Introducció a la programació orientada a objectes

Vicent Moncho Mas

PID_00191133

UOC
Universitat Oberta de Catalunya

Els textos i imatges publicats en aquesta obra estan subjectes –llevat que s'indiqui el contrari– a una llicència de Reconeixement-NoComercial-SenseObraDerivada (BY-NC-ND) v.3.0 Espanya de Creative Commons. Podeu copiar-los, distribuir-los i transmetre'ls públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), no en feu un ús comercial i no en feu obra derivada. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.ca

Índex

1. Introducció 5
- 1.1. La programació 5
- 1.2. Models de desenvolupament 6
  - 1.2.1. Programació imperativa 7
  - 1.2.2. Programació funcional 8
  - 1.2.3. Programació lògica 8
  - 1.2.4. Programació orientada a objectes 9

2. Conceptes i característiques principals 11
- 2.1. El model orientat a objectes 11
  - 2.1.1. Classes 12
  - 2.1.2. Objectes 16
  - 2.1.3. Herència 19
- 2.2. Característiques de la programació orientada a objectes 21
  - 2.2.1. Abstracció 21
  - 2.2.2. Ocultament o encapsulació 22
  - 2.2.3. Polimorfisme 24
  - 2.2.4. Destrucció d'objectes 28
  - 2.2.5. Anàlisi i disseny orientat a objectes 29

3. POO en els diversos llenguatges de programació 31
- 3.1. Smalltalk 31
- 3.2. Eiffel 33
- 3.3. C++ 34
- 3.4. ActionScript 3.0 35
- 3.5. Ada 36
- 3.6. Perl 38
- 3.7. PHP 39
- 3.8. C# 40
- 3.9. Java 41
- 3.10. JavaScript 44

1. Introducció

1.1. La programació

La programació és un procés pel qual s'escriu, es prova, es depura i es modifica el codi. Els programes com a elements que formen el programari són un conjunt d'instruccions que s'executen en el maquinari amb l'objectiu de dur a terme una tasca determinada.

Per a desenvolupar programes de certa envergadura o complexos, amb certes garanties de qualitat, cal seguir algun dels models de desenvolupament de programari que hi ha, en els quals la programació és només una de les etapes del procés de desenvolupament del programari.

Un **programa** és un conjunt d'instruccions que s'executen amb l'objectiu de resoldre un cert problema. Els programes es componen en diversos algorismes, i cadascun d'aquests algorismes resol una part del problema inicial. D'aquesta manera, la complexitat de cadascuna de les parts és més petita que la del programa complet (aquesta tècnica es coneix com a *divideix i venceràs*).

Segons Niklaus Wirth, un programa és format per algorismes i una estructura de dades.

Niklaus Wirth (Winterthur, 1934)

Doctorat el 1963 a Berkeley, de 1963 a 1967 va ser professor d'informàtica a Stanford i a la Universitat de Zuric. A partir de 1968, va passar a ser professor d'informàtica a l'ETH de Suïssa i es va agafar dos anys sabàtics a la Xerox PARC de Califòrnia.

Va ser el cap de disseny dels llenguatges de programació Euler, Algol W, Pascal, Modula, Modula-2 i Oyeron, i va ocupar una bona part del temps en l'equip de disseny i implementació de sistemes operatius Lilith i Oberon per al Lola en el disseny del maquinari digital i el sistema de simulació.

El seu article de desenvolupament d'un programa per refinament successiu ("Program Development by Stepwise Refinement") és considerat un text clàssic en l'enginyeria del programari, i també el seu llibre *Algoritmos + estructuras de datos = programas*, que va rebre un reconeixement ampli i que encara avui és útil en l'ensenyament de la programació. Va rebre el premi Turing pel desenvolupament d'aquests llenguatges de programació el 1984.

La programació té com un dels **objectius** principals crear programari de qualitat, i els factors principals que indiquen el **nivell de qualitat d'un programa** són els següents:

1) Correcció: un programa és correcte si fa el que ha de fer tal com es va establir. Per a determinar si un programa actua correctament, és molt important tenir una especificació del que ha de resoldre el programa abans de desenvolupar-lo; d'aquesta manera, una vegada implementat s'ha de comprovar que realment l'implementa.

2) Claredat: és fonamental que el codi sigui tan clar i llegible com sigui possible, per a facilitar-ne així el desenvolupament i el manteniment. S'ha de mirar que l'estructura sigui senzilla i coherent, i també tenir cura de l'estil en l'edició; d'aquesta manera, es facilita la feina del programador, tant en la fase de creació com en les fases posteriors de correcció d'errors, ampliacions, modificacions, etc., fases que fins i tot les pot fer un altre programador, amb la qual cosa la claredat és encara més necessària perquè aquests altres programadors continuïn la feina fàcilment.

3) Eficiència: a més de fer allò per a què es va crear, ho ha de fer minimitzant els recursos que utilitza. L'eficiència es basa en el temps que triga a fer la tasca per a la qual s'ha creat i en la quantitat de memòria que necessita. Hi ha altres recursos que també s'han de considerar: espai de disc que utilitza, trànsit de xarxa que genera, etc.

4) Portabilitat: es basa en la capacitat de poder-se executar en una plataforma, sia maquinari o programari, diferent de la plataforma en què es va elaborar. La portabilitat és una característica molt desitjable per a un programa, ja que, per exemple, permet, a un programa que s'ha desenvolupat per a sistemes GNU/Linux executar-se també en la família de sistemes operatius Windows, fins i tot en diferents màquines virtuals de Java o en els diferents navegadors que hi ha en el mercat.

1.2. Models de desenvolupament

S'entén per *models de desenvolupament* o *paradigmes de programació* els diversos enfocaments o les diverses filosofies que s'han anat creant per a construir el programari. No hi ha un model millor que un altre, sinó que, depenent de la mena de problema, un model pot ser més apropiat que un altre.

Els models de desenvolupament més comuns són els següents:

- Programació imperativa.
- Programació funcional.
- Programació lògica.
- Programació orientada a objectes.

1.2.1. Programació imperativa

La programació imperativa es basa en un conjunt d'instruccions que indiquen al maquinari com ha de fer una tasca. Es considera la més comuna i la representen, per exemple, llenguatges com C o BASIC.

El maquinari està dissenyat per a executar codi de màquina, escrit d'una manera imperativa i seqüencial. Des d'aquesta perspectiva de baix nivell, les sentències són instruccions en el llenguatge de màquina natiu de l'ordinador (per exemple, el llenguatge d'assemblador).

Els primers llenguatges imperatius van ser els llenguatges d'assemblador, que són condicionats pel maquinari que els havia d'implementar. En aquests llenguatges, les instruccions van ser molt simples, cosa que va provocar la implementació de maquinari fàcil, però va obstruir la creació de programes complexos. Fortran, que va començar a desenvolupar el 1954 John Backus a IBM, va ser el primer gran llenguatge de programació que va superar els obstacles que presentava el codi màquina en la creació de programes complexos.

Els llenguatges següents implementen la programació imperativa: BASIC, C, C#, C++, Fortran, Pascal, Java, Perl, PHP.

John Backus (Filadèlfia, 1924 - Oregon, 2007)

Informàtic nord-americà. John Backus va guanyar el premi Turing el 1977 per la feina en sistemes de programació d'alt nivell, en especial per la feina amb Fortran.

Per a evitar les dificultats de programació de les calculadores de la seva època, el 1954 Backus es va encarregar de dirigir un projecte de recerca a IBM per a projectar i dur a terme un llenguatge de programació més proper a la notació matemàtica normal. D'aquest projecte va sorgir el llenguatge Fortran, el primer dels llenguatges de programació d'alt nivell que va tenir un gran impacte, fins i tot comercial, en la comunitat informàtica emergent.

Després d'haver portat a cap el Fortran, Backus va ser un membre molt actiu del comitè internacional que es va encarregar del projecte de llenguatge Algol. En aquest context va proposar una notació per a representar les gramàtiques que es feien servir en la definició d'un llenguatge de programació (les anomenades *gramàtiques lliures de context*). Aquesta notació es coneix com a *BNF* o *Forma de Naur i Backus* (*Backus-Naur Form*), i uneix el nom de Backus al de Peter Naur, un informàtic europeu del comitè Algol que va contribuir a definir-lo.

A la dècada de 1970, Backus es va interessar sobretot per la programació funcional i va projectar el llenguatge de programació FP, que es descriu en el text que li va servir per a guanyar el premi Turing: "Can Programming be Liberated from the Von Neumann Style?". Es tracta d'un llenguatge d'ús fonamentalment acadèmic que, tanmateix, va animar una bona colla d'investigacions. El projecte FP, transformat en FL, es va acabar quan Backus es va jubilar a IBM, el 1991.

Font: Viquipèdia.

1.2.2. Programació funcional

La programació funcional és un paradigma de programació que es basa a utilitzar funcions matemàtiques. El màxim representant d'aquest paradigma és el llenguatge Lisp.

L'objectiu és aconseguir llenguatges expressius i matemàticament elegants, en què no faci falta baixar al nivell de la màquina per a descriure el procés que ha dut a terme el programa. Els programes són constituïts únicament per definicions de funcions; aquestes definicions s'entenen no com a subprogrames clàssics d'un llenguatge imperatiu sinó com a funcions purament matemàtiques.

Algunes altres característiques pròpies d'aquests llenguatges són que no tenen assignacions de variables i que no tenen construccions estructurades com la seqüència o la iteració. És el que obliga, a la pràctica, que totes les repeticions d'instruccions es duguin a terme per mitjà de funcions recursives.

Hi ha dues grans categories de llenguatges funcionals: els **funcionals purs** (Haskell i Miranda) i els **híbrids** (Scala, Lisp, Scheme, Ocaml, SAP i Standard ML). La diferència entre els uns i els altres és que els llenguatges funcionals híbrids són menys rígids que els purs, ja que admeten conceptes dels llenguatges imperatius, com les seqüències d'instruccions o l'assignació de variables.

Es pot incloure el Perl com a llenguatge funcional híbrid, ja que, encara que és un llenguatge de propòsit general, s'hi poden implementar programes usant exclusivament funcions que ha definit l'usuari.

1.2.3. Programació lògica

Generalment, en els llenguatges de programació imperatius, quan ens enfrontem a un problema complex, la implementació de la solució d'aquest problema en pot ocultar o dificultar la comprensió. En aquest sentit, la lògica matemàtica és la manera més senzilla d'expressar formalment problemes complexos i de resoldre'ls aplicant regles, hipòtesis i teoremes.

La programació lògica té l'hàbitat natural en aplicacions d'intel·ligència artificial. Per exemple:

- Sistemes experts. El programa imita les recomanacions d'un expert sobre algun domini de coneixement.

- Demostració automàtica de teoremes. El programa genera teoremes nous sobre una teoria existent.

- Reconeixement de llenguatge natural. El programa és capaç de comprendre (amb limitacions) la informació que conté una expressió lingüística humana.

El llenguatge principal de programació lògica és el Prolog.

1.2.4. Programació orientada a objectes

La programació orientada a objectes té l'origen en el llenguatge Simula 67, que van crear Ole-Johan Dahl i Kristen Nygaard al Centre Noruec de Còmput d'Oslo.

