Conceptes bàsics de programació estructurada

Programes i algorismes

Un ordinador és una màquina electrònica dotada d'una memòria de gran capacitat i de mètodes de tractament de la informació, capaç de resoldre problemes matemàtics i lògics mitjançant la utilització automàtica de programes informàtics.

Els components principals d'un ordinador són el processador central, la memòria i els perifèrics. Entre els perifèrics destaquen el teclat, la pantalla i els discs d'emmagatzemament (dics durs).

El **processador central** és el cervell de l'ordinador i realitza operacions aritmètico-lògiques sobre la informació que recupera de la memòria.

Els components físics de l'ordinador són el **maquinari**. Les dades i els programes constitueixen el **programari**.

El sistema operatiu de l'ordinador és la capa de programari que s'encarrega de la gestió i control del maquinari i proporciona eines per gestionar la informació i els programes.

La programació d'ordinadors s'utilitza per resoldre problemes. Generalment, els problemes a resoldre consisteixen en processar informació i obtenir un resultat o prendre accions o decisions.

El procediment per resoldre un problema concret s'anomena **algorisme**

Un algorisme consisteix en un conjunt ordenat d'operacions que permeten resoldre un problema en un nombre finit de passos i en un temps finit. Pot acceptar un conjunt de **dades d'entrada** i un conjunt de **dades de sortida**.

La codificació d'un algorisme en un ordinador s'anomena **programa**.

Activitat Prova de programar el moviment per arribar a la destinació Blockly:laberint

El processador de la CPU només entén el **codi màquina**. Per facilitar la tasca de programació, existeix el codi **assemblador**, el qual es caracteritza perquè cada *mnemònic* equival a una instrucció de màquina. Els **programes assembladors** tradueixen aquests mnemònics a instruccions de màquina en codi binari i generen l'estructura de programa a partir de les directives d'estructura definides pel programador. Els principals avantatges de la programació en assemblador són la rapidesa d'execució, l'estalvi de memòria, la potència a l'hora de controlar perifèrics, però té com a inconvenients que el codi no és transportable a sistemes amb altres processadors i és difícil d'escriure i depurar. Es tracta d'un **llenguatge de programació de baix nivell**. Per aquests motius, no és un llenguatge adient per dissenyar grans aplicacions.

Els programadors dissenyen les aplicacions en **llenguatges d'alt nivell**, els quals estan dotats d'estructures complexes per definir el codi i també les dades. Aquests llenguatges són independents de l'arquitectura de l'ordinador i faciliten les tasques de desenvolupament, depuració i

manteniment. Com a contrapartida, necessiten un programa que tradueixi les seves instruccions d'alt nivell a instruccions de màquina. Aquests traductors són els **compiladors** i els **intèrprets**.

Els **intèrprets** processen i tradueixen cada instrucció d'alt nivell just abans d'executar-la. El principal inconvenient és la poca velocitat d'execució, atès que cal traduir les instruccions d'alt nivell cada vegada que el programa s'executa.

Els **compiladors** tradueix per complet tot el programa a codi màquina. La compilació i l'execució són processos independents, de manera que l'execució es fa sobre el codi màquina sense cap traducció prèvia, aconseguint una millor rapidesa d'execució.

Les fases del procés de traducció amb un compilador són: **compilació** del **codi font** a **codi objecte** amb la corresponent detecció d'errors i **muntatge** (*link*) del codi objecte (que pot residir en diversos mòduls) amb les **biblioteques del llenguatge** aportades pel programa compilador.

Els algorismes han de ser mètodes generals per a resoldre tots els casos possibles del mateix problema i, per tant, han de ser independent de les dades d'entrada de qualsevol cas concret. Per a comprendre completament aquest concepte d'*algorisme* ens caldrà definir ara els conceptes d'*entorn, acció, procés* i *processador*.

L'**entorn** és el conjunt d'objectes necessaris per a portar a terme una tasca determinada. L'**estat de l'entorn** en un moment determinat és la descripció de l'estat dels objectes de l'entorn en aquell moment concret. Un algorisme actua de manera que fa canviar progressivament l'estat del seu entorn.

Una **acció** és un esdeveniment finit en el temps i que té un efecte definit i previst. Una acció pot actuar sobre un entorn i el pot modificar, és a dir, es parteix d'un estat inicial i s'arriba a un estat final diferent. Una **acció elemental** és una acció que el destinatari d'un algorisme entén i sap processar.

Un **procés** és l'execució d'una o diverses accions. L'algorisme expressa unes pautes que cal seguir per a portar a terme una tasca concreta. L'encarregat de dur a terme aquesta tasca és el processador. Un **processador** és una entitat capaç de comprendre i executar eficaçment un algorisme. El destinatari de l'algorisme és, doncs, el processador.

Un algorisme es defineix com una descripció no ambigua i precisa de les accions que cal fer per a resoldre un problema ben definit en un temps finit.

El Diccionari de la Llengua de la Gran Enciclopèdia Catalana defineix: *"Un algorisme és un procediment de càlcul que consisteix a acomplir un seguit ordenat i finit d'instruccions que condueix, un cop especificades les dades, a la solució que el problema genèric en qüestió té per a les dades considerades"*.

Exemple d'entorn i estat: En el cas d'una recepta de cuina, l'entorn vindria donat pels estris de cuina (olles, paelles, etc.) i els ingredients. L'estat de les paelles, per exemple, canviarà de netes a brutes.

Un processador pot ser una persona, una rentadora, un ordinador, etc. Ara que ja sabem què és un algorisme ens cal decidir com el podem expressar. Haurem de trobar un llenguatge que ens permeti fer una descripció no ambigua i precisa de les accions que componen els nostres algorismes.

Anomenem **llenguatge natural** el llenguatge que normalment utilitzem per a comunicar-nos. Ara bé, atesa la seva complexitat i ambigüitat, veurem que el llenguatge natural no ens permet definir les accions amb la precisió i claredat que volem. De fet, si nosaltres actuem com a processadors d'algú que ens indica com s'ha de fer una tasca, sovint demanem puntualitzacions i aclariments de què cal fer. Necessitem, doncs, un llenguatge més reduït i precís. Els algorismes es poden representar mitjançant diagrames o usant un llenguatge algorísmic (pseudocodi) que permeti el disseny de l'algorisme amb la major independència del **llenguatge de programació** que s'utilitzi per codificar-lo amb posterioritat.

Exemples de llenguatges de programació són Pascal, C, C++, Cobol, Fortran, Java, Python, PHP, etc.

