

Elements bàsics

Objectius

- S'han creat i usat blocs de sentències.
- S'han identificat els diferents tipus de variables i la utilitat específica de cadascun.
- S'ha diferenciat entre variables de tipus primitiu i de tipus classe.
- S'han definit variables amb visibilitat diferents.
- S'ha modificat el codi d'un programa per a crear i utilitzar variables.
- S'han creat i utilitzat constants i literals.
- S'han classificat, reconegut i utilitzat en expressions els operadors de Java.
- S'ha comprovat el funcionament de les conversions de tipus explícites e implícites (càsting).
- S'han utilitzat mètodes de objectes proporcionats pel llenguatge.
- S'han realitzat entrades i eixides d'informació.
- S'ha utilitzat la consola per a realitzar operacions d'entrada i eixida d'informació.
- S'han aplicat formats en la visualització de la informació.
- S'ha utilitzat l'entorn integrat de desenvolupament en la creació i compilació programes.
- S'han introduït comentaris en el codi.

Sentències i blocs

Expressió

Una **expressió** és una construcció amb variables, constants, literals, operadors i cridades a mètodes (anirem veient cada concepte més avant).

```
nombre + 12 // variable anomenada nombre, operador + per a la suma i el literal per a l'enter 12  
"Hola " + getNom() // literal per al text "Hola ", operador + per a la concatenació i crida al mètode getNom
```

→ Una expressió se substitueix pel seu resultat.

Sentència

Una **sentència** és una construcció amb expressions i elements del llenguatge que defineixen una acció. La solució lògica que has creat en el llenguatge natural es transforma a sentències, una per cada frase del llenguatge natural.



Una sentència, a Java, finalitza amb punt i coma.

```
int suma = nombre + 12;  
System.out.println("Hola " + getNom());
```

La sentència `int suma = nombre + 12;` correspon a la frase, en el llenguatge natural, "defineix una variable de tipus enter anomenada `suma` i assigna-li el valor que guarda la variable `nombre` més 12", la variable `nombre` s'ha definit anteriorment, es recupera el seu valor, ha de ser un enter perquè l'operador `suma` puga sumar-li 12.

La sentència `System.out.println("Hola " + getNom());` correspon a la frase, en el llenguatge natural, "escriu el nom d'una persona precedit del text 'Hola' en la consola", el mètode `getNom()` obté el nom d'una persona i l'operador concatenació el junta amb el text "Hola".

Bloc

Un **bloc** és un grup de zero o més sentències tancades entre claus `{}`. Un bloc reuneix sentències lògicament relacionades.

```
public class Principal {  
    public static void main(String[] args) {  
        int num = 123;  
        System.out.println("num = " + num);  
    }  
}
```

En l'exemple, hi ha dos blocs, el que correspon a la classe `Principal` i el que correspon al mètode `main`.

La sentència `int num = 123;` correspon a la frase, en el llenguatge natural, "defineix una variable de tipus enter anomenada `num` i assigna-li el valor 123"

La sentència `System.out.println("num = " + num);` correspon a la frase, en el llenguatge natural, "escriu el text '`num =` ' seguit del valor de la variable `num` en la consola", es recupera el valor de `num` i es transforma a text, ja que l'operador `+` (concatenació) junta els textos, el resultat de l'expressió és la cadena de text "`num = 123`" que es visualitza en la consola.

Les sentències estan dins del bloc de sentències del mètode `main` que està dins del bloc de la classe principal. Una clau de tancament `}` s'aparella amb la clau d'obertura `{` prèvia, cal tindre molt clar que aquest és el bloc que es vol tancar.

→ Els blocs s'escriuen un dins de altre (estan niats).

Tot el codi d'una aplicació s'escriu en blocs, són els contenidors del codi i determinen els seus límits.

Estructura el teu codi mitjançant blocs, així serà més fàcil de mantindre.



→ Crea un bloc, encara que continga una única sentència.

Un bloc es comporta com una sola sentència.

Una variable definida dins d'un bloc, sols és accessible en aquest bloc.

Bloc estàtic

→ El concepte **estàtic** està associat al fet de tindre una única còpia de l'element en temps d'execució.

Un **bloc estàtic** és un bloc de codi que s'executa en el moment de la càrrega de la classe en memòria, és a dir, s'executa una única vegada. S'utilitza per a inicialitzar dades complexes (matrius, llistes, mapes, etc.), també, s'anomena **bloc d'inicialització**, s'utilitza per no repetir el mateix codi en construcció de cada objecte.

És estàtic, per tant, només pot manejar elements estàtics.

```
private static int[] nums; // declaració de la matriu d'enters nums
static {                  // bloc estàtic per a carregar la matriu nums amb valors inicials
    nums = new int[10];   // crea una matriu de 10 enters
    for (int i = 0; i < 10; i++) { // bucle que ompli la matriu amb els valors del 0 al 9
        nums[i] = i;      // assignació del valor de i a l'element de matriu nums[i]
    }
}
```

Comentaris

Comentaris per al codi

Per a comentar el codi, tens:

- un comentari d'una línia, s'escriu després de `//`, normalment, després del codi.
- un comentari de més d'una línia, tancat entre els caràcters `/*` i `*/`

```
/* llegim de teclat els costats del triangle
   calculem l'àrea del triangle
   calculem el perímetre del triangle
*/
Scanner lector = new Scanner(System.in); // crea un Scanner per a introduir dades de teclat
// es lliguen els costats del triangle
System.out.println("Introdueix els tres costats del triangle");
double costat1 = lector.nextDouble();
double costat2 = lector.nextDouble();
double costat3 = lector.nextDouble();
double perimetre = costat1 + costat2 + costat3;
// per al càlcul de l'àrea s'usa fórmula d'Herón
double p = perimetre / 2; // el semi perímetre del triangle
double area = Math.sqrt(p * (p - costat1) * (p - costat2) * (p - costat3)); // fórmula d'Herón
// visualitza el perímetre i l'àrea del triangle
System.out.println("Perímetre = " + perimetre + " àrea = " + area);
```



Comentaris per a l'aplicació

Per a documentar una aplicació usa els **Javadoc**, són uns comentaris especials que permeten documentar els mètodes de les classes, l'entorn els usa per a presentar informació dels mètodes.

Els Javadoc s'escriuen davant de les classes o dels mètodes, i comencen amb `/**` i acaben amb `*/`.

Amb els Javadoc de les classes d'un projecte, pots crear de manera automàtica tota la documentació en format HTML `Run > Generate Javadoc (nomprojecte)`.

El resultat és una pàgina web amb un resum de totes les classes, atributs i mètodes que formen el API del projecte.

El format més habitual d'un Javadoc, és una primera línia amb el que fa el mètode, després més comentaris si són necessaris. Amb `@param` s'indica la informació que rep el mètode i amb `@return` s'indica la informació que retorna el mètode.

```
/**
 * retorna el valor màxim d'una llista de nombres.
 * si hi ha més d'un retorna l'últim valor màxim de la llista
 * @param llista és la llista de nombres a recórrer
 * @return l'enter més gran de la llista de nombres
 */
```

Admeten codi HTML i hi ha més anotacions.

```
/**
 * indica si un client té saldo suficient.
 * @param client és el client a comprovar
 * @return <code>true</code> si el client té saldo suficient<br>
 * <code>false</code> si el client té no saldo suficient
 * @author Eduardo
 */
```

Dades

Una dada és la representació d'una informació, de manera que puga ser tractada per un ordinador, una dada es transforma a una seqüència de bits en memòria.

Per exemple, la frase "*la base d'un rectangle és de 12.4 centímetres*" per a l'ésser humà, pot representar una sola dada, ja que associa de manera automàtica el valor (12.4) i el significat (és la base d'un rectangle i està en centímetres).

Per a l'ordinador, en la frase hi ha molts conceptes: base, rectangle, 12.4, centímetres, etc., alguns es converteixen en dades (base i el seu valor 12.4) uns altres s'associen al codi (rectangle i centímetres), no és l'única forma de definir aquestes dades.



A l'hora de resoldre els problemes has de veure les dades com a elements abstractes que descriuen una determinada lògica, sense pensar com s'implementen en l'ordinador.

No has de preocupar-te pel contingut d'una dada, és un element canviant. La base ara val 12.4, després valdrà 3.56 o qualsevol altre valor, és a dir, has de manejar el concepte lògic de base, no els seus possibles valors.

Elegir un tipus de dades

La lògica d'un problema indica quin tipus de dada triar per a la informació a manejar.

Per a calcular la superfície d'un rectangle necessites dues dades d'entrada, la base i l'altura, que són dos nombres que hauràs de multiplicar.

En l'exemple anterior la base val 12.4, és a dir el tipus de dada que es necessita per a emmagatzemar la base, és un nombre amb decimals. Si la base mai usa decimals, llavors pots triar el tipus nombre enter.

La lògica del problema determina el tipus de les dades a utilitzar.

- si vaig a manejar el nom d'una persona, trie el tipus cadena de text.
- si vaig a manejar una edat, trie el tipus enter.
- si vull saber si una persona és major d'edat o no, trie el tipus booleà.

El tipus de dada triat defineix el conjunt de valors que pot prendre la dada i les operacions que es poden realitzar amb ella.

Si tries una dada de tipus enter, el rang de valors és -2.147.483.648 a 2.147.483.647 i pots sumar-la, restar-la, multiplicar-la però no pots passar-la a majúscules.

Si tries una dada de tipus cadena de text, pots guardar qualsevol combinació de caràcters en ella, pots afegir-li caràcters, passar-los a majúscules però no pots multiplicar-la.

Variables

Una variable és una dada que canvia de valor durant l'execució del programa.

→ Una variable necessita una posició de memòria on guardar els diferents valors que va adquirint durant l'execució.

Quan es defineix una variable, se l'hi dona un nom i un tipus.

El nom és la posició de memòria on es guarda la variable. El nom permet accedir a la posició de memòria per a poder llegir o modificar la variable.

El tipus defineix l'espai que es reserva en memòria.



Declarar

→ Totes les variables de Java s'han de declarar abans d'usar-se.

La sintaxi de la declaració de variables és

```
TipusDeDada identificador [ = valor ] [,identificador [ = valor ] ... ]; // [] indica que l'element és optatiu
```

Primer escriu el tipus de dada de la variable, després l'identificador (la referència) i de manera opcional assigna-li un valor inicial.

```
int numJugadors; // defineix la variable numJugadors de tipus int (un enter)
String nomJugador; // defineix la variable nomJugador de tipus String (una cadena de text)
int num = 10; // defineix num de tipus int i li dona valor 10
Point punt = new Point(5, 8); // defineix punt de tipus Point i li dona valor (un Point en la posició 5,8)
```

Pots declarar més d'una variable del mateix tipus en una sentència, cal separar els identificadors amb comes.

```
double saldo, total; // defineix les variables saldo i total de tipus double
int edat, gruix = 2; // defineix edat i gruix de tipus int, a la variable gruix li dona valor 2
```

Les variables que declares, sense valor inicial, prenen el seu valor per defecte.