L'objectiu del llenguatge era implementar simulacions de naus d'una manera simple i més òptima. Van pensar a agrupar els diferents tipus de naus en diverses classes d'objectes, considerant cada classe d'objectes responsable de definir les seves pròpies dades i el seu propi comportament.

Ole-Johan Dahl (Mandal, 1931-2002)

És un dels científics de la computació més famosos a Noruega. Juntament amb Kristen Nygaard, Ole-Johan Dahl va produir les primeres idees sobre programació orientada a objectes durant dècada de 1960 al Centre Noruec de Còmput (NCC), com a part dels llenguatges de programació per a simulació Simula I (1961-1965) i Simula 67 (1965-1968). Dahl i Nygaard van ser els primers a desenvolupar els conceptes d'*objecte*, *classe*, *herència*, *creació dinàmica d'objectes*, etc., que són aspectes importants del paradigma de l'OO.

Dahl va aconseguir una plaça de professor a la Universitat d'Oslo el 1968, en la qual va destacar com un educador i investigador privilegiat. Allà va treballar en *Estructuras Jerárquicas de Programas*, probablement la seva publicació més influent, que va escriure juntament amb C. A. R. Hoare i Edsger Dijkstra en el famós llibre *Structured Programming*, de 1972, potser el llibre sobre programari més conegut d'aquella dècada.

A mesura que avançava la seva carrera, Dahl es va interessar en l'ús de mètodes formals, per exemple, per a raonar rigorosament sobre orientació a objectes. Com tot bon científic de la computació, la seva experiència comprenia des de l'aplicació pràctica de les seves idees fins a la demostració matemàtica de la correcció d'aquestes idees per a assegurar la validesa del seu enfocament. Va rebre el premi Turing per la feina feta el 2001, un any abans de morir.

Font: Viquipèdia.

Kristen Nygaard (1926-2002)

Va ser un matemàtic noruec, un pioner de la informàtica i un polític. Kristen Nygaard va obtenir el títol de màster en Matemàtiques a la Universitat d'Oslo el 1956. La seva tesi sobre la teoria de probabilitat abstracta es va titular *Theoretical Aspects of Monte Carlo Methods* ('Aspectes teòrics dels mètodes de Montecarlo'). Entre 1948 i 1960, Nygaard va desenvolupar diverses tasques en el Departament de Defensa noruec, incloses tasques investigadores. Va ser cofundador i primer president de la Societat Noruega d'Investigacions Operacionals (1959-1964). El 1960, va ser contractat pel Centre Noruec de Computació (NCC), com a responsable d'establir l'NCC com un institut important de recerca durant la dècada de 1960. Allà, al costat d'Ole-Johan Dahl, va desenvolupar els llenguatges Simula I (1961-1965) i Simula 67.

Va treballar per als sindicats noruecs sobre planificació, control i processament de dades, tot això avaluat segons els objectius de la mà d'obra organitzada (1971-1973, juntament amb Olav Terje Bergo). Finalment, també va dedicar una mica d'esforç a estudiar l'impacte social de les tecnologies de la computació, i també el llenguatge general de descripció de sistemes Delta (1973-1975, juntament amb Erik Holbaek-Hanssen i Petter Haandlykken). Nygaard va ser professor a Aarhus (Dinamarca) durant el curs 1975-1976, i va ser nomenat professor emèrit a Oslo (a temps parcial des de 1977, i a temps complet de 1984 a 1996).

Font: Viquipèdia.

El novembre de 2002, va rebre, juntament amb Dahl, la medalla John von Neumann de l'IEEE, "per haver introduït els conceptes subjacents de la programació orientada a objectes, mitjançant el disseny i la implementació de Simula 67".

Aquests conceptes es van perfeccionar en el llenguatge Smalltalk, dissenyat per a ser un sistema completament dinàmic en què els objectes es podrien crear i modificar "sobre la marxa" en comptes de tenir un sistema basat en programes estàtics.

La programació orientada a objectes va prendre posició com l'estil de programació dominant a mitjan la dècada de 1980, en bona part per la influència de C++, una extensió del llenguatge de programació C. El domini que va tenir es va consolidar gràcies a l'apogeu de les interfícies gràfiques d'usuari, per a les quals la programació orientada a objectes està particularment ben adaptada.

Les característiques d'orientació a objectes es van agregar a molts llenguatges que hi havia llavors, com ara Ada, BASIC, Lisp o Pascal. L'addició d'aquestes característiques als llenguatges que no s'hi havien dissenyat de bon principi va conduir sovint a problemes de compatibilitat i a la capacitat de manteniment del codi.

Els llenguatges orientats a objectes "purs", d'altra banda, no tenien les característiques que molts programadors solien utilitzar. Per a evitar aquest problema, van aparèixer una sèrie d'iniciatives per a crear llenguatges nous basats en mètodes, però orientats a objectes, que mantenien d'aquesta manera algunes de les característiques imperatives.

El llenguatge Eiffel va ser el primer que complia aquests objectius, però va ser reemplaçat per Java, en bona part per l'aparició d'Internet i per la implementació de la màquina virtual de Java en la majoria de navegadors.

El llenguatge PHP, a partir de la versió 5, suporta una orientació completa a objectes i compleix totes les característiques pròpies de l'orientació a objectes.

En el cas de JavaScript, és un llenguatge orientat a objectes en què l'herència s'implementa seguint el paradigma de la programació basada en prototips, això és, les classes noves es generen clonant les classes base i ampliant-ne els mètodes i les propietats.

2. Conceptes i característiques principals

La tecnologia orientada a objectes ja no s'aplica tan sols en els llenguatges de programació, sinó que també s'aplica en l'anàlisi i el disseny de programari, igual que en les bases de dades. És un dels models més productius, que es deu a les grans capacitats i els grans avantatges que té davant de la resta de models de programació.

L'orientació a objectes es basa en la divisió del programa en petites unitats lògiques de codi; aquestes unitats es coneixen com a *objectes*. Aquests objectes són unitats independents que es comuniquen entre elles mitjançant missatges.

Els **avantatges principals** d'un llenguatge orientat a objectes són els següents:

- El foment de la reutilització i extensió del codi.
- L'elaboració de sistemes més complexos.
- La relació del sistema amb el món real.
- La creació de programes visuals.
- La construcció de prototips.
- L'agilitació del desenvolupament de programari.
- La facilitació del treball en equip.
- El manteniment més senzill del programari.

Un dels aspectes més interessants de la programació orientada a objectes és que proporciona conceptes i eines amb què es modela i representa el món real tan fidelment com sigui possible.

En els subapartats següents es mostraran exemples utilitzant el llenguatge de programació Java.

S'ha triat el llenguatge Java per dos motius principalment: en primer lloc, compleix tots els conceptes de la programació orientada a objectes que s'explicaran i, en segon lloc, la sintaxi que té és senzilla i, per tant, facilita la comprensió.

Els exemples no estan per a reproduir-los, sinó per a ajudar a comprendre els conceptes que s'expliquen. En el mòdul següent l'estudiant podrà practicar part d'aquests conceptes amb JavaScript, ja que aquest no cobreix tot el paradigma de la programació orientada a objectes.

2.1. El model orientat a objectes

El model de programació orientada a objectes es basa en els conceptes fonamentals següents:

- Classes.
- Objectes.
- Herència.
- Tramesa de missatges.

2.1.1. Classes

L'**estandardització** és una tècnica que provoca que hi hagi diferents objectes d'un mateix tipus, que comparteixen el procés de fabricació.

Exemples

Alguns exemples d'estandardització són el telèfon mòbil, el cotxe, l'ordinador i la televisió.

En el món de la programació orientada a objectes, el nostre telèfon mòbil pot ser una instància d'una classe que es podria dir *telèfon*. Cada telèfon mòbil té un seguit de característiques, com la marca, el model, el sistema operatiu i el tipus de pantalla i de teclat, i alguns dels comportaments que té són fer i rebre trucades, enviar missatges i transmetre dades.

En el procés de fabricació s'aprofita el fet que els mòbils tenen característiques i es construeixen models o plantilles comuns, a partir dels quals es creen milers de telèfons mòbils del mateix model.

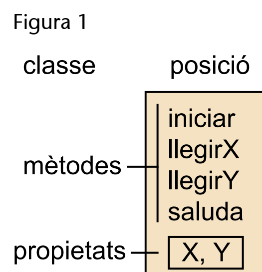
En el món de la programació orientada a objectes, la plantilla es coneix com a *classe* i els equips que traiem a partir d'aquesta classe s'anomenen *objectes* o *instància de classe*. D'aquesta manera, d'una sola classe es poden crear molts objectes del mateix tipus.

La **classe** és un model o prototip, que defineix les variables i els mètodes comuns a tots els objectes, o una plantilla genèrica per a un conjunt d'objectes de característiques semblants.

La instància és un objecte d'una classe en particular.

La figura següent presenta un exemple d'una classe posició, en què hi ha quatre mètodes (iniciar, llegirX, llegirY i saluda) i dues propietats, X i Y:

Figura 1



A partir d'aquesta classe es poden crear tants objectes com faci falta i tots tenen els mètodes i les propietats que s'han definit en la classe.

A Java la creació d'una classe es fa utilitzant la paraula clau reservada *class*, seguida del nom de la classe, i s'acaba amb un bloc de codi delimitat per dues claus que defineixen el cos de la classe. Per exemple:

```
class elMeuPunt {


}
```

Encara que la classe elmeuPunt és sintàcticament correcta, se la coneix com a *classe buida* (ja que no disposa de mètodes ni propietats).

Una classe, però, ha de definir les propietats i els mètodes d'un objecte. La sintaxi d'una definició de classe és la següent:

```
class Nom_De_Classe {
       tipus_de_variable nom_de_propietat1;
       tipus_de_variable nom_de_propietat2;
       // . . .
       tipus_retornat nom_de_metode1( llista_de_parametres ) {
              cos_del_metode1;
       }
       tipus_retornat nom_de_metode2( llista_de_parametres ) {
              cos_del_metode2;
       }
       // . . .
}
```

Els tipus tipus_de_variable i tipus_retornat han de ser tipus simples Java o noms d'altres classes que ja s'han definit abans.

Les propietats es defineixen a dins de la classe declarant variables. Per exemple, tot seguit s'afegeixen a la classe elmeuPunt dues propietats, x i y:

```
class elMeuPunt {
       int x, y;
}
```

Les propietats es poden declarar amb dues classes de tipus de dades: un tipus simple del llenguatge o el nom d'una classe (és una referència a un objecte).

Els mètodes són funcions que defineixen la interfície d'una classe, les capacitats que tenen i el comportament. Aquests mètodes es declaren al mateix nivell que les variables dins de la definició de classe.

