U.T. 1 - INTRODUCCIÓ A LA PROGRAMACIÓ

1. CONCEPTES BÀSICS DE PROGRAMACIÓ

programació

> És el desenvolupament de la seqüència d'accions a executar per l'ordinador per a resoldre un problema o realitzar una tasca.

programador

> Qui desenvolupa la seqüència d'accions a executar per l'ordinador.

programa

> La tasca en si o la seqüència d'accions.

Altres conceptes: maquinari (o *hardware*), programari (o *software*), memòria RAM, disc dur, codi font ... La màquina només entén un alfabet amb dos signes: 0 i 1 (codi binari o codi màquina).

Exemple: 01110010 --> es pot interpretar com el valor numèric enter 114 o la lletra 'r', segons el tipus de dada.

Per desenvolupar la seqüència d'accions utilitzem els llenguatges de programació, perque a la màquina no podem dir-li directament "escriu el document doc1.txt a la impressora". Per tant, seria alguna cosa així com *print (DEVICE (printer), doc1.txt)*.

En definitiva, haurem d'utilitzar un sistema de codificació: el llenguatge de programació.

https://ca.wikipedia.org/wiki/Llenguatge_de_programaci%C3%B3

https://www.softzone.es/noticias/general/lenguajes-programacion-aprender-trabajo-seguro/

2. PROCESSAMENT DE DADES

Els programes que farem funcionen segons el següent esquema d'esquerra a dreta:

Perifèrics d'entrada (lectura de dades) -------> Procés --------> Perifèrics d'eixida (presentació de resultats)

En el nostre cas, per a aplicacions a executar en un terminal en mode text, el perifèric d'entrada que utilitzem per introduir dades al procés (el nostre programa) serà el **teclat**, mentre que el perifèric d'eixida per defecte serà la **pantalla** o monitor, ja que en ell es mostraran els resultats d'eixida que produeix el procés.

Els perifèrics poden ser de 3 tipus:

- d'entrada (teclat, ratolí, escàner, micròfon)
- d'eixida (monitor, impressora, altaveus)
- d'entrada i eixida (unitats d'emmagatzematge, com el disc dur)

Dada

és la informació sobre una entitat susceptible de ser processada. Cada dada va a constar de:

- identificador (el nom que li donem)
- tipus (si és numèric, enter o real, de text, etc.)
- valor (el contingut, per aquesta dada, en memòria interpretat segons el seu tipus). Com hem dit abans, la mateixa seqüència, per exemple 01110010, pot ser 114 ó 'r'.

Un *identificador* seguirà unes regles concretes, però dependran del llenguatge. Les regles habituals per a l'elecció de l'identificador són:

- constituïts per lletres (a, ..., z, A, ..., Z), dígits (0,1, ..., 9) i, segons el llenguatge, alguns caracters més.
- ha de començar per lletra
- no ha de contenir espais en blanc (*numero_primo*)
- la longitud màxima depèn del llenguatge i del compilador
- no s'utilitzaran paraules reservades del llenguatge (*if, while, ...*)
- convé que siga prou descriptiu per augmentar la llegibilitat del codi.
- segons el llenguatge es pot distingir, o no, entre majúscules i minúscules. En Pascal, NUM, Num i num són la mateixa variable. En C o Java, són variables diferents.

Tipus -> determina la mida requerida per a guardar la dada en memòria. Ha d'estar d'acord amb els valors a contenir.

Valor -> contingut en memòria per aquesta dada en un instant determinat.

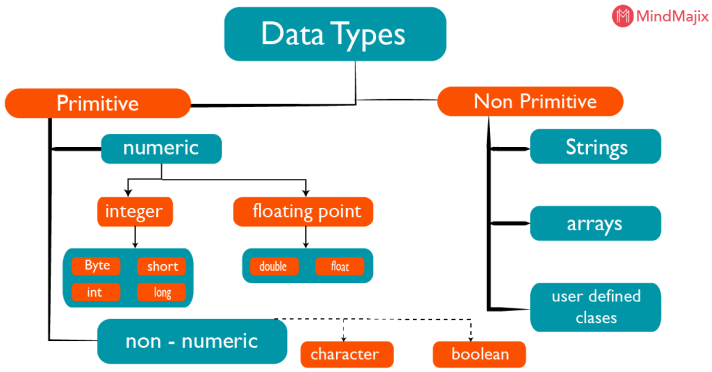
3. TIPUS DE DADES

Segons el seu valor puga, o no, ser modificat durant l'execució del programa:

- constants
- variables

Segons els possibles valors a contenir:

```
* Dades bàsiques
    - numèrics (sencer, real)
    - caracter
    - alfanumèric
    - lògic
* Dades indirectes
    - punter
* Dades estructurades
    - interns
            + estàtics
                - taules (arrays)
                - registres (structs)
            + dinàmics
                - lineals
                    * llistes
                    * piles
                    * cues
                - no lineals
                    * arbres
                    * grafs
    - externs
        + fitxers
        + Bases de dades
```



Representació interna (de les dades bàsiques):

```
* numèrics
    - enters
        + Sense signe
            * Binari pur
            * BCD
        + Amb signe
            * Signe-magnitud
            * Complement a 1
            * Complement a 2
            * Complement a 2 elevat a n-1
            * Altres ...
    - reals
```

```
        + Simple precisió
        + Doble precisió
```

Enters:

Binari pur (sense signe)

57 decimal =? binari. Es divideix successivament entre 2 i es recull l'últim quocient i restes en ordre invers -> 111001 en binari -> 00111001 afegint dos bits a l'esquerra per completar el byte

Quin és el màxim valor representable amb 1 byte, amb 2 i amb 4?

- 1B -> 255
- 2B -> 65535
- 4B -> uns 4 mil milions

BCD

representa cada dígit decimal amb 4 bits.