Bàsicament, hi ha dos tipus de llenguatges de programació: els **llenguatges imperatius** o **llenguatges procedimentals** i els **llenguatges declaratius**, que a la vegada es divideixen en **llenguatges funcionals** i **llenguatges lògics**. En aquesta assignatura estudiarem només la programació imperativa, que és la més estesa i utilitzada. Alguns llenguatges de programació faciliten el seguiment d'una metodologia concreta de programació. Una de les més àmpliament acceptada actualment és l'orientació a l'objecte. **L'orientació a l'objecte** determina un estil de programació que es caracteritza per la manera de manipular la informació, centrant-se sobretot en les dades i la seva estructura més que en el codi i incorporant el codi que ha de manipular les dades dintre de la definició de les mateixes dades. En la programació imperativa, els programes són seqüències d'instruccions que s'han de dur a terme com una recepta o guió per a resoldre un problema determinat.

Exemple d'algorisme: Pensem el una rentadora de roba com un autòmat. Si demanem a la nostra rentadora que ens renti la roba blanca, la rentadora seguirà aquest procés:

- Agafar aigua i sabó del calaix.
- Escalfar l'aigua a 40 graus.
- Donar voltes durant 20 minuts (rentar).
- Expulsar l'aigua.
- Agafar més aigua.
- Donar voltes durant 10 minuts (primera esbandida).
- Expulsar l'aigua.
- …
- Donar voltes durant 10 minuts (quarta esbandida).
- Expulsar l'aigua.
- Donar voltes molt ràpidament durant 1 minut (centrifugar)

Tot aquest conjunt d'accions que la rentadora ha dut a terme seria l'algorisme que aquest aparell segueix per rentar la roba blanca. Si ara demanem que ens renti la roba delicada, les accions que

executarà la rentadora seran segurament diferents (per exemple, la temperatura serà inferior, el temps de rentat serà més curt, hi haurà menys esbandides, etc.). Això equival a dir que l'algorisme per a rentar la roba delicada és diferent de l'algorisme per a rentar la roba blanca.

També que cal que es compleixin unes condicions inicials perquè el problema es resolgui correctament (que la roba blanca estigui dins la rentadora, que la rentadora estigui connectada al corrent elèctric i a l'entrada d'aigua, i també que hi hagi sabó al calaixet). Per tal de poder arribar a l'estat final desitjat, que en el nostre cas seria tenir la roba neta al final, l'entorn hauria d'estar en aquest estat inicial. Si no partim de l'estat inicial correcte, segurament no aconseguirem el nostre objectiu (penseu, per exemple, que si la rentadora no està connectada al corrent, el nostre algorisme no donarà el resultat esperat).

Cicle de vida d'un programa

El procés que se segueix des del plantejament d'un problema fins a tenir una solució instal·lada en la computadora, i en funcionament pels usuaris finals, es denomina **cicle de vida d'una aplicació informàtica**.

Les fases principals que es poden considerar són:

- Anàlisi de requeriments
- Disseny de l'algorisme
- Implentació del programa: codificació, compilació i muntatge
- Prova i depuració
- Explotació i manteniment

L'anàlisi ha de permetre determinar quina informació ha de processar el programa, què ha de fer amb la informació, quin mecanismes de comunicació té amb els usuaris, etc. Algunes de les tècniques que es poden utilitzar en l'anàlisi són els diagrames de flux de dades, els models de dades (entitat-relació, formes normals, tipus abstractes de dades, …), els diccionaris de dades i la definició de la comunicació amb l'usuari.

Durant la fase de disseny s'han de modelar les dades i definir la solució al problema (algorisme) utilitzant diagrames gràfics o pseudocodi.

Els diagrames permeten una visualització gràfica de l'estructura del codi (**diagrama de flux**) o de les dades (**diagrama UML**).

Les etapes bàsiques del disseny són la definició dels prerequisits i postrequisits sobre les dades d'entrada i sortida del programa, el disseny de dades, el disseny modular (partició del algorisme en mòduls i definició de la relació i la comunicació entre ells) i l'especificació de cada mòdul.

La fase de codificació consisteix en transcriure l'algorisme utilitzant llenguatges de programació, com per exemple java, C#, php, python, … El resultat és el **codi font** del programa.

Els llenguatges més propers a les característiques i arquitectura de l'ordinador es diuen **llenguatges de baix nivell**, com ara el *llenguatge màquina* i el *llenguatge assemblador*.

Els llenguatges més propers al programador, amb característiques més complexes s'anomenen **llenguatges d'alt nivell**, com per exemple, Fortran, C, Cobol, Pascal, Java, etc.

Per tal que l'ordinador pugui executar el programa, cal traduir el codi font al *codi màquina*, específic del sistema on s'ha d'executar. Això es realitza a les fases de compilació i muntatge.

Segons el tipus de traducció que es realitza, els llenguatges es classifiquen en interpretats i compilats.

Als llenguatges **interpretats**, cada instrucció es tradueix just abans de la seva execució.

Als llenguatges **compilats**, en primer lloc un programa anomenat compilador fa la traducció completa del codi font. El resultat és el **codi objecte**, el qual encara s'ha d'enllaçar amb les biblioteques del llenguatge (**link**), obtenint al final el **codi executable**. Els llenguatges compilats proporcionen, en general, més velocitat d'execució.

Un tipus especial de llenguatges compilats són els que tenen un codi objecte intermig entre el codi font i el codi màquina. Aquests llenguatges necessiten una capa virtual que interpreti el codi intermig per executar-lo. Tot i que la velocitat no serà la mateixa que en el cas de codi objecte natiu, tenen com a avantatge que són multiplataforma, és a dir, que es poden executar un cop compilats en qualsevol sistema, ja que és aquesta capa virtual l'unica que cal instal·lar de forma específica a cada sistema. És el cas del llenguatge **java**.

A la fase de prova i depuració es verifica que el programa funciona d'acord amb les especificacions requerides i es corregeixen els errors que s'hi trobin.

A les fases d'explotació i manteniment, el programa és utilitzat pel usuaris finals i els programadors realitzen canvis, correccions i modificacions d'acord amb els requeriments dels usuaris.

Errors d'un programa

Els errors són funcionaments anòmals o manca de funcionament absoluta del programa en determinades circumstàncies. Cal fer un procés de prova intensiu amb dades d'entrada prou diverses per detectar possibles errors i poder-los corregir.

Els errors es clasifiquen en:

- de compilació
- d'execució
- de lògica
- d'especificació

Els errors de compilació corresponen a incompliment de les regles de sintaxi del llenguatge i són generats pel compilador o intèrpret.

Els errors d'execució es produeixen quan alguna operació no es pot realitzar sobre les dades subministrades. Són més difícils de detectar, ja que es produeixen en temps d'execució i només en circumstàncies molt concretes, depenent de les dades d'entrada.

Els errors de lògica produeixen resultats no correctes. Per detectar-los cal usar un joc de dades d'entrada prou extens en realitzar les proves.

Els errors d'especificació es produeixen per deficiències de comunicació entre client i desenvolupador. El resultat és que el programa no respon a les especificacions demanades pel client i acostuma a obligar a refer gran part del programa.