En la declaració dels mètodes es defineix el tipus de valor que tornen i una llista de paràmetres d'entrada separats per comes. En l'exemple següent, el mètode torna la suma de dos enters:

```
int metodeSuma( int paramX, int paramY ) {
       return ( paramX + paramY );
}
```

En cas que no es vulgui tornar cap valor, s'ha d'indicar com a tipus la paraula reservada *void*. Així mateix, si no es volen paràmetres, la declaració del mètode ha d'incloure un parell de parèntesis buits:

```
void metodeBuit( ) { };
```

Els mètodes són cridats indicant una instància individual de la classe, que té el seu propi conjunt únic de variables d'instància, de manera que els mètodes s'hi poden referir directament. El mètode inicia() següent es fa servir per a establir els valors de les dues variables d'instància:

```
void inicia( int paramX, int paramY ) {
        x = paramX;
        y = paramY;
}
```

Tot seguit, presentarem un exemple complet en què es defineix una classe amb mètodes i propietats. Les característiques principals d'aquesta classe són les següents:

- La classe modelitza un punt d'espai i, per tant, disposa de dues propietats que defineixen les coordenades que té.

- La classe disposa de dos mètodes: el primer fa la suma de les dues coordenades i torna el valor, i el segon calcula la distància entre el punt propi i les coordenades del punt passat com a paràmetre.

```
//Definim una classe elmeuPunt amb:
//un mètode metodeSuma que ens retorna la suma de les dues coordenades que es passen com a paràmetres
//un mètode distancia que calcula la distància entre el punt passat com a paràmetre i el punt de
//l'objecte
//un mètode metodeBuit que no fa res!



class elmeuPunt{
  //Definim dos propietats x e y, que defineixen les coordenades de l'objecte punt
  int x, y;

   //Definim un mètode que torna el valor de la suma dels dos valors enters que se li passen com a
```

```
 //paràmetres
 int metodeSuma( int paramX, int paramY ) {
           return ( paramX + paramY );
 }

//Definim una funció que ens torna la distància entre el punt objecte i el que és passat per paràmetre
   double distancia( int x, int y ) {
 //Assignem a la variable dx la diferència entre la coordenada x de l'objecte (this.x) i la
 //coordenada x passada pel paràmetre
    int dx= this.x - x;
 //Assignem a la variable dy la diferència entre la coordenada y de l'objecte (this.y) i la
 //coordenada y passada pel paràmetre
             int dy = this.y - y;
//Tornem el resultat de l'arrel quadrada de la suma de dx i dy, per tant la distància entre dos punts
    return Math.sqrt( dx*dx + dy*dy );
   }
 //Definim un mètode buit que no fa res.
   void metodeBuit( ) { }

 //Definim un mètode que inicialitza l'objecte amb els valors que es passen com a paràmetre
 //de forma que ja tenim el primer objecte elmeuPunt amb les coordenades definides
   void inicia( int paramX, int paramY ) {
            x = paramX;
            y = paramY;
}

 //Definim un nou mètode inicialitzador, però al no ficar la paraula this, no s'està assignant a
 //la variable //d'instancia, sino a la mateixa que tenim al mètode, per lo que en aquest mètode
 //realment sols es modifica la variable y
   void inicia2( int x, int y ) {
              x = x;
              this.y = y;
   }

//Definim un mètode constructor elmeuPunt, on tenim dos paràmetres que s'assignen a les
//propietats de la instància x e y
   elmeuPunt( int paramX, int paramY ) {
            this.x = paramX;   // En aquesta línia this es pot obviar, ja que x correspon
                               // a la variable de la funció però com que es tracta
                               // del constructor es la pròpia instància de la classe
            y = paramY;        // Aquesta línia es equivalent a l'anterior
   }

//Definim un mètode constructor elmeuPunt, on com no rebem cap paràmetre s'inicialitza amb les
//coordenades (-1,-1)
   elmeuPunt() {
            inicia(-1,-1);
```

```
        }
}
```

2.1.2. Objectes

Hi ha moltes definicions del concepte o*bjecte*. En primer lloc, en el món real un objecte és qualsevol cosa que veiem al nostre voltant. Ara mateix sou davant d'uns materials en format paper o digital, i tots dos són objectes, com el telèfon mòbil, la televisió o un vehicle.

Si analitzem un objecte del món real com un ordinador portàtil, veiem que és format pels components següents: la placa base, el processador, el disc dur i els mòduls de memòria. És la feina coordinada de tots els components el que fa que l'ordinador porti a cap les funcions.

Cadascun d'aquests components és summament complex, és fabricat per companyies diferents amb materials diferents i fins i tot amb dissenys diferents. No fa falta, però, saber com funcionen internament cadascun d'aquests components, perquè cadascun és una unitat autònoma i tot el que en cal saber és com interactuen entre ells. Un processador i un mòdul de memòria són compatibles amb la placa base si la interacció que tenen és correcta.

La programació orientada a objectes funciona de la mateixa manera: els programes són construïts segons components diferents, cadascun dels quals té un paper específic, i tots els components es poden comunicar entre ells de maneres definides.

> Un **objecte** és una unitat de codi format per propietats, que en defineixen l'estat, i mètodes, que en defineixen el comportament.

En el món real, tot objecte té dos components: **característiques** i **comportament**. En el cas de l'exemple del vehicle, tenim les característiques o propietats següents: marca, model, color, velocitat màxima, etc.; el comportament es basa en les accions següents: frenar, accelerar, recular, omplir combustible, canviar llantes, etc.

En el món de la programació, els objectes també tenen característiques i comportaments; d'aquesta manera, els objectes en la programació emmagatzemen les característiques en variables i implementen el comportament amb funcions.

Exemple

Imaginem-nos que en un garatge hi ha estacionat un Ford Focus de color blau capaç d'arribar a 180 km/h; si es passa al món de la programació, podem considerar un objecte Automòbil amb les propietats següents:

- Marca = Ford.

- Model = Focus.
- Color = Blau.
- Velocitat = 180 km/h.

Quan s'assignen valors a les propietats de l'objecte, l'objecte adquireix un estat; d'aquesta manera, les variables emmagatzemen els estats d'un objecte en un determinat moment.

Seguint la classe plantejada en el subapartat anterior, tot seguit es pot veure el procés de creació d'un objecte o instanciació de la classe.

Cada vegada que es crea una classe s'afegeix un altre tipus de dada que es pot utilitzar igual que una dels tipus simples de dades; per aquest motiu, quan es declara una variable nova, es pot utilitzar un nom de classe com a indicador de tipus:

```
elmeuPunt p;
```

És una declaració d'una variable *p*, que és una referència a un objecte de la classe elmeuPunt, de moment amb el valor per defecte null.

Les classes implementen un mètode especial que s'anomena *constructor*; aquest mètode inicia l'objecte immediatament després d'haver-se creat i té exactament el mateix nom que la classe que l'implementa; és a dir, no hi pot haver cap altre mètode que comparteixi el nom amb el de la seva classe. Una vegada definit, es crida automàticament el constructor quan es crea un objecte d'aquesta classe (quan s'utilitza l'operador new).

El constructor no torna cap tipus, ni tan sols void. La missió que té és iniciar tot estat intern d'un objecte (les seves propietats), i fer que l'objecte es pugui utilitzar immediatament, reservar memòria per a les variables, iniciar els valors, etc.

Per exemple:

```
elmeuPunt( ) {
      inicia( -1, -1 );
}
```

Aquest constructor anomenat *constructor per defecte*, com que no té paràmetres, estableix el valor –1 a les variables d'instància *x* i *y* dels objectes que construeix.

Quan no troba un constructor de la classe, el compilador crida el constructor de la superclasse, que, si no n'hi ha, és de la classe Object().

Aquest altre constructor, tanmateix, rep dos paràmetres:

```
elmeuPunt( int paraX, int paraY ) {
```

```
        inicia( paraX, paraY );
}
```

La llista de paràmetres especificada després del nom d'una classe en una sentència new es fa servir per a passar paràmetres al constructor. Es crida el mètode constructor just després de crear la instància i abans que new retorni el control al punt de la crida.

L'operador new crea una instància d'una classe (un objecte) i torna una referència a aquest objecte. Per exemple:

```
elmeuPunt p2 = new elmeuPunt(2,3);
```

Aquest exemple crea una instància de l'objecte elmeuPunt() inicialitzada pels valors (2,3) i és referenciat per la variable *p2*.

Les referències a objectes realment no contenen els objectes als quals referencien, sinó l'adreça en què s'emmagatzema l'objecte; d'aquesta manera, es poden crear múltiples referències a un mateix objecte. Per exemple:

```
elmeuPunt p3 = p2;
```

Només s'ha creat un objecte elmeuPunt, però hi ha dues variables (*p2* i *p3*) que el referencien. Qualsevol canvi que es fa a l'objecte que referencia *p2* afecta l'objecte que referencia *p3*. L'assignació de *p2* a *p3* no reserva memòria ni modifica l'objecte.

L'assignació següent al valor null de *p2* te com a efecte que *p2* ja no apuntarà a l'objecte, però aquest últim encara existeix i a més *p3* encara hi fa referència:

```
p2 = null; // p3 encara apunta l'objecte creat amb new
```

Quan no hi ha cap variable que fa referència a un objecte, l'intèrpret de Java allibera automàticament la memòria que ha utilitzat l'objecte (aquest procés es coneix com a ***garbage collector***).

Quan es fa una instància d'una classe (mitjançant new), es reserva en la memòria un espai per al conjunt de dades que defineixen els atributs de la classe que s'indica en la instanciació. Aquest conjunt de variables es diu **variables d'instància**.

La potència de les variables d'instància es basa a crear un conjunt diferent de variables per a cadascun dels objectes nous creats, és a dir, cada objecte té la seva pròpia còpia de les variables d'instància de la seva classe, de manera que els canvis sobre les variables d'instància d'un objecte no tenen efecte sobre les variables d'instància d'un altre objecte.

Accés a l'objecte

L'operador punt (.) s'utilitza per a accedir a les propietats de la instància i als mètodes, mitjançant la seva referència a objecte:

```
referencia_a_objecte.nom_de_variable_d_instancia;
referencia_a_objecte.nom_de_metode( llista-de-parametres );
```

En l'exemple següent es pot veure l'ús de la sintaxi anterior:

```
elmeuPunt p3 = new elmeuPunt( 100, 200 );
p3.inicia( 300, 400 );
```

D'aquesta manera, es crea un objecte elmeuPunt amb les coordenades (100,200), però tot seguit es crida el mètode inicia que actualitza les coordenades anteriors a (300,400).

2.1.3. Herència

L'*herència* és un dels conceptes més importants en la programació orientada a objectes i consisteix en la possibilitat de crear classes a partir d'altres classes que ja hi ha. El que fa tan potent l'herència és que la classe nova pot heretar de la primera les propietats i els mètodes (apareixen així els conceptes *de classe pare* o *superclasse* i *classe filla* o *subclasse*).

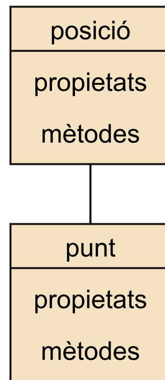
D'aquesta manera, una subclasse pot tenir incorporades les propietats i els mètodes heretats de la classe pare i, a més, pot afegir propietats i mètodes propis als heretats.