- Nombres 0
- Caràcter '\u0000'
- Booleà false
- Classe null

Usa una línia diferent per a cada declaració, encara que siguin del mateix tipus, així podràs posar comentaris per a totes les declaracions.

```
double saldo; // saldo del compte corrent
double total; // total de despeses del mes
```

Una declaració de variable és una expressió, per tant, es pot declarar una variable en altres posicions del codi que no són una sentència.

Identificador

→ L'identificador és un text que substitueix la posició de memòria (un enter) de l'objecte declarat.

Per a un ésser humà, un text és més comprensible que un nombre.

→ Utilitza NOMS SIGNIFICATIUS per als identificadors, el text de l'identificador ha d'estar relacionat amb la informació a la qual dona accés..

```
int edat = 12; // es defineix un enter per a guardar una edat i se li dona el valor 12
int e = 12; // es defineix un enter per a guardar una edat i se li dona el valor 12
```

L'elecció de l'identificador de la segona variable és dolenta, no és prou significatiu.



💀💀💀💀 L'ús d'identificadors dolents BAIXA LA NOTA 💀💀💀💀

Un identificador usa totes les lletres de l'alfabet, incloses lletres amb accent o la ñ, els dígit, el guió baix i el dòlar. Per convenció el dòlar no s'usa, i es desaconsella l'ús del guió baix.

Un identificador no pot començar amb un dígit.

Un identificador és sensible a les majúscules (`maxim` és diferent de `maXim`),

→ Utilitza minúscules per a l'identificador.

Quan un identificador està format per més d'una paraula, a partir de la segona paraula, escriu la primera lletra de cada paraula en majúscula (`dadesVehicles`, `preuVendaMínim`, `matrículaAlumneActiu`). Aquest tipus de notació es diu **CamelCase**.

Paraules reservades

Les paraules reservades tenen un significat definit pel llenguatge.

`abstract, continue, for, new, switch, assert, default, goto, package, synchronized, boolean, do, if, private, this, break, double, implements, protected, throw, byte, else, import, public, throws, case, enum, instanceof, return, transient, catch, extends, int, short, try, char, final, interface, static, void, class, finally, long, strictfp, volatile, const, float, native, super, while`

→ No pots utilitzar les paraules reservades de Java per als teus identificadors.

Constants

Una dada constant no canvia de valor durant l'execució de l'aplicació, hi ha dos tipus:

- amb posició de memòria, és una dada **constant**.
- sense posició de memòria, és un **literal**.

Una **constant** es declara com una variable, però està precedida per la paraula `final`, cal donar-li un valor en la declaració, ja que qualsevol intent de modificació en el codi del programa provoca un error.

```
final tipusDada identificador = valor;
```

L'identificador d'una constant usa el mateix conjunt de caràcters que les variables, però, per convenció s'escriu en majúscules. Si hi ha més d'una paraula, aquestes se separen amb un guió baix.

```
final int AMPLE = 122;
final double PI = 3.1416;
final String NOM_EMPRESA = "T.I.S. SA";
final int LONGITUD_LINIA = 80;
```



La constant **AMPLE** té més significat que el literal 122.

→ Una constant es declara en el bloc de la classe, és a dir, és un camp de la classe.

Un **literal** no té assignat cap posició de memòria, és el compilador qui l'interpreta i el converteix al valor que s'incrusta en el codi del programa.

```
final int AMPLE = 122; // 122 és un literal de tipus enter
final String NOM_EMPRESA = "T.I.S. SA"; // "T.I.S. SA" és un literal de tipus cadena de text
System.out.println("radi = " + (perímetre / 2)); // "radi = " i 2 són literals de tipus text i enter respectivament
```

Només hi ha literals per a dades primitives, en les definicions dels tipus de dades següents s'indica el format del seu literal.

☞ Les primitives es veuen més avant.

Visibilitat

La visibilitat d'un element, és sinònim d'accés, i defineix qui pot accedir (veure) a l'element per a treballar amb ell.

Els diferents **tipus d'accés** a un element són: **públic**, **paquet**, **protegit** o **privat**.

☞ Els tipus d'accés es veuen amb més profunditat en temes següents.

Les referències dins d'un bloc són **privades**, és a dir, només són accessibles en el bloc, es diu que són **locals** al bloc.

☞ En aquest tema, només usaràs referències locals.

<pre>{ int a = 3 { int b = a * 3 { int c = a + b a = a + b + c } } a = a + b }</pre>	<pre>// inici del bloc, defineix a, es veu a // inici del bloc, defineix b, es veu a i b // inici del bloc, defineix c, es veu a, b i c // es veu a, b i c // final del bloc, es destrueix c // final del bloc, es destrueix b // es veu a, ERROR no es veu la variable b ja està destruïda // final del bloc, es destrueix a</pre>
--	---

Les variables es declaren el més prop d'on s'usen. S'usen i després es destrueixen quan ja no es necessiten, i es reutilitza l'espai de memòria assignat.

<pre>{ int a = 3; int b = a * 3 { int b = 5 b = a * b System.out.println("b = " + b) } System.out.println("b = " + b) }</pre>	<pre>// inici del bloc, defineix un enter a {3} // defineix un enter b {9} // inici del bloc, defineix un enter b {5}, és un altre espai de memòria // multiplica a {3} per b {5} i ho guarda en b (el 15 substitueix al 5) // visualitza b = 15 // final del bloc, destrueix l'enter b, el definit en aquest bloc (el segon) // visualitza b = 9 // final del bloc, destrueix l'enter b, el definit en aquest bloc (el primer)</pre>
---	---



Pots usar el mateix identificador en blocs diferents, les referències `b` són posicions de memòria diferents, el codi accedeix a la referència definida més a prop.

→ Quan es tanca un bloc es destrueixen totes les referències locals que conté.

→ Si es destrueix la referència es perd l'objecte associat.

El **Garbage Collector** de Java és l'encarregat de recuperar la memòria ocupada pels objectes que no tenen cap referència. És un procés que s'està executant contínuament, però amb prioritat molt baixa.

Tipus de dades

→ Un tipus de dada defineix el conjunt de valors vàlids que poden prendre unes dades i el conjunt de transformacions que s'hi pot fer.

Per exemple, si vols definir una distància, com un enter (`int`) aquesta distancia no tindrà decimals, podrà valdre de -2147483648 a 2147483647, i podràs fer operacions aritmètiques amb ella (sumar, restar, dividir, etc.).

Classes

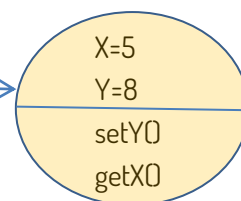
A Java les classes funcionen com un tipus de dada, el llenguatge ens ofereix una multitud de classes, `String`, `Point`, `LocalDate`, `Calendar`, `Boolean`, etc.

```
Point punt; // defineix la referència punt de tipus Point que val null, ja que no s'ha creat cap objecte
```

→ El valor per defecte d'una referència a una classe és `null`, és a dir, l'objecte no existeix i no podem demanar-li res.

La sentència `new` crea l'objecte associat a la referència, reserva espai en memòria i crida al constructor de l'objecte del tipus de la classe.

```
Point punt = new Point(5, 8);
```



La classe `Point` té els camps `X`, `Y` (les propietats d'un `Point`) i els mètodes `setY` i `getX` (comportaments que es poden demanar a un `Point`).

La referència de l'objecte permet manejar els seus camps i mètodes.

Una classe creada per nosaltres és un tipus de dada nou, per exemple, vull manejar vehicles per a definir la seua marca, el seu preu, si ha passat la ITV o no, qui és el seu propietari, etc., per a això crearé la classe `Vehicle` amb aquests comportaments.



Operador punt (.)

Per a accedir als camps o als mètodes d'una classe s'utilitza l'operador punt, només dona accés als elements visibles la classe (API).

L'operador punt, també es diu **missatge**.

Has d'escriure una referència o un nom de la classe, seguit de l'operador punt seguit del nom de l'element de l'objecte que volem manejar.

Amb la referència, l'accés és dinàmic i tens accés als elements dinàmics i estàtics.

Amb el nom de la classe, l'accés és estàtic i tens accés, únicament, als elements estàtics.

```
String cad = new String("Hola Món"); // creació del String cad amb el text "Hola Món"
System.out.println(cad.toString()); // visualitza "Hola Món" en la consola
```

La classe `System` accedeix a l'objecte `out` que accedeix al mètode `println` que visualitza un text, el text el proporciona la referència `cad` que accedeix al mètode `toString` que retorna un text.

```
double num = Math.random() * 100; // num és un nombre aleatori amb decimals
```

La classe `Math` accedeix al mètode `random` que retorna un valor aleatori entre 0 i 1 (un 0 amb decimals) aquest valor es multiplica per 100 i s'assigna a la variable `num`.

Primitives

Les dades primitives no necessiten un `new` per a crear-les, la creació de l'objecte està associada a l'assignació. Són una forma d'escriure heretada de llenguatges anteriors.

→ Una primitiva sempre té un valor (un objecte) i no té mètodes.

→ Una classe pot no tindre valor (`null`) i té mètodes.

Hi ha 8 tipus de primitives `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` i `double`, i únicament contenen un valor del tipus de la dada.



→ La referència d'una primitiva dona accés únicament al valor de l'objecte.

S'usen per a manejar

- nombres sense decimals: `byte`, `short`, `int` i `long`
- nombres amb decimals: `float` i `double`
- text: `char`
- valors booleans: `boolean`



Totes les primitives tenen una **classe embolcall** (**wrapper**) que ofereix diversos mètodes, per exemple, la classe **Integer** és la classe embolcall de **int**. Les primitives s'emboiquen i desemboliquen de manera automàtica en la seua classe embolcall.

Els objectes d'una classe embolcall d'una primitiva són **immutables**, és a dir, no canvien una vegada s'han creat. Els canvis de valor d'una primitiva suposa la creació d'objectes nous.

Nombre sense decimals

Per a representar un nombre sense decimals, un **enter**, tens els tipus següents:

- **byte** és un enter de 8 bits, el rang de valors és de -2^7 a 2^7-1 (inclusivament), és a dir, de **-128** a **127**, s'usa per a estalviar memòria o per a nombres compresos en aquest rang.
- **short** és un enter de 16 bits, el rang de valors és de -2^{15} a $2^{15}-1$ (inclusivament), és a dir, de **-32768** a **32767**, s'usa per les mateixes raons que byte.
- **int** és un enter de 32 bits, el rang de valors és de -2^{31} a $2^{31}-1$ (inclusivament), és a dir, de **-2147483648** a **2147483647**
- **long** és un enter de 64 bits, el rang és de -2^{63} a $2^{63}-1$ (inclusivament), és a dir, de **-9223372036854775808** a **9223372036854775807**

```
int a;           // declara un enter de tipus int amb la referència a
int base = 23;   // declara un enter de tipus int amb la referència base i li dona valor 23
byte mascara;    // declara un enter de tipus byte amb la referència mascara
```

Habitualment, usarem **int** o **Integer** per a representar un enter.