En l'etapa de proves es tracta de provar el programa resultant amb diferents dades d'entrada que reben el nom de **jocs de proves**. L'èxit d'aquestes proves dependrà en gran mesura de la qualitat del disseny fet abans.

Els jocs de proves només serveixen per a comprovar que el programa no és correcte (si algun joc de proves no dóna els resultat esperat), però no per a assegurar que sí que ho és (llevat que es pràcticament impossible comprovar totes les entrades possibles).

Els algorismes i els programes resultants han de complir les següents característiques:

a) Correctesa: l'algorisme fa allò que realment es demana.
b) Intel·ligibilitat: l'algorisme ha de ser clar i fàcil d'entendre, atès que s'escriurà un sol cop, però caldrà llegir-lo molts més per a poder-lo mantenir i/o modificar. c) Eficiència: l'algorisme ha de portar a terme la tasca que se li ha encomanat en un temps raonable.
d) Generalitat: amb pocs canvis, l'algorisme s'ha de poder adaptar a altres enunciats semblants.
Quan dissenyeu els vostres algorismes tingueu en compte sempre aquests quatre criteris. La metodologia que proposarem en aquesta assignatura us ajudarà a dissenyar algorismes que compleixin aquests criteris. Però penseu també que no només és necessari seguir la metodologia, sinó que per a assolir una certa destresa en el disseny cal força pràctica, ja que tant els conceptes com la metodologia necessiten un temps d'assimilació, i això només s'aconsegueix amb la pràctica contínua.

Elements d'un programa

En el codi d'un programa informàtic podem trobar diversos elements. El programa es divideix en **instruccions o sentències**, cada una de les quals realitza una acció concreta i definida. Segons el tipus de sentència, poden contenir diferents elements:

Variables i constants

Serveixen per emmagatzemar **dades** dintre del programa.

Han de tenir un tipus (defineix el tipus de dada contindrà), un valor (la dada concreta emmagatzemada) i un identificador (el nom amb què ens referirem a la variable o constant dintre del programa).

En el cas de les constants, no es permet canviar el valor durant l'execució del programa.

Operadors

Són connectors de dades que simbolitzen accions sobre les dades.

En funció del tipus de dades que connecten i de l'operació que representen poden ser numèrics, alfanumèrics, lògics, etc.

Expressions

Són combinacions de dades i operadors que proporcionen un resultat.

A les expressions, els operadors s'avaluen aplicant els criteris de prioritat establerts pel llenguatge. Si tenen igual grup de prioritat, s'avaluen d'esquerra a dreta.

Representació de programes

Una manera convenient de reprentar gràficament un algorisme o procés és el diagrama de flux o ordinograma.

Un altre sistema per representar un algorisme és el pseudocodi. Consisteix en un llenguatge intermedi entre el llenguatge natural i el llenguatge de programació i permet definir l'estructura del codi i el processat de les dades sense entrar en els detalls del llenguatge de programació a utilitzar.

Característiques del llenguatge Java

El llenguatge Java va ser creat l'any 1995 per l'empresa Sun Microsystems, la qual va ser comprada l'any 2009 per Oracle.

És un llenguatge compilat, d'alt nivell i amb sintaxi estricta. El compilador genera un codi intermig (bytecodes) que s'executa posteriorment sobre una entorn d'execució (màquina virtual). Això permet que sigui multiplataforma, ja que el codi compilat és el mateix per a totes les plataformes. Només cal tenir instal·lat l'entorn virtual (JRE: Java Runtime Environment) corresponent a la plataforma concreta.

Java és un llenguatge orientat a objectes. Això implica que les funcions (anomenades mètodes) del llenguatge es troben encapsulades dintre d'objectes contenidors. El paradigma de l'orientació a objectes facilita molt el disseny i la construcció d'aplicacions grans.

Es pot escriure un programa en Java amb qualsevol editor de text, tot i que un IDE (Integrated Development Environment) és convenient per disposar d'eines de compleció i evitar errors d'escriptura. Els fitxers de codi font porten l'extensió .java. el compilador genera fitxers en format bytecode amb extensió .class.

Tot programa en Java es compon d'un conjunt de classes, cadascuna de les quals acostuma a estar en un fitxer amb el mateix nom que la classe. La classe que es carrega primer quan s'executa el programa ha de contenir un mètode main().

Heus aquí un exemple d'aplicació petita en Java:

```
/**
 * Hola món
 * @author ProvenSoft
 */
public class HolaMon {
    public static void main(String[] args) {
        //imprimir missatge
        System.out.println("Hola món!");
    }
}
```

El fitxer s'ha de dir *HolaMon.java*.

Els comentaris poden ser multilínia (/* */) o unilínia (//).

La classe *HolaMon* conté un únic mètode *main*. Les claus {} encerclen blocs de codi, en aquest cas, el corresponent al mètode *main*. Conté una única instrucció, la qual usa el mètode **println** per mostrar una cadena de text ("Hola món!"). El mètode println pertany a un objecte (*out*), el qual está contingut a la classe *System*.

Per als noms s'utilitza la notació **CamelCase**:

- Noms de classes: inicial en majúscules i majúscula la inicial de cada nova paraula.
- Noms de mètodes i variables: inicial en minúscules i majúscula la inicial de cada nova paraula.
- Noms de constants: tots els caràcters en majúscula i paraules separades per guions baixos.

Paradigmes de programació

Programació modular

El programa es divideix en mòduls, cadascun dels quals executa una única activitat i es codifiquen independentment. Cada programa conté un mòdul principal que transfereix el control a altres mòduls (d'ara endavant subprogrames) que tornen el control al mòdul principal en acabar la seva tasca. Cada subprograma pot transferir el control a altres subprogrames, però cada un ha de tornar el control al mòdul que el va invocar. És aconsellable dimensionar els mòduls de manera que no ocupin més de 30 o 40 línies d'instruccions.

