud** i **longitud** (les coordenades), ...

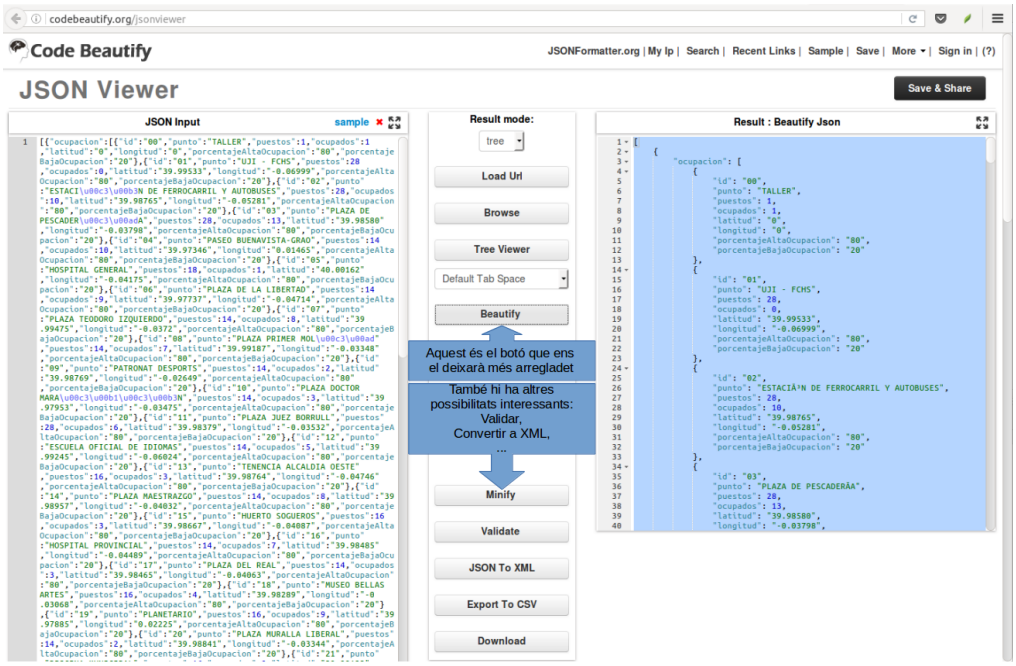
Nota

En realitat ens apareixerà tota la informació molt més apegada, perquè realment està en una única línia.

Per a poder observar millor l'estructura podem utilitzar un **visor** de json. Normalment el navegador Firefox els visualitza bé, encara que també depén de la versió. Si tenim instal·lada una versió que admet la visualització de JSON, ho intentarà interpretar, encara que segurament la millor manera de veure el format JSON és,tiar les opcions **Dades sense processar --> Format d'impressió**, que és la que veiem a la dreta:



Si la versió nostra de Firefox no visualitza el format JSON, podem buscar un visor dels molts que hi ha per internet. En la figura n'hem utilitzat un, i es pot observar com facilita molt la lectura.



Un altre exemple. Un WebService de GeoNames (una Base de Dade geogràfica gratuïta i accessible a través d'Internet) ens proporciona informació dels llocs que troba dins d'un rectangle delimitat per un latitud al nord i al sud, i una longitud a l'esti a l'oest (en l'exemple: nord 40.01, sud 39.9, est 0.1 i oest -0.1). Per exemple, http://api.geonames.org/citiesJSON?north=40.01&south=39.99&east=0.01&west=-0.01&lang=ES&username=demo torna el següent:


```
{
  "geonames": [
    {
      "lng": -0.04935,
      "geonameId": 2519752,
      "countrycode": "ES",
      "name": "Castelló de la Plana",
      "fclName": "city, village,...",
      "toponymName": "Castelló de la Plana",
      "fcodeName": "seat of a second-order administrative division",
      "wikipedia": "en.wikipedia.org/wiki/Castell%C3%B3n_de_la_Plana",
      "lat": 39.98567,
      "fcl": "P",
      "population": 180005,
```

```
      "fcode": "PPLA2"
    },
    {
      "lng": -0.06313,
      "geonameId": 2521909,
      "countrycode": "ES",
      "name": "Almazora",
      "fclName": "city, village,...",
      "toponymName": "Almassora",
      "fcodeName": "populated place",
      "wikipedia": "en.wikipedia.org/wiki/Almassora",
      "lat": 39.94729,
      "fcl": "P",
      "population": 24963,
      "fcode": "PPL"
    },
    ...
    ]
}
```

A partir de l'arrel (que ara sí que és un objecte), tenim un membre: **geonames**, que és un array (un element per cada "lloc" trobat), on cada element té informació diversa, com el nom del lloc, les coordenades, la població, ...

Nota

De fa uns mesos que Googel limita el servei anterior, i ha de ser amb un usuari validat. No valdrà la pena, per al poc profit que li trauríem. Mostrem en què consisteix el servei únicament a nivell il·lustratiu

5.2 - API Simple JSON

Hi ha més d'una llibreria per a poder accedir i analitzar els documetns json.

Les que més es comenten per Internet són **GSON** (de Google) i **Jackson**. Però nosaltres anem a utilitzar una altra, per la seua senzillesa: **JSON.simple**.

En el moment de fer aquestos apunts l'última versió estable és la 3.1.0. En aquest enllaç teniu d'on us la podeu baixar: https://jar-download.com/artifacts/com.github.cliftonlabs/json-simple. Una vegada baixada l'haurem d'incorporar al nostre projecte (**Project --> Properties --> Java Build Path --> Libraries --> Add Externals JARs**)

En ella trobem el més bàsic:

- **JsonObject**: equivaldrà a un objecte
- **JsonArray**: equivaldrà a un array
- **Jsoner**: contindrà tota l'altra funcionalitat, com l'analitzador (mètode **deserialize()**), que agafarà com a entrada el document Json, i tornarà l'objecte arrel. També té el mètode **serialize()** que fa el procés invers.