En el procés de fabricar models de vehicles s'utilitza d'una manera intensiva l'herència; per exemple, a partir d'una base de xassís un mateix fabricant construeix models diferents que comparteixen propietats i comportaments, amb la particularitat que s'afegeixen propietats i comportaments nous que els defineixen com un model nou, encara que, en realitat, una bona part de l'estructura interna és comuna a altres models i compartida per altres models.

L'herència és un mecanisme que s'utilitza per a definir semblança entre classes i simplifica definicions de classes semblants detingudes prèviament. Hi ha **dos tipus d'herència**:

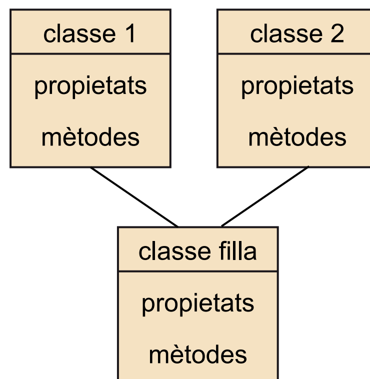
- L'**herència simple** es produeix quan el llenguatge només permet que una classe derivi d'una classe. En l'exemple següent es veu com la classe punt hereta de la classe posició:

Figura 2



- L'**herència múltiple** es produeix quan una classe es pot derivar de més d'una classe. L'exemple següent mostra com a partir de dues classes que defineixen objectes de so i imatge es pot crear una classe nova que hereti de les dues anteriors:

Figura 3



L'**herència** permet la reutilització del codi d'una manera molt simple pel programador.

Per a indicar que una classe hereta d'una altra, i que n'hereta tant les propietats com els mètodes, a Java s'usa el terme *extends*, com en l'exemple següent:

```
public class SubClasse extends SuperClasse {
        // Contingut de la classe
}
```

Per exemple, tot seguit es crea una classe elmeuPunt3D, que hereta de la classe elmeuPunt:

```
class elmeuPunt3D extends elmeuPunt {
        int z;
        elmeuPunt3D( ) {
              x = 0; // Heretat del elmeuPunt
```

```
                y = 0; // Heretat del elmeuPunt
                z = 0; // Nova propietat
        }
}
```

La paraula clau *extends* s'utilitza per a indicar que es crea una subclasse de la classe que és anomenada després; en l'exemple anterior, elmeuPunt3D és filla d'elmeuPunt.

La classe Object

La **classe Object** és la superclasse de totes les classes de Java. Totes les classes en deriven, directament o indirectament. Si quan es defineix una classe nova no hi ha la clàusula extends, Java considera que aquesta classe descendeix directament d'Object.

La classe Object aporta una sèrie de mètodes bàsics comuns a totes les classes:

- public boolean equals(Object obj): s'utilitza per a comparar, en valor, dos objectes. Torna true si l'objecte que rep per paràmetre és igual, en valor, que l'objecte des del qual es crida el mètode. Si es volen comparar dues referències a objecte es poden utilitzar els operadors de comparació (==) i (!=).
- public int hashCode(): torna un codi hash per a aquest objecte, per a poder-lo emmagatzemar en una Hashtable.
- protected Object clone() throws CloneNotSupportedException: torna una còpia d'aquest objecte.
- public final Class getClass(): torna l'objecte concret, de tipus Class, que representa la classe d'aquest objecte.
- protected void finalize() throws Trowable: fa accions durant la recuperació de memòria.

2.2. Característiques de la programació orientada a objectes

2.2.1. Abstracció

L'abstracció és un exercici mental pel qual a partir d'un objecte complex som capaços d'extreure'n les propietats i el comportament essencial deixant de banda els aspectes que no són rellevants.

Exemple

En descriure el cos humà, es refereix al cap, al braç (o braços), a la cama (o cames), etc.

Gràcies a l'abstracció es poden representar les característiques essencials d'un objecte sense preocupar-se de les altres característiques (no essencials). L'abstracció se centra en la vista externa d'un objecte, de manera que serveixi per a separar el comportament essencial d'un objecte de la implementació que té.

En els llenguatges de programació orientada a objectes, el concepte de Classe és la representació i el mecanisme pel qual es gestionen les abstraccions.

En POO, es pot considerar una Persona, per exemple, com un objecte que té propietats (nom, alçada, pes, color dels cabells, color dels ulls, etc.) i mètodes (parlar, mirar, caminar, córrer, parar, etc.).

Amb l'abstracció, un objecte Tren pot manipular objectes Persona sense tenir en compte ni les seves propietats ni els seus mètodes, ja que només li interessa, per exemple, calcular la quantitat de persones que hi viatgen en aquell moment, sense tenir present cap més informació relacionada amb aquestes persones, com l'alçada, el nom o el color dels ulls.

Hi ha situacions en què s'ha de definir una classe que representi un concepte abstracte i, per tant, no es pugui proporcionar una implementació completa d'alguns dels seus mètodes.

Quan creem una classe, podem declarar un determinat mètode com a abstract. Fent això forcem que qualsevol subclasse hagi d'implementar obligatòriament aquest mètode, o bé declarar-lo igualment abstact.

Qualsevol classe que contingui mètodes declarats abstract també s'ha de declarar abstract, i no es poden crear instàncies d'aquesta classe.

Tot seguit, es presenta un exemple de classes abstractes:

```
abstract class classeA {
       abstract void metodeAbstracte();
       void metodeConcret() {
             //El mètode concret de classeA;
       }
}
class classeB extends classeA {
       void metodeAbstract(){
             //El mètode abstracte de classeB;
       }
}
```

La classe abstracta classeA ha implementat el mètode concret metodeConcret(), però el mètode metodeAbstracte() era abstracte i per això s'ha hagut de redefinir en la classe filla classeB.

2.2.2. Ocultament o encapsulació

És la capacitat d'ocultar els detalls interns del comportament d'una classe i exposar públicament tan sols els detalls que són necessaris per a la resta del sistema.

L'ocultació permet dues característiques fonamentals:

- **Restringir l'ús de la classe:** hi ha cert comportament privat de la classe a què no poden accedir altres classes.

- **Controlar l'ús de la classe:** hi ha certs mecanismes per a modificar l'estat de la classe i és amb aquests mecanismes que es valida que algunes condicions es compleixin.

D'aquesta manera, cada objecte està aïllat de l'exterior i cada tipus d'objecte exposa una interfície a altres objectes que especifica com es pot interactuar. L'aïllament protegeix les propietats d'un objecte contra el fet que qui no té dret d'accedir-hi les modifiqui, i només són els mètodes interns de l'objecte els que poden accedir al seu estat.

Això assegura que no es pugui canviar l'estat intern d'un objecte de maneres inesperades, perquè elimina efectes secundaris i interaccions inesperades.

Si es continua l'exemple anterior de la classe Persona on es té una propietat que es diu DNI que conté el número de DNI. Es defineix un mètode NIF que retorna el valor del NIF. Dins de la classe hi haurà definit un mètode que a partir d'un número de DNI retorna la lletra del NIF. Ara bé, aquest mètode no és accessible, no es pot cridar directament per obtenir només la lletra. S'ha de cridar el mètode NIF (a més, l'operació que el crida tampoc coneix si és un mètode o una propietat) i obtenir directament el valor del NIF sencer.

En el llenguatge Java, totes les propietats i els mètodes d'una classe són accessibles des del codi de la mateixa classe, el control de l'accés des d'altres classes i de l'herència per les subclasses; els membres (atributs i mètodes) de les classes tenen **tres modificadors** possibles que defineixen el tipus de control d'accés:

- Public: les propietats o els mètodes declarats amb public són accessibles des de qualsevol lloc en què sigui accessible la classe, i els hereten les subclasses.

- Private: les propietats o els mètodes declarats private només són accessibles des de la classe.

- Protected: les propietats o els mètodes declarats protected només són accessibles per a les seves subclasses.

En cas que no s'especifiqui cap modificador, les propietats i mètodes són accessibles des de qualsevol mètode que pertany a la classe.

Per exemple:

Package

Un package en el llenguatge Java és un mecanisme d'agrupar classes similars en un tipus de biblioteca.

Una característica interessant és que no hi pot haver dues classes amb el mateix nom en un mateix package, però sí si pertanyen a packages diferents.

```
class Pare { // Hereta d'Object
	// Atributs
	private int numeroFavorit, nascutFa, dinersDisponibles;
	// Mètodes
	public int getAposta() {
		return numeroFavorit;
	}
	protected int getEdat() {
		return nascutFa;
	}
	private int getSaldo() {
		return dinersDisponibles;
	}
}
class Filla extends Pare {
	// Definició
}
class Visita {
	// Definició
}
```

En l'exemple anterior, un objecte de la classe Filla hereta els tres atributs (numeroFavorit, nascutFa i dinersDisponibles) i els tres mètodes (getAposta(), getEdat() i getSaldo()) de la classe Pare, i els pot cridar. Quan es crida el mètode getEdat() d'un objecte de la classe Filla, es torna el valor de la variable d'instància nascutFa d'aquest objecte, i no d'un de la classe Pare.

Tanmateix, un objecte de la classe Filla no pot cridar el mètode getSaldo() d'un objecte de la classe Pare, de manera que evita que el fill sàpiga l'estat del compte corrent d'un Pare.

La classe Visita només pot accedir al mètode getAposta() per esbrinar el número favorit d'un pare, però de cap manera no en pot saber ni el saldo ni l'edat.

2.2.3. Polimorfisme

La paraula *polimorfisme* ve del grec i significa 'que té formes diferents'. Aquest concepte és un dels conceptes essencials d'una programació orientada a objectes. De la mateixa manera que l'herència està relacionada amb les classes i la seva jerarquia, el polimorfisme ho està amb els mètodes.

El **polimorfisme** permet modificar el comportament d'un operador depenent dels operands.

En general, hi ha **tres tipus de polimorfisme:**

1) Polimorfisme de sobrecàrrega: ocorre quan funcions que tenen el mateix nom tenen una funcionalitat semblant, però en classes que són completament independents les unes de les altres (no han de ser classes filles de la classe objecte).

Exemple

La classe Punt, la classe Image i la classe Link poden tenir totes la funció "display". Això vol dir que no cal preocupar-se pel tipus d'objecte amb què es treballa si l'única cosa que es vol és visualitzar-lo a la pantalla.

D'aquesta manera, el polimorfisme de sobrecàrrega permet definir operadors el comportament dels quals varia d'acord amb els paràmetres que s'hi apliquin. Així es pot agregar l'operador + i fer que es comporti de manera diferent quan fa referència a una operació entre dos nombres enters (suma) o bé quan és entre dues cadenes de caràcters (concatenació).

2) Polimorfisme paramètric: és la capacitat per a definir diverses funcions utilitzant el mateix nom, però usant paràmetres amb diferents noms o tipus. D'aquesta manera se selecciona automàticament el mètode correcte que s'ha d'aplicar segons el tipus de dades passades en el paràmetre. Per tant, es poden definir diversos mètodes homònims de suma() fent una suma de valors, de manera que, si els paràmetres són numèrics, torni el valor de la suma d'aquests paràmetres i, si són cadenes de text, torni la concatenació de les cadenes.