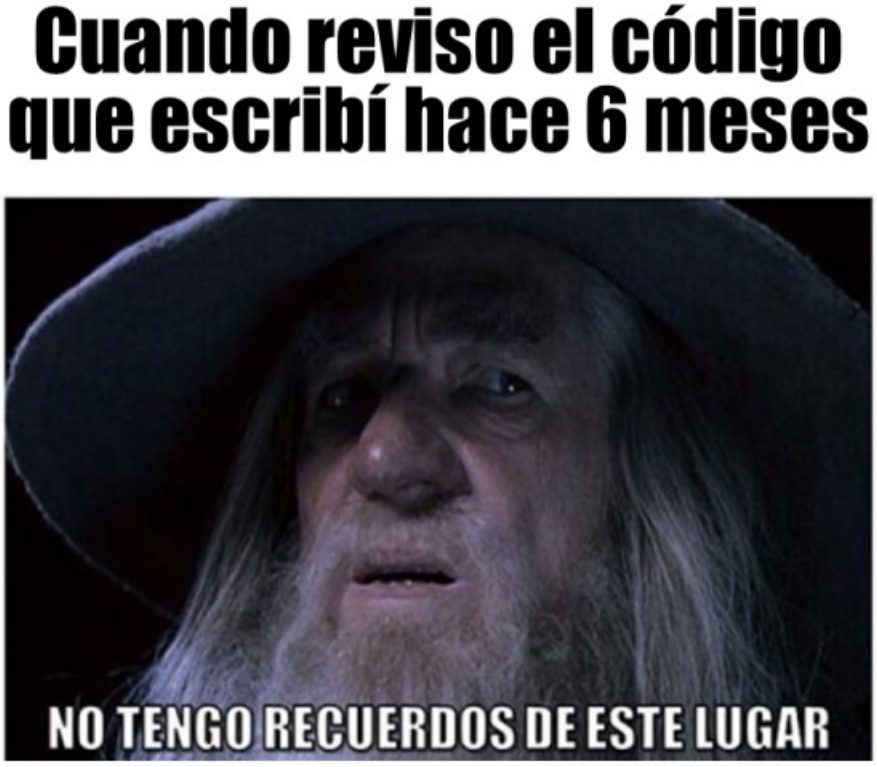
Podem definir un mòdul com una o diverses instruccions físicament contigües i lògicament encadenades, tals que es poden referenciar mitjançant un nom i ser anomenades des de diferents punts d'un programa, així com el conjunt de dades privades d'aquest mòdul requerides per les instruccions. Els mòduls han de tenir la màxima cohesió i el mínim acoblament. La sortida del mòdul ha de ser funció només de la vostra entrada.

Programació estructurada

Per augmentar l'eficiència de la programació i del manteniment, cal dotar els programes d'una estructura. Això, a més, assegura que els programes siguin adaptables, manejables, fàcilment comprensibles i transportables.

Al maig de 1966, Böhm i Jacopini van demostrar el teorema de la programació estructurada: qualsevol programa propi pot ser escrit utilitzant només tres tipus d'estructures de control: seqüencials, condicionals i iteratives. Un programa es defineix com a propi si té només un punt d'entrada i un de sortida o fi, si hi ha camins des de l'entrada fins a la sortida que passen per totes les parts del programa i totes les instruccions són executables i no hi ha bucles sense fi.

Així, doncs, en el disseny dels nostres programes ens limitarem a aquestes estructures de control.



Dades, operadors i expressions

Dades i tipus de dades

El propòsit d'un programa és processar dades. Aquestes dades s'han d'emmagatzemar en memòria. Per accedir a les dades en memòria, els programes utilitzen el concepte de **variable**.

Una variable ve definida pels seus atributs:

- **Nom o identificador**: El nom ens permet identificar unívocament l'objecte.
- **Tipus**: El tipus indica el conjunt de valors que pot tenir i quines operacions s'hi poden aplicar.
- **Valor**: El valor d'un objecte no ha de ser necessàriament un número. El valor d'un objecte serà un element del conjunt al qual pertany i que ve indicat pel seu tipus corresponent. Un objecte podrà ser constant si el seu valor no és modificable o variable si el seu valor es pot modificar.

Un objecte és una **instància** de la classe o tipus de dada de l'objecte. Instanciar un objecte implica reservar-li un espai de memòria, inicialitzar el seu valor i associar la seva ubicació en memòria amb un identificador que pugui ser utilitzat pel programador en el disseny de l'algorisme per accedir-hi.

Java té dues grans categories de tipus de dades:

- tipus primitius
- tipus referencials

Els **tipus primitius** estan incorporats dintre de la sintaxi del llenguatge de programació, i s'utilitzen per construir-ne d'altres més complexos. Les variables de tipus primitius es creen en memòria en definir-les i el seu valor és el de la dada emmagatzemada en memòria.

Els **tipus referencials** són tota la resta. Corresponen a estructures de dades més complexes i a objectes. El valor de la variable de tipus referencial no és el que està emmagatzemat en memòria, sinó una referència (un apuntador) al lloc on està emmagatzemat. Aquestes variables no s'inicialitzen en declarar-les, sinó que cal contruir els objectes amb un operador del llenguatge (***new***).

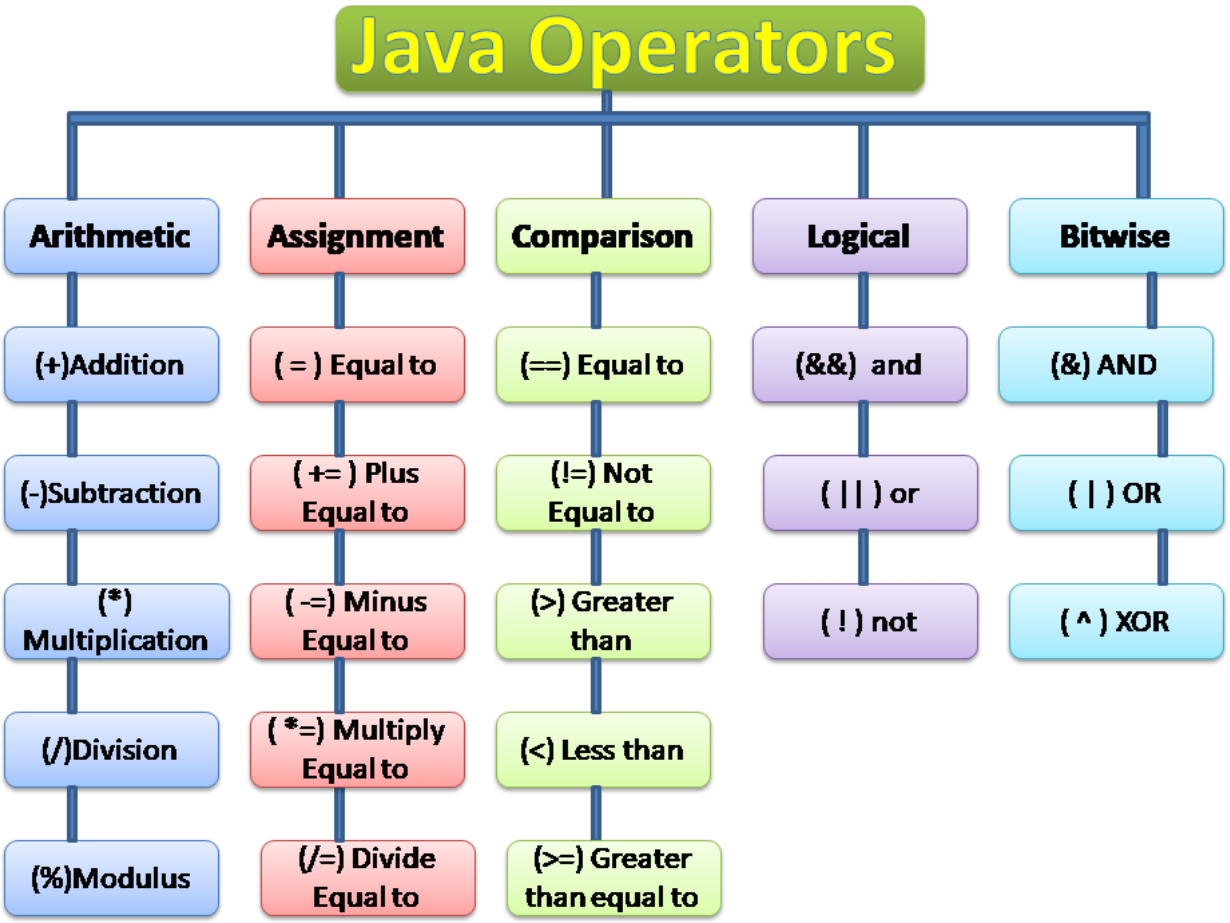
TIPOS DE DATOS PRIMITIVOS EN JAVA				
Tipo	**Valores**	**Por defecto**	**Tamaño**	**Rango**
byte	enteros con signo	0	8 bits	-128 a 127
short	enteros con signo	0	16 bits	-32768 a 32767
int	enteros con signo	0	32 bits	-2147483648 a 2147483647
long	enteros con signo	0	64 bits	-9223372036854775808 a 9223372036854775807
float	IEEE 754 punto flotante	0.0	32 bits	±1.4E-45 a ±3.4028235E+38 y ±∞, ±0, NAN (*Not A Number*)
double	IEEE 754 punto flotante	0.0	64 bits	±4.9E-324 a ±1.7976931348623157E+308 y ±∞, ±0, NAN (*Not A Number*)
char	Carácter Unicode	\u0000	16 bits	\u0000 a \uFFFF
boolean	true, false	false	Entero de 32 bits (sólo usa un bit)	