Versió 1.1.1

Ja des de la versió 2 van canviar prou les coses respecte de les versions anteriors. Són compatibles les classes de versions anteriors, però estan marcades com a **deprecated**.

En aquestes notes de color blau us posaré com es feien. **Evidentment, si utilitzeu la nova versió no cal que feu les corresponents a versions anteriors**.

Si utilitzem la versió 1.1.1, la llibreria és json-simple-1.1.1.jar

Els objectes en aquella versió eren (observeu els canvis de majúscules i minúscules):

- **JSONObject**: equivaldrà a un objecte
- **JSONArray**: equivaldrà a un array
- **JSONParser**: l'analitzador, que agafarà com a entrada el document Json amb el mètode **parse**, i tornarà l'objecte arrel

Jsoner

El primer que haurem de fer serà analitzar el document per a obtenir l'element arrel (que com hem vist normalment serà un objecte, però de vegades pot ser un array).

El mètode **deserialize** de **Jsoner** admet com a paràmetre un **string** amb tot el document json, però també admet un **Reader**, que normalment serà el més còmode.

En el següent exemple agafem el json directament d'un **string**, on teníem els números parells.

```
{ "p1" : 2 , "p2" : 4 , "p3" : 6 , "p4" : 8 , "p5" : 10 }
```

Observeu com per a la definició del string ens ha tocat escapar les dobles cometes.

```
import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.Jsoner;

public class Exemple_3_51 {

    public static void main(String[] args) throws JsonException {
        String cadena = "{ \"p1\" : 2 , \"p2\" : 4 , \"p3\" : 6 , \"p4\" : 8 ,
\"p5\" : 10 }";

        JsonObject arrel = (JsonObject) Jsoner.deserialize(cadena);
```

```
        System.out.println(arrel.get("p1"));
    }
}
```

Mentre que en aquest exemple accedim al fitxer **parelles.json** (que assumirem que té el mateix contingut). Podríem passar el contingut del fitxer a una cadena, però és molt més còmode passar el **Reader** com paràmetre, i ell s'encarrega de llegir-lo tot.

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

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.Jsoner;


public class Exemple_3_52 {

    public static void main(String[] args) throws JsonException, IOException
{

        Reader r_json = new FileReader("parelles.json");

        JsonObject arrel = (JsonObject) Jsoner.deserialize(r_json);

        System.out.println(arrel.get("p1"));
    }
}
```

JSONParser

En la versió 1.1.1, hem d'utilitzar el mètode **parse** de **JSONParser** per a analitzar. També admetia com a paràmetre un **string** o un **Reader**.

Ara el primer exemple quedarà:

```
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class Exemple_3_51 {

    public static void main(String[] args) throws ParseException
{
        String cadena = "{ \"p1\" : 2 , \"p2\" : 4 , \"p3\" : 6
, \"p4\" : 8 , \"p5\" : 10 }";

        JSONParser parser = new JSONParser();
        JSONObject arrel = (JSONObject) parser.parse(cadena);

        System.out.println(arrel.get("p1"));
    }
}
```

Mentre que el segon, el que llig del fitxer **parelles.json**, queda així:

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

import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class Exemple_3_52 {

    public static void main(String[] args) throws IOException,
ParseException {
        Reader r_json = new FileReader("parelles.json");

        JSONParser parser = new JSONParser();
        JSONObject arrel = (JSONObject) parser.parse(r_json);

        System.out.println(arrel.get("p2"));
    }
```

```
}
```

JsonObject

Contindrà un **objecte json**. Recordeu que l'**objecte** es delimita amb les claus: **{ }**

El mètode més important serà **get(*clau*)** al qual li passem la clau del membre que volem obtenir. Si el membre és una parella clau-valor, obtindrem directament el valor. Si és un altre objecte, doncs obtindrem l'objecte o també podria ser un array. El cas que siga una parella clau-valor, ja l'hem vist en l'exemple anterior.

En el següent exemple, en el fitxer **empleat.json** tindrem un empleat

```
{ "empleat" :
  { "num": 1 ,
    "nom": "Andreu" ,
    "departament": 10 ,
    "edat": 32 ,
    "sou": 1000.0
  }
}
```

Observeu com ara de l'element arrel hem d'agafar **empleat**, que és un objecte amb les parelles clau-valor **num**, **nom**, ...

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

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.Jsoner;

public class Exemple_3_53 {

    public static void main(String[] args) throws IOException, JsonException {
        Reader r_json = new FileReader("empleat.json");

        JsonObject arrel = (JsonObject) Jsoner.deserialize(r_json);

        JsonObject empleat = (JsonObject) arrel.get("empleat");

        System.out.println(empleat.get("nom") + " (" + empleat.get("sou") +
")");
    }
}
```

> En la versió 1.1.1, l'objecte es deia **JSONObject** (observeu el canvi de majúscules)

JSONArray

Serà l'array, que recordem que ve limitat per els claudàtors: **[ ]**

Per a obtenir els elements de l'array utilitzarem el mètode **get(*index*)**, on l'índex és el número d'ordre de l'element que volem obtenir.

Posem el primer exemple sobre un json que només té un array els elements del qual són valors:

```
[ 5 , 7 , 8 , 7 ]
```

Observeu com ara l'arrel és un array, i ens muntem un bucle per obtenir tots els elements.