Literal

- En base 10, el literal usa els dígit del **0** al **9** i els signes **+** i **-** (si no s'escriu res, es considera el nombre positiu). Un literal per a un nombre de tipus **long** finalitza amb una **L**.

```
short curt = -2689;
int valor = +26;    // el valor 26 en base 10
long àtoms = 1254994761616L;    // el literal per a un long té L al final
```

- En base 8, el literal usa els dígit del **0** al **7** i els signes **+** i **-**, i comença amb **0**.

```
int valor = 032;    // el valor 26 en base 10
```

- En base 16, el literal usa els dígit del **0** al **9**, els caràcters de la **A** a la **F** (minúscules o majúscules), els signes **+** i **-**, i comença amb **0x**.

```
int valor = 0x1a;    // el valor 26 en base 10
int max = 0xFFFF;   // el valor 65535 en base 10
```

- En base 2, el literal usa els dígit **0** i **1**, els signes **+** i **-**, i comença amb **0b**. No se solen posar signes als valors binaris.

```
int valor = 0b11010;    // el valor 26 en base 10
long esGran = 0b1101001001101001100101001001001;    // el valor 1765067337 en base 10
```



Guió baix en els literals numèrics

Pots usar el guió baix per a separar dígitos en un literal numèric i així donar-li més significat al literal, però no afecta el valor del literal.

Pots posar un o més guions baixos en un literal, junts o separats.

NO pots posar un guió a l'inici o final del literal, al costat del punt decimal, al costat de la lletra **x**, al costat de la lletra **b** i abans del sufix **L** o **F**.

```
long numTargeta = 1234_5678_9012_3456L;      // es guarda 1234567890123456L
long nombreSeguridadSocial = 999_99_9999L;    // es guarda 9999999999L
long hexBytes = 0xFF_EC_DE_5E;               // es guarda 0xFFECDE5E
long hexWords = 0xBEBE_CAFE;                  // es guarda 0xBEBECAFE
long maxLong = 0x7fff_ffff_ffff_ffffL;       // es guarda 0x7fffffffffffffffL
byte nybbles = 0b0010_0101;                  // es guarda 0b00100101
long bytes = 0b11010010_01101001_10010100_10010010; // es guarda 0b110100100110100110010010010010
```

També, pots usar guions baixos per a literals amb decimals (es veuen més avant)

```
float pi = 3.14_15f; // es guarda 3.1415f
```

Classes embolcalls

Les classes embolcall per a la primitives de tipus enter són: **Byte**, **Short**, **Integer** i **Long** (la majúscula indica que és una classe).

Pots escriure expressions que mesclen el tipus primitiu i la seua classe embolcall.

```
Integer a = new Integer(10); // la classe Integer amb el valor 10, es pot escriure Integer a = 10;
int b = 23;                  // la primitiva int amb el valor 23
int c = a + b;               // es sumen els dos valors i s'assigna a una primitiva
Integer d = a + b;           // es sumen els dos valors i s'assigna a una classe
```

Els objectes d'una classe embolcall són immutables (no poden canviar)

```
Integer a = 13;              // el literal 13 s'embolcalla en un Integer i s'assigna a a
System.out.println("a= " + a + " idhc=" + System.identityHashCode(a));
a = a + 3;                   // el literal 3 s'embolcalla en un Integer que se suma a a i s'assigna a a
System.out.println("a= " + a + " idhc=" + System.identityHashCode(a));
// visualitza
a= 13 idhc=363771819
a= 16 idhc=793589513
```

Es visualitza el valor de **a** i el seu **codi hash** (quan es crea un objecte se li assigna un codi hash únic). El codi hash canvia, ja que l'objecte associat a **a** ha canviat, la suma es guarda en un objecte nou amb el valor 16.

Les classes embolcalls tenen constants que donen informació del tipus de dada. Per exemple, **Integer** té **BYTES**, **MAX_VALUE**, **MIN_VALUE**, **SIZE**

```
System.out.println("mínim " + Integer.MIN_VALUE); // visualitza mínim -2147483648
```



Mètodes de la classe Integer

`static int parseInt(String s)`

Analitza el text `s` i intenta transformar-lo a un enter amb signe, usant la base 10.

`static int parseInt(String s, int radix)`

Analitza el text `s` i intenta transformar-lo a un enter amb signe, usant la base `radix`.

`long longValue()`

Retorna el valor de l'enter como un `long`.

`static String toBinaryString(int i)`

Retorna una cadena de text amb l'enter `i` sense signe en base 2.

`static String toHexString(int i)`

Retorna una cadena de text amb l'enter `i` sense signe en base 16.

La classe `Integer` té més mètodes.

Altres classes per als enters

La classe `BigInteger` ofereix enters de major rang i precisió. Les classes `AtomicInteger` i `AtomicLong` s'usen en aplicacions multi-fil.

```
BigInteger res = new BigInteger("240958240958240958"); // declaració d'un BigInteger
res = res.add(BigInteger.valueOf(789));                // suma de dos BigInteger
```

Nombre amb decimals

Per a representar un nombre amb decimals, un **real**, tens els tipus següents:

- **float** és un real de 32 bits, el rang és de **1.40129846432481707e-45** a **3.40282346638528860e+38** (positiu o negatiu)
- **double** és un real de 64 bits, el rang és de **4.94065645841246544e-324** a **1.79769313486231570e+308** (positiu o negatiu)

```
float pes; // declara un float amb referència pes
double costat = 12.6; // declara un double amb referència costat amb el valor 12.6
```

Habitualment, usarem `double` o `Double` per a representar un real.

Literal

Un literal per a un real, sempre està en base 10, usa els dígit del 0 al 9 i els signes `+` i `-`, el punt decimal `.` i la lletra `E` o `e`.

```
double pes = 10.17;
double pendent = -2.6;
```



La **e**, en minúscula o majúscula, s'utilitza en la notació científica d'un nombre que s'expressa com **cEn** i equival a **c * 10ⁿ**, on **c** és el coeficient (un valor real major que 1 i menor que 10) i **n** és l'exponent (un valor enter), **c** i **n** poden ser negatius.

```
double valor = 1.234e2;    // notació científica de 123.4
double pro = 1.234E-2;    // notació científica de 0.01234
double valor = 1.0e200;    // és el valor 1 seguit de dos-cents 0 (1000000000 ... 000000000)
double pro = 1.0E-200;    // és el valor 0. seguit de dos-cents 0 i un 1 (0.000000000 ... 00000001)
```

Per a un literal de tipus **float** cal afegir-li el sufix **f** o **F**.

```
float pes = 10.17F;
float pesGran = 5.436e4f;
```

Precisió

L'elecció del tipus **float** o **double** no sols afecta el rang de valors, sinó també a la precisió del nombre, és a dir al nombre de xifres diferents de zero.

```
float nf = 12345678901234567890.0f;
double nd = 12345678901234567890.0;
System.out.println("float = " + nf + "\ndouble = " + nd);
// visualitza
float    = 1.2345679E19
double   = 1.2345678901234567E19
```

En l'exemple, definim dues variables **nf** i **nd** definides com a **float** i **double**, i assignem el mateix literal (valor) a les dues. Com es pot apreciar, quan es visualitzen les variables, el **float** té 8 xifres diferents de zero i el **double** té 17.

Classes embolcalls

Les classes embolcall són **Float** i **Double**.

Mètodes de la classe Double

static double parseDouble(String s)

Analitza **s** i intenta transformar-lo a un **double** amb signe.

static double max(double a, double b)

Retorna el valor major de **a** i **b**.

static int compare(double d1, double d2)

Retorna la comparació de **d1** i **d2**, torna 0 si són iguals, un valor negatiu si **d1** és menor que **d2** i un valor positiu si **d1** és major que **d2**.

La classe **Double** té més mètodes.

Altres classes per a nombres reals

La classe **BigDecimal** ofereix un rang i precisió major.



```
BigDecimal prestec = new BigDecimal("122115.32"); // declaració
BigDecimal interes = BigDecimal.valueOf(0.49); // declaració
BigDecimal deute = prestec.multiply(interres); // multiplicació
```

→ Usa `int` per a nombres enters.

→ Usa `double` per a nombres reals.

→ Usa `String` (cadena de text) per a nombres amb els quals no fas càlculs aritmètics.

Caràcter

Un caràcter s'utilitza per a representar una lletra, una xifra (que no s'usa en operacions aritmètiques) o un símbol.

Un caràcter es codifica com un nombre enter sense signe, el seu valor prové d'un mapa de símbols, en el qual s'associa cada caràcter amb un valor. Aquests mapes de representació de caràcters s'anomenen taules, les més conegudes són la taula `ASCII` i la `UNICODE`.

Java treballa amb representació `UNICODE`, que usa 16 bits, per tant, pot representar un valor enter des del 0 a 65535, és a dir, 65536 caràcters.

El tipus primitiu és `char`, i la seua classe embolcall és `Character`.

Literal

El literal per a representar un caràcter, és el caràcter tancat entre cometes simples.

```
char lletra = 'A'; // defineix la referència lletra i li assigna el caràcter 'A'
```

Els caràcters que usen la codificació Unicode (UTF-16), es poden expressar mitjançant la seua `seqüència d'escapament`, aquesta comença amb `\u` seguit del valor en hexadecimal del caràcter, tot escrit entre cometes simples (és un únic caràcter). S'usa per a obtenir caràcters que no estan en el teclat.

```
char eñe = '\u00F1'; // seqüència d'escapament del caràcter 'ñ'
char perMil = '\u2031'; // seqüència d'escapament del caràcter '‰'
char calavera = '\u2620'; // seqüència d'escapament del caràcter '☠'
```

La visualització dependrà de l'element que la realitza.

També, es pot utilitzar el valor en base 10 del codi del caràcter.

```
char titlla = 126; // valor en base 10 del caràcter '~'
char eñe = 241; // valor en base 10 del caràcter 'ñ'
char setRoma = 8550; // valor en base 10 del caràcter 'ⅤⅡ'
```

En l'adreça <https://unicode-table.com/es/> pot tots els caràcters Unicode.

Hi ha seqüències d'escapament amb un significat predefinit: `\b` (retrocés), `\t` (tab), `\n` (línia nova), `\f` (pàgina nova), `\r` (retorn de carro), `\"` (cometes dobles), `\'` (cometes simples), i `\\` (contrabarra), és un caràcter.



Classe embolcall

La seua classe embolcall és `Character`.

Mètodes de la classe `Character`

`static boolean isDigit(char ch)`

Retorna `true` si `ch` és un dígit i `false` si no ho és.

`static boolean isLetter(char ch)`

Retorna `true` si `ch` és una letra i `false` si no ho és.

`static char toUpperCase(char ch)`

Retorna el valor `ch` passat a majúscules.