Operadors i expressions

Per realitzar transformacions i operacions amb les dades s'utilitzen els **operadors**.

Normalment, els operadors actuen sobre dades (**operands**) d'un mateix tipus.

Els operadors que actuen sobre un únic operand s'anomenen **unaris**. Els que actuen sobre dos operands s'anomenen **binaris**.



Les **expressions** són combinacions de constants, variables, operadors, parèntesis i noms de funció. Cada expressió avalua a un valor que ve donat per l'aplicació dels operadors als operands i que correspon a un dels tipus de dades. En llenguatge algorísmic considerarem que existeix **sobrecàrrega d'operadors**, és a dir, que un mateix símbol d'operador crida a diferents funcions segons el tipus d'objectes que té com a operands. Per exemple, a/b realitzarà la divisió entera de a i b si les dues variables són enteres, però calcularà la divisió real si es tracta de variables reals. Si els operands són de diferent tipus, és necessari realitzar un canvi de tipus previ (***type cast***).

L'avaluació d'una expressió es realiza avaluant els parèntesis, començant pels més interns i aplicant als operadors la seva precedència.

Per ser correcta, una expressió ha de tenir **correctesa sintàctica** (ordre correcte de les parts que componen l'expressió) i **correctesa semàntica** (l'aplicació de l'operador sobre els seus operands té sentit).

La manera natural d'especificar que una variable ha de tenir un valor donat és l'**operació d'assignació**, que en llenguatge Java realitza l'*operador* =. Es tracta d'una operació destructiva perquè es perd el valor que contenia prèviament la variable assignada. El valor assignat a la variable ha de ser del seu mateix tipus. Cas contrari, ha de fer-se prèviament una conversió de tipus (*type*

cast). Cal tenir molta cura amb aquestes conversions, que molts llenguatges efectuen de manera silenciosa, perquè poden ocultar errors en temps d'execució difícils de detectar.

El segon membre de l'assignació pot ser qualsevol expressió en què es tingui una combinació de variables, constants i operadors, el resultat de la qual serà el valor que s'assignarà a la variable indicada. La forma general de l'enunciat d'assignació és, doncs,

```
variable = expressió;
```

Exemple:

```
int a, b, x;
...
x = (a + b) /2;
```

Declaració de variables i constants

Tota variable ha d'estar declarada abans de ser usada dintre d'un programa.

La **declaració** consisteix en donar-li un **identificador** (el seu nom amb el qual serà coneguda dintre del programa) i un **tipus de dada**, el qual no podrà canviar durant tota la seva existència en memòria. Addicionament, es pot **inicializar** (donar-li un valor inicial).

El format de la declaració és:

```
tipus identificador;
```

Exemples:

```
int comptador;
char lletra;
float amplada;
double salari;
int repeticions = 4;
float area = 3.5f;
int a, b, c;
```

L'àmbit d'un identificador de variable és el context en què està declarada. Els idenficadors de variable es poden utilitzar només dintre del bloc de codi en què s'han declarat.

Si una dada no ha de ser modificada durant l'execució del programa, cal definir-la com a **constant**, utilitzant el modificador **final** a la seva definició. En aquest cas, cal inicialitzar-la en el moment de la declaració.

```
final int NOMBRE_DE_COSTATS = 3;
```

Conversió de tipus

La conversió de tipus consisteix en la transformació d'una dada d'un tipus a un altre.

La conversió la pot fer de forma **implícita** el propi compilador si es tracta de tipus i valor compatibles i es requereix per a l'avaluació d'una expressió.

```
float a = 3.0;
int b = 2;
float suma = a + b;
```

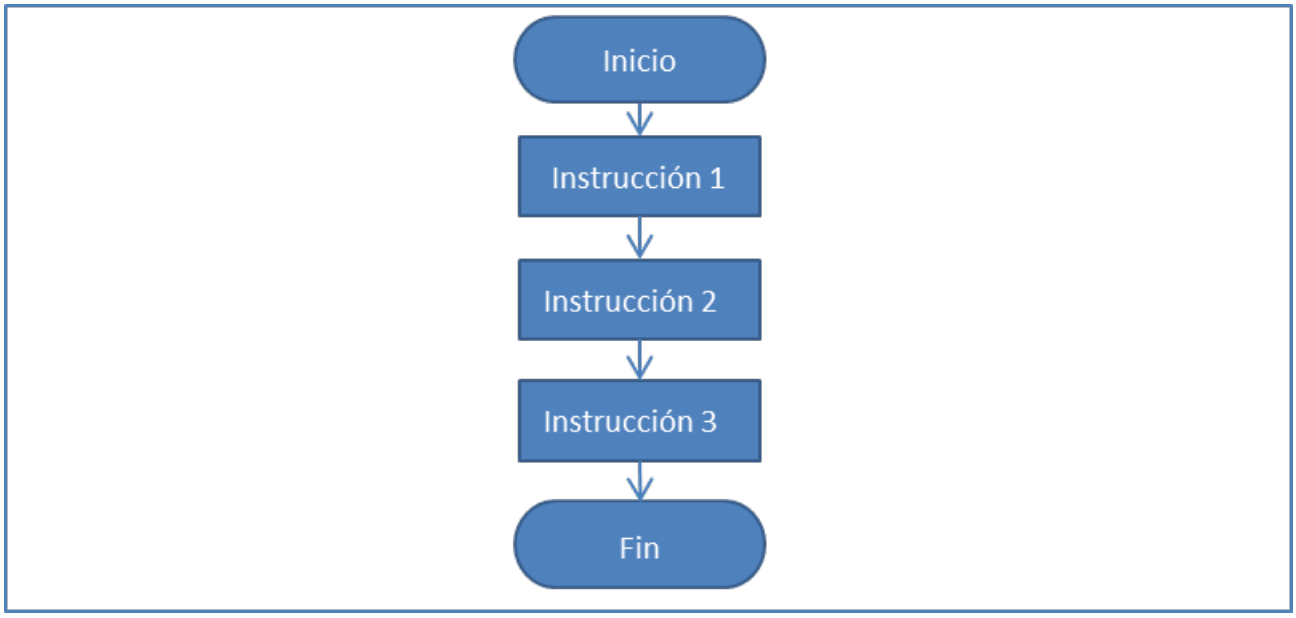
També es pot expressar de forma **explícita** indicant el tipus entre parèntesis abans de la variable.

```
int iVal = 3;
double dVal = (double) iVal;
```