3) Polimorfisme de subtipificació: es basa en l'habilitat de redefinir un mètode en classes que s'hereten d'una classe base. Per tant, es pot cridar un mètode d'objecte sense haver-ne de saber el tipus intrínsec; així es permet no tenir en compte detalls de les classes especialitzades d'una família d'objectes i emmascarar-los amb una interfície comuna (aquesta és la classe bàsica).

Exemple

Un exemple pot ser un joc d'escacs amb els objectes Rei, Reina, Alfil, Cavall, Torre i Peó, en què cadascun hereta de l'objecte Peça. El mètode Moviment podria fer, usant polimorfisme de subtipificació, el moviment corresponent d'acord amb la classe objecte que es crida.

Així, doncs, es tracta d'una característica que permet que una classe tingui diversos procediments amb el mateix nom, però amb diferent tipus o nombre d'arguments. D'aquesta manera, s'obtenen comportaments diferents associats a objectes diferents, però que comparteixen el mateix nom, i quan es criden per aquest nom s'utilitza el comportament corresponent a l'objecte que es fa servir.

Per exemple, una companyia paga els treballadors setmanalment. Aquests treballadors es classifiquen en quatre tipus: treballadors assalariats, que reben un salari setmanal fix, sense tenir en compte el nombre d'hores treballades; treballadors per hores, que reben un sou per hora i el pagament per temps extra

per totes les hores treballades que superin quaranta hores; treballadors per comissió, que reben un percentatge de les seves vendes, i treballadors assalariats per comissió, que reben un salari base i un percentatge de les seves vendes.

Per a aquest període de pagament, la companyia ha decidit recompensar els assalariats ocupats per comissió, i els agrega un 10% al salari base. En aquest exemple, es definiria el mètode Pagament utilitzant polimorfisme de subtipificació.

En el llenguatge de programació Java, el polimorfisme es pot implementar amb les tècniques següents:

1) Selecció dinàmica de mètode

Les dues classes implementades tot seguit tenen una relació subclasse/superclasse simple amb un sol mètode que se sobreescriu en la subclasse:

```
class classeA {
        void metodeDinamic() {
                // El mètode dinàmic de classeA;
        }
}
class classeB extends classeA {
        void metodeDinamic() {
                // El mètode dinàmic de classeB;
        }
}
```

Per tant, si s'executen les sentències següents:

```
classeA referenciaA = new classeB();
referenciaA.metodeDinamic();
```

s'executa el metodeDinamic definit en la classeB.

Es declara la variable de tipus classeA, i després s'hi emmagatzema una referència a una instància de la classe classeB. Quan es crida el mètode metodeDinamic() de classeA, el compilador de Java verifica que la classeA té un mètode anomenat metodeDinamic(), però l'intèrpret de Java observa que la referència és realment una instància de classeB, de manera que crida el mètode metodeDinamic() de classeB en comptes del de classeA.

Aquesta manera de polimorfisme dinàmic en temps d'execució és un dels mecanismes més poderosos que ofereix el disseny orientat a objectes per a suportar la reutilització del codi i la seva robustesa.

2) Sobreescriptura d'un mètode

Durant una jerarquia d'herència ens pot interessar tornar a escriure el cos d'un mètode, per a dur a terme una funcionalitat de manera diferent depenent del nivell d'abstracció en què siguem. Aquesta modificació de funcionalitat es diu *sobreescriptura d'un mètode*.

Per exemple, en una herència entre una classe EsserViu i una classe filla Persona, si la classe EsserViu té un mètode Alimentarse(), s'ha de tornar a escriure en el nivell Persona, ja que una persona no s'alimenta ni com un animal ni com una planta.

La millor manera de veure la diferència entre sobreescriptura i sobrecàrrega és amb un exemple. Tot seguit, es pot veure la implementació de la sobrecàrrega de la distància en 3D i la sobreescriptura de la distància en 2D.

```
class elmeuPunt3D extends elmeuPunt {
       int x,y,z;
       double distancia(int pX, int pY) { // Sobreescriptura
              int retorn=0;
              retorn += ((x/z)-pX)*((x/z)-pX);
              retorn += ((y/z)-pY)*((y/z)-pY);
              return Math.sqrt( retorn );
       }
}
```

S'inicien els objectes amb les sentències:

```
elmeuPunt p3 = new elmeuPunt(1,1);
elmeuPunt p4 = new elmeuPunt3D(2,2);
```

i es criden els mètodes de la manera següent:

```
p3.distancia(3,3); //Mètode elmeuPunt.distancia(pX,pY)
p4.distancia(4,4); //Mètode elmeuPunt3D.distancia(pX,pY)
```

Els mètodes se seleccionen segons el tipus de la instància en temps d'execució, no de la classe en què s'executa el mètode actual. Això es diu **selecció dinàmica de mètode**.

3) Sobrecàrrega de mètode

Pot ser que faci falta crear més d'un mètode amb el mateix nom, però amb llistes de paràmetres diferents. Això s'anomena *sobrecàrrega de mètode*. La sobrecàrrega de mètode s'utilitza per a proporcionar a Java un comportament polimòrfic.

Un exemple d'ús de la sobrecàrrega és crear constructors alternatius segons les coordenades, tal com es feia en la classe elmeuPunt:

```
elmeuPunt( ) { //Constructor per defecte
      inicia( -1, -1 );
}

elmeuPunt( int paramX, int paramY ) { // Parametritzat
      this.x = paramX;
      y = paramY;
}
```

Es crida el constructor basant-se en el nombre i tipus de paràmetres que s'hi passen. El nombre de paràmetres amb tipus d'una seqüència específica es diu **signatura de tipus**. Java utilitza aquestes signatures de tipus per a decidir quin mètode ha de cridar. Per a distingir entre dos mètodes, no es consideren els noms dels paràmetres formals sinó els seus tipus i/o el seu nombre.

```
elmeuPunt p1 = new elmeuPunt(); // Constructor per defecte
elmeuPunt p2 = new elmeuPunt( 5, 6 ); // Constructor parametritzat
```

2.2.4. Destrucció d'objectes

Un **destructor** és un mètode de la classe que fa la tasca oposada al seu constructor: allibera la memòria que es va assignar a l'objecte que va crear el constructor. Val la pena que es cridi implícitament el destructor quan l'objecte abandona el bloc en què es va declarar.

El destructor permet al programador despreocupar-se d'alliberar la memòria que deixa d'utilitzar i no haver de córrer el risc que aquesta memòria se saturi.

A Java, la destrucció es pot fer d'una manera automàtica o d'una manera personalitzada, segons les característiques de l'objecte.

La destrucció per defecte: recuperació de memòria

L'intèrpret de Java té un sistema de recuperació de memòria que, en general, permet que el programador no s'hagi de preocupar d'alliberar la memòria assignada explícitament.

El recuperador de memòria és l'encarregat d'alliberar una zona de memòria dinàmica que havia estat reservada mitjançant l'operador new, quan l'objecte ja no s'ha d'utilitzar durant el programa (per exemple, quan surt de l'àmbit d'utilització o no es torna a referenciar).

El sistema de recuperació de memòria s'executa periòdicament, i busca objectes que ja no estan referenciats.

La destrucció personalitzada: finalize

A vegades, una classe manté un recurs que no és de Java, com un descriptor d'arxiu o un tipus de lletra del sistema de finestres. En aquest cas, és recomanable utilitzar l'acabament explícit, per a assegurar-se que aquest recurs s'allibera. Per a especificar una destrucció personalitzada, s'afegeix un mètode a la classe amb el nom finalize:

```
class ClasseFinalitzada{
       ClasseFinalitzada() { // Constructor
             // Reserva del recurs no Java o recurs compartit
       }
       protected void finalize() {
             // Alliberament del recurs no Java o recurs compartit
       }
}
```

L'intèrpret de Java crida el mètode finalize() quan ha de destruir l'objecte.

2.2.5. Anàlisi i disseny orientat a objectes

Per a desenvolupar programari orientat a objectes no n'hi ha prou d'usar un llenguatge orientat a objectes, sinó que també cal fer una anàlisi i un disseny orientat a objectes.

La modelització visual és la clau per a dur a terme l'anàlisi orientada a objectes. Des del començament del desenvolupament de programari orientat a objectes hi ha hagut diferents metodologies per a implementar aquesta modelització, però, sens dubte, el llenguatge universal de modelització (UML) va posar fi a la guerra de metodologies.

Segons els mateixos dissenyadors del llenguatge UML, aquest llenguatge té la finalitat de modelar qualsevol tipus de sistemes (no solament de programari) usant els conceptes de l'orientació a objectes. A més, aquest llenguatge ha de ser entenedor tant per als programadors com per a les màquines.

Actualment, en la indústria del desenvolupament de programari, l'UML és un estàndard *de facto* en la modelització de sistemes. Va ser la companyia Rational qui va crear aquestes definicions i especificacions de l'estàndard UML, i més endavant ho va publicar en el mercat.

La mateixa empresa va crear un dels programes més populars per a aquesta finalitat: el Rational Rose. També hi ha altres programes, però, com el Poseidon, que disposa de llicències del tipus *community edition*, que permeten usar-lo lliurement.

L'UML consta de tots els elements i diagrames que permeten modelitzar els sistemes des del paradigma orientat a objectes. Quan es construeixen d'una manera correcta, els models orientats a objectes són fàcils de comunicar, canviar, expandir, validar i verificar.

Aquesta modelització en UML és flexible al canvi i permet crear components plenament reutilitzables.

3. POO en els diversos llenguatges de programació

L'objectiu d'aquest apartat és presentar els principals llenguatges de programació que han incorporat la POO en la seva implementació. Per a això, en cadascun d'aquests llenguatges se'n farà una revisió històrica breu i se'n presentaran les característiques principals.

3.1. Smalltalk

Smalltalk va ser desenvolupat a Xerox Parc (Palo Alto Research Center) amb l'impuls d'Alan Kay durant la dècada de 1970. Al començament, havia de ser un llenguatge per a un ordinador personal anomenat *Dynabook* adreçat a tota mena d'usuaris (inclosos els infants). Havia de ser, per tant, un sistema amb un entorn intuïtiu i fàcil de programar. Encara que el projecte Dynabook no es va completar mai, el llenguatge va adquirir vida pròpia i va continuar el seu camí.

Alan Kay (1940)

Informàtic nord-americà pioner en la programació orientada a objectes i el disseny de sistemes d'interfícies d'usuari. És professor adjunt de la Universitat de Califòrnia, a Los Angeles. Una de les seves frases més cèlebres és aquesta: "la millor manera de predir el futur és inventar-lo".

No és gaire coneguda la gran importància que va tenir aquest desenvolupament en l'evolució de la informàtica. En parteixen moltes de les idees que avui són la base de les interfícies d'usuari, com l'ús de gràfics, el ratolí, les finestres i els menús desplegables.

Smalltalk és un llenguatge orientat a objectes pur (el mateix terme, si no el concepte, el va inventar Alan Kay) i inclou tots els conceptes clau, com *classes*, *mètodes*, *missatges* i *herència*. Tot el programa és una cadena de missatges enviats a objectes.

Les característiques principals del llenguatge són les següents:

- Orientació a objectes pura.
- Tipus dinàmics.
- Herència simple.
- Compilació en temps d'execució o interpretat.