- 0 -> 0000
- 1 -> 0001
- 2 -> 0010
- 3 -> 0011
- ...
- 9 -> 1001

Així 57 en decimal seria en BCD 01010111 (0101 corresponent a 5, i 0111 corresponent a 7).

Amb signe:

-signe-magnitud: es reserva el primer bit, per l'esquerra, per indicar el signe (0 per a indicar positiu, 1 per a negatiu)

01101010 -> +106 a decimal

10110001 -> -49 a decimal

El rang en aquest cas varia de -127 a +127 (per 1B) o -32767 a +32767 (per 2B)

Complement a 1:

- positius: Igual que signe-magnitud
- negatius: intercanviant 0s per 1s i viceversa, només per als bits de magnitud

-49 en decimal -> 10110001 en signe-magnitud -> serà 11001110 en complement a 1

Complement a 2:

- positius: Igual que signe-magnitud
- negatius: intercanviant 0s per 1s i viceversa, en els bits de magnitud i sumem 1

Reals:

- de simple precisió (4B, 32 bits)
- de doble precisió (8B, 64 bits)

Tots dos responen a la fórmula de mantissa multiplicada per la base elevada a un exponent:

Mantissa * Base ^ exponent

La base no es representa, és sempre 2.

Simple precisió:

- Bit 0: signe
- Bits 1 a 8: exponent en complement a 2 ^ 7
- Bits 9-31: mantissa en binari pur

Doble precisió:

- Bit 0: signe
- Bits 1 a 11: exponent en complement a 2 ^ 10
- Bits 12-63: mantissa en binari pur

http://evanw.github.io/float-toy/

Caracters:

Es representen mitjançant alguna taula de codis tipus ASCII o Unicode.

ASCII és un codi de 7 bits que representa 128 caràcters (lletres angleses, números i símbols bàsics). **Unicode** és un estàndard més ampli que inclou ASCII i amplia la capacitat a més de 140.000 caràcters per representar gairebé totes les llengües i símbols del món.

Cada caracter es representa amb 1 Byte (ASCII) o 2 Bytes o més (Unicode). En el primer cas permet representar 256 caracters o signes de control, entre ells:

- caracters de l'alfabet (en majúscules i també en minúscules)
- vocals amb les diferents accentuacions (à, á, ä ..... ü)
- dígits (0 ... 9)
- signes de puntuació (,;:. , etc.)
- altres caracters (alguns poden ser no imprimibles)

Cada caracter es representa amb el valor corresponent, en binari natural, a la seva posició a la taula ASCII (man ASCII)

Per exemple, 'm' està en la posició 109 -> 01101101

Alfanumèric (cadenes de text):

Segons el llenguatge de programació, pot existir aquest tipus o, en el seu lloc, es pot utilitzar l'array (o conjunt) de caracters.

'm' <> "m" (no és el mateix, observa les cometes). Al primer cas és caracter, només un, mentres que al segon es un alfanumèric, es a dir, una sequència de caracters.

Lògic (booleà):

És aquell que només pot contenir dos possibles valors del tipus veritable / fals, on / off o 1 / 0.

4. ALGORISMES

Algorisme -> el terme prové del matemàtic persa Muhammad ibn Musa *Al-khwarizmi*.

És la seqüència d'operacions a executar per resoldre un problema o desenvolupar una tasca. Un algorisme, per ser correcte, ha de tenir 3 qualitats: ser finit, precís i eficient.

Cadascuna de les operacions simples que componen un algorisme són instruccions.

Tipus d'instruccions:

- d'entrada
- d'eixida
- d'assignació
- alternatives
- repetitives

Format d'una **instrucció d'assignació**:

```
nom_variable = expressió (composta d'operadors i operands segons les regles d'aquells)
```

Exemple: n2 = n1 + 3, a la variable n 2 se li assigna el resultat de l'operació n1 +3

ATENCIÓ!: No és vàlid fer al contrari: expressió = variable

El resultat d'una expressió serà, generalment, un valor numèric o un resultat lògic (veritable o fals).

Operadors numèrics:

- + (Suma)
- - (resta)
- * (Producte)
- / (Divisió)
- Mod (residu de la divisió entera)

Operadors lògics:

not, and, or i xor.

A	NOT A
0	1
1	0

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Altres operadors que poden aparèixer en expressions lògiques són els operadors relacionals: <,>, <=,> =, =, <>

Exemple: (5 < 7) and (8>= 6)

Exercici: obtenir una expressió per a la següent condició lògica: número comprès entre 1 i 100, ambdós inclosos, i múltiple de 5.

A l'hora de resoldre una expressió s'han de tenir en compte els nivells de prioritat dels operadors:

1. Parèntesi (començant pels més interns)
2. Potència (^)
3. Producte i divisió
4. Sumes i restes

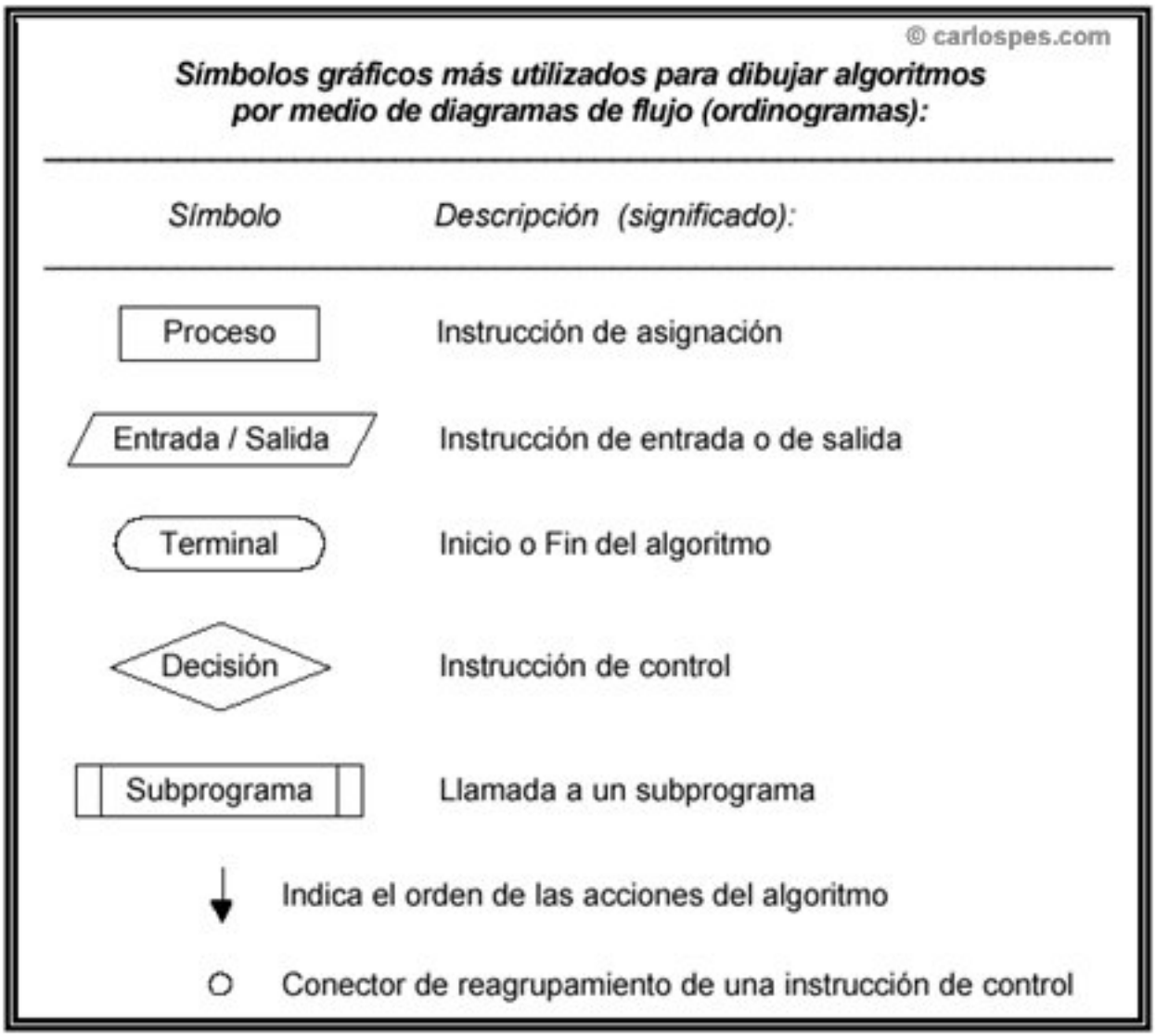
5. Concatenació de cadenes de caracters
6. Relacionals (<,>, ...)
7. Negació (not)
8. Conjunció (and)
9. Disjunció (or)

5. REPRESENTACIÓ D'ALGORISMES

2 tècniques:

- pseudocodi (representació textual)
- ordinogrames (representació gràfica)

La representació mitjançant **ordinogrames** utilitza la següent simbologia:



Exemple: Programa que demana 2 valors numèrics a introduir des teclat i imprimeix en pantalla el resultat del producte de tots dos valors.

El **pseudocodi** és un llenguatge fictici molt proper al llenguatge humà. És utilitzat per plasmar solucions algorítmiques prèvies a la codificació en algun llenguatge de programació concret. Característiques:

- la seva sintaxi no és estricta
- no és compilable

Esquema general d'un programa en pseudocodi:

```
Programa NOM_PROGRAMA
Dades:
    constants:
        (Declaració de les constants utilitzades)
    variables:
        (Declaració de les variables utilitzades)
Algorisme:
    INICI
        (Conjunt d'instruccions)
    FI
```

Veguem un exemple (l'anterior exercici):

```
Programa: PRODUCTE
Dades:
    variables:
        num1, num2 sencer
Algorisme:
    INICI
        Escriure "Introduïu 2 valors numèrics"
        Llegir num1, num2
        Escriure "El producte és:", num1 * num2
    FI
```

Cal observar en aquest exemple el funcionament de la instrucció **Escriure**. Aquesta instrucció escriurà literalment a pantalla, sense cap alteració, tot allò que vagi entre cometes dobles. Però també permet, per les dades, imprimir els seus valors si aquests s'escriuen SENSE cometes. Cal observar que amb una sola instrucció *Escriure* es poden escriure literals i, en qualsevol moment, tancant les cometes i separant per comes, escriure els valors d'algunes dades o expressions (com en aquest cas el producte *num1*num2*).

Exercici: programa que demane el radi d'una circumferència i obtinga el seu perímetre:

```
Programa CIRCUMFERÈNCIA
dades:
    constants:
        PI 3.1416
    variables:
        radi Real

Algorisme:
INICI
    Escriure "Introduïu el radi de la circumferència:"
    llegir radi
    Escriure "El seu perímetre és", 2 * PI * radi
FI
```

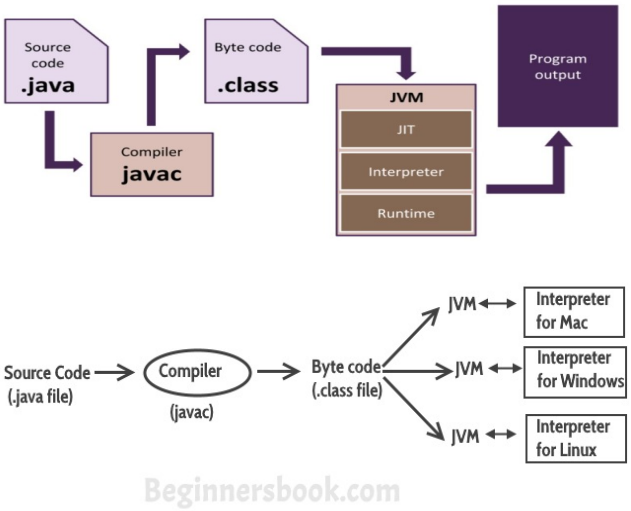
La sintaxi del pseudocodi no és estricta!