Estructures seqüencials

Les estructures seqüencials són les més senzilles, ja que consisteixen en l'execució ordenada de les instruccions, una darrera de l'altra, en el mateix ordre en què s'ha escrit.

El flux d'execució del programa conté una única línia que va de l'inici de l'algorisme fins al final, passant per cadascuna de les seves instruccions seguint l'ordre d'escriptura.



Exemples

```
import java.util.Scanner;
/**
 * Llegeix una distància en milles marines i la converteix a metres.
 * @author Jose
 */
public class MillesAMetres {
    public static void main(String[] args) {
        final double MILLES_A_METRES = 1852;  //factor conversió constant
        Scanner lector = new Scanner(System.in);
        //llegir distància en milles
        System.out.print("Entra la distància en milles: ");
        double distanciaEnMilles = lector.nextDouble();
        //calcular conversió de milles a metres
        double distanciaEnMetres =
                distanciaEnMilles * MILLES_A_METRES;
        //imprimir resultat a l'usuari
        System.out.println(
          distanciaEnMilles + " milles equivalen a "
                  + distanciaEnMetres + " metres"
        );
    }
}
```

```
import java.util.Scanner;
import java.util.InputMismatchException;
```

```
/**
 * Llegeix el radi d'un cercle i calcula i imprimeix la seva àrea
 * Captura excepció de format de dades invàlides
 * @author Jose
 */
public class AreaCercle {
    public static void main(String[] args) {
        System.out.print("Entra el radi del cercle: ");
        Scanner lector = new Scanner(System.in);
        try {
            //llegir el radi
            double radi = lector.nextDouble();
            //calcular l'àrea
            double area = Math.PI * radi * radi;
            //imprimir resultat
            System.out.format(
                "L'àrea del cercle de radi %.2f és %.2f\n",
                radi, area);
    //        System.out.println(
    //            "L'àrea del cercle de radi " + radi + " és " + area
    //         );
        } catch (InputMismatchException e) {
            System.out.println("Dades invàlides");
        }
    }
}
```

Estructures condicionals

Introducció

Les sentències condicionals contenen instruccions que es poden executar o no en funció del valor d'una condició.

En funció del nombre de branques de codi, es poden classificar en:

- condicional simple
- condicional doble
- condicional múltiple

Condicional simple

Només una branca de codi amb execució opcional. Si es compleix la condició, s'executa el bloc de codi, si no es compleix, no s'executa cap instrucció del condicional i el control passa a la instrucció que segueix l'estructura.

Es un cas simplificat del condicional doble.

L'expressió de la condició pot utilitzar els operadors relacionals i lògics.

Operadors relacionals

operador	significat
>	Major que
<	Menor que
==	Igual a
>=	Major que o igual a
<=	Menor que o igual a
!=	Diferent de

Operadores lògics

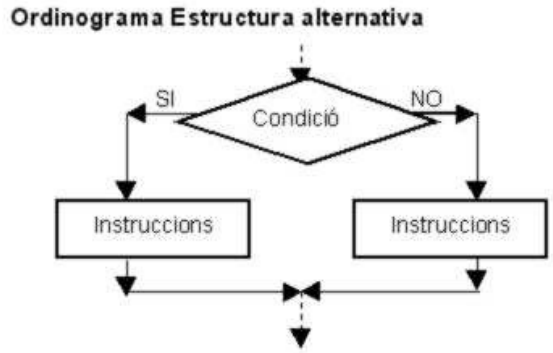
operador	significat
&&	AND
||	OR
!	NOT

AND (Y), OR (O), NOT (NO)

Condicional doble

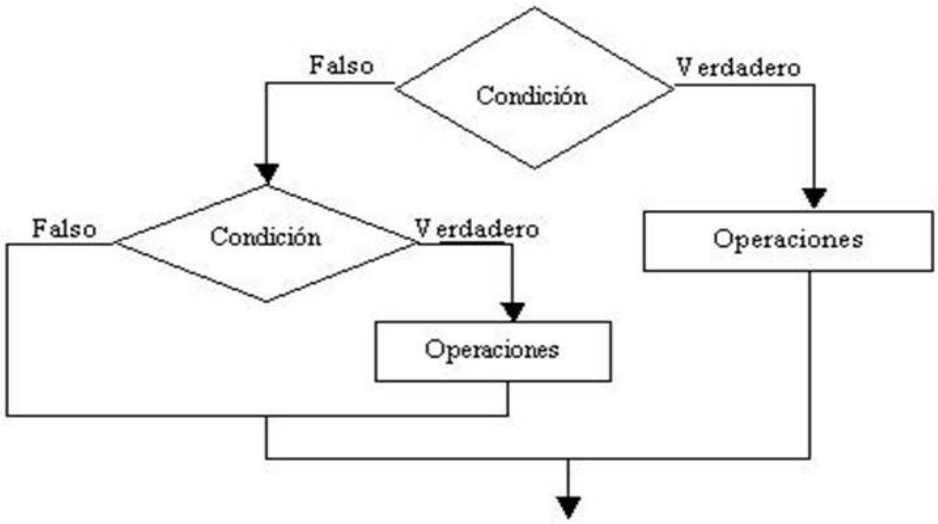
La part lògica de l'ALU (la unitat aritmètico-lògica del processador) dota l'ordinador de la capacitat de prendre decisions. Això s'implementa en llenguatge algorísmic mitjançant la contrucció **si-**

llavors-sinó. La decisió s'especifica en una expressió lògica, la qual ha d'avaluar a cert (*true*) o fals (*false*).




```
si              "condició"
llavors "alternativa certa"
[sino   "alternativa falsa"]
[fi_si]
```

Les estructures condicionals es poden **enniuar**, es a dir, que una de les branques pot contenir una altra estructura condicionals si és necessari.