Smalltalk és un model pur orientat a objectes, que vol dir que, en l'entorn, tot es tracta com un objecte. Entre els llenguatges orientats a objectes, Smalltalk és el més consistent pel que fa al maneig de les definicions i propietats del paradigma orientat a objectes.

Es pot afirmar que és més que un llenguatge: és un entorn de desenvolupament amb més de dues-centes classes i diversos milers de mètodes. Smalltalk conté els components següents:

- Un llenguatge.

- Un model d'objecte, que defineix com actuen els objectes i implementa l'herència, el comportament de classes i instàncies, l'associació dinàmica, el maneig de missatges i les col·leccions.

- Un conjunt de classes reutilitzables, que disposa d'una bona quantitat de classes que es poden fer servir en qualsevol programa. Aquestes classes proveeixen les funcions bàsiques en el llenguatge, a més del suport per a la portabilitat a diferents plataformes, inclosa la portabilitat de les interfícies gràfiques d'usuari.

- Un conjunt d'eines de desenvolupament, que habiliten els programadors per mirar i modificar les classes que ja hi ha, i també per rebatejar, agregar i esborrar classes. També proveeixen de detecció d'errors, inclosa l'habilitat d'agregar parades en l'execució, veure els valors de les variables, modificar el valor de variables en execució i fer canvis en el codi en temps d'execució d'un programa.

- Un entorn en temps d'execució, que permet als usuaris posar fi al cicle "compilar-enllaçar-executar" d'un programa. Això permet als usuaris executar un programa a Smalltalk mentre es canvia el codi font, de manera que els canvis que s'han fet en el codi font són reflectits a l'acte en l'aplicació que s'executa.

Una de les característiques millors de Smalltalk és l'**alt grau de reutilització** del codi que té, ja que inclou un gran conjunt d'objectes que es poden utilitzar directament o modificar d'una manera senzilla per a satisfer la necessitat d'una aplicació en general.

Smalltalk **no té una notació explícita** per a descriure un programa sencer. Sí que es fa servir una sintaxi explícita per a definir certs elements d'un programa, com ara mètodes, però la manera com s'estructuren aquests elements dins d'un programa sencer és definida generalment per les múltiples implementacions.

La **sintaxi** de Smalltalk tendeix a ser minimalista, cosa que implica que hi ha un grup reduït de paraules reservades i declaracions en comparació de la majoria dels llenguatges populars. Smalltalk té un grup de cinc paraules reservades: *self*, *super*, *nil*, *true* i *false*.

Les **implementacions** utilitzen tècniques de recuperació de memòria per a detectar i reclamar espai en memòria associat amb objectes que ja no s'utilitzaran més en el sistema. La manera d'execució del recuperador de memòria és en *background*, és a dir, com un procés de baixa prioritat no interactiu, encara que en algunes implementacions es pot executar a demanda. La freqüència i les característiques de la recuperació depenen de la tècnica que fa servir la implementació.

Web recomanat

En el web de Smalltalk hi ha informació ampliada sobre el desenvolupament i les característiques d'aquest llenguatge.

3.2. Eiffel

És un llenguatge de programació que va escriure Bertrand Meyer. A diferència de Smalltalk, inclou un preprocessador que permet traduir el codi Eiffel al llenguatge C. És popular en el camp de l'enginyeria de programari, ja que permet l'encapsulació, el control d'accés i l'àmbit de les modificacions. Per les capacitats tècniques que té, és, presumiblement, el millor llenguatge orientat a objectes pur.

Bertrand Meyer (1950)

Va estudiar a l'Escola Politècnica de París, va obtenir un màster a la Universitat de Stanford i es va doctorar en Filosofia a la Universitat de Nancy. La seva principal via es basa en el fet que els llenguatges de programació han de ser simples, elegants i fàcils d'usar. Va ser el dissenyador inicial del llenguatge i del mètode Eiffel. Una de les seves frases més celebres és aquesta: "un element de programari no és correcte ni incorrecte per si mateix: és correcte si es comporta d'acord amb la seva especificació".

En aquest llenguatge els programes consisteixen a declarar col·leccions de classes que inclouen mètodes i en els quals s'associen els atributs. D'aquesta manera, el punt primordial d'un programa a Eiffel és la declaració de classes. Les classes i els atributs són accessibles a partir de la implementació d'un concepte anomenat *característica*, que és, alhora, una agrupació de dades i una manera típica de tractar-los.

A Eiffel, una declaració de classe pot incloure les llistes següents:

- Una llista de característiques exportables.
- Una llista de les classes antecessores: classes de què aquesta hereta.
- Una llista de declaracions de característiques.

Les **característiques** principals del llenguatge són aquestes:

- Es tracta d'un llenguatge orientat a objectes pur.

- És un llenguatge de programació orientat a dissenyar grans aplicacions. Les propietats anteriors fan que vagi molt bé per a dissenyar aplicacions en grups de treball.

- El pas intermedi a codi C es pot considerar un avantatge i no un inconvenient, ja que les seccions que són difícils de tractar amb Eiffel es poden elaborar a partir de codi C. La compatibilitat amb C assegura també la portabilitat cap a altres sistemes operatius.

- El maneig de la memòria, un punt delicat en tots els llenguatges orientats a objectes, no és transparent com en el cas de Smalltalk.

- Les biblioteques de classes són reduïdes.

Web recomanat

En el web de la companyia Eiffel Software, en què s'ofereix un entorn de desenvolupament amb la llicència GPL, hi ha informació ampliada sobre el desenvolupament i les característiques d'aquest llenguatge.

- El rendiment és més gran que el de Smalltalk, però, davant de la necessitat d'incloure un mòdul Run-time dins de l'executable, la mida creix i el rendiment baixa.

3.3. C++

És un llenguatge de programació que va dissenyar a mitjan dècada de 1980 Bjarne Stroustrup, de manera que amplia el llenguatge de programació C amb mecanismes que permeten manipular objectes. Per aquest motiu, el llenguatge C++ es considera un llenguatge híbrid.

Bjarne Stroustrup (1950)

És un científic de la computació i catedràtic de Ciències de la Computació a la Universitat A&M de Texas. Ha destacat per haver desenvolupat el llenguatge de programació C++ i haver escrit el manual de referència del llenguatge, *The C++ Programming Language*.

Més endavant, s'hi van afegir facilitats de programació genèrica, que es va sumar als altres dos paradigmes que ja eren admesos (programació estructurada i programació orientada a objectes). Per això, se sol dir que el C++ és un **llenguatge multiparadigma**. Actualment, hi ha un estàndard, anomenat *ISO C++*, a què s'han adherit la majoria dels fabricants de compiladors més moderns.

La contribució més important que fa C++ a C és que introdueix el tipus classe, ja que les classes permeten definir conjunts de propietats i els mètodes que les manipulen.

C++ disposa de **tres tipus de mètodes constructors** que s'executen quan una instància d'una classe es crea amb l'objectiu d'inicialitzar o definir l'estat de l'objecte:

- **Constructor predeterminat**. És el constructor que no rep cap paràmetre en la funció. Si no es defineix cap constructor, el sistema en proporciona un de predeterminat.

- **Constructor de còpia**. És un constructor que rep un objecte de la mateixa classe i fa una còpia dels atributs que té. Igual que el predeterminat, si no es defineix, el sistema en proporciona un.

- **Constructor de conversió**. Aquest constructor rep com a únic paràmetre un objecte o una variable d'un altre tipus diferent que el seu. És a dir, converteix un objecte d'un tipus determinat en un altre objecte del tipus que genera.

De la mateixa manera que fan falta mètodes constructors, en fan falta de destructors, que tan sols són funcions membres especials que tenen la comesa d'alliberar els recursos que ha adquirit l'objecte d'aquesta classe en temps d'execució quan ha expirat aquest temps. Els destructors són cridats automàticament quan el flux del programa assoleix l'objectiu de l'àmbit en què és declarat l'objecte.

Hi ha **dos tipus de destructors:**

- **Públics:** es poden cridar des de qualsevol part del programa.
- **Privats:** quan no es permet que l'usuari destrucció de l'objecte per part de l'usuari.

En C++ es poden definir classes abstractes, que estan dissenyades només com a classe pare, de les quals han de derivar classes filla. Una classe abstracta s'usa per a representar les entitats o els mètodes que després s'implementen en les classes derivades, però la classe abstracta per si mateixa no conté cap codi específic sinó que només representa els mètodes que s'han d'implementar. Per això, no es pot instanciar una classe abstracta, però sí una classe concreta que implementi els mètodes que s'hi han definit.

En C++ hi ha **tres maneres d'herència simple** que es diferencien en la manera de manejar la visibilitat dels components de la classe resultant:

- **Herència pública:** amb aquest tipus d'herència es respecten els comportaments originals de les visibilitats de la classe pare en la classe filla.

- **Herència privada:** amb aquest tipus d'herència tot component de la classe pare és privat en la classe filla.

- **Herència protegida**: amb aquest tipus d'herència tot component públic i protegit de la classe base és protegit en la classe derivada, i els components privats continuen essent privats.

La sobrecàrrega d'operadors en C++ és una manera d'implementar polimorfisme; d'aquesta manera, es pot definir el comportament d'un operador del llenguatge perquè treballi amb tipus de dades que ha definit l'usuari. No tots els operadors de C++ són factibles en el cas de sobrecarregar-los, i, entre els que es poden sobrecarregar, s'han de complir certes condicions.

Lectura recomanada

Bruce Eckel és autor de llibres i articles sobre programació. Les seves obres més conegudes són *Thinking in Java* i *Thinking in C++*, adreçades a programadors amb poca experiència en la programació orientada a objectes.

Hi ha un web oficial en què es porta a cap la traducció al castellà del llibre de Bruce Eckel *Thinking in C++*.

3.4. ActionScript 3.0

ActionScript es va desenvolupar, de bon principi, amb la finalitat d'agregar interactivitat al format d'animació vectorial Flash. A partir d'aquell moment, els dissenyadors Flash es van haver de convertir en programadors per a afegir totes les possibilitats que podia proporcionar el llenguatge.

ActionScript és un llenguatge de programació orientat a objectes que es fa servir en les aplicacions web animades que es creen en l'entorn Flash. El llenguatge es va introduir a partir de la versió 4 de Flash i des de llavors ha evolucionat en cadascuna de les versions noves.

Es tracta de llenguatge script basat en especificacions estàndard d'indústria ECMA-262, un estàndard per a JavaScript, i per això ActionScript s'assembla tant a JavaScript. La versió més estesa actualment és ActionScript 3.0, que va significar una millora en el maneig de programació orientada a objectes perquè s'ajustava més bé a l'estàndard ECMA-262.

Indiquem tot seguit les **característiques** principals del llenguatge:

- Disposa d'una màquina virtual que és l'encarregada d'interpretar el codi, independentment de la plataforma en què s'executa aquest codi.
- Té una sintaxi del llenguatge que s'ajusta a l'estàndard ECMAScript (ECMA-262).
- Disposa d'una interfície de programació que permet un control de baix nivell dels objectes que componen les pel·lícules Flash.
- Té una API XML basada en l'especificació d'ECMAScript per a XML (E4X) (ECMA-357, edició 2). E4X és una extensió del llenguatge ECMAScript que afegeix l'XML com un tipus de dades natiu del llenguatge.
- Disposa d'un model d'esdeveniments basat en l'especificació d'esdeveniments DOM (model d'objectes de document) de nivell 3.