6. INTRODUCCIÓ AL LLENGUATGE JAVA

James Gosling treballava per *Sun Microsystems* quan va dissenyar Java molt a principis dels anys 90. El seu primer nom va ser OAK i va tenir com a referents, quant a sintaxi, C i C++ (encara que la seva filosofia de funcionament és totalment diferent). La idea primitiva de *Sun* va ser utilitzar per a la programació de xicotets electrodomèstics, projecte que no va prosperar. Passats uns 4 anys aquesta idea primitiva de llenguatge va ser represa per a un propòsit molt diferent, el que actualment té Java: llenguatge d'ús general **multiplataforma**.

La primera característica que anem a destacar de Java és que es tracta d'un llenguatge orientat a objectes, o llenguatge de la programació orientada a objectes (**POO**). La filosofia de la POO, per contraposició a la programació estructurada clàssica, no estarà tan basada en el disseny dels algorismes, sinó en el de les dades o objectes que intervenen en el programa.

El particular èxit de Java, d'entre la resta de llenguatges orientats a objectes, rau en ser un llenguatge multiplataforma. Això ho aconsegueix fent ús d'una màquina virtual en el sistema destí, evitant recompilar per a cada sistema operatiu. Es tracta per tant d'un llenguatge, en última instància, interpretat que, per a major eficiència, utilitza un codi intermedi (*bytecode*) resultat d'una compilació prèvia. L'esmenat *bytecode* és independent de l'arquitectura i pot ser executat en qualsevol sistema. Per a poder funcionar d'aquesta manera, Java requerirà tenir instal·lada a la màquina destí la màquina virtual Java (JVM, Java Virtual Machine).



Per tant, Java és a el mateix temps compilat i interpretat. El compilador compila a bytecode i l'intèrpret s'encarregarà d'executar aquest codi intermedi en la màquina real.

7 - L'ENTORN DE DESENVOLUPAMENT EN JAVA

Java SE vs Java EE: Principals diferències

1. Java SE (Standard Edition)

- **Propòsit**: És el **nucli bàsic** de Java, amb tot el necessari per a desenvolupar aplicacions genèriques.
- **Contingut**:
  - Llenguatge Java i API fonamentals
  - JVM (Màquina Virtual de Java).
  - Eines com `javac` (compilador) i `java` (executor).

- **Casos d'ús**:
    - Aplicacions d'escriptori (amb JavaFX o Swing).
    - Programes de consola.
    - Servidors web simples (com l'exemple de `HttpServer` de Java 18).

2. Java EE (Enterprise Edition, ara Jakarta EE)

- **Propòsit**: És una **extensió de Java SE** pensada per a aplicacions empresarials complexes (backend, microserveis, etc.).
- **Contingut**:
    - APIs per a web (Servlets, JSP, JSF), persistència (JPA) per a treball amb bases de dades, transaccions (JTA), seguretat (JAAS), etc.
    - Servidors d'aplicacions com WildFly o TomEE.

Taula comparativa

Característica	Java SE	Java EE (Jakarta EE)
Enfocament	Aplicacions generals	Empresarial (backend/web)
APIs clau	Collections, IO, JDBC	Servlets, JPA, JAX-RS, EJB
Servidor necessari	No	Sí (TomEE, WildFly, etc.)
Complexitat	Baixa	Alta (arquitectures complexes)
Exemple d'ús	Eines de consola, apps d'escriptori	APIs REST, sistemes bancaris

Quan triar cadascun?

- **Java SE**:
    - Si desenvolupes aplicacions senzilles o que no requereixin infraestructura empresarial.
    - Exemples: Eines CLI, aplicacions amb interfície gràfica.
- **Java EE/Jakarta EE**:
    - Si necessites gestió de transaccions, seguretat integrada, o escalabilitat en entorns corporatius.
    - Exemples: APIs per a mòbils, sistemes distribuïts.

Alternatives modernes:

Frameworks com **Spring Boot** (basat en Spring, que usa parts de Java EE) o **Quarkus** (optimitzat per a Jakarta EE) ofereixen aproximacions més lleugeres.

En el nostre cas, i al llarg del present curs, treballarem amb el JDK de Java SE.

El **JDK** (*Java Development Kit*), tot i que no conté cap eina gràfica per al desenvolupament de programes, sí que conté aplicacions de consola i eines de compilació, documentació i depuració. El JDK inclou el JRE (Java Runtime Environment), que consta dels mínims components necessaris per executar una aplicació Java, com són la màquina virtual i les llibreries de classes.

El JDK conté, entre altres, les següents eines de consola:

- **javac**. Compilador de Java.
- **java**. Intèrpret de Java.

- **javap**. Desensamblador de classes.
- **jdb**. Depurador de consola.
- **javadoc**. Generador de documentació.
- **appletviewer**. Visor de *applets*.

Per a conèixer la versió de Java amb la qual estem treballant n'hi ha prou amb executar:

```
java -version
```

Els programes o aplicacions en Java es componen d'una sèrie de fitxers *.class* que són els fitxers amb el bytecode corresponent a les classes del programa. Aquests fitxers no tenen perquè estar tots en un mateix directori, sinó que poden estar distriuits en diversos discos o, fins i tot, en diverses màquines.

L'aplicació s'executa des del mètode principal anomenat *main()* dins d'alguna classe, el qual va creant objectes a partir de les classes.