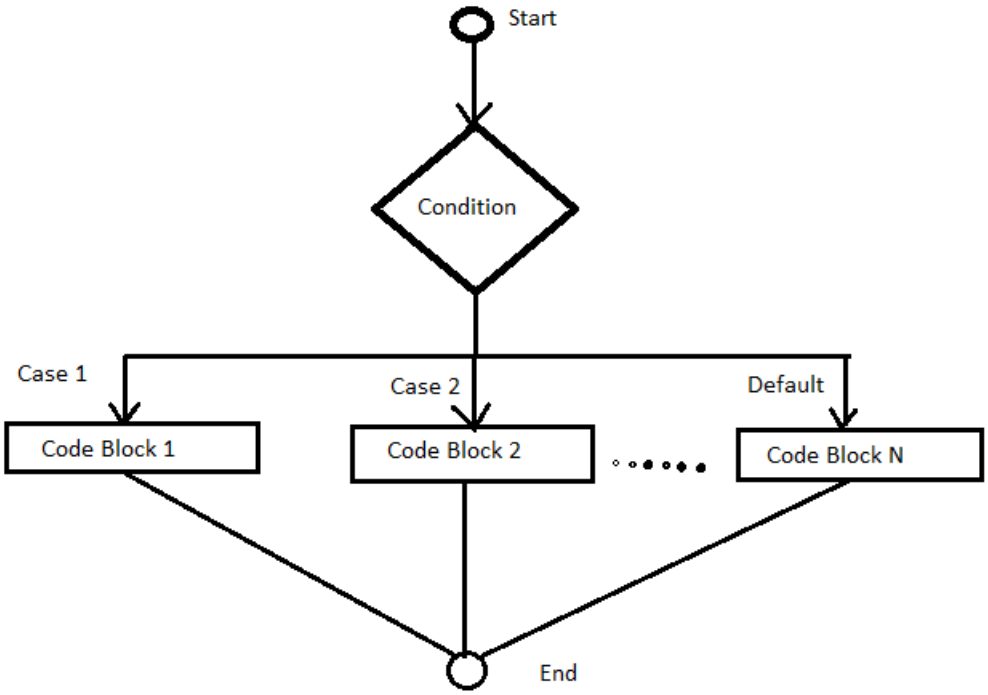



```
int edat;
...
String missatge;
if (edat >= 0) {
```

```
    if (edat >= 67) {
        missatge = "estàs jubilat";
    } else {
        missatge = "no estàs jubilat";
    }
} else {
    missatge = "edat no vàlida";
}
```

Condicional múltiple

Quan la condició a avaluar pot prendre més de dos valors, s'utilitza el condicional múltiple.



Exemples

```
import java.util.Scanner;
/**
 * Llegeix el número de dia de la setmana i
 * mostra el dia de la setmana
 * @author Jose
 */
public class DiaSetmana {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        //llegir número de l'1 al 7 (dia de la setmana)
        System.out.print("Entra el dia de la setmana: ");
        int dia = lector.nextInt();
        System.out.println("Has entrat el dia: " + dia);
        //mostrar dia de la setmana
```

```
        switch (dia) {
            case 1:
                System.out.println("Oh, avui és dilluns");
                break;
            case 2:
                System.out.println("Avui és dimarts");
                break;
            case 3:
                System.out.println("Avui és dimecres");
                break;
            case 4:
                System.out.println("Avui és dijous");
                break;
            case 5:
                System.out.println("Avui és divendres. Hurra!, demà serà
dissabte");
                break;
            case 6:
                System.out.println("Avui és dissabte");
                break;
            case 7:
                System.out.println("Avui és diumenge. Ohh! demà serà dilluns");
                break;
            default:
                System.out.println("Número de dia incorrecte");
                break;
        }
    }
}

import java.util.Scanner;
/**
 * Llegeix qualificació i mostra comentari
 * @author Jose
 */
public class Qualificacio {
    public static void main(String args[]) {
        Scanner lector = new Scanner(System.in);
        //llegir la qualificació
        System.out.print("Entra la teva qualificació: ");
        char qualificacio = lector.next().toUpperCase().charAt(0);
        //mostrar la qualificació entrada
        System.out.println("La teva qualificació és " + qualificacio);
        //mostrar comentari
        switch (qualificacio) {
            case 'A':
                System.out.println("Excel·lent!");
                break;
            case 'B':
            case 'C':
                System.out.println("Ben fet!");
                break;
            case 'D':
                System.out.println("Has aprovat");
            case 'E':
                System.out.println("Insuficient");
            case 'F':
                System.out.println("Torna-ho a intentar");
```

```
                break;
            default:
                System.out.println("Qualificació no vàlida");
        }
    }
}

import java.util.Scanner;
/**
 * Calculadora aritmètica senzilla
 * @author Jose
 */
public class CalculadoraSenzilla {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        //llegir 1r operand
        System.out.print("Entra el primer operand: ");
        int operand1 = lector.nextInt();
        //llegir 2n operand
        System.out.print("Entra el segon operand: ");
        int operand2 = lector.nextInt();
        //llegir operador
        System.out.print("Entra l'operador: ");
        char operador = lector.next().charAt(0);
        //mostrar expressió a avaluar
        System.out.format("Expressió a avaluar: %d %c %d\n",
            operand1, operador, operand2);
        int result=0;  //resultat de l'operació
        String missatge;  //missatge a mostrar
        //calcular
        switch (operador) {
            case '+':
                result = operand1 + operand2;
                missatge = "resultat: "+result;
                break;
            case '-':
                result = operand1 - operand2;
                missatge = "resultat: "+result;
                break;
            case '*':
                result = operand1 * operand2;
                missatge = "resultat: "+result;
                break;
            case '/':
                result = operand1 / operand2;
                missatge = "resultat: "+result;
                break;
            default:
                missatge = "operació no vàlida";
                break;
        }
        //mostrar resultat
        System.out.println(missatge);
    }
}

import java.util.Scanner;
/**
```

```
 * Llegeix de l'usuari el mes i imprimeix els dies d'aquest mes
 * (suposant que l'any no és de traspàs)
 * @author Jose
 */
public class Mesos {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        //llegir el mes
        System.out.print("Entra el mes: ");
        String mes = lector.next();
        //escriure el mes
        System.out.println("Has escrit el mes: " + mes);
        //imprimir els dies del mes
        int dies=-1;  //dies que té el mes
        switch (mes) {
            case "gener":
            case "març":
            case "maig":
            case "juliol":
            case "agost":
            case "octubre":
            case "desembre":
                dies = 31;
                break;
            case "febrer":
                dies = 28;
                break;
            case "abril":
            case "juny":
            case "setembre":
            case "novembre":
                dies = 30;
                break;
            default:
                dies =-1;
                break;
        }
        //imprimir resposta a l'usuari
        if ( dies<0 ) {
            System.out.println("Invàlid nom de mes!");
        } else {
            System.out.println(mes + " té " + dies + " dies");
        }
    }
}
```

Estructures iteratives

Introducció

Els ordinadors estan especialment dissenyats per a les aplicacions en les quals una operació o una sèrie s'han de repetir moltes vegades. La construcció de programació corresponent a aquest cas és el llaç o el **bucle**.