La classe `Character` té més mètodes: `isAlphabetic`, `isLetterOrDigit`, `isLowerCase`, `isUpperCase`, `isWhitespace`, `compare`, `compareTo`, `digit`, `equals`, `toString`, `toLowerCase`, `toUpperCase`, `valueOf`, etc.

```
char set = 8550; // valor en base 10 del caràcter 'VII'
System.out.println(Character.isDigit(set)); // visualitza false, 'VII' no és un dígit
set = '7';
System.out.println(Character.isDigit(set)); // visualitza true, '7' és un dígit
```

Cadenes de text

Una cadena de text s'utilitza per a representar un conjunt de caràcters.

A Java, una cadena de text es defineix amb la classe `String`, però el llenguatge ens ofereix dos operadors `=` i `+` que permeten usar aquesta classe com un tipus primitiu.

```
String text;
System.out.println(text); // visualitza null que és el valor per defecte d'un objecte
```

Literal

El literal per a una cadena de caràcters es defineix tancant els caràcters que formen la cadena entre cometes dobles, hi pot haver zero o molts caràcters.

```
String text = "Torna arrere!"; // és una cadena amb el text "Torna arrere!"
String cad = ""; // és una cadena buida
String lletra = "A"; // cadena amb una sola lletra
String telefon = "666012345"; // cadena de text amb xifres únicament
```

Entre les cometes dobles es pot tancar qualsevol caràcter del teclat, junt amb qualsevol seqüència d'escapament.

```
String acierto = "S\u00ED se\u00F1or"; // és el text "Sí señor"
String interes = "interés de 3\u2031"; // és el text "interés de 3%"
String dos = "línea 1\nlínea 2"; // el \n baixa de línia
String valor1 = "\tEuros"; // el \t és una tabulació
String bravo = "\BRAVO" n° '5'; // les cometes necessiten \ davant, el text és ""BRAVO" n° '5'"
```



Usa variables de tipus cadena per a guardar un caràcter, abans que variables de tipus caràcter, la classe `String` ofereix més possibilitats.

➡ Més avant es veuen mètodes de la classe `String`.

Booleà

Un booleà és un tipus de dada que, únicament, té dos estat: veritat i falsedat.

Per a declarar una variable booleana utilitzarem el tipus primitiu `boolean` que usa 1 bit, o la classe embolcall `Boolean`.

```
boolean majorDedat;
```

Literal

Un booleà, té dos literals, `true` per a vertader i `false` per a fals

```
boolean majorDedat = false;    // declaració i inicialització a fals
Boolean majorDedat = true;     // declaració i inicialització a vertader
```



Exercicis

1.

Quin tipus de dada triaries per a representar:

- El nom d'una persona
- El telèfon d'una persona
- El nombre pi = 3.1416
- El preu d'una tomaca
- Una nota de l'assignatura de programació
- Estar aprovat o no
- El nombre d'assignatures de DAM
- La marca d'un ordinador portàtil
- El nombre d'ordinadors de l'aula d'informàtica
- El nombre de bastidor d'un cotxe
- El mes de març
- El fet que una persona siga major d'edat o no

2.

Un establiment vol controlar les persones que entren en ell, les dades que li interessin són, l'edat, el pes, l'altura i si és un home, una dona o un xiquet. Quin tipus de dada triaries per a representar una persona?



Operadors

Punt .

L'operador punt permet accedir a un element d'un objecte el seu format és `objecte.element`. També, s'anomena missatge, ja que transmet un missatge a l'objecte.

El significat del punt canvia segon l'element accedit

- per a un mètode, significa que l'objecte ha d'executar el mètode
- per a una referència, significa que l'objecte ha de donar accés a la referència

```
punt.getDistancia() // l'objecte punt executa el mètode getDistancia
matriu.length       // l'objecte matriu dona accés a la referència length
```

En una sentència pot haver-hi més d'un operador punt.

```
System.out.println(punt.getDistancia()); // visualitza el valor retornat pel mètode getDistancia de punt
```

La classe `System` envia un missatge a `out` per a accedir a l'objecte, que al seu torn aquest objecte envia un missatge a `println` per a executar-lo, entre parèntesis la referència `punt` envia un missatge a `getDistancia` per a executar-lo.

```
int numElements = matriu.length; // assigna el valor de length de matriu a numElements
```

La referència `matriu` accedeix a la referència `length` i recupera el seu valor que s'assigna a `numElements`.

Aritmètics

Els operadors aritmètics s'utilitzen per a realitzar operacions amb nombres

- **+** **Suma** de dos nombres
- **-** **Resta** de dos nombres
- ***** **Multiplicació** de dos nombres
- **/** **Divisió** de dos nombres
- **%** **Mòdul** de dos nombres (és la resta de la divisió entera dels nombres)

Si mescles nombres enters i reals, llavors els enters es converteixen en reals i el resultat és real, és a dir, es mantenen els decimals.

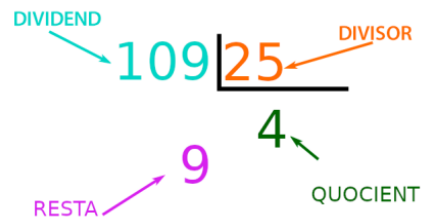
```
int a = 10; // declaració de les variables enteres
int b = 3;
int c;
double d = 2.1; // declaració de les variables reals
double e;
c = a + b; // suma a(10) i b(3) i guarda 13 en c
e = a + b; // suma a(10) i b(3) i guarda 13.0 en e
d = a + d; // suma a(10) i d(2.1) i guarda 12.1 en d, 10 es converteix a 10.0
d = d * e; // multiplica d(12.1) per e(13) i guarda 157.2999 en d, 13 es converteix en 13.0
d = d - a; // resta a(10) de d(157.2999) i guarda 147.2999 en d, 10 es converteix en 10.0
```



L'operador `/` retorna el quocient de la divisió.

Si divideixes dos enters, llavors la divisió és entera i el resultat és enter (es perden els decimals).

Si el dividend o el divisor és real, llavors la divisió és real i es mantenen els decimals.



```
int quocient = 109 / 25;    // quocient val 4, es perden els decimals
double div = 109.0 / 25;   // div val 4.36, es transforma el 25 a 25.0
div = 109 / 25.0;          // div val 4.36, es transforma el 109 a 109.0
div = 109.0 / 25.0;        // div val 4.36, el dividend i el divisor són reals
int a = 10;
System.out.println(a/3);    // visualitza 3, divideix a(10) per 3, dos enters, per tant divisió entera
System.out.println(a/3.0); // visualitza 3.3333, divideix a(10) per 3.0, enter i real, per tant divisió real
```

L'operador `%` retorna la resta de la divisió entera.

```
int resta = 109 % 25;    // el mòdul val 9
resta = 25 % 109;        // el mòdul val 25
```

El mòdul és la resta de divisió entera, també funciona amb valors reals, però el resultat té decimals.

```
double restad = 25.0 % 10.9;    // el mòdul val 3.1999999999999993
// els càlculs que es realitzen són els següents:
double quocient = 25.0 / 10.9;  // el quocient val 2.293577981651376
int quocientEnter = (int) quocient; // el quocient enter val 2, s'eliminen els decimals (casting)
double restad = 25.0 - quocientEnter * 10.9; // la resta val 3.1999999999999993
```

Una expressió pot contindre tots els operadors que vulgues.

```
int a = 10, b = 3, c;    // declaració de les variables enteres
c = a + b * b + a / b;    // c val 22.33333
```

El resultat de l'expressió el defineix la **precedència** dels operadors (orde d'execució), per exemple el producte s'executa abans que la suma.

☞ La taula de precedència es veu més avant.

→ Posa parèntesi per a aclarir les expressions

Els parèntesis tenen una precedència major a la dels operadors i s'avaluen abans.

```
c = (a + b) * ((b + a) / b);    // c val 56.33333
```

L'avaluació d'una expressió comença en els parèntesis més interns, en l'exemple anterior el primer que s'avalua és $(b + a)$.

Matemàtics

La classe `Math` ofereix un munt de mètodes per a realitzar operacions matemàtiques.



- `abs(n)` valor absolut de n (el valor positiu)
- `asin(a)` arc sinus de l'angle a
- `ceil(n)` arrodoniment a l'alça de n (n = 3.33 -> n = 4.0)
- `floor(n)` arrodoniment a la baixa de n (n = 3.63 -> n = 3.0)
- `round(n)` arrodoniment de n (n = 3.49 -> n = 3.0, n = 3.50 -> n = 4.0)
- `max(a, b)` màxim d'a i b
- `min(a, b)` mínim d'a i b
- `pow(a, b)` a elevat a b
- `random()` valor aleatori entre 0 i 1 (0<= valor <1)
- `sqrt(n)` arrel quadrada de n (n ha de ser positiu)

Hi ha molts més mètodes.

```
System.out.println(Math.round(3.49));
double aleatori = Math.random() * 100;
```

Assignació

L'operador d'assignació és el `=`, que assigna el valor de la seua dreta a la variable de la seua esquerra.

variable = valor
←

→ L'element receptor ha de ser una referència per a poder guardar el valor en la seua posició de memòria.

Si la referència és una primitiva, llavors s'ha creat un nou objecte amb el valor, i l'objecte s'assigna a la referència.

```
a = 2; // assigna el valor del literal 2 a la referència a
b = -2; // assigna el valor del literal -2 a la referència b
c = (a + 8) * (8 - a) * (a - b); // avalua l'expressió (val 240) i assigna el valor a la referència c
```

→ L'assignació és l'última acció que s'executa en una sentència (té la precedència més baixa)

Si escrivim l'assignació després d'un operador aritmètic (`+=`, `-=`, `*=`, `/=`, `%=`), llavors es fa una **acumulació**, ja que es realitza l'operació i es guarda el resultat en el mateix operand. Els dos caràcters s'escriuen junts.

```
a = 10;
b = 4;
a += 3; // equival a = a + 3; ara a val 13
a -= b; // equival a = a - b; ara a val 9
a *= b - 2; // equival a = a * (b - 2); ara a val 18
a /= 4; // equival a = a / 4; ara a val 4 (divisió entera)
a %= 3; // equival a = a % 3; ara a val 1
```



Unaris

Els operadors unaris s'apliquen a un sol operand

- **+** **positiu**, no canvia el valor del nombre
- **-** **negatiu**, canvia el signe al valor del nombre
- **++** **increment**, suma 1 al valor del nombre i l'assigna a la variable
- **--** **decrement**, resta 1 al valor del nombre i l'assigna a la variable
- **!** **no lògic**, inverteix el valor del booleà

```
int a;  
int b = 3;  
a = -b; //obté el valor de b (3) el canvia de signe (-3) i l'assigna a a  
System.out.println("b = " + b); // visualitza b = 3 b no ha canviat, el + és la concatenació de cadenes  
System.out.println("a = " + a); // visualitza a = -3  
esMajor = !esMajor; // inverteix el valor de la variable booleana
```

Els operadors **+**, **-** i **!** no canvien el contingut de la variable a la qual s'apliquen, els operadors **++** i **--**, sí.

Els operadors **++** i **--** funcionen de forma diferent si estan escrits davant (notació prefixa) o darrere (notació postfixa) de l'operand.