És el moment de veure el nostre primer exemple, el HolaMon:

```
public class HolaMon
{
    public static void main (String args [])
    {
        // mostra el missatge per pantalla
        System.out.println ( "Hola món!");
    }
}
```

A la vista de tan escàs codi, ja podem fer algunes puntualitzacions:

- en Java no existeixen funcions independents. Totes les funcions estaran incloses en alguna classe i, com a tal, han de ser mètodes d'aquestes classes. És per això que, a diferència de C++, es tracta d'un llenguatge orientat a objectes pur o amb un sentit més estricte de l'orientació a objectes. En el cas de C++ sí que hi han funcions independents, la mateixa *main()* per exemple, que no s'inclou en cap classe.
- els comentaris (// per a una sola línia i /* ... */ multilínia) són els ja coneguts de C i C++.

En Java generalment cada classe és un fitxer diferent. Si hi ha diverses classes en el mateix fitxer, la classe que s'anomena amb el nom del fitxer hauria de portar el modificador *public* i és la que es pot utilitzar des de fora de l'arxiu. Les classes tenen el mateix nom que el fitxer *.java* sense canvis de majúscules o minúscules.

El mètode **main** té les següents propietats:

- és públic, per poder ser utilitzat des de qualsevol lloc.
- és estàtic, per poder ser utilitzat sense haver d'instanciar la classe.
- no torna cap valor (modificador *void*).
- admet opcionalment una sèrie de paràmetres (String args[]) que en aquest exemple no són utilitzats.

En l'anterior exemple també pots veure com imprimir text en pantalla. Per a això utilitzem la classe *System*, que pot ser cridada des de qualsevol punt d'un programa i conté un atribut *out* que inclou dos mètodes molt utilitzats: *print()* i *println()*. La diferència entre tots dos és que en el segon s'afegeix un retorn de línia al text introduït.

8 - TIPUS DE DADES I OPERADORS DEL LLENGUATGE

Els tipus de dades s'utilitzen generalment al declarar variables i són necessaris perquè l'intèrpret o compilador conega per endavant el tipus d'informació que contindrà una variable. Els tipus de dades bàsics o primitius en Java són els següents.

Tipo	Significado	Cantidad de bytes por cada valor
byte	Números enteros cortos [-128..0..127]	1
short	Números enteros cortos [-32768..0..32767]	2
int	Números enteros	4
long	Números enteros largos	8
float	Números con coma flotante o decimales [de 7 dígitos decimales]	4
double	Números con coma flotante o decimales [de 14 dígitos decimales]	8
char	Caracteres simples	2
boolean	Valores lógicos [true;false]	1

Com en C, és usual en Java utilitzar majúscules per als identificadors de les constants, mentre que les variables utilitzen minúscules. Les constants es declaren seguint el següent format:

```
final [static] tipusDada identificador = valor;
```

El qualificador **final** identifica la dada com a constant, mentre que *static* és opcional i es veurà el seu significat més endavant. Un exemple seria:

```
final static double PI = 3.14159;
```

L'àmbit de cada dada, variable o constant, depèn d'on siga declarat. En general, cada dada podrà ser utilitzada al llarg de tot el bloc on ha estat declarat, entenent com a bloc la porció de codi delimitada pels claus d'obertura i tancament { } dins de les quals ha estat declarat. D'aquesta manera, una variable definida, per exemple, dins d'un bucle només podrà utilitzar-se des de l'interior d'aquest i no un des de l'exterior. Les variables locals a un mètode es poden utilitzar al llarg de tot el mètode, mentre que els atributs podran ser accedits des de qualsevol localització interior a la classe, en general en qualsevol dels seus mètodes.

A part de les dades primitives, qualsevol altra classe (pròpia de Java o creada per nosaltres) actuarà com a tipus de dada igualment utilitzable en els nostres programes com ja veurem més endavant.

Prior.	Operador	Tipo de operador	Operación
1	++ -- +, - ~ ¡	Aritmético Aritmético Aritmético Integral Booleano	Incremento previo o posterior (unario) Incremento previo o posterior (unario) Suma unaria, Resta unaria Cambio de bits (unario) Negación (unario)
2	(tipo)	Cualquiera	
3	*, /, %	Aritmético	Multiplicación, división, resto
4	+, - +	Aritmético Cadena	Suma, resta Concatenación de cadenas
5	<< >> >>>	Integral Integral Integral	Desplazamiento de bits a izquierda Desplazamiento de bits a derecha con inclusión de signo Desplazamiento de bits a derecha con inclusión de cero
6	<, <= >, >= instanceof	Aritmético Aritmético Objeto, tipo	Menor que, Menor o igual que Mayor que, Mayor o igual que Comparación de tipos
7	== ¡= == ¡=	Primitivo Primitivo Objeto Objeto	Igual (valores idénticos) Desigual (valores diferentes) Igual (referencia al mismo objeto) Desigual (referencia a distintos objetos)
8	& &	Integral Booleano	Cambio de bits AND Producto booleano
9	^ ^	Integral Booleano	Cambio de bits XOR Suma exclusiva booleana
10	| |	Integral Booleano	Cambio de bits OR Suma booleana
11	&&	Booleano	AND condicional
12	||	Booleano	OR condicional
13	? :	Booleano, cualquiera, cualquiera	Operador condicional (ternario)
14	= *=, /=, %= +=, -= <<=, >>= >>>= &=, ^=, |=	Variable, cualquiera	Asignación Asignación con operación

9. PROGRAMACIÓ ESTRUCTURADA

És una tècnica sorgida en els anys 60 que pretenia facilitar el manteniment dels programes. Proposa evitar a tota costa l'ús abusiu de la instrucció *goto* (instrucció de salt). Suposa desenvolupar tots els programes utilitzant només 3 estructures:

- seqüència: una instrucció a continuació d'una altra
- alternatives: instruccions que s'executen només si es compleix una determinada condició lògica
- repetitives (o iteratives o bucles): instruccions que s'executen repetides vegades mentre siga certa una determinada condició lògica.