Per tant, es tracta d'un llenguatge de programació encastat a les pel·lícules creades amb Flash, però que compleix els estàndards ECMAScript, com JavaScript, de manera que tots dos llenguatges comparteixen una bona part de la sintaxi.

Web recomanat

Hi ha manuals, programes d'aprenentatge, articles, etc. en el web de la comunitat de programadors d'ActionScript: www.actionscript.org.

3.5. Ada

Durant la dècada de 1970, el Departament de Defensa dels Estats Units tenia projectes que es desenvolupaven en un conjunt variat de llenguatges de programació. Aquesta varietat comportava un cert problema, i la solució es va basar a cercar un sol llenguatge que complís certes normes obligatòries. Es va fer un concurs i, de les diverses propostes que hi va haver, el maig de 1979 es va seleccionar la de Honeywell Bull, que es va anomenar *Ada*.

Origen d'Ada

El nom es va escollir en commemoració d'Ada Augusta Byron (1815-1852), comtessa de Lovelace, filla del poeta George Byron, a la qual es considera la primera programadora de la història, per la seva col·laboració i relació amb Charles Babbage, creador de la màquina analítica.

El Departament de Defensa i els ministeris equivalents de diversos països de l'OTAN exigien l'ús d'aquest llenguatge en els projectes que contractaven (aquesta obligació es deia *Ada mandate*). L'obligatorietat, en el cas dels Estats Units, es va acabar el 1997.

La sintaxi del llenguatge s'inspira en Pascal, de manera que el poden llegir fins i tot programadors que no coneguin aquest llenguatge. Es tracta d'un llenguatge que no escatima la longitud de les paraules clau, ja que un dels principis que té és que un programa s'escriu una vegada, es modifica desenes de vegades i es llegeix milers de vegades (la llegibilitat és més important que la rapidesa d'escriptura).

Es va dissenyar amb el propòsit principal de generar programes de màxima qualitat, amb l'objectiu d'aconseguir la confiança dels usuaris. S'hi pot implementar qualsevol tipus de programari, però l'ús principal que se n'ha fet és en programari de control en temps real i de missió crítica.

D'altra banda, Ada, com a llenguatge que promou les bones pràctiques en enginyeria del programari, es fa servir molt en l'ensenyament de la programació a moltes universitats d'arreu del món.

Es tracta d'un llenguatge de programació imperatiu, orientat a objectes, concurrent i distribuït. Les **característiques** principals són aquestes:

- Té una sintaxi inspirada en Pascal que es llegeix fàcilment.
- És un llenguatge *case insensitive*, és a dir, els identificadors i les paraules clau són equivalents amb independència de l'ús de majúscules i minúscules.
- És un llenguatge amb tipificació forta: assigna a cada objecte un conjunt de valors clarament definit, cosa que impedeix la confusió entre conceptes lògicament diferents. Això fa que el compilador detecti més errors que en altres llenguatges.
- Està preparat per a construir grans programes. Per a crear programes sostenibles i transportables, de qualsevol mida, fan falta mecanismes d'encapsulació per a compilar separadament i per a gestionar biblioteques.
- Disposa de mecanismes que permeten manejar excepcions. D'aquesta manera, els programes es construeixen per capes i es limiten les conseqüències dels errors en qualsevol de les parts.
- Se separen, amb l'abstracció de dades, els detalls de la representació de les dades i les especificacions de les operacions lògiques sobre aquestes dades per a obtenir més portabilitat i més bon manteniment.
- Disposa de la capacitat de processament paral·lel i, així, evita que s'hagin d'afegir aquests mecanismes per mitjà de crides al sistema operatiu, de manera que aconsegueix més fiabilitat i portabilitat.

- Disposa de l'opció de crear unitats genèriques; aquestes unitats són necessàries, ja que una part d'un programa pot ser independent del tipus de valors que s'han de manipular. Per a això, cal que s'utilitzi aquest mecanisme que permet crear parts d'un programa semblants a partir d'una plantilla.

Web recomanat

A GRB ADA95 hi ha una guia bàsica d'aprenentatge del llenguatge de programació: www.gedlc.ulpgc.es/docencia/NGA/index.html.

3.6. Perl

Larry Wall (1954)

Programador, lingüista i autor, Larry Wall és conegut pel fet de ser el creador del llenguatge de programació Perl. El 1998 va rebre el primer premi de la Fundació del Programari Lliure per a l'avenç del programari lliure. És el coautor del llibre *Programming Perl* (comunament anomenat *llibre del camell*), que és el recurs bàsic dels programadors de Perl.

El creador de Perl, Larry Wall, va anunciar la versió 1.0 del llenguatge el 18 de desembre de 1987. Durant els anys següents, aquest llenguatge es va expandir d'una manera molt ràpida, encara que fins a 1991 l'única documentació de Perl era una simple (i cada vegada més llarga) pàgina de manual Unix.

El 26 d'octubre de 1995, es va crear la Xarxa d'Arxius Perl Completa (CPAN). La CPAN és una col·lecció de llocs web que emmagatzemen i distribueixen fonts en Perl, binaris, documentació, scripts i mòduls. Al començament, s'havia d'accedir a cada lloc CPAN amb el seu propi URL, però, actualment, www.cpan.org els encamina automàticament a un dels centenars de repositoris mirall de la CPAN.

Perl és un llenguatge de propòsit general que al començament es va desenvolupar per a manipular text i que ara es fa servir per a un rang ampli de tasques que inclouen administrar sistemes, desenvolupar webs, programar en xarxa i desenvolupar GUI. Es va preveure que fos pràctic (facilitat d'ús, eficient, complet) en comptes de bonic (petit, elegant, mínim).

Les característiques principals que té són que és fàcil d'usar, que suporta tant la programació estructurada com la programació orientada a objectes i la programació funcional i que incorpora un sistema poderós de processament de text i una col·lecció enorme de mòduls disponibles.

L'estructura completa de Perl deriva àmpliament del llenguatge C; per tant, és un llenguatge imperatiu, amb variables, expressions, assignacions, blocs de codi delimitats per claus, estructures de control i subrutines.

Perl també agafa característiques de la programació Shell, de manera que disposa de moltes funcions integrades per a tasques comunes i per a accedir als recursos del sistema.

En la versió 5 de Perl es van afegir característiques per a suportar estructures de dades complexes, funcions de primer ordre (per exemple, clausures com a valors) i un model de programació orientada a objectes. Aquests objectes inclouen referències, paquets i una execució de mètodes basada en classes i la introducció de variables d'àmbit lèxic, que va fer més fàcil escriure codi robust. Una característica important introduïda en Perl 5 va ser l'habilitat d'empaquetar codi reutilitzable en estructures de mòduls.

Totes les versions de Perl implementen tipificació automàtica de dades i gestió de memòria. D'aquesta manera, l'intèrpret sap el tipus i els requeriments d'emmagatzemament de cada objecte en el programa, i hi reserva i allibera espai quan fa falta. Les conversions legals de tipus es fan de manera automàtica en temps d'execució; les conversions il·legals es consideren errors fatals.

S'ha usat des dels primers dies del Web per a escriure guions (scripts) CGI. És una de les "tres *P*" (Perl, Python i PHP), que són els llenguatges més populars per a crear aplicacions web, i és un component integral de la popular solució LAMP per al desenvolupament web.

Alguns dels projectes importants que s'han escrit en Perl són Slash, IMDb i UseModWiki; a més, llocs web amb un nivell alt de trànsit, com Amazon.com i Ticketmaster.com, utilitzen aquest llenguatge. També es fa servir molt en finances i bioinformàtica, en què és apreciat pel desenvolupament ràpid que té, tant d'aplicacions com de desplegament, i també per l'habilitat de manejar grans volums de dades.

Lectura recomanada

Teniu en línia un PDF que introdueix el programador en el model de programació orientada a objectes del llenguatge Perl: www.gwolf.org/files/poo_perl.pdf.

3.7. PHP

Al començament, PHP es va dissenyar en Perl, basant-se en l'escriptura d'un grup de CGI binaris que va escriure en el llenguatge C el programador danès-canadenc Rasmus Lerdorf l'any 1994 per mostrar el seu currículum i guardar certes dades, com la quantitat de trànsit que rebia el seu lloc web. El 8 de juny de 1995 es va publicar "Personal Home Page Tools" després que Lerdorf el va combinar amb el seu propi Form Interpreter per crear PHP/FI.

Rasmus Lerdorf (1968)

Programador nascut a Groenlàndia a qui es considera un dels creadors de PHP més importants. El 1995, Lerdorf va crear un CGI en Perl que mostrava el nombre de visites que havia tingut el seu lloc web; aquest script el va anomenar *PHP* (pàgina d'inici personal) i va ser el detonador del llenguatge script nou. Va crear una llista de correu per a intercanviar opinions, suggeriments i correccions que va provocar la base que va acabar formalitzant PHP com una eina de programari lliure en què l'aportació de la comunitat mundial ha fet que sigui un dels llenguatges de programació web més utilitzats.

PHP és un llenguatge de programació que s'ha dissenyat especialment per a desenvolupar webs i que també es pot incrustar als mateixos llocs de la Xarxa; d'altra banda, l'interpreta el servidor, que normalment torna com a resultat un lloc web nou. La sintaxi de PHP és semblant a la dels llenguatges Perl i C, i això fa que la corba d'aprenentatge del llenguatge sigui molt curta.

Quan el client fa una petició al servidor perquè hi enviï un lloc web, el servidor executa l'intèrpret de PHP. Aquest intèrpret processa l'script i genera el contingut d'una manera dinàmica (per exemple, obtenint informació d'una base de dades). El resultat l'envia l'intèrpret al servidor, que, al seu torn, l'envia al client. Mitjançant extensions també es poden generar arxius PDF, Flash i imatges en diferents formats.

PHP és portable i, per tant, es pot executar en la majoria dels sistemes operatius: Unix, Linux, Mac OS X i Windows.

Tot seguit presentem les **característiques** principals del llenguatge:

- Es tracta d'un llenguatge multiplataforma.
- Està completament orientat al Web.
- Té capacitat de connexió amb la majoria dels motors de base de dades que s'utilitzen en l'actualitat; en aquest sentit, en destaca la connectivitat amb MySQL i PostgreSQL.
- Té capacitat d'expandir el seu potencial utilitzant la quantitat enorme de mòduls de què disposa.
- És programari lliure, de manera que es presenta com una alternativa fàcil d'accedir-hi.
- Implementa el paradigma de la Programació Orientada a Objectes.
- Disposa d'una biblioteca nativa de funcions summament àmplia i inclosa.
- No requereix definició de tipus de variables, encara que les variables que té es poden avaluar també pel tipus que manegen en temps d'execució.
- Implementa la capacitat de manejar excepcions.
- L'ofuscació de codi és l'única manera d'ocultar les fonts.