Si l'operador està sols, llavors les dues notacions tenen el mateix resultat.

```
int n = 10;  
n++; //Incrementa el valor de n, n val 11  
n = 10;  
++n; //Incrementa el valor de n, n val 11
```

Una expressió s'avalua d'esquerra a dreta

- si el primer que es troba és l'operador, llavors es fa l'operació i després es recupera el seu valor.
- si el primer que es troba és l'operand, llavors es recupera el seu valor i després es fa l'operació.

```
int n = 10;  
int x = ++n; // primer incrementa n, després assigna n a x, n val 11 i x val 11  
n = 10;  
int x = n++; // primer assigna n a x, després incrementa n, n val 11 i x val 10
```

En l'exemple anterior, l'increment està associat amb l'assignació i la posició de l'operador canvia el valor assignat, però en els dos casos s'incrementa **n**.

☞ En els exercicis d'aquest tema, escriuràs el codi en el mètode **main** de la classe principal i cridaràs a mètodes d'objectes proporcionats pel llenguatge Java.

Exercicis

3. increment

Quin valor es visualitza en la consola el programa següent? a) 1 b) 2 c) 3 d) 6 e) 7 f) 8

```
public class Principal {
```



```

static int x, y;
public static void main(String[] args) {
    {int x = 5;}
    x--;
    y = x++ + ++x;
    y += x + ++x;
    System.out.println(y); // visualitza en la consola el valor de y
}
}

```

Primer fes-ho sobre paper, després ho proves en l'ordinador, crea el projecte “increment” i la classe principal s’anomena “Principal”.

4. opsaritmetics

Què val la variable `a` després de cada sentència? Totes les sentències van seguides, per tant, la variable `a` té el valor obtingut en la sentència anterior.

```

a = 3;
b = -3;
a = a + 1;
a = a + 2 * b;
a = a - 2 * b - 2;
a = a / b - 3;
a = (a - b * 4) % a;

```

Primer fes-ho sobre paper, després ho proves en l'ordinador, crea el projecte “opsaritmetics”, la sentència per a mostrar `a` en la consola és `System.out.println(a);`.

Primer defineix `a` i `b` com a enter i després com a real.

Relacionals

Els operadors relacionals són els següents:

- `==` igual que
- `!=` diferent que
- `>` major que
- `>=` major o igual que
- `<` menor que
- `<=` menor o igual que

→ Aquests operadors sols funcionen amb primitives.

→ No confongues el `=` (assignació) i el `==` (comparació).

→ Sols pots comparar dades del mateix tipus.

→ Si pots triar, usa `>`, `>=`, `<`, `<=` abans que `!=`, `==` (`>`, `>=`, `<`, `<=` es compleixen per a molts valors, `!=`, `==`, només es compleixen per a un únic valor).

El resultat d'aquests operadors és un booleà (vertader o fals).

```

int x = 10;
int i = 3;
x == i // és false      x != i // és true
x > i  // és true       x < i   // és false
x >= i // és true       x <= i  // és false

```



Per a comprovar la igualtat de dos objectes s'utilitza el mètode `equals` que retorna `true` si són iguals o `false` si són diferents.

```
boolean res = ref1.equals(ref2); // l'objecte és ref1 i ref2 és el paràmetre
```

El mètode `compareTo(referència)` o `compare(referència, referència)` compara dos objectes, retorna un enter:

- **0** si aquest objecte és igual al paràmetre.
- **Positiu** si aquest objecte és major que el paràmetre.
- **Negatiu** si aquest objecte és menor que el paràmetre.

```
Double num = 24.9; // defineix num (objecte) amb el valor 24.9
double n = 2;      // defineix n (primitiva) amb el valor 2
int res = num.compareTo(n); // demana a num que es compare amb n, res val 1, num > n
res = Double.compare(n, 3.3); // demana a Double que compare n i 3.3, res val -1, n < 3.3
```

Lògics

Els operadors lògics permeten realitzar operacions amb valors booleans (normalment resultats d'expressions relacionals).

- `&&` **I** lògic
- `||` **O** lògic
- `!` **NO** lògic

Les taules de veritat dels operadors són les següents:

a	!a
true	false
false	true

a	b	a && b	a b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

```
int a = 10, b = 3;
boolean seSuma = false;
a > b && b > 0 // true && true dona true,
a > 0 && a < 0 // true && false dona false,
a > 0 || b < 0 // true || false dona true,
!(a > 0) || b < 0 // false || false dona false,
a > 0 || b < 0 && a < b // true || false && false dona true, es calcula primer l'i lògic
(a > 0 || b < 0) && a < b // (true || false) && false dona false, es calcula primer els parèntesis
(a > 0 || b < 0) && (a < b || b < 0) // (true || false) && (false || false) dona false
seSuma && a < b // false || false dona false
!seSuma && a > b // true || true dona true
```

→ Si una expressió té `&&` i `||` combinats, llavors, s'avalua primer el `&&`.

→ Si una expressió té més de dos operadors lògics, uneix les condicions de dues en dues amb parèntesis (facilita les coses).



En l'exemple següent, la variable `esDeTraspàs` indica si un any és de traspàs (`true`) o no (`false`). Un any és de traspàs, si és divisible per 4 però no per 100 però sí per 400 (usa l'operador `%` per a saber si l'any és divisible).

```
int any = 1998; // comprova
boolean esDeTraspàs = (any % 4 == 0); // si any és divisible per 4
esDeTraspàs = esDeTraspàs && (any % 100 != 0); // I si any no és divisible per 100
esDeTraspàs = esDeTraspàs || (any % 400 == 0); // O si any és divisible per 400
// totes les operacions expressades en una sola línia
esDeTraspàs = (any % 4 == 0) && (any % 100 != 0) || (any % 400 == 0);
```

Una dada booleana no es compara amb `true` o `false`, només es posa la referència, l'identificador ha de ser significatiu, normalment comença amb "es" (valencià, castellà) o "is" (anglès).

```
if (esDeTraspàs) { // es comprova si esDeTraspàs és vertader
    System.out.println("és un any de traspàs");
}
if (!esMajorDedat) { // es comprova si esMajorDedat és fals
    System.out.println("no pots conduir");
}
```

L'avaluació d'una expressió amb operadors lògics finalitza quan s'ha determinat el resultat final, encara que queden expressions per a avaluar.

```
int n = 3;
int v = 7;
boolean res = n < -3 || n == 3 || v == 7 // || v == 7 no s'avalua, ja que no afecta el resultat final
```

Una expressió s'avalua d'esquerra a dreta.

```
n < -3 || n == 3 || v == 7 // s'avalua n < -3 que dona false
false || n == 3 || v == 7 // s'avalua n == 3 que dona true
false || true || v == 7 // s'avalua false || true que dona true
true || v == 7 // true || qualsevol cosa dona true, per tant v == 7 no s'avalua
```

```
n == 5 && (n > 6 || v < 12)
false && (n > 6 || v < 12) // s'avalua n == 5 que dona false
false && (n > 6 || v < 12) // false && qualsevol cosa dona false, per tant (n > 6 || v < 12) no s'avalua
```

Aquesta qualitat és útil en expressions com la següent

```
if (ref != null && ref.getEdad() > 18)
```

L'expressió `ref.getEdad()` dona error si `ref` és `null`, però la segona expressió no arriba a avaluar-se si `ref` és `null`, per tant, no es produeix l'error.

 En els exercicis següents, el nom del projecte està junt al nombre de l'exercici.



Exercicis

5. opslogics

Què visualitzen les expressions següents (primer paper, després ordinador)?

```
boolean b = true;
```




```

boolean c = false;
int nw = 2;
int nx = 20;
int ny = 10;
int nz = 5;
System.out.println("!b " + (!b));
System.out.println("0 != 1 " + (0 != 1));
System.out.println("b != !b " + (b != !b));
System.out.println("b == !b " + (b == !b));
System.out.println("!b " + (! !b));
System.out.println("c && !b " + (c && !b));
System.out.println("b || nx > 0 " + (b || nx > 0));
System.out.println("nx > ny && nz > nw " + (nx > ny && nz > nw));
System.out.println("nx < ny && nz > nw " + (nx < ny && nz > nw));
System.out.println("nx < ny || nz > nw " + (nx < ny || nz > nw));
System.out.println("nx - 4 < ny || nz > nw + 10 " + (nx - 4 < ny || nz > nw + 10));
// els && s'avaluen abans que els ||
System.out.println("10 > 5 && !(10 < 9) || 3 <= 4 " + (10 > 5 && !(10 < 9) || 3 <= 4));
System.out.println("b && !c || b " + (b && !c || b));
System.out.println("b && !(c || b)) " + (b && !(c || b)));
System.out.println("nx != 0 && ny == 0 && nz != 0) " + (nx != 0 && ny == 0 && nz != 0));
System.out.println("nx != 0 || ny == 0 || nz != 0) " + (nx != 0 || ny == 0 || nz != 0));
System.out.println("nx != 0 && ny == 0 || nz != 0) " + (nx != 0 && ny == 0 || nz != 0));
System.out.println("nw > 10 && nx > 20 || ny > 0 && nz != 0) " + (nw > 10 && nx > 20 || ny > 0 && nz != 0));
System.out.println("nw != 2 && nx == 20 || ny > 0 && nz == 10) " + (nw != 2 && nx == 20 || ny > 0 && nz == 10));

```

Primer s'avalua l'expressió entre parèntesis, després es passa a text i finalment es concatena al literal.

6.

Què val l'expressió **tinc maduixes i (tinc llet o tinc iogurt)** per als casos següents:

- Si tinc llet i iogurt però no tinc maduixes
- Si no tinc maduixes i ni llet, però tinc iogurt
- Si tinc maduixes i gelat
- Si no tinc llet i ni iogurt
- Si tinc llet i maduixes
- Si tinc iogurt i maduixes
- Si no tinc res

De cadenes

String és una classe (el seu nom comença amb majúscula), té constructors i mètodes per a treballar amb les cadenes de text, però el llenguatge té dos operadors per a poder treballar amb les cadenes com si foren dades primitives (es considera la novena primitiva).

- = **assignació** que inclou la creació de la cadena `new String()`
- + **concatenació** de cadenes (el mètode `concat` fa el mateix)

```
String txt = "hola";
```

S'ha realitzat l'operació `new` per a crear un objecte de tipus **String** amb el valor "hola" i després s'ha assignat a la referència `txt`.



`String` és una classe immutable, no es pot canviar el seu contingut una vegada creat, per tant, cada assignació suposa crear un nou objecte amb el nou valor.

```
txt = txt + " món";
```

En la sentència anterior es realitzen les operacions següents:

- obté el valor de l'objecte referenciat per `txt`
- concatena el text recuperat "hola" amb el literal " món"
- crea un nou objecte de tipus `String` amb el valor "hola món"
- assigna l'objecte a la referència `txt` (l'objecte anterior es perd).

```
String txt = "hola";  
System.out.println(txt + " > " + System.identityHashCode(txt)); // visualitza hola > 883049899  
txt = txt + " món";  
System.out.println(txt + " > " + System.identityHashCode(txt)); // visualitza hola món > 1854731462
```

Quan es crea un objecte se li assigna un `hashCode`, en un principi diferent per a cada objecte. En l'exemple es mostra el hashcode de `txt`, i és diferent després de la concatenació.