La conclusió d'aquesta tècnica és que la instrucció *goto* s'ha d'intentar evitar sempre. Si s'utilitza és per falta d'habilitat del programador.

10. INSTRUCCIONS ALTERNATIVES

Alternatives: simple, doble i múltiple.

Alternativa simple.

Si una determinada condició lògica es compleix s'executa el conjunt d'instruccions contingudes dins del Si (if). En cas contrari, no es fa res.

Sintaxi de la *alternativa simple*:

```
if (condicio)
{
    bloc d'instruccions;
}
```

Nota: a partir d'ara sempre que el bloc siga d'una sola instrucció, s'ometran les claus.

exemple:

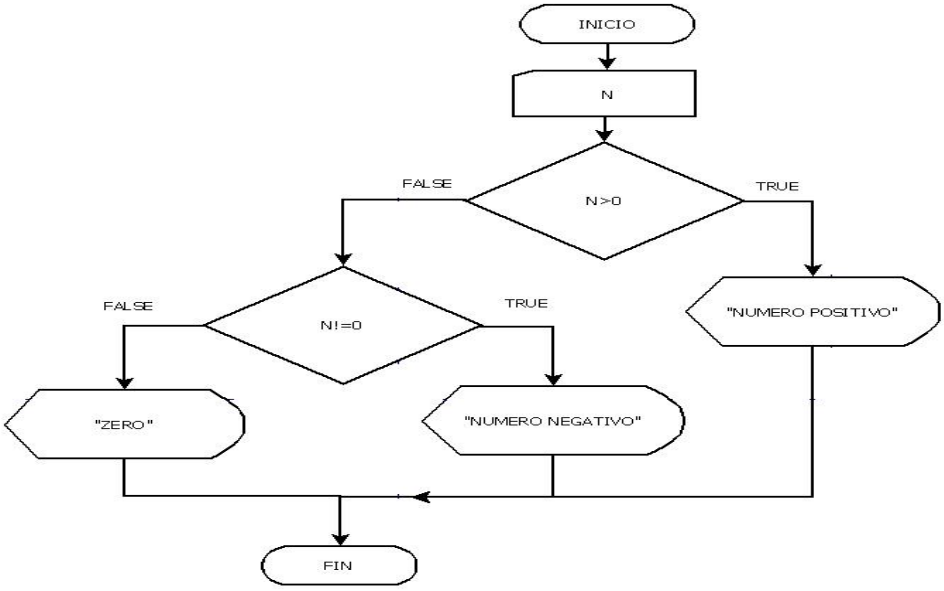
```
if (num > 0)
    System.out.println("El número és positiu\n");
```

Alternativa doble.

Si la condició lògica es compleix s'executa un bloc d'instruccions, si no s'executarà un altre alternatiu:

```
if (condicio)
{
    bloc de instruccionsA;
}
else
{
    bloc de instruccionsB;
}
```

Exemple: programa que, donat un número enter, indique si és positiu, negatiu o zero.



En alguns llenguatges, com Java, es posible utilitzar l'**operador condicional simplificat** (d'ús en assignacions) amb la següent sintaxi:

```
variable = condició? valor_si_es_compleix: valor_si_no_es_compleix;
```

exemple:

```
max = n1> n2? n1: n2;
```

En aquest cas s'avalua la condició *n1> n2*. Si aquesta es compleix s'assigna a *max* el valor de n1; en cas contrari se li assigna el valor de n2.

Alternativa múltiple.

No està subjecta a una condició lògica, sinó a una expressió generalment entera:

```
switch (expressió)
{
    case valor1: conjuntInstruccions1; break;
    case valor2: conjuntInstruccions2; break;
    ...
    case valorN: conjuntInstruccionsN; break;
```

```
    default: conjuntInstruccionsN +1;
};
```

- l'expressió ha de donar com a resultat un enter o un caracter
- *default* és opcional
- en alguns casos podrà interessar ometre algun *break*

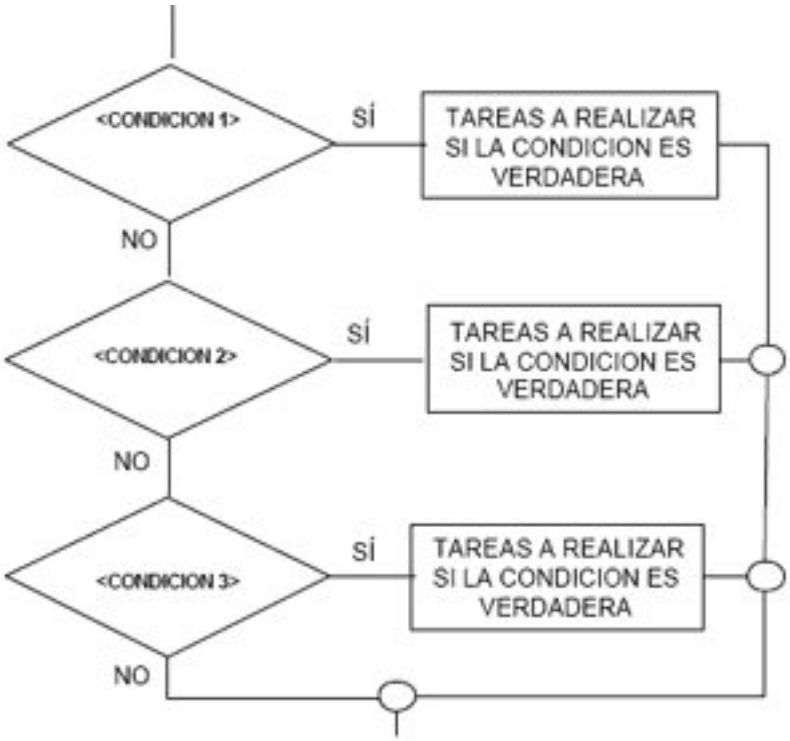
exemple:

```
int dia;

... // lectura des de teclat del valor per a la variable dia
switch (dia)
{
    case 1: System.out.println("És dilluns"); break;
    case 2: System.out.println("És dimarts"); break;
    ...
    default: System.out.println("Dia incorrecte");
};
```

Molt usual serà també trobar-nos amb alternatives niuades, això és, una dins d'una altra. Per exemple:

En pseudocodi:



11. INSTRUCCIONS REPETITIVES

Estructures iteratives.