```
import com.github.cliftonlabs.json_simple.JsonArray;
import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.Jsoner;

public class Exemple_3_54 {

    public static void main(String[] args) throws JsonException {
        String cadena = "[ 5 , 7 , 8 , 7 ]";
```

```
        JsonArray arrel = (JsonArray) Jsoner.deserialize(cadena);

        for (int i=0; i<arrel.size();i++)
            System.out.println(arrel.get(i));
    }
}
```

També podem fer servir els bucles foreach, però lamentablement no podem obtenir directament un JsonObject, i per tant l'hem de reconvertir. Ho mostrarem en el següent exemple en el qual agafem tots els empleats de l'empresa, on **empresa.json** és així:

```
{ "empresa" :
    { "empleats":
            [
        { "num":1, "nom": "Andreu", "departament": 10, "edat": 32, "sou":
1000.0}  ,
        { "num":2, "nom": "Bernat", "departament": 20, "edat": 28, "sou":
1200.0}  ,
        { "num":3, "nom": "Clàudia", "departament": 10, "edat": 26, "sou":
1100.0}  ,
        { "num":4, "nom": "Damià", "departament": 10, "edat": 40, "sou":
1500.0}
       ]
    }
}
```

Tindrem un objecte arrel, amb només un objecte, **empresa**, que té un únic membre empleat que és un array amb 4 elements, cadascun dels empleats

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

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.Jsoner;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.JsonArray;

public class Exemple_3_55 {

    public static void main(String[] args) throws IOException, JsonException {
        Reader r_json = new FileReader("empresa.json");

        JsonObject arrel = (JsonObject) Jsoner.deserialize(r_json);

        JsonObject empresa = (JsonObject) arrel.get("empresa");

        JsonArray empleats = (JsonArray) empresa.get("empleats");

        for (Object e : empleats) {
            JsonObject emp = (JsonObject) e;
            System.out.println(emp.get("nom") + " (" + emp.get("sou") + ")");
        }
    }
}
```

> En la versió 1.1.1, l'objecte es deia **JSONArray** (observeu el canvi de majúscules)

5.3 - Accés complet des de Java

Com ja havíem vist en els últims exemples del punt anterior, la qüestió es tracta d'anar agafant objectes i arrays, per l'estructura del JSON.

Mirem algun exemple ja més elaborat, on ens tocarà analitzar amb detall l'estructura json. Fem-lo sobre l'exemple de **BICICAS**. Podeu tornar a fer la consulta de l'estat actual en aquest moment a la pàgina http://gestiona.bicicas.es/apps/apps.php , seleccionar-ho tot i guardar-lo en el fitxer **bicicas.json**. És molt possible que tinguem problemes amb els caràcers especials, com les vocals accentuades, a causa de que el navegador utilitzat no les reconega, i en copiar-les al fitxer no tinguem ja la codificació correcta. No li donarem importància en aquest moment. Recordem ací l'estructura:

```
[
  {"ocupacion":
    [  {"id":"01","punto":"UJI -
FCHS","puestos":28,"ocupados":11,"latitud":"39.99533","longitud":"-0.06999",
       "porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      {"id":"02","punto":"ESTACIÓN DE FERROCARRIL Y
AUTOBUSES","puestos":28,"ocupados":8,"latitud":"39.98765",
       "longitud":"-
0.05281","porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      {"id":"03","punto":"PLAZA DE
PESCADERÍA","puestos":28,"ocupados":13,"latitud":"39.98580","longitud":"-
0.03798",
       "porcentajeAltaOcupacion":"80","porcentajeBajaOcupacion":"20"},
      ...
    ]
  }
]
```

Com podem observar, comença per un **array**, no per un objecte, com sol ser habitual. L'únic que ens interessa és el **primer element** de l'array, ja que en posteriors anirien en tot cas missatges. El primer element és un **objecte** que té un únic membre **ocupacion** (o si en té més no ens interessen), que és un **array** amb totes les estacions. Cada estació és un objecte amb la informació que ens interessa.

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

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.Jsoner;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.JsonArray;


public class mirarBicicas {

    public static void main(String[] args) throws IOException, JsonException {
        JsonArray arrel = (JsonArray) Jsoner.deserialize(new
FileReader("bicicas.json"));
        JsonObject obj = (JsonObject) arrel.get(0);
        JsonArray estacions = (JsonArray) obj.get("ocupacion");

        for (int i=0; i<estacions.size();i++){
            JsonObject e = (JsonObject) estacions.get(i);
            System.out.println(e.get("id") + ".- " + e.get("punto") + " (" +
e.get("ocupados") + "/" + e.get("puestos") + ")");
        }
    }
}
```

Hem utilitzat la manera habitual, meny liosa però més llarga: agafar l'array, d'ell agafar el primer objecte i d'ell agafar l'array. I hem utilitzat un bucle **for**, amb tantes iteracions com la grandària de l'array.

Crear estructures JSON

Ens falta veure com escriure un document JSON. És molt fàcil anar creant l'estructura JSON. Només hem de recordar la manera d'afegir al **JsonObject** i al **JsonArray**.

- **put("nom", element)** per afegir un nou membre a un **JsonObject**
- **add(element)** per afegir un nou element a un **JsonArray**

Una vegada tinguem l'estructura, podrem passar-la a un **string** amb el mètode **toJson**, i la podrem guardar directament en un fitxer, per exemple.

Farem més d'un exemple, per poder practicar.

En el primer exemple generarem un JSON a partir de dades definides en el mateix programa, per mig de vectors. Intentarem generar aquest fitxer JSON:

```
{ "empresa":
   { "empleat":
     [  {
           "num": "1",
           "nom": "Andreu",
           "departament": "10",
           "edat": "32",
           "sou": "1000.0"
        },
        {
           "num": "2",
           "nom": "Bernat",
           "departament": "20",
           "edat": "28",
           "sou": "1200.0"
        },
        {
           "num": "3",
           "nom": "Clàudia",
           "departament": "10",
           "edat": "26",
           "sou": "1100.0"
        },
        {
           "num": "4",
           "nom": "Damià",
           "departament": "10",
           "edat": "40",
           "sou": "1500.0"
        }
     ]
   }
}
```

Analitzem l'estructura. Tenim un objecte arrel, que consta d'un únic membre, empresa, que és un objecte. Aquest objecte té un únic membre que és un array. Cada element de l'array és un objecte, i els seus membres ja són clau-valor.

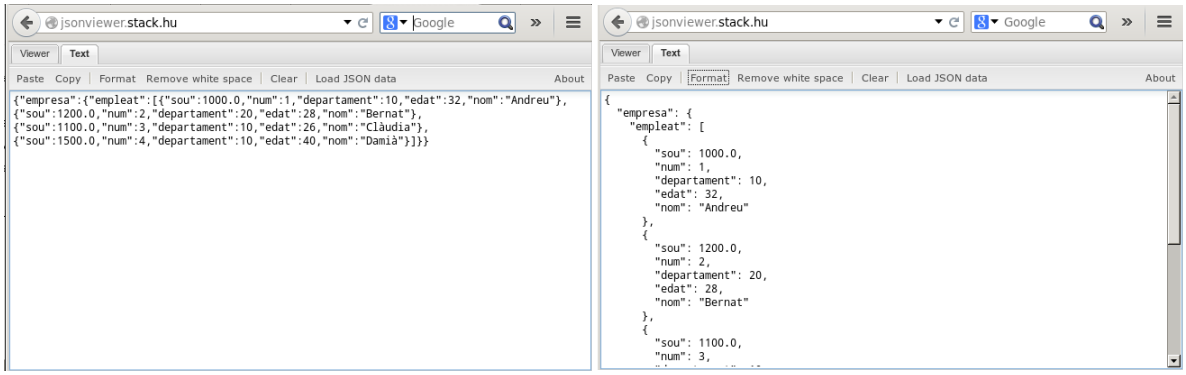
```
import java.io.FileWriter;
import java.io.IOException;

import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.JsonArray;

public class transformarEmpleats {

    public static void main(String[] args) throws IOException {
        String[] noms = {"Andreu","Bernat","Clàudia","Damià"};
        int[] departaments = {10,20,10,10};
        int[] edats = {32,28,26,40};
        double[] sous = {1000.0,1200.0,1100.0,1500.0};

        JsonObject arrel = new JsonObject();
        JsonObject empresa = new JsonObject();
        arrel.put("empresa", empresa);
        JsonArray empleats = new JsonArray();
        empresa.put("empleat", empleats);

        for (int i=0;i<4;i++){
            JsonObject emp = new JsonObject();
            emp.put("num", i+1);
            emp.put("nom", noms[i]);
            emp.put("departament", departaments[i]);
            emp.put("edat", edats[i]);
            emp.put("sou", sous[i]);
            empleats.add(emp);
        }

        FileWriter f = new FileWriter("Empleats.json");
        f.write(arrel.toJson());
        f.close();
    }
```

```
}
```

El resultat és el següent:

```
{"empresa":{"empleat":
[{"num":1,"sou":1000.0,"nom":"Andreu","edat":32,"departament":10},
{"num":2,"sou":1200.0,"nom":"Bernat","edat":28,"departament":20},
{"num":3,"sou":1100.0,"nom":"Clàudia","edat":26,"departament":10},
{"num":4,"sou":1500.0,"nom":"Damià","edat":40,"departament":10}]}}
```

Per a poder-ho estudiar millor, tornem a utilitzar un **visor JSON** online, en aquesta ocasió un altre diferent a l'utilitzat en la pregunta 5.1.



I un altre problema és que els membres de cada empleat estan **desordenats**. No és que haja eixit malament, perquè hem de recordar que un objecte JSON és un conjunt no ordenat de membres.

Si volem que apareguen ordenats, és tan senzill com substituir el **JsonObject** que volem que aparega ordenat per un **LinkedHashMap**. Aquest és el fragment de programa substituït:

```
        for (int i=0;i<4;i++){
            Map emp = new LinkedHashMap();
            emp.put("num", i+1);
            emp.put("nom", noms[i]);
            emp.put("departament", departaments[i]);
            emp.put("edat", edats[i]);
            emp.put("sou", sous[i]);
            empleats.add(emp);
        }
```

Un altre exemple, el de **Bicicas**, però donant-li una altra estructura. Per tant llegirem l'original, i anirem construint la següent estructura:

```
{ "bicicas":
    [ {"num":"01","nom":"UJI - FCHS","llocs":28,"ocupats":11,"lliures":17},
      {"num":"02","nom":"ESTACIÓN DE FERROCARRIL Y
AUTOBUSES","llocs":28,"ocupats":8,"lliures":20},
      {"num":"03","nom":"PLAZA DE
PESCADERÍA","llocs":28,"ocupats":13,"lliures":15},
      ...
    ]
  }
]
```

Per a veure més possiblitats hem utilitzat ara la manera de fer-lo en una línia, que com hem de fer cast (posar el tipus) a cada moment, queda prou engorrós en quant a parèntesis. I hem utilitzat ara el **foreach** per a recórrer l'array.

Els mètodes per a afegir són:

- Afegir a un objecte: **put(clau,valor)**
- Afegir a un array: **add(valor)**

on valor pot ser també un objecte, ...

Observeu sobretot la manera tan senzilla d'escriure el fitxer JSON. No cal fer cap transformació com en el cas de XML. Senzillament utilitzem el mètode de convertir JSON a string, i ho escrivim en el fitxer.

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

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

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.Jsoner;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.JsonArray;

public class transformarBicicas {

    public static void main(String[] args) throws IOException, JsonException {
        Reader r_json = new FileReader("bicicas.json");

        JsonArray estacions = (JsonArray) ((JsonObject) ((JsonArray )
Jsoner.deserialize(r_json)).get(0)).get("ocupacion");

        JsonArray destEstacions = new JsonArray();

        for (Object est : estacions){
            JsonObject e = (JsonObject) est;
            JsonObject destE = new JsonObject();
            destE.put("num", e.get("id"));
            destE.put("nom", e.get("punto"));
            destE.put("llocs", e.get("puestos"));
            destE.put("ocupats", e.get("ocupados"));
            int lliures = Integer.parseInt(e.get("puestos").toString())-
Integer.parseInt(e.get("ocupados").toString());
            destE.put("lliures", lliures );
            destEstacions.add(destE);
        }

        JsonObject bicicas = new JsonObject();
        bicicas.put("bicicas",destEstacions);

        Writer w_json = new FileWriter("bicicas2.json");
        w_json.write(bicicas.toJson());
        w_json.close();
    }
}
```