Java té les classes `StringBuilder` i `StringBuffer` que tracten les cadenes de manera dinàmica (`StringBuffer` ofereix mètodes sincronitzats).

```
StringBuilder cad = new StringBuilder("hola"); // crea cad amb el text "hola"  
cad.append(" món").append("cruel"); // a cad se li afeg el text " món " i "cruel", cad conté "hola món cruel"  
cad.delete(8, 11); // a cad se li esborren els caràcters del 8 (inclusivament) al 11 (exclusivament)  
System.out.println(cad); // visualitza hola mónuel el valor de cad  
String copia = cad.toString(); // el mètode toString torna el String de cad
```

Comparar

Dues cadenes es comparen pel text que les formen, és a dir, es compara la codificació dels caràcters que formen les cadenes. La cadena "ana" és major que "aNaBel" ja de la segona lletra "n" de "ana" és major la segona lletra "N" de "aNaBel".

Per a comparar cadenes cal utilitzar mètodes oferits per la classe `String`:

equals

`equals` retorna `true` si les dues cadenes són iguals i `false` si són diferents, compara la cadena referenciada amb la cadena que rep com a paràmetre.

```
String cad = "uno";  
System.out.println(cad.equals("Uno")); // visualitza false, cad ("uno") i "Uno" són diferents  
String cadd = "uno";  
System.out.println(cad.equals(cadd)); // visualitza true, cad ("uno") i cadd ("uno") són iguals
```

equalsIgnoreCase

`equalsIgnoreCase` funciona igual que `equals`, però no distingeix majúscules de minúscules



```
System.out.println(cad.equalsIgnoreCase("Uno")); // visualitza true, cad ["uno"] i "Uno" són iguals
System.out.println("Uno".equalsIgnoreCase(cad)); // visualitza true, "Uno" i cad ["uno"] són iguals
```

compareTo

El seu format és `ref.compareTo(cad)` compara les cadenes `ref` i `cad`, retorna

- 0 si les cadenes són iguals
- negatiu si la cadena `ref` és menor que `cad`
- positiu si la cadena `ref` és major que `cad`

compareToIgnoreCase

Igual que `compareTo`, però no distingeix majúscules de minúscules

```
String cad = "uno";
System.out.println("una".compareTo(cad)); // visualitza -14, "una" és menor que "uno"
System.out.println("uña".compareTo("una")); // visualitza 131, "uña" és major que "una"
System.out.println(cad.compareToIgnoreCase("Uno")); // visualitza 0, "uno" és igual a "Uno"
System.out.println("Miel".compareToIgnoreCase("MISERIA")); // visualitza -14, "Miel" és menor que "MISERIA"
System.out.println("Muleta".compareTo("Motocicleta")); // visualitza 6, "Muleta" és major que "Motocicleta"
```

==

L'operador `==` amb les cadenes compara les posicions en memòria dels dos objectes no es seu contingut.

En l'exemple següent, totes les sentències s'escriuen seguides.

→ sols existeix una copia de l'objecte associat a un literal.

```
String cad = "uno";
String cadd = "Uno";
System.out.println("cad == cadd >> " + cad + " == " + cadd + ">> " + (cad == cadd));
// visualitza cad == cadd >> uno == Uno >> false
```

`cad` i `cadd` són dues cadenes de text que fan referència a literals diferents que estan en posicions diferents, la comparació és `false`.

```
String cad3 = "uno";
System.out.println("cad == cad3 >> " + cad + " == " + cad3 + ">> " + (cad == cad3));
// visualitza cad == cad3 >> uno == uno >> true
```

`cad` i `cad3` són dues cadenes de text que fan referència al mateix literal "uno" que està en una única posició, la comparació és `true`.

```
String cad4 = new String("uno"); // es crea un objecte de tipus String amb el valor "uno"
System.out.println("cad == cad4 >> " + cad + " == " + cad4 + ">> " + (cad == cad4));
// visualitza cad == cad4 >> uno == uno >> false
```

`cad` i `cad4` són dues cadenes de text que fan referència al mateix contingut ("uno") però que estan en posicions de memòria diferents, la comparació és `false`.



```
System.out.println("cad.equals(cad4) >> " + cad + " és igual a " + cad4 + " >> " + (cad.equals(cad4)));
// visualitza cad.equals(cad4) >> uno és igual a uno >> true
```

`cad1` i `cad4` són dues cadenes de text que fan referència al mateix contingut, per tant, `equals` torna `true`.

```
cad1 = "Uno"; // assigna el literal "Uno"
System.out.println("cad == cadd >> " + cad + " == " + cadd + " >> " + (cad == cadd));
// visualitza cad == cadd >> Uno == Uno >> true
```

`cad` i `cadd` són dues cadenes de text que fan referència al mateix literal ("Uno") que està en una única posició, la comparació és `true`.

```
cad4 = "Uno"; // assigna el literal "Uno"
System.out.println("cad == cad4 >> " + cad + " == " + cad4 + " >> " + (cad == cad4));
// visualitza cad == cad4 >> Uno == Uno >> true
```

`cad` i `cad4` són dues cadenes de text que fan referència al mateix literal ("Uno") que està en una única posició, la comparació és `true`.

```
cad = new String("Uno"); // es crea un objecte de tipus String
System.out.println("cad == cad4 >> " + cad + " == " + cad4 + " >> " + (cad == cad4));
// visualitza cad == cad4 >> Uno == Uno >> false
```

`cad` i `cad4` són dues cadenes de text que fan referència al mateix contingut però que estan en posicions diferents, la comparació és `false`.

```
System.out.println("cad.equals(cad4) >> " + cad + " és igual a " + cad4 + " >> " + (cad.equals(cad4)));
// visualitza cad.equals(cad4) >> Uno és igual a Uno >> true
```

`cad` i `cad4` són dues cadenes de text que fan referència al mateix contingut, per tant, `equals` torna `true`.

Tractar caràcters individuals

`charAt`

Retorna el caràcter de la cadena en la posició indicada per `index`.

→ El primer caràcter d'una cadena té índex 0

```
String cad = "Ana";
char c0 = cad.charAt(0); // el caràcter en la posició 0
int n0 = cad.charAt(0); // el codi del caràcter en la posició 0
System.out.println("codi de " + c0 + " = " + n0); // visualitza codi de A = 65
char c2 = cad.charAt(2); // el codi del caràcter en la posició 2
int n2 = c2;
System.out.println("codi de " + c2 + " = " + n2); // visualitza codi de a = 97
```

El mètode `charAt` retorna un caràcter, però està codificat com un enter, per tant, pots utilitzar de les dues formes.

```
c0++; // suma 1 al caràcter
```



```
System.out.println("codi de " + c0 + " = " + (int) c0); // visualitza codi de B = 66 s'ha de fer un casting
c2 = (char) ++n2; // s'ha de fer un casting
System.out.println("codi de " + c2 + " = " + n2); // visualitza codi de b = 98
```

toCharArray

Retorna tots els caràcters de la cadena de text en una matriu de caràcters.

```
String s = "cadena";
char[] cars = s.toCharArray(); // veurem les matrius en un tema posterior
```

Regex


La classe `String` té mètodes que reben una expressió regular, `matches`, `replaceAll`, `replaceFirst`, `split`.

Una expressió regular és una seqüència de caràcters que forma un patró de cerca, pot ser un sol caràcter o un patró més complicat.

Pots utilitzar les expressions regulars per a realitzar tot tipus d'operacions de cerca i reemplaçament de text.

```
String cad = "pose un exemple"; // cadena on es cerca el patró
String regex = "pose"; // cerca el text "pose"
System.out.println(cad.matches(regex)); // false, "pose un exemple" no és igual a "pose"
regex = "(.*)se(.*)"; // cerca el text "se" amb qualsevol quantitat de caràcter per davant o per darrere
System.out.println(cad.matches(regex)); // true, "se" està en cad
cad = "Anais"; // cadena on es cerca el patró
regex = "^a...s$"; // cerca 5 caràcter que comencen amb "a" i terminen amb "s"
System.out.println(cad.matches(regex)); // false, comença amb "A"
regex = "[aA]...s$"; // cerca 5 caràcter que comencen amb "a" o "A" i terminen amb "s"
System.out.println(cad.matches(regex)); // true, comença amb "a" o "A" i terminen amb "s"
```

Les expressions regulars són molt potents i ofereixen moltes possibilitats, durant el curs usarem algunes molt simples.

 Mira el tema `tva03_regex`.

Altres mètodes de String

format

```
String format(String formatCad, Object... args)
```

El mètode rep una cadena `formatCad` amb el text i els caràcters de substitució (un `%` i un caràcter que indica el tipus de dada) que es substituiran pels valors de `args`. El mètode retorna una cadena amb el text amb les substitucions realitzades.

```
int a = 7, b = 15;
String cad = String.format("%d + %d = %d", a, b, a + b); // assigna "7 + 15 = 22" a cad,
cad = String.format("la suma de %d i %d val %d", a, b, a + b); // assigna "la suma de 7 i 15 val 22" a cad
// %d és el caràcter de substitució per a representar un enter
```



☞ En l'apartat **Eixida amb format** es veuen els caràcters de substitució.

indexOf

Retorna la posició de la primera ocurrència d'una cadena en altra. Si la cadena buscada no està, llavors retorna -1.

```
String text = "En un lugar de la Mancha, y en otro lugares hay buenos manjares";  
int pos = text.indexOf("ar");           // dona 9, és la primera aparició de "ar" en el text  
pos = text.indexOf("ara");             // dona -1, "ara" no està en text
```

lastIndexOf

Igual que `indexOf`, però retorna l'última ocurrència.

```
pos = text.lastIndexOf("ar");          // dona 59, és l'última aparició de "ar" en el text
```

length

Retorna la longitud d'una cadena

```
int numCaracters = text.length();     // dona 63
```

substring

Retorna una subcadena d'una cadena des de la posició inicial (inclòsa) a una final (exclòsa). La posició del primer caràcter és 0. Es pot posar, únicament la posició inicial i s'arriba fins al final de la cadena.

```
text.substring(19, 24)                 // és "ancha"  
text.substring(44)                    // és "y buenos manjares"  
text.indexOf("ar") + text.substring(text.indexOf("ar") + "ar".length(), text.indexOf("ar")) // és 37 posició del segon "ar"
```

valueOf

Converteix l'objecte rebut a cadena

```
String cad = String.valueOf(45365.66454); // s'assigna "45365.66454" a cad
```

La classe `String` té més mètodes: `contains`, `endsWith`, `concat`, `getBytes`, `isEmpty`, etc.

→ La classe `String` s'usa molt, és convenient dominar els mètodes que ofereix.