Permeten que un bloc d'instruccions s'execute un nombre repetit de vegades (en general de 0 a múltiples, fins i tot infinites, vegades). Tenim 3 tipus:

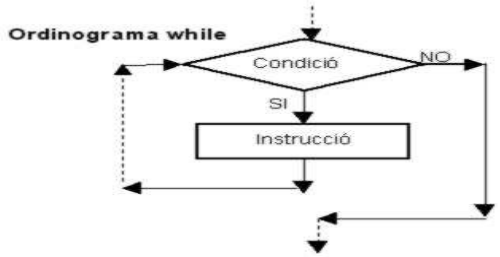
- mentre (while)
- repetir-mentre (do-while)
- per (for)

Sintaxi de **while** (mentre):

```
while (condició)
{
    bloc d'instruccions;
}
```

El bloc s'executarà de 0 a N vegades. En pseudocodi es representa:

L'ordinograma seria:



Exemple: programa que demane una nota però forçant que siga vàlida:

```
System.out.println("Introdueix una nota entre 0 i 10:");
... // s'assigna un valor a nota des de teclat
while ( (nota < 0) || (nota > 10) )
{
    System.out.println("Nota no acceptada, introdueix nota entre 0 i 10:");
    ... // s'assigna un nou valor a nota des de teclat
}
```

Vegem ara la sintaxi del **do ... while**:

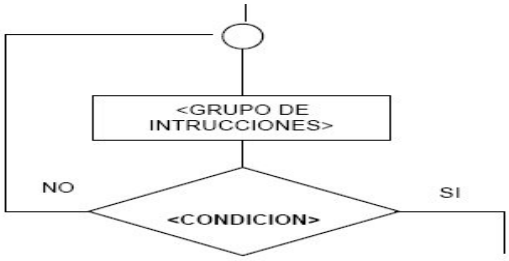
```
do
{
    bloc d'instruccions;
} while (condició);
```

Aquest bucle avalua en primer lloc la condició lògica. Si aquesta és certa, executa una primera vegada el bloc d'instruccions. A continuació, torna a avaluar la condició i, si aquesta es segueix complint, executa una segona vegada el bloc d'instruccions, per tornar a avaluar la condició lògica, etc. Només quan la condició lògica deixa de complir-se s'executarà la següent instrucció a continuació del *do...while*. Per tant, el nombre d'execucions del bloc no està entre 0 i N, sinó entre 1 i N.

Exemple: el mateix anterior, però resolt amb un **do ... while**:

```
do
{
    System.out.println("Introdueix una nota entre 0 i 10:");
    ... // s'assigna un valor a nota des de teclat
} while ((nota < 0) || (nota > 10));        // nota negativa o superior a 10
System.out.println("Aquesta nota si és valida\n");
```

En pseudocodi:

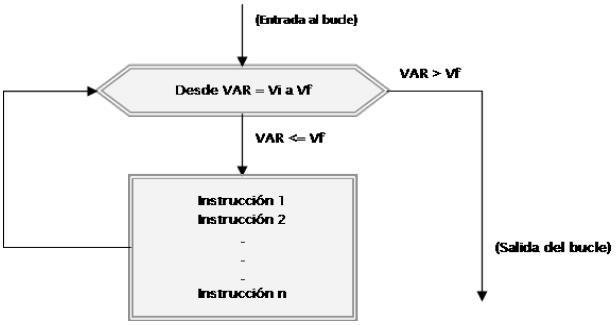


Sintaxi del **for**:

```
for (inicialització ; condició; increment)
{
    bloc d'instruccions;
}
```

En aquest cas tenim un bucle controlat per una variable comptadora, que partint d'un cert valor inicial va canviant. El funcionament d'aquest bucle serà com segueix: aquesta variable contadora *i* inicialment prendrà el valor inicial indicat, a continuació s'executarà el bloc d'instruccions, i posteriorment s'incrementarà el valor d'i segons un valor x amb alguna fórmula del tipus *i = i + x*. Si i no sobrepassa el valor final (vf) es torna a entrar al bucle per tornar a executar el bloc i tornar a incrementar la variable comptadora en la mateixa quantitat, i així successivament. El bucle acaba quan i supera el valor final vf.

El seu ordinograma:



Important:

- qualsevol *for* es podria implementar també amb un *mentre*
- se sol utilitzar quan s'ha de fer un nombre de passades concret pel bucle. Exemple:

```
int i;
for (i = 1 ; i <= 10 ; i++)
    System.out.println(i); /* Imprimeix 1 a 10 */
```

Qualsevol de les 3 parts, inicialització, condició i increment o decrement, són opcionals i poden ometre's. Exemple:

```
float nota;
boolean notaValida = false;
for ( ; !notaValida ; ) // mentre no arribe una nota vàlida
{
    System.out.println("Introdueix nota entre 0 i 10);
    ... // s'assigna un valor a nota des de teclat
    if ( ( nota >= 0 ) && ( nota <= 10 ) ) // si és vàlida
        notaValida = true;
```

```
    else
        System.out.println("Nota no valida");
}
```

A més, la condició de repetició pot ser una condició composta, no necessàriament simple. Però ... molt d'ull!: un error molt freqüent sol ser posar punt i coma després del tancament del parèntesi del for i abans de les instruccions contingudes en el bucle. És a dir, alguna cosa com:

```
for (i = 0; i <10 ; i ++); // Error: no hem de posar un punt i coma final
    System.out.println(i);
```

Al executar aquest codi esperem que ens mostre els dígits del 0 a el 9 i, en canvi, ens mostra un 10. Això és perquè al caure en l'error comentat, el bucle acaba amb el punt i coma, és a dir, el *System. out.println* queda fora del bucle i per tant s'executa una sola vegada (al acabar les 10 passades del bucle).