El resultat seria aquest (he tabulat per a una millor visualització):

```
{"bicicas":
[{"num":"01","llocs":28,"ocupats":11,"lliures":17,"nom":"UJI - FCHS"},
{"num":"02","llocs":28,"ocupats":8,"lliures":20,"nom":"ESTACIÓN DE FERROCARRIL
Y AUTOBUSES"},
{"num":"03","llocs":28,"ocupats":13,"lliures":15,"nom":"PLAZA DE PESCADERÍA"},
{"num":"04","llocs":14,"ocupats":5,"lliures":9,"nom":"PASEO BUENAVISTA-GRAO"},
{"num":"05","llocs":14,"ocupats":7,"lliures":7,"nom":"HOSPITAL GENERAL"},
...
]}
```

L'únic problema és que ha eixit desordenat: l'array està ordenat, però cada objecte no ha eixit amb l'ordre que havíem creat, i així el nom de cada objecte ha eixit al final.

Si volem que ens aparega ordenat, utilitzarem **LinkedHashMap**, igual que en l'exemple anterior.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.io.Writer;
import java.util.LinkedHashMap;
import java.util.Map;

import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.Jsoner;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.JsonArray;

public class transformarBicicas2 {

    public static void main(String[] args) throws IOException, JsonException {
        Reader r_json = new FileReader("bicicas.json");

        JsonArray estacions = (JsonArray) ((JsonObject) ((JsonArray )
Jsoner.deserialize(r_json)).get(0)).get("ocupacion");

        JsonArray destEstacions = new JsonArray();
```

```
        for (Object est : estacions){
            JsonObject e = (JsonObject) est;
            Map destE = new LinkedHashMap();
            destE.put("num", e.get("id"));
            destE.put("nom", e.get("punto"));
            destE.put("llocs", e.get("puestos"));
            destE.put("ocupats", e.get("ocupados"));
            int lliures = Integer.parseInt(e.get("puestos").toString())-
Integer.parseInt(e.get("ocupados").toString());
            destE.put("lliures", lliures );
            destEstacions.add(destE);
        }

        JsonObject bicicas = new JsonObject();
        bicicas.put("bicicas",destEstacions);

        Writer w_json = new FileWriter("bicicas2.json");
        w_json.write(bicicas.toJson());
        w_json.close();
    }
}
```

Ara el resultat sí que està ordenat:

```
{"bicicas":
[{"num":"01","nom":"UJI - FCHS","llocs":28,"ocupats":11,"lliures":17}
,{"num":"02","nom":"ESTACIÓN DE FERROCARRIL Y
AUTOBUSES","llocs":28,"ocupats":8,"lliures":20},
{"num":"03","nom":"PLAZA DE PESCADERÍA","llocs":28,"ocupats":13,"lliures":15},
{"num":"04","nom":"PASEO BUENAVISTA-GRAO","llocs":14,"ocupats":5,"lliures":9},
{"num":"05","nom":"HOSPITAL GENERAL","llocs":14,"ocupats":7,"lliures":7},
...
]}
```

Exercicis

Exercici 3_1

En el projecte anomenat **Tema3**, crea't un paquet anomenat **Exercicis** on col·locarem tot el relatiu als exercicis d'aquest tema. Copia't dins del projecte el fitxer **Rutes.dat** que us passarà el professor. En ell tenim dades prèviament guardades que seran unes **rutes** consistents en una sèrie de punts amb una descripció. Cada punt seran unes coordenades (com en un mapa).

L'estructura de les dades guardades és la següent

- nom de la ruta (string)
- desnivell (int)
- desnivell acumulat (int)
- número de punts (int)
- Per cada punt:
  - nom (string)
  - latitud (double)
  - longitud (double)

Observa que la quarta dada és un enter amb el número de punts de la ruta.

Fes una classe executable (amb **main**) anomenada **Llegir_Rutes_Serial** que agafe les dades del fitxer (hi ha 2 rutes, però ho heu de fer genèric per a un número indeterminat de rutes) i les traga per pantalla amb aquest aspecte:

```
Ruta: Pujada a Penyagolosa
Desnivell: 530
Desnivell acumulat: 530
Té 5 punts
Punt 1: Sant Joan (40.251036,-0.354223)
Punt 2: Encreuament (40.25156,-0.352507)
Punt 3: Barranc de la Pegunta (40.247318,-0.351713)
Punt 4: El Corralico (40.231708,-0.348859)
Punt 5: Penyagolosa (40.222632,-0.350339)

Ruta: La Magdalena
Desnivell: 51
Desnivell acumulat: 84
Té 7 punts
Punt 1: Primer Molí (39.99385,-0.032941)
Punt 2: Segon Molí (39.99628,-0.029427)
Punt 3: Caminàs (40.00513,-0.022569)
Punt 4: Riu Sec (40.006765,-0.02237)
Punt 5: Sant Roc (40.017906,-0.02289)
Punt 6: Explanada (40.034048,-0.00633)
Punt 7: La Magdalena (40.034519,-0.005856)
```

Exercici 3_2

Construeix les següents classes:

Coordenades, que implementarà **Serializable** (i que és convenient posar-li el número de versió per defecte: **private static final long serialVersionUID = 1L;**), i que contindrà:

- **latitud** (double)
- **longitud** (double)

Haurà de tenir un constructor **public Coordenades(double latitud, double longitud)**

Haurà de tenir també els mètodes **public double getLatitud()** i **public double getLongitud().**

Encara que no ens faran falta del tot, crea també els mètodes **set**, és a dir **void setLatitud(double)** i **setLongitud(double)**.

PuntGeo, que implementarà **Serializable** (i que és convenient posar-li el número de versió per defecte: **private static final long serialVersionUID = 1L;**), i que contindrà:

- **nom** (String)
- **coord** (Coordenades)

Haurà de tenir dos constructors:

- **public PuntGeo(String nom, Coordenades coord)**
- **public PuntGeo(String nom, double latitud, double longitud)**

> **Nota importantíssima**
>
> Els dos constructors han d'aconseguir que estiga creat un objecte del tipus **Coordenades**

Haurà de tenir també els mètodes

- **public String getNom()**
- **public Coordenades getCoord**
- **public double getLatitud()** (per a major comoditat)
- **public double getLongitud()** (per a major comoditat)

Ruta . Aquesta classe us la passarà el professor.

També implementa **Serializable** i conté:

- **nom** (String)
- **llistaDePunts**: un ArrayList de PuntGeo
- **desnivell** (int)
- **desnivellAcumulat** (int)

Observa com té els mètodes get i set per a les propietats normals, i que per a l'ArrayList de punts té els mètodes addPunt (afegirà un nou PuntGeo a la llista) i a banda de **getLlistaPunts()** que torna tota la llista en un array, també té per a major comoditat **getPunt(int)**, al qual se li passa un número i torna aquest punt. També té **getPuntNom(int), getPuntLatitud(int)** i **getPuntLongitud(int)** als quals se'ls passa l'índex del punt que es vol i tornarà el nom, latitud o longitud respectivament. També té el mètode **length()** que ens dóna el número de punts guardats en la llista.

Fes un mètode nou en la classe **Ruta** anomenat **mostraRuta**, que mostre el contingut de la ruta amb aquest aspecte:

```
Ruta: Pujada a Penyagolosa
Desnivell: 530
Desnivell Acumulat: 530
Té 5 punts
Punt 1: Sant Joan (40.251036,-0.354223)
Punt 2: Encreuament (40.25156,-0.352507)
Punt 3: Barranc de la Pegunta (40.247318,-0.351713)
Punt 4: El Corralico (40.231708,-0.348859)
Punt 5: Penyagolosa (40.222632,-0.350339)
```

Per últim en una última classe **PassarRutesSerialObj** (executable, és a dir amb main) agafa el fitxer **Rutes.dat** , guarda la informació en un objecte **Ruta**, visualitza la seua informació amb **mostraRuta** i per últim guarda la informació de l'objecte en un fitxer anomenat **Rutes.obj**

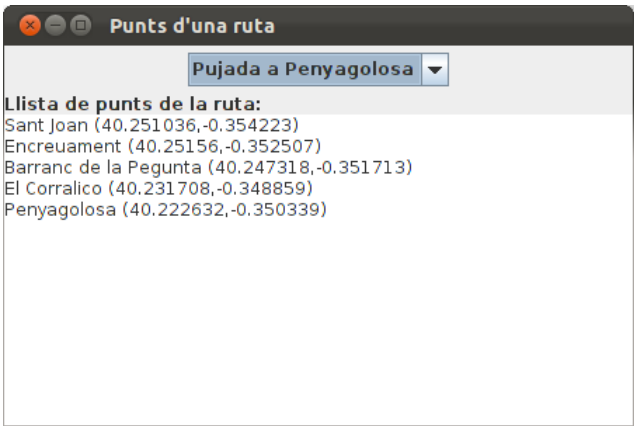
Exercici 3_3

Fer una classe anomenada **PassarRutesObjXML** (amb main) que passe el fitxer **Rutes.obj** a un fitxer XML anomenat **Rutes.xml** amb aquest aspecte. Els punts suspensius indiquen que hi ha més d'un punt eb cada ruta, i que hi ha més d'una ruta

```
<rutes>
   <ruta>
      <nom>Pujada a Penyagolosa</nom>
      <desnivell>530</desnivell>
      <desnivellAcumulat>530</desnivellAcumulat>
      <punts>
         <punt num="1">
            <nom>Sant Joan</nom>
            <latitud>...</latitud>
            <longitud>...</longitud>
         </punt>
         ...
      </punts>
   </ruta>
   ...
</rutes>
```

Exercici 3_4

Fer una aplicació gràfica que llegirà el fitxer **Rutes.xml** per a que apareguen els noms de les rutes en un JComboBox. Quan se seleccione una, ha d'aparèixer la llista de punts (nom, latitud i longitud) en un JTextArea. L'aspecte podria ser el següent:



L'esquelet del programa seria aquest (si voleu el podeu fer directament vosaltres amb WindowBuilder):

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;

import org.w3c.dom.Document;


public class Vis_Rutes_XML_Pantalla extends JFrame implements
ActionListener{

    JComboBox combo;
    JTextArea area = new JTextArea();
    Document doc;
```

```
    public void iniciar(){
        // sentències per a omplir doc


        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());

        JPanel panell1 = new JPanel(new FlowLayout());
        JPanel panell2 = new JPanel(new BorderLayout());
        this.add(panell1,BorderLayout.NORTH);
        this.add(panell2,BorderLayout.CENTER);

        ArrayList<String> llista_rutes = new ArrayList<String>();
        // sentències per a omplir l'ArrayList amb el nom de les rutes


        combo = new JComboBox(llista_rutes.toArray());

        panell1.add(combo);

        panell2.add(new JLabel("LLista de punts de la
ruta:"),BorderLayout.NORTH);
        panell2.add(area,BorderLayout.CENTER);

        this.setVisible(true);
        combo.addActionListener(this);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == combo){
            //accions quan s'ha seleccionat un element del combobox, i
que han de consistir en omplir el JTextArea


        }
    }
}
```

I el programa principal que crida a aquest JFrame:

```
public class Vis_Rutes_XML {

    public static void main(String[] args) {
        Vis_Rutes_XML_Pantalla finestra = new Vis_Rutes_XML_Pantalla();
        finestra.iniciar();
    }
}
```

Exercici 3_5

Fer una classe anomenada **PassarRutesObjJSON** (amb main) que passe el fitxer **Rutes.obj** a un fitxer JSON **Rutes.json** amb aquest aspecte:

```
{ "rutes" :
   [
      {"nom":"Pujada a Penyagolosa" ,
       "desnivell":530 ,
       "desnivellAcumulat":530 ,
       "punts":
        [  {"num":1" ,
             "nom":"Sant Joan" ,
             "latitud": ... ,
             "longitud": ...
           } ,
          ...
        ]
       } ,
    ...
   ]
}
```