Altres operadors

<< >> >>>

~ & | ^

instanceof

? :

desplaçament de bits a la dreta o esquerra

operadors lògics a nivell de bits no, i, o, o exclusiu

comprova si un objecte és una instància d'una classe

operador ternari de condició



Ordre de precedència

Precedència	operador	Associació
15	() <i>parèntesis</i> [] <i>accés matriu</i> . <i>accés objecte</i>	esquerra a dreta
14	<i>expr</i> ++ <i>expr</i> --	dreta a esquerra
13	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ! <i>expr</i> ~ <i>expr</i> (tipus) <i>casting</i> new classe	dreta a esquerra
12	* / %	esquerra a dreta
11	+ <i>suma</i> - + <i>concatenació</i>	esquerra a dreta
10	<< >> >>>	esquerra a dreta
9	< <= > >= instanceof	esquerra a dreta
8	== !=	esquerra a dreta
7	&	esquerra a dreta
6	^	esquerra a dreta
5		esquerra a dreta
4	&&	esquerra a dreta
3		esquerra a dreta
2	? :	dreta a esquerra
1	= += -= *= /= %= &= ^= = <<= >>= >>>=	dreta a esquerra

La precedència més alta indica que l'operador es tracta abans. En un mateix valor de precedència, els operadors es tracten seguint l'ordre d'escriptura en la taula.

Com s'avalua aquesta expressió?

(3*(4+5) > 8) && !(10 < 12) || !('A' == 'a')

∇

(3 * 9 > 8) && !(10 < 12) || ('A' == 'a')

∇

(27 > 8) && !(10 < 12) || ('A' == 'a')

∇

(true) && !(true) || (false)

∇

(15) parèntesis més interns, (11) suma

primer parèntesi, (12) producte

(9) major, (8) igualtat

(13) negació



```

true && false    || false
  ∇
false || false
  ∇
false

```

(4) &&
(3) ||

L'associació indica en quina direcció s'uneixen dos operadors del mateix ordre. *expr* indica la posició de l'expressió sobre la qual s'està aplicant l'operador. Com s'avalua aquesta expressió?

```
double divisio = 72.0 / 2.0 / 3.0;    // dona 12
```

Totes les operacions són divisions, les divisions s'associen d'esquerra a dreta, per tant, es divideix 72 per 2, i el resultat es divideix per 3, i el resultat s'assigna a *divisio*.

→ Usa parèntesis per a aclarir les expressions.

```

double divisio = (72.0 / 2.0) / 3.0;    // dona 12
double divisio = 72.0 / (2.0 / 3.0);    // dona 108

```

Conversions de tipus

Un exercici demana "incrementa el preu un 20%", és a dir, multiplica el preu per 1.2

```

int preu = 37;
preu = preu * 1.2;    // el producte per 1.2 incrementa preu un 20%

```

Aquest codi sembla correcte, però el compilador ens informa d'un error: "incompatible types: possible lossy conversion from double to int". Per què ocorre això?

- L'ordinador vol realitzar el producte `preu * 1.2`
- `preu` és un enter i `1.2` és un double, com sols pot fer operacions amb elements del mateix tipus, converteix `preu` a double, és a dir, `37` passa a `37.0`
- Ja pot fer la multiplicació `37.0 * 1.2 = 44.4` el resultat és double
- El problema ve quan vol realitzar l'assignació `preu = 44.4`, no hi ha conversió automàtica d'un double `44.4` a un enter `44` i dona l'error.

Perquè es produïska la conversió és necessari realitzar un càsting (canviar el format d'una dada) hi ha dos tipus de càsting: **implícit** i **explícit**

Càsting implícit

El càsting implícit (*Widening Casting*) es realitza quan el tipus de destinació és més gran que el tipus d'origen (té més bits en memòria), per tant, no es perd informació.

Els dos tipus han de ser compatibles. No es pot convertir una cadena de text en un nombre usant el càsting (dona error).



Per als nombres, la seqüència de càsting implícit és: `byte -> short -> char -> int -> long -> float -> double`, són compatibles, ja que tots venen de la classe `Number`.

```
int a = 65;
double b = 4.35;
long c = 124578369L;
long d = 1234567890123456789L;
b = a + b; // int(65) passa a double(65.0), se suma al double(4.35) i es guarda el double(69.35)
b = c;     // long(124578369) passa a double(124578369.0)
b = d;     // long(1234567890123456789) passa a double(1.23456789012345677E18) es perden dígit significatiu
a = 'E';   // el literal 'E' és un char, s'obté el seu codi i es converteix a int(69)
```

Càsting explícit

El càsting explícit (*Narrowing Casting*) és necessari quan el tipus de destinació és més xicotet que el d'origen (té menys bits en memòria).

Has de realitzar un càsting al tipus que vols obtenir, el tipus de destinació s'escriu entre parèntesis davant de l'element a transformar. Els dos tipus han de ser compatibles.

Per als nombres, la seqüència de càsting explícit és: `double -> float -> long -> int -> char -> short -> byte`

```
double d = 65 * 0.33;           // d val 21.45, el 65 passa a double (implícit) i es realitza el producte
int e = 65 * (int) 0.33;        // e val 0, el casting afecta al 0.33 i el transforma a int 0
short s = (short) 65 * (short) 3.33; // s val 195, el casting afecta al 65 i al 3.33 (el transforma a 3)
s = (short) (65 * 3.33);        // s val 216, el 65 passa a double (implícit) i el càsting s'aplica al resultat
byte b = (byte) (65 * 0.33);    // b val -40,
```

Com es passa a un espai de menor grandària, llavors, es perden els bits superiors. Cal anar amb compte, en l'últim càsting el valor 216 no cap en un byte, però es copien els bits inferiors del `short` en el `byte` i aquests representen el valor -40 en base 10, però `s` i `b` tenen els mateixos bits inferiors.

Les operacions de càsting produeixen una excepció (error) si no es pot convertir un tipus en l'altre.

 Veuràs les excepcions en un tema posterior.

L'entorn avisa de tipus diferents i demana realitzar el càsting explícit necessari.

```
int preu = 37;
preu = preu * 1.2;           // es necessita fer un càsting a enter (int)
```

```
preu = (int) (preu * 1.2);    // es perden els decimals del producte
```

Càsting amb mètodes

Hi ha classes que tenen mètodes que converteixen un valor d'un tipus al tipus de la classe, per exemple els mètodes `Integer.parseInt` i `Double.parseDouble`



```
String cadena = "65";           // defineix una cadena amb el text "65"
int a = Integer.parseInt(cadena); // parseInt converteix cadena en int (65)
double b = Double.parseDouble(cadena); // parseDouble converteix cadena en double (65.0)
```

El nom dels mètodes que converteixen un text en altre tipus, contenen el verb *parse*, que significa analitzar gramaticalment un text per a comprovar si compleix les normes de transformació, si no s'acompleixen, llavors es llança un excepció.

Escriure en la consola

La consola permet visualitzar text, únicament.

La consola està associada al flux d'eixida estàndard *out* que es maneja amb la classe *System*. El flux *out* té els mètodes, *print*, *println*, *printf*, *flush*, *write*, *append*, *format*, etc. usarem els tres primers per a escriure en la consola.

```
int x = 3;
double y = -23.085;
System.out.print(x + "\n");           // concatena l'enter x amb el canvi de línia "\n"
System.out.println(y);                // transforma el real y a cadena, afeg un canvi de línia
System.out.println("x= " + x + " y= " + y); // concatena els literals i els valors, afeg un canvi de línia
System.out.printf("x= %d y= %f", x, y); // substitueix el %d pel valor de x i %f pel valor d'y
System.out.println(x + y);            // suma de x i y, i la transforma a cadena, afeg un canvi de línia
// visualitza
3
-23.085
x= 3 y= -23.085
x= 3 y= -23,085000-20.085
```

L'expressió entre parèntesis dels mètodes *print* i *println* s'avalua i es transforma en una cadena de text que és mostra en la consola en la posició del cursor. El mètode *println* afeg una nova línia al final text passat com a paràmetre.

Pots canviar el flux estàndard d'eixida a un fitxer.

Eixida amb format

El mètode *printf* o *format* reben una cadena de format i la informació a visualitzar, la cadena de format conté caràcters de substitució per a la informació a visualitzar.

```
System.out.printf("x= %d", x);         // "x= %d" és la cadena de format, x és la informació
System.out.format("y= %f", y);         // "y= %f" és la cadena de format, y és la informació
System.out.printf("\nx= %.2f y= %.2f suma = %.2f\n", (double) x, y, x + y); // %f espera double, fa el càsting de x
// visualitza
x= 3y= -23,085000
x= 3,00 y= -23,09 suma = -20,09
```

→ La classe *String* té el mètode *format* que funciona de la mateixa forma.

```
String txt = String.format("la divisió %d/%.2f val %.2f\n", x, y, x / y);
```



```
System.out.println(txt); // visualitza la divisió 3/-23,09 val -0,13
```

Cadena de format

La cadena de format conté text i caràcters de substitució. La sintaxi d'un caràcter de substitució és:

```
%[amplària][.precisió]caràcter // els [] indiquen que l'element és opcional
```

- el caràcter %
- *amplària* és el nombre de caràcters mínims que ocuparà el text que substitueix el caràcter. Si el valor supera l'espai mínim, llavors s'escriu el valor complet (no es trunca).
- *precisió* és la grandària màxima. Per a nombres reals indica quantes xifres es visualitzaran en la part decimal. No és aplicable a enters.
- *caràcter* defineix el tipus de dada:
 - "d", "o", "x", "X" per a un nombre enter
 - "f", "e", "E" per a un nombre real
 - "c", "C" per a un caràcter
 - "s", "S" per a una cadena de text
 - "b", "B" per a un valor booleà

quan el caràcter està en majúscula, llavors el text que el substitueix s'escriu en majúscules.

```
int a = 109;
System.out.printf("num = %d\n", a); // visualitza num = 109
System.out.printf("num = %+6d\n", a); // visualitza num = +109 el nombre ocupa 6 caràcters
System.out.printf("num = %X\n", a); // visualitza num = 6D
System.out.printf("car = %c\n", a); // visualitza car = m
double b = 32.147;
System.out.printf("num = %f\n", b); // visualitza num = 32,147000
System.out.printf("num = %E\n", b); // visualitza num = 3,214700E+01
System.out.printf("num = %5.2f\n", b); // visualitza num = 32,15
String t = "Hola";
System.out.printf("text = %s\n", t); // visualitza text = Hola
System.out.printf("text = %S\n", t); // visualitza text = HOLA
System.out.printf("text = %10s\n", t); // visualitza text =          Hola ocupa 10 caràcters
boolean x = true;
System.out.printf("bool = %b\n", x); // visualitza bool = true
```

Es posa `\n` en la cadena de format per a baixar de línia en la consola.