IMPORTANT!: Per a tots els tipus de bucle s'ha de tenir en compte la següent condició per a un correcte disseny: el bloc d'instruccions ha de ser capaç de modificar la condició lògica (ja que, si no, obtindríem un bucle infinit).

A més dels bucles vistos, molts llenguatges, com C i Java, compten amb instruccions per trencar el normal funcionament dels bucles. En Java tenim **break** i **continue**.

- *break* -> força l'eixida del bucle (continua a la següent instrucció fora del bucle).
- *continue* -> força l'eixida de la passada actual pel bucle, tornant a avaluar la condició de repetició del bucle i executant una nova passada si aquesta es compleix (ignorant les instruccions que quederen per executar en l'anterior passada).

Exemple: programa que demane nombres enters, fins a acabar amb un 0, i si és parell indique quina és la seva meitat:

```
do
{
    System.out.println("Introduïu un enter:");
    ... // s'assigna un valor a n des de teclat
    if ((n % 2) == 1) /* si és imparell */
        continue; // Es salta el System.out.println següent i torna a dalt
    System.out.println("La meitat és" + (n / 2));
} while (n != 0);
```

Per a treballar amb algorismes i convertir de pseudocodi a ordinograma, o viceversa, podem fer servir aplicacions com *Pseint*.

12. INSTRUCCIONS D'ENTRADA I EIXIDA

La classe *System* inclou 3 atributs *in*, *out* i *err* per a lectura (el primer) i escriptura de text (els dos últims) des de la consola. Vegem un exemple del primer:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
System.out.println("Introdueix el teu nom:");
String nom = br.readLine();
System.out.println("Et dius" + nom);
```

En l'anterior exemple, com en altres anteriors, apart de l'ús de *in* (entrada estàndard de dades que, per defecte, és el teclat) veiem l'ús de *out* (eixida estàndard de dades que, per defecte, és el monitor o pantalla).

Finalment, *err* (eixida d'errors) també s'utilitza per enviar missatges que, per defecte, aniran a pantalla si no es redireccionen. Mentre l'eixida habitual de resultats s'han de mostrar utilitzant *out*, els missatges d'error es podran dirigir opcionalment a algun altre dispositiu si així ho desitgem. Vegem un exemple:

```
int num = 5, den = 0, result;

try
{
    result = num / den;
}
catch (ArithmeticException e)
{
    System.err.println("No es pot dividir per zero");
}
```

Pots provar a executar aquest codi i comprovaràs que el missatge d'error apareix en pantalla (comportament per defecte) i tornar-lo a executar redirigint, per exemple, a fitxer.

13. OPERACIONS MÉS FREQÜENTS

- *Comptador de successos*. Utilitzarà una variable sencera que, usualment, anomenarem cont o similar. Requerirà iniciar la variable a zero inicialment (cont = 0) i, a continuació, cada vegada que es produeix el succés incrementar aquesta variable en una unitat (cont = cont + 1).
- *Sumar un conjunt de valors*. Definirem una variable d'igual tipus que els valors a sumar (l'anomenarem suma o alguna cosa semblant), la inicialitzarem a zero (suma = 0) i, a continuació, utilitzarem una variable per llegir els diferents valors de la sèrie (per exemple, num) i sumarem cada un d'ells (suma = suma + num).
- *Mitjana*. La mitjana comprèn les 2 operacions anteriors ja que suposa sumar tots aquests valors i comptar quants són (comptador de successos). La mitjana serà el quocient entre la suma i el comptador, però ... amb compte!, vigila que el quocient (el comptador) no valga zero ja que la divisió per zero , com ja hem comentat, provoca un error d'execució:

  ```
  Si cont != 0
      Escriure "La mitjana és", suma / cont
  fINSI
  ```

- *Intercanviar el valor de 2 variables*: requerirà d'una tercera variable (variable auxiliar) d'igual tipus i farem:

  ```
  aux = num1
  ```

Guardem el valor de num1 en la variable aux. D'aquesta manera, no es perd el valor de num1 a l'fer a continuació:

```
num1 = num2
```

D'aquesta manera, num1 ja té el valor que tenia num2. Ara fem que num2 prenga el valor que tenia num1 (que ja no està en num1, sinó en aux):

```
num2 = aux
```

- *Obtenir el màxim d'una sèrie de valors*. Utilitzarem una variable anomenada *max* o similar, de la mateixa mena que els valors de la sèrie. Generalment farem que *max* prenga com a primer valor el del primer valor de la sèrie i, a continuació, per a cada nou valor (num) comprovarem si és més gran que el màxim, en eixe cas assignarem a max el valor de num. Serà alguna cosa com:

```
Escriure "Introdueix un número"
llegir num
max = num
Mentre ...
        Si num> max
                max = num
        FiSi
Escriure "Introdueix un número"
Llegir num
FiMentre
```

- *Obtenir el mínim*. Igual que l'anterior, anomenarem *min* o semblant a la variable, prendrà com a valor inicial el primer valor de la sèrie i, per a cada nou valor, comprovarem si és menor que el mínim, assignant a min el valor de num si fos així.

En alguns casos, pot interessar donar-li a les variables *max* i *min* uns valors inicials concrets. Per exemple, si la llista de valors són notes (entre 0 i 10) podem fer que el màxim inicial siga -1 (inferior a la nota més baixa) i el mínim inicial siga 11 (superior a la nota més alta). Això convindrà fer-ho en els casos en què la sèrie només puga prendre valors compresos dins d'un rang.