Web recomanat

En el web oficial de PHP trobareu més informació sobre aquest llenguatge: www.php.net/.

3.8. C#

Els primers rumors que Microsoft desenvolupava un llenguatge de programació nou van sorgir el 1998, amb referència a un llenguatge que llavors anomenaven *COOL* i que deien que era molt semblant a Java. El juny de 2000, Microsoft va aclarir tots els dubtes i va alliberar l'especificació d'un llenguatge nou anomenat *C#*. De seguida, en va seguir la primera versió de prova de l'entorn de desenvolupament estàndard .NET, que incloïa un compilador de C#.

C# és un llenguatge de programació orientat a objectes que ha desenvolupat i estandarditzat Microsoft com a part de la seva plataforma .NET. La sintaxi bàsica deriva de C/C++ i utilitza el model d'objectes de la plataforma .NET, que és semblant al de Java, encara que inclou millores derivades d'altres llenguatges.

C#, com a part de la plataforma .NET, és normalitzat per ECMA des del desembre de 2001 (ECMA-334 "Especificació del llenguatge C#").

El 7 de novembre de 2005 es va publicar la versió 2.0 del llenguatge, que incloïa millores com els tipus genèrics, els mètodes anònims, els iteradors, els tipus parcials i els tipus anul·lables. El 19 de novembre de 2007 es va publicar la versió 3.0 de C#, entre les millores de la qual destacaven els tipus implícits, els tipus anònims i el LINQ (consulta integrada en el llenguatge).

Encara que C# forma part de la plataforma .NET (aquesta plataforma és una interfície de programació d'aplicacions), es tracta d'un llenguatge de programació independent que es va dissenyar per a generar programes sobre aquesta plataforma.

Microsoft

.NET és la proposta de Microsoft a la programació en entorns web i neix amb l'objectiu de competir amb la plataforma Java. Com la majoria de productes de Microsoft, la seva gran basa és que proporciona una manera ràpida i econòmica de desenvolupar aplicacions.

Actualment, hi ha un compilador GNU de C# (Mono) que genera programes per a diferents plataformes com Win32, Unix i Linux.

Web recomanat

Trobareu més informació sobre el llenguatge C# en la pàgina Centre de desenvolupadors en C#, que dirigeix Microsoft: msdn.microsoft.com/es-es/vcsharp/default.aspx.

3.9. Java

El començament de Java es remunta a l'any 1991, quan un grup d'enginyers que dirigien Patrick Naughton i James Gosling volia dissenyar un llenguatge petit de programació que es pogués fer servir en dispositius de consum com els equips de televisió per cable (aquest projecte es va dir Green Project); com que aquests dispositius no tenen una capacitat de memòria gran, el llenguatge havia de ser simple i generar codi molt reduït.

James Gosling (1956)

Doctorat per la Universitat de Carnegie Mellon i conegut com el creador del llenguatge de programació Java, va fer el disseny original, la implementació del compilador original i la màquina virtual Java, per la qual cosa va ser elegit membre de l'Acadèmia Nacional d'Enginyeria dels Estats Units (NAE).

D'altra banda, els fabricants d'aquest tipus de dispositius electrònics solen canviar els xips sovint. L'aparició d'un xip nou més barat i, generalment, més eficient condueix aquests fabricants a incloure'l en les sèries noves de les seves cadenes de producció, ja que aquesta diferència de preu, per petita que sigui, pot generar un estalvi considerable en dispositius de tirada massiva.

Si s'usaven llenguatges com C o C++, havien de compilar tots els programes amb el compilador d'aquest xip nou i això encaria els desenvolupaments. Per tant, com que els fabricants podien escollir diferents CPU, era important no estar lligat a una sola arquitectura: era molt important la portabilitat. Si s'aconseguia un llenguatge que produís un codi independent de la CPU, s'evitava haver de compilar tots els programes que hi havia (per a diferents

aparells electrònics) quan aparegués una CPU nova. Simplement, haurien de tenir un intèrpret d'aquest codi per a la CPU nova (que es podia donar als fabricants si el llenguatge es popularitzava prou).

L'abril de 1991, Gosling va començar a treballar en el llenguatge nou de Green Project, i va decidir que els avantatges que aportava l'eficiència de C++ no compensaven els grans costos de proves i depuració del codi. Com va dir Gosling, "el llenguatge era una eina, no la finalitat", de manera que va desenvolupar un llenguatge de programació que, fins i tot partint de la sintaxi de C++, mirava de remeiar els aspectes de C++, que eren la causa de la majoria dels problemes.

Gosling va anomenar el seu llenguatge *Oak* (se suposa que es referia a un roure que hi havia davant de la finestra del seu lloc de treball, a Sun). Els primers programes en Oak es van executar l'agost de 1991. Més endavant, a Sun, es van adonar que ja hi havia un llenguatge anomenat *Oak* i el van rebatejar *Java* (en anglès nord-americà significa 'cafè'; l'equip que desenvolupava aquest llenguatge es reunia en una cafeteria a la vora de les instal·lacions de Sun per discutir d'una manera distesa el projecte).

El 1992, el Green Project va comercialitzar el seu primer producte, anomenat **7* (Star Seven). Era una mena de barreja entre PDA i control remot extremament intel·ligent, que es va dissenyar per a fer un control integrat d'una casa amb tots els aparells electrònics que hi ha. El sistema presentava una interfície basada en la representació de la casa i el control es duia a terme amb una pantalla tàctil.

En el sistema apareixia Duke, la mascota actual de Java. Malauradament, no hi va haver gaire interès en aquest sistema i en aquell moment l'equip de Green Project es va embarcar en un concurs que va convocar la Time Warner per a dissenyar un equip per a la televisió per cable que fos capaç d'ocupar-se de serveis nous de cable com el vídeo per encàrrec. És a dir, s'aplicava Java a la interfície de la televisió interactiva.

Cap d'aquests dos projectes no es va convertir en un sistema comercial, però es van desenvolupar enterament en un Java primitiu i li van servir de baptisme de foc.

A mitjan 1994, la popularitat del Web va cridar l'atenció dels directius de Sun. Bill Joy, cofundador de Sun i un dels desenvolupadors principals de Unix de Berkeley, va considerar que Internet podria arribar a ser el camp de joc adequat per a disputar a Microsoft la supremacia gairebé absoluta en el terreny del programari, i va veure en Oak/Java l'instrument idoni per a dur a terme aquests plans. Es va adonar que els requisits per al programari dels dispositius electrònics i els equips de televisió (*set top boxes*) eren els mateixos que per al Web (codi senzill, independent de plataforma, segur i fiable).

Van decidir programar un navegador fent servir la tecnologia Java. Aquell primer programa, anomenat *WebRunner*, va estar a punt el maig de 1995 (Patrick Naughton va escriure un prototip d'aquest navegador en un cap de setmana d'inspiració) i, quan van veure les possibilitats enormes que tenia, van decidir millorar el navegador. El 23 de maig de 1995, en la Sun World 95 de San Francisco, Sun presenta el navegador nou HotJava i Netscape anuncia que té la intenció d'integrar Java al seu navegador.

A partir d'aquell estiu, els esdeveniments es desenvolupen vertiginosament per al món Java, sobretot després de la comercialització i distribució lliures de Java Development Kit (JDK) 1.0. Al final de 1995 (al cap de només sis mesos de la comercialització de JDK!), Java havia firmat acords amb les firmes de programari principals perquè poguessin utilitzar Java en els seus productes –entre d'altres, Netscape, Borland, Mitsubishi Electronics, Dimension X, Adobe, IBM, Lotus, Macromedia, Oracle i SpyGlass–, però la cosa més espectacular va ser l'anunci que va fer Bill Gates, president i director executiu de Microsoft, el 7 de desembre de 1995, amb referència a la voluntat de Microsoft d'obtenir la llicència d'ús de Java.

Aquest anunci mostrava clarament que Microsoft considerava Java com una part important en l'estratègia global d'Internet. És un anunci significatiu si es té en compte el menyspreu que feia uns mesos havia mostrat Bill Gates envers Java, quan s'hi va referir com "un llenguatge més". El mateix director general de Microsoft a Espanya havia qualificat Java de "llenguatge per a torradores".

Durant el 1996, es va plantejar el debat de crear un "terminal ximple" anomenat *NC* (Network Computer) que únicament servís per a connectar-se al World Wide Web. Al començament, es va projectar que aquest terminal seria governat per un sistema operatiu Java. La idea de l'NC no va quallar com s'esperava i en va aparèixer una altra d'intermèdia entre l'NC i el PC, la del NetPC, que tampoc no va tenir massa èxit. Tanmateix, la idea de produir un sistema operatiu sí que s'ha desenvolupat: es diu *JavaOS*.

El lema de Java és aquest: "escriu una vegada i executa'l a qualsevol lloc". La idea principal que transmet aquest lema és la portabilitat de Java: una vegada escrit el codi font i traduït a *bytecode*, es pot executar a qualsevol màquina amb qualsevol sistema operatiu (fins i tot encara que no sigui un ordinador), sense haver-lo de recompilar.

Per a aconseguir la portabilitat, el codi font Java "es compila" per a una màquina fictícia que s'anomena *màquina virtual Java* (JVM o *Java virtual machine*), cosa que genera un codi anomenat *codi d'octets* o *bytecode*. El codi font també és portable, però fer servir aquest codi intermedi anomenat *codi d'octets* té diversos avantatges:

- Permet un rendiment més bo, ja que una bona part del procés de traducció del codi font a les instruccions d'una CPU específica ja està fet.

- Permet mantenir en secret el codi font original, cosa que pot ser important en cert tipus de programes en què el que es vol és que siguin portables i alhora difícils de manipular, tafanejar, copiar, etc. També se'n pot extrapolar una altra idea important: a qualsevol lloc (navegadors) es pot executar codi Java (miniaplicacions) sense haver de pensar d'on ve, ja que no hi ha gaires perills de seguretat (com ara virus o programes que atemptin contra la privacitat).

Bàsicament, la idea que proposa és que l'usuari només necessita tenir al costat seu un mer element d'interacció (a vegades anomenat *terminal ximple*) sense massa potència de processament o capacitat d'emmagatzemament (per exemple, sense disc dur) i que el veritable ordinador (elements de computació i emmagatzemament) pot estar distribuït en una xarxa.

De bon principi Java es va dissenyar com un llenguatge orientat a objectes. Els objectes agrupen en estructures encapsulades tant les dades com els mètodes (o funcions) que manipulen aquestes dades. La tendència del futur, a la qual Java s'afegeix, apunta cap a la programació orientada a objectes, especialment en entorns cada vegada més complexos i basats en una xarxa.

Web recomanat

En el web oficial Sun hi ha més informació sobre el llenguatge de programació Java: de java.sun.com/.

3.10. JavaScript

JavaScript és un llenguatge que es va introduir en la versió 2.0 del Netscape Navigator, al començament de 1996, i que Microsoft va acceptar més endavant perquè el seu Internet Explorer guanyés quota de mercat, encara que totes dues versions tenen característiques incompatibles.

Vegeu també

Les característiques de l'orientació a objectes en JavaScript s'estudien en detall en el mòdul "Orientació a objectes en JavaScript".