Els bucles es poden classificar en funció de la condició de sortida del mateix de dues maneres:

- Bucles condicionals.
- Bucles comptats.

Bucles condicionals

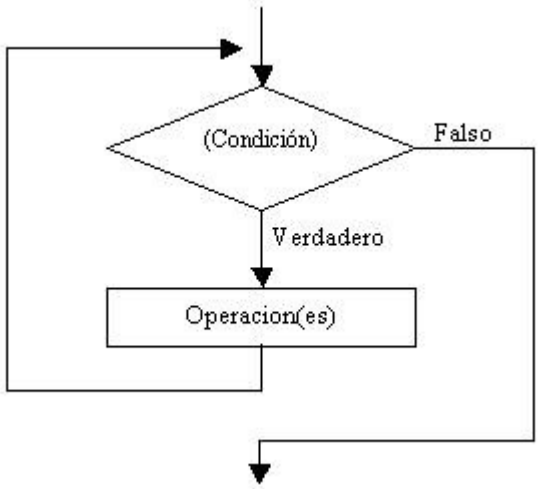
El bloc de codi a repetir s'executa mentres que se satisfà una certa condició. Quan la condició és falsa, se surt del bucle i es continua executant la següent instrucció que segueix.

A cada iteració (repetició) s'avalua novament la condició. En funció del moment en què s'avalua la condició de manteniment del bucle es classifiquen en:

- Bucles condicionals provats a l'inici
- Bucles condicionals provals al final

Bucle condicional provat a l'inici (while)

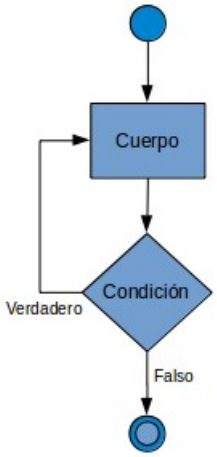
La condició s'avalua abans d'executar el bloc intern. Per tant, si la condició no es compleix, el codi del bucle no s'executa cap vegada.




```
while (condicio) {
  //bloc a executar mentre es compleixi la condicio
  //si no es compleix la primera vegada, aquest bloc no s'executa mai
}
```

Bucle condicional provat al final (do-while)

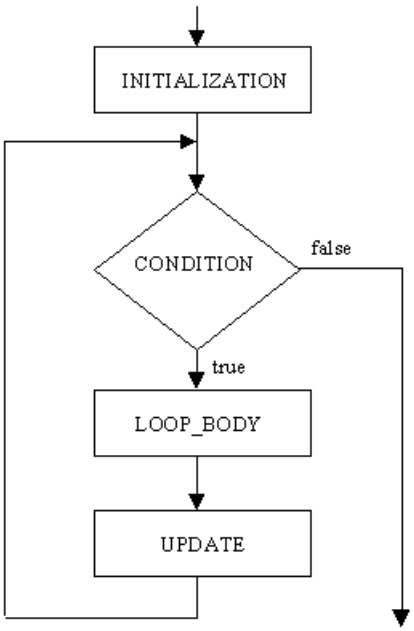
La condició s'avalua al final del bloc intern (després de la seva execució). Per tant, sempre s'executa el bloc intern almenys una vegada.




```
do {
  //bloc a executar mentre es compleixi la condicio
  //encara que la condició sigui falsa la primera vegada, el bloc s'executa una vegada
} while (condicio);
```

Bucles comptats (for)

S'utilitzen generalment quan es coneix el nombre de vegades que el bloc del bucle s'ha de repetir. Fan servir un comptador per controlar el nombre d'iteracions.



Els bucles *for* en llenguatge *Java* són molt potents i tenen moltes més possibilitats d´us que l'especificat per al recompte d'iteracions.

```
for (inicialitzacio; condicio; actualitzacio) {
  //bloc a executar mentre es compleixi la condicio
  //si no es compleix la primera vegada, aquest bloc no s'executa mai
}
```

La inicialització s'executa només la primera vegada, en entrar al bucle.

La condició s'avalua i es comprova cada vegada que s'itera.

L'actualització s'executa cada vegada que s'itera al bucle, al igual que el bloc intern.

Enniuament de bucles

El codi de l'interior del bucle pot també contenir altres bucles, generant estructures per respondre a problemes complexos.

Exemples

```
import java.util.InputMismatchException;
import java.util.Scanner;
/**
 * Programa que llegeix un nombre enter positiu N
 * i mostra els primers N nombres enters positius
 * @author Jose
 */
public class PrimersNNaturals {
    public static void main(String[] args) {
        Scanner lector = new Scanner(System.in);
        //llegir el número
        System.out.print("Entra el número enter positiu: ");
        try {
            int limit = lector.nextInt();
            if (limit > 0) {
                //amb bucle for
                System.out.println("\nAmb bucle for");
                for (int num = 1; num <= limit; num++) {
                    System.out.format("%d ", num);
                }
                //amb bucle while (provat a l'inici)
                System.out.println("\nAmb bucle while (provat a l'inici)");
                int iter1=1;
                while (iter1 <= limit) {
                    System.out.format("%d ", iter1);
                    iter1++;
                }
                //amb bucle do while (provat al final)
                System.out.println("\nAmb bucle do while (provat al final)");
                int iter2=1;
                do {
                    System.out.format("%d ", iter2);
                    iter2++;
                } while (iter2 <= limit);
```

```
            } else {
                System.out.println("Has d'entrar un nombre positiu");
            }
        } catch (InputMismatchException e) {
            System.out.println("Has d'entrar un número enter positiu");
        }
    }
}

/**
 * Imprimeix els primers 20 nombres naturals
 * @author Jose
 */
public class Print20Enters {
    public static void main(String[] args) {
        final int LIMIT = 20;
        //amb bucle while
        //inicialitzar comptador
//        int comptador = 1;
//        while (comptador <= LIMIT) {
//            System.out.println(comptador);
//            comptador++;
//        }
        //amb bucle for
//        for (int i=1; i<=LIMIT; i++) {
//            System.out.println(i);
//        }
        //amb bucle do while
        int comptador = 1;
        do {
            System.out.println(comptador);
            comptador++;
        } while (comptador <= LIMIT);
    }
}
```