Els caràcters de substitució s'associen als arguments per ordre d'escriptura

```
System.out.printf("%.2f+%.2f=%.2f\n", (double) a, b, a + b); // visualitza 109,00+32,15=141,15
```

En l'exemple hi ha tres arguments `a`, `b` i `a + b`. Els tres es tracten com `double`, per tant s'utilitza el caràcter de substitució `%f`, un per cada argument a escriure. El `%.2f` indica que sols es van a veure dos decimals, el valor 32.147 s'arrodoneix a 32.15. El primer `%.2f` de la cadena se substitueix pel que val `a`, com no és un enter cal canviar el seu



format, es fa un càsting a `double`, el segon `%.2f` se substitueix pel que val `b` i el tercer pel que val `a + b`, el resultat és un `double`

Locale

El mètode `printf` permet canviar l'eixida aplicant un `Locale`, aquest objecte defineix aspectes locals a un país o idioma, per defecte s'usa el `Locale` del sistema (en el nostre cas és l'espanyol).

Hi ha `Locale` predefinits, també, pots crear `Locale` indicant l'idioma, el país i la variant (l'únic obligatori és l'idioma), en la pàgina <https://www.localeplanet.com/java/> tens els codis per als arguments de `Locale`.

```
Locale.FRANCE    Locale.FRENCH    // Locale predefinits per a la França i per al francès
Locale.UK        Locale.ENGLISH  // Locales predefinits per a la Gran Bretanya o l'anglès
Locale.setDefault(new Locale("ca")); // canvia el Locale a català
Locale.setDefault(new Locale("ca", "ES_VALENCIA")); // canvia el Locale a català - valencià
Locale.setDefault(new Locale("es")); // canvia el Locale a espanyol
Locale.setDefault(new Locale("es", "BZ")); // canvia el Locale a espanyol - Belize
```

Si vols el punt com a separador dels decimals en un nombre has d'usar el `Locale.UK` o el `Locale.US` o el `new Locale("es", "BZ")`.

```
System.out.printf("num = %f\n", b); // num = 32,147000 la separació dels decimals és la coma
System.out.printf(Locale.UK, "num = %f\n", b); // num = 32.147000 la separació dels decimals és el punt
```

Mira: <http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html>

En NetBeans la consola utilitza la codificació **UTF-8** i no representa correctament els caràcters que no són anglesos.

```
System.out.println("pàlā viña dél sèñor"); // visualitza p0l0 vi0a d0l s00or
```

Per a poder veure els caràcters correctament, cal canviar, en les propietats del projecte, la codificació a **Windows-1252** o **ISO-8859-1**

Encoding: → Encoding:

Llegir de teclat

La classe `Scanner` permet entrar informació des de teclat, és el flux d'entrada estàndard `System.in`.

```
Scanner teclat = new Scanner(System.in);
System.out.println("entra un nombre");
int num1 = teclat.nextInt();
```

L'entrada d'informació des de teclat es pot realitzar d'altres formes.



```
BufferedReader lector = new BufferedReader(new InputStreamReader(System.in));
System.out.println("entra un nombre");
String entrada = lector.readLine();
int num1 = Integer.parseInt(entrada);
```

Els dos exemples anteriors, permeten llegir un enter, nosaltres usarem `Scanner`.

En els exemples posteriors, la instància de `Scanner` s'associa a la referència `teclat`.

```
Scanner teclat = new Scanner(System.in);
```

La referència `teclat` duu a un objecte de tipus `Scanner` associat al flux d'entrada estàndard `System.in` (el teclat), l'objecte recull les pulsacions de teclat fins a prémer la tecla *retorn de carro* que finalitza l'entrada.

Els caràcters teclejats se separa mitjançant els `javaWhitespace`: l'*espai en blanc*, el *tab* i el *retorn de carro*. Pots canviar els `javaWhitespace` amb el mètode `useDelimiter(pattern)` on `pattern` és una expressió regular (*regex*) on es defineixen els `javaWhitespace` de les entrades.

Cada "tros" de l'entrada s'assigna a un element diferent.

La classe `Scanner` permet llegir fitxers, per a això cal canviar el flux d'entrada.

Llegir un text

```
String entrada = teclat.next();
```

El mètode `next` recull les pulsacions del teclat (caràcters) fins a prémer el *retorn de carro* i separa els caràcters usant els delimitadors.

```
String cad1, cad2, cad3;
cad1= teclat.next(); // recupera el text fins al primer delimitador
cad2= teclat.next(); // recupera el text fins al segon delimitador
cad3= teclat.next(); // recupera el text fins al tercer delimitador
```

Amb els delimitadors per defecte, si introdueixes "hola que tal¿", tens dos *espais en blanc* i el *retorn de carro*, per tant, "hola" s'assigna a `cad1`, "que" a `cad2` i "tal" a `cad3`.

Si introdueixes "hola món¿", tens un *espai en blanc* i el *retorn de carro*, "hola" s'assigna a `cad1`, "món" a `cad2` i l'execució es deté a l'espera de la tercera entrada.

Si introdueixes "hola que tal Pascual¿", tens tres *espais en blanc* i el *retorn de carro*, "hola" s'assigna a `cad1`, "que" a `cad2`, "tal" a `cad3` i "Pascual" es queda en el **buffer d'entrada** (zona de memòria intermèdia) i serà recuperat pel `next` següent.

Per a assegurar la lectura d'un `next` (de qualsevol tipus), s'obté el contingut del buffer fins al *retorn de carro* amb un `nextLine` i es descarta la informació recuperada.

```
String uno, dos, tres;
teclat.nextLine(); // recupera els caràcters del buffer d'entrada fins al primer retorn de carro
```



```
uno= teclat.next();      // el buffer d'entrada està buit, per tant, l'usuari ha de teclejar la nova entrada
teclat.nextLine();
dos= teclat.next();
teclat.nextLine();
tres= teclat.next();
```

Llegir una línia de text

```
String nom = teclat.nextLine();
```

El mètode `nextLine` recull les pulsacions del teclat fins a prémer el *retorn de carro*. L'únic delimitador que es té en compte és el *retorn de carro*, per tant, el mètode retorna una única entrada, el text sense el *retorn de carro*.

Conjunt de caràcters

Si Netbeans utilitza la codificació "UTF-8", si lliges una cadena de text que conté títles i la ñ, no obtens el que has teclejat.

```
System.out.print("Entrada: ");
String cadena = teclat.nextLine();
System.out.printf("Salida: %s%n", cadena);
// visualitza
Entrada: las 3 ñññ más 3 NNN
Salida: las 3  m s 3
```

Has de definir la codificació "windows-1252" o el "ISO-8859-1" per al `Scanner`.

```
Scanner teclat = new Scanner(System.in, "windows-1252");
```

Si Netbeans utilitza la codificació "windows-1252" o el "ISO-8859-1", has de definir la codificació "UTF-8" per al `Scanner`.

```
Scanner teclat = new Scanner(System.in, "UTF-8");
```

Llegir un enter

```
int hores = teclat.nextInt();
```

El mètode `nextInt` recull els caràcters teclejats fins al retorn de carro, i transforma el text a un enter. Si no introdueixes un enter, falla la transformació i es llança l'excepció `InputMismatchException`.

Llegir un real

```
double preuHora = teclat.nextDouble();
```

El mètode `nextDouble` recull els caràcters teclejats fins al *retorn de carro*, i transforma el text a un `double`. Si no introdueixes un `double`, es llança l'excepció `InputMismatchException`.



`Scanner` utilitza la configuració local del sistema, és a dir, l'espanyola que utilitza la coma com a separador dels decimals, l'entrada 12.7 llança l'excepció `InputMismatchException`. Si vols que `Scanner` utilitze el punt com a separador dels decimals, canvia el `Locale` del `Scanner` al de Gran Bretanya (UK).

```
teclat.useLocale(Locale.UK);
```

Llegir un booleà

```
boolean esHoraExtra = teclat.nextBoolean();
```

El mètode `nextBoolean` recull els caràcters teclejats fins al retorn de carro, transforma el text a un booleà.

Els valors per a l'entrada d'un booleà són el text `true` o `false`.

Per a una major coherència en el GUI (*Graphical User Interface*) és millor llegir un text i després transformar-ho al nostre gust, per exemple, si el GUI està en valencià, llegir el text "sí" o "no", i transformar-los a `true` o `false`.

En l'exemple següent s'utilitzen la majoria dels formats per a llegir informació

```
Scanner lector = new Scanner(System.in);
System.out.println("Com et dius?");
String nom = lector.nextLine();
System.out.println("Quantes hores vas treballar?");
int hores = lector.nextInt();
System.out.println("Són hores extra?");
boolean esHoraExtra = lector.nextBoolean();
System.out.println("Quant et paguen per hora?");
lector.useLocale(Locale.UK);
double preuHora = lector.nextDouble();
```

 Usaràs la consola, únicament, en aquest tema i el següent.

Exercicis

7. eurospesetes

Escriu un programa que llig de teclat una quantitat expressada en euros i la visualitza en pesetes (1€=166.386Pts). Fes dues visualitzacions, usant `println` i `printf`, amb `printf` limita el nombre de decimals a 2. Usa `Locale` per a usar el punt com a separador dels decimals.

8. fahrenheitcelsius

Escriu un programa que llig de teclat una temperatura en graus Fahrenheit i presenta en pantalla la temperatura en graus Celsius. La fórmula per a convertir graus Fahrenheit (`gFahr`) a Celsius (`gCel`) és: $gCel = (gFahr - 32) * (5 / 9)$. En la visualització usa `printf`, i limita el nombre de decimals a 2, mostra la temperatura en les dues escales.

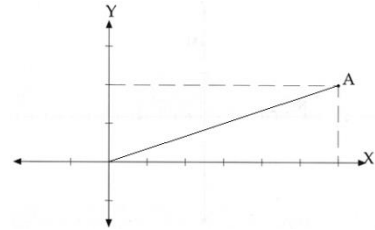
9. cercle



Escriu un programa que calcula l'àrea i el perímetre d'un cercle. Llig de teclat el radi del cercle (té decimals). L'àrea i el perímetre es calculen amb les fórmules: **àrea** = $\pi * \text{radi}^2$, **perímetre** = $2 * \pi * \text{radi}$, el valor de π s'obté amb [Math.PI](#).

10. distancia2punts

Escriu un programa que llig de teclat la posició x i y d'un punt A(xA, yA) en el pla i calcula la seua distància a l'origen (0, 0).



Variació: es lligen la posició de dos punts A(xA, yA) i B(xB, yB), i es visualitza la distància entre ells. La fórmula de la distància és

$$D = \sqrt{(xA - xB)^2 + (yA - yB)^2}$$

El mètode per a calcular l'arrel quadrada és [Math.sqrt\(n\)](#).

11. soubar

Escriu un programa que calcula i visualitza el que ha de pagar-nos el nostre cap pel nostre treball en un bar (són euros, tenen 2 decimals).

Ens paga 7€ per hora de dilluns a divendres, i 11.5€ per hora el dissabte i diumenge. Hi ha 5 treballadors i les propines es reparteixen entre tots. Finalment, cal descomptar el preu dels menjars que hem realitzat, cada menjar val 4.5€.

12. eliminar

Escriu un programa que llig una cadena de text i un caràcter, elimina el caràcter del text i visualitza el text modificat i quants caràcters s'han eliminat. Usa el mètode [replaceAll](#) per a reemplaçar caràcters d'un text.