Exercici 3_6

Replicar l'exercici 3_4, però ara llegint del fitxer **Rutes.json**, en compte de **Rutes.obj**

Fer una aplicació gràfica que llegirà el fitxer **Rutes.json** i que aparega el nom de les rutes en un JComboBox. Quan se seleccione una, ha d'aparèixer la llista de punts (nom, latitud i longitud) en un JTextArea. L'aspecte podria ser el següent:

L'esquelet del programa seria aquest (si voleu el podeu fer directament vosaltres amb WindowBuilder):

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;

import com.github.cliftonlabs.json_simple.JsonArray;
import com.github.cliftonlabs.json_simple.JsonObject;


public class Vis_Rutes_JSON_Pantalla extends JFrame implements
ActionListener{
```

```
    JComboBox combo;
    JTextArea area = new JTextArea();

    JsonObject arrel;
    JsonArray rutes;

    public void iniciar() throws FileNotFoundException, IOException {
        // sentències per a omplir arrel i rutes


        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());

        JPanel panell1 = new JPanel(new FlowLayout());
        JPanel panell2 = new JPanel(new BorderLayout());
        this.add(panell1,BorderLayout.NORTH);
        this.add(panell2,BorderLayout.CENTER);

        ArrayList<String> llista_rutes = new ArrayList<String>();
        // sentències per a omplir l'ArrayList amb el nom de les rutes


        combo = new JComboBox(llista_rutes.toArray());

        panell1.add(combo);

        panell2.add(new JLabel("LLista de punts de la
ruta:"),BorderLayout.NORTH);
        panell2.add(area,BorderLayout.CENTER);

        this.setVisible(true);
        combo.addActionListener(this);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == combo){
            //accions quan s'ha seleccionat un element del combobox, i
que han de consistir en omplir el JTextArea


        }
    }
}
```

I el programa principal quedaria així

```
import java.io.FileNotFoundException;
import java.io.IOException;

public class Vis_Rutes_JSON {

    public static void main(String[] args) throws
FileNotFoundException, IOException {
        Vis_Rutes_JSON_Pantalla finestra = new
Vis_Rutes_JSON_Pantalla();
        finestra.iniciar();
    }
}
```

Exercici 3_7 (voluntari)

Per a practicar un poquet més, anem a fer un altre exercici per a construir un fitxer JSON.

Feu la classe amb main **PassarCotxesXMLJSON**, que haurà de passar el fitxer **cotxes.xml** al fitxer **cotxes.json**. Aquest és el fitxer **cotxes.xml**:

```
<oferta>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
```

```
    <motor combustible="gasolina">duratorc 1.4</motor>
    <matricula>1234AAA</matricula>
    <kilometros>12500</kilometros>
    <precio_inicial>12000</precio_inicial>
    <precio_oferta>10000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="300">llantas</extra>
    <foto>11325.jpg</foto>
    <foto>11326.jpg</foto>
  </vehiculo>
  <vehiculo>
    <marca>ford</marca>
    <modelo color="gris">focus</modelo>
    <motor combustible="diesel">duratorc 2.0</motor>
    <matricula>1235AAA</matricula>
    <kilometros>125000</kilometros>
    <precio_inicial>10000</precio_inicial>
    <precio_oferta>9000</precio_oferta>
    <extra valor="250">pintura metalizada</extra>
    <extra valor="200">spoiler trasero</extra>
    <extra valor="500">climatizador</extra>
    <foto>11327.jpg</foto>
    <foto>11328.jpg</foto>
  </vehiculo>
</oferta>
```

I aquest ha de ser l'aspecte de **cotxes.json**:

```
{
  "oferta": {
    "vehiculo": [
      {
        "marca": "ford",
        "modelo": {
          "color": "gris",
          "nombre_modelo": "focus"
        },
        "motor": {
          "combustible": "gasolina",
          "nombre_motor": "duratorc 1.4"
        },
        "matricula": "1234AAA",
        "kilometros": "12500",
        "precio_inicial": "12000",
        "precio_oferta": "10000",
        "extra": [
          {
            "valor": "250",
            "nombre_extra": "pintura metalizada"
          },
          {
            "valor": "300",
            "nombre_extra": "llantas"
          }
        ],
        "foto": [
          "11325.jpg",
          "11326.jpg"
        ]
      },
      {
        "marca": "ford",
        "modelo": {
          "color": "gris",
          "nombre_modelo": "focus"
        },
        "motor": {
          "combustible": "diesel",
          "nombre_motor": "duratorc 2.0"
        },
        "matricula": "1235AAA",
        "kilometros": "125000",
        "precio_inicial": "10000",
        "precio_oferta": "9000",
        "extra": [
          {
            "valor": "250",
            "nombre_extra": "pintura metalizada"
          },
          {
            "valor": "200",
            "nombre_extra": "spoiler trasero"
          },
```

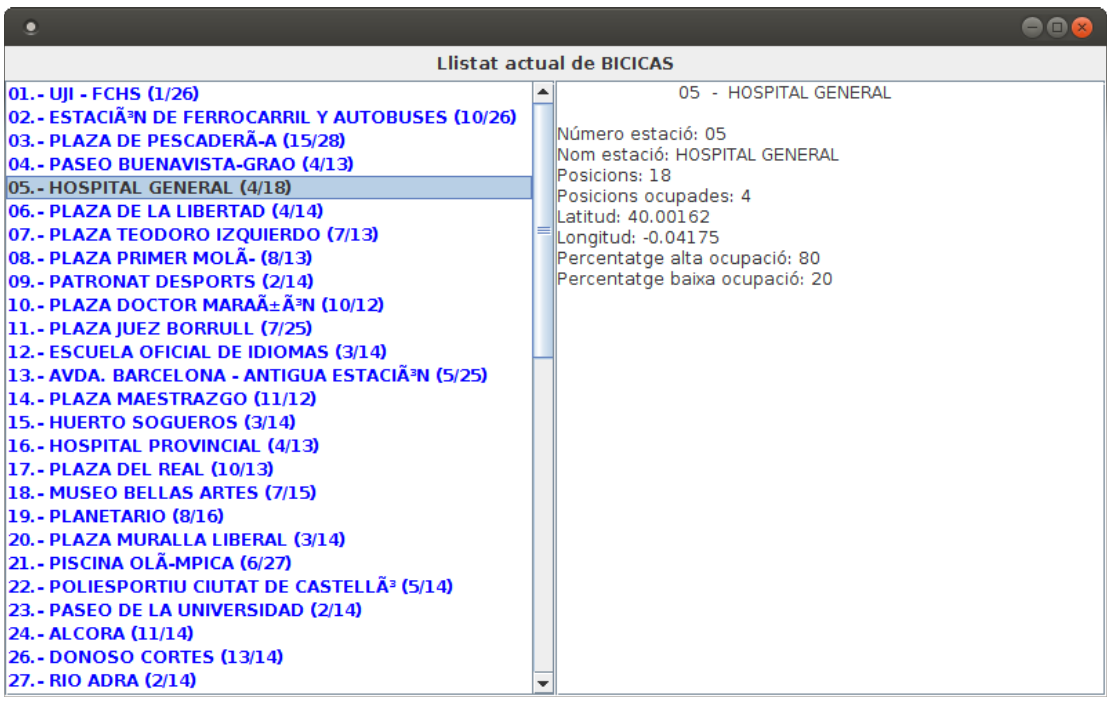
```
            {
              "valor": "500",
              "nombre_extra": "climatizador"
            }
          ],
          "foto": [
            "11327.jpg",
            "11328.jpg"
          ]
        }
      ]
    }
}
```

Exercici 3.8 (voluntari)

Visualitzar en un programa gràfic l'estat actual de Bicicas, agafant-lo directament de la pàgina http://gestiona.bicicas.es/apps/apps.php.

- En el mètode **agafarBicicas()** farem les instruccions per agafar-lo directament d'Internet, utilitzant un **URL**, el qual:
  - Inicialitzarem amb la pàgina que busquem
  - Obrim la connexió amb el mètode **openConnection()**
  - Després obtindrem el contingut amb el mètode **getInputStream()**, que com el seu nom indica serà un InputStrem
- En el mètode **mostrarEstacions()** introduirem les instruccions per a visualitzar les estacions, utilitzant un **JList** (el que està a la part de l'esquerra), que s'ha de construir a partir d'un **DefaultListModel**. Per anar col·locant els diferents elements del JList, els haurem d'anar afagint al **DefaultListModel**, i aniran apareixent en el JList. Intentarem mostrar el número d'estació, nom i número de bicicletes (posicions ocupades) i el total de posicions.
- El mètode **visualitzaEstacio()** s'activa quan seleccionem un element del JList, i en ell mostrarem les propietats de l'estació seleccionada en el **TextArea** que està a la part de la dreta.

Aquest seria l'aspecte:



Aquest és l'esquelet del programa:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;

import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import com.github.cliftonlabs.json_simple.JsonArray;
import com.github.cliftonlabs.json_simple.JsonException;
import com.github.cliftonlabs.json_simple.JsonObject;
import com.github.cliftonlabs.json_simple.Jsoner;


public class Pantalla_Vis_Bicicas_Json_JList extends JFrame implements ListSelectior

    private static final long serialVersionUID = 1L;
    DefaultListModel listModel = new DefaultListModel();
    JList list = new JList(listModel);
    JTextArea area = new JTextArea(5,15);

    JsonObject arrel;
    JsonArray estacions;

    public void iniciar() throws JsonException, IOException {
        this.setBounds(100, 100, 450, 300);
        this.setLayout(new BorderLayout());

        JPanel panell1 = new JPanel(new GridLayout(1,2));
        JPanel panell2 = new JPanel(new FlowLayout());
        this.add(panell1,BorderLayout.CENTER);
        this.add(panell2,BorderLayout.NORTH);
        list.setForeground(Color.blue);
        JScrollPane scroll1 = new JScrollPane(list);
        panell1.add(scroll1);
        JScrollPane scroll2 = new JScrollPane(area);
        panell1.add(scroll2);

        JLabel et1 = new JLabel("Llistat actual de BICICAS");
        panell2.add(et1);


        agafarBicicas();

        mostrarEstacions();

        setVisible(true);

        list.addListSelectionListener(this);
    }


    @Override
    public void valueChanged(ListSelectionEvent e){
        JList l = (JList) e.getSource();
        if (l.getSelectedIndex()>=0){
            visualitzaEstacio(l.getSelectedValue().toString());
        }
    }

    private void agafarBicicas() throws JsonException, IOException {
        // Instruccions per a llegir de la pàgina de Bicicas i col·locar en arrel
        URL bicicas = new URL("http://gestiona.bicicas.es/apps/apps.php");
        JsonArray arrel = (JsonArray) Jsoner.deserialize(new InputStreamReader(bicic
        // Instruccions per a col·locar les estacions en estacions (JsonArray)

    }

    private void mostrarEstacions() {
        // Instruccions per a introduir en el JList les estacions
        // La manera d'anar introduint informació en el JList és a través del Defaul
```

```
        // listModel.addElement("Linia que es vol introduir ")
        // Una manera de solucionar el problema de la cometa simple és utilitzar con
        // Una altra és utilitzar paràmetres amb PreparedStatement

    }

    private void visualitzaEstacio(String estacio){
        // Instruccions per a mostrar les característiques en el area, el JTextArea

    }
}
```

I aquest seria el programa principal:

```
import java.io.FileNotFoundException;
import java.io.IOException;

import org.json.simple.parser.ParseException;

public class Vis_Bicicas_Json_JList {

    public static void main(String[] args) throws FileNotFoundException, IOExceptio
        Vis_Rutes_JSON_Pantalla finestra = new Vis_Rutes_JSON_Pantalla();
        finestra.iniciar();
    }
}
```