As Linguaxes e Os Paradigmas de Programación

Introdución

As linguaxes de programación nos permiten indicar as instrucións que o ordenador debe levar a cabo dun xeito facilmente comprensible, en contraposición a introducir directamente os códigos binarios das operacións que desexamos realizar.

Existen numerosos tipos de linguaxes de programación con aproximacións e orientacións diferentes, veremos algúns de eles.

A Linguaxe Ensamblador

Nun principio, para evitar ter que introducir directamente o código binario das instrucións dos programas, se deseñaron uns códigos mnemotécnicos para recordalas facilmente. As CPU dos ordenadores son capaces de levar a cabo un conxunto de instrucións moi limitados, ás que se asignou un código alfabético. Deste xeito, por exemplo **MOV AX,23** indicaría "mover o número 23 ao rexistro da CPU AX", **ADD BX,3** "sumar a 3 ao contido do rexistro BX almacenando o resultado en BX" ou **CMP AX,5** , **JZ AB567F** "comparar o contido do rexistro AX con 5 e si é igual pasar a executar o contido da dirección de memoria AB567F".

Para poder executar os programas escritos de este modo, e necesario transformalos a código binario mediante un programa denominado *ensamblador,* e a este tipo de linguaxe na que cada instrución corresponde cunha instrución propia da CPU se lle denomina ***código ensamblador***.

Como escribir programas deste xeito era moi longo e propenso a erros, creáronse linguaxes que permitían o uso de estruturas máis complexas e próximas á linguaxe humana, coñecidas coma *linguaxes de alto nivel*.

Para poder executar os programas escritos con linguaxes de alto nivel existen dúas aproximacións diferentes: As *linguaxes compiladas* e as *linguaxes interpretadas*.

Linguaxes Interpretadas

Algúns linguaxes de alto nivel dispoñen dun programa que se encarga de ir traducindo cada bloque de instrucións a medida que se van executando, interpretando o código escrito sobre a marcha para xerar o código máquina correspondente. Este tipo de aproximación se coñece como linguaxe interpretada.

Para a execución dun programa nunha linguaxe interpretada unicamente necesitaremos o intérprete apropiado para esta linguaxe

Son exemplos típicos de linguaxes interpretada, as linguaxes de Shell como ***PowerShell*** ou ***bash***, e as linguaxes de "scripting" como ***JavaScript***, ***Python***, **PHP** ou ***Perl***.

As principais vantaxes das linguaxes interpretadas son:

- Notificación inmediata dos erros sintácticos e facilidade de modificación.
- Facilidade ***multiplataforma***, xa que unicamente necesitamos un intérprete apropiado.
- Facilidade de modificación, xa que unicamente é necesario cambiar o código fonte.

En canto aos inconvenientes, principalmente podemos destacar unha menor velocidade xa que é preciso interpretar as ordes para ser executadas.

Linguaxes Compiladas

Outra aproximación para a execución dos programas escritos en linguaxes de alto nivel e a súa tradución ao código binario propia da CPU e sistema onde se quere executar o programa, xerando o que se coñece como ***código obxecto***. Este proceso se denomina ***compilado*** e é realizado por un ***compilador*** propio da linguaxe empregada.

Este código obxecto precisa dun bloque coas instrucións de carga para o sistema operativo, e mais módulos xa existentes que se queiran agregar ao noso programa. Ademais, o programa pode estar composto por varios módulos compilados de xeito independente. Para agregar todo este código en un único ficheiro denominado ***código executable*** é necesario un programa denominado ***enlazador*** ou ***linker***, e o proceso se coñece como *enlazado*.

En resumo, o proceso cando se utiliza unha linguaxe compilada é :

1. Creación do *código fonte* cun editor de textos
2. Compilación do código fonte a *código obxecto*
3. Enlazado do programa par producir o *código executable*

O código executable producido é específico da plataforma para a que se compila, e polo tanto non é executable en plataformas distintas, a diferencia das linguaxes interpretadas, nas que basta con dispoñer dun intérprete que se encargará de interpretar e executar o código fonte. Por outra banda, cada corrección no código fonte precisa dun novo compilado e enlazado para poder executar o programa.

As vantaxes de este tipo de aproximación son que a distribución do software non inclúe o código fonte, sendo máis compacta, e sobre todo a súa velocidade de execución, moi superior ás das linguaxes interpretadas.

Linguaxes salientables con esta aproximación son ***C/C++*** e ***Pascal***

O p-code

Co obxectivo de manter a portabilidade do software entre diferentes plataformas coa mínima perdida de velocidade posible, se deseñou unha "***máquina virtual***[7]" capaz de executar un código binario específico, de xeito que para executar un programa bastaría con dispoñer da "máquina virtual" específica da plataforma.

As linguaxes de programación se compilarían entón ao código binario específico da "máquina virtual" en lugar do código da plataforma onde desexamos executar o programa. Este tipo de linguaxe máquina denomínase ***p-code***.

A velocidade de execución do p-code e moi superior a das linguaxes interpretadas, xa que as instrucións p-code son moito máis sinxelas de executar que a interpretación dunha linguaxe de

7 O sentido de "Máquina Virtual" non é o mesmo que unha máquina virtual como VirtualBox. Se refire simplemente a unha simulación dunha CPU que leva a cabo un xogo de instrucións específico. E algo moito máis simple e rápido que un Hypervisor de máquinas virtuais como VMWare, VirtualBox ou KVM.

programación e permiten numerosas optimizacións moi difíciles de obter cunha interpretación pura, como a compilación Just In Time (***JIT***[8]). En algúns casos específicos e aínda que pareza incrible o p-code pode chegar a executarse a maior velocidade que un programa compilado nativo.

Mediante esta aproximación se consigue a portabilidade do executable, simplemente si dispoñemos da máquina virtual apropiada, pero o código fonte precisa dun proceso similar á compilación.

As linguaxes máis salientables que utilizan esta aproximación son **Java** e ***.NET*** (***C#***).

Paradigmas de Programación

Baseándose nas súas características, as linguaxes de programación poden clasificarse en múltiples "paradigmas". Un paradigma é un "estilo" concreto de analizar e xestionar as solucións aos problemas.

Algunhas clasificacións en "paradigmas" se fixan principalmente na forma de analizar levar a cabo as solucións, outras se refiren a como se organiza o código, mentres que outras se centran principalmente no estilo da sintaxe e a gramática da linguaxe. Sen embargo, podemos distinguir os seguintes estilos principais de programación xerais: ***Programación Declarativa*** e ***Programación Imperativa***.

Programación Declarativa

Na programación declarativa se describe a lóxica de computación necesaria para resolver un problema, sen seguir unha secuencia de ordes fixas. Na programación declarativa non é necesario definir algoritmos xa que o que se detalla é a solución ao problema, e non como se chega a esa solución. A solución se alcanza mediante mecanismos internos de control pero non se especifica exactamente como chegar a ela.

No paradigma declarativo descríbese o problema sen fixarnos no algoritmo necesario para conseguila solución.

Un exemplo de un uso declarativo dunha linguaxe son as instrucións de consulta, inserción, modificación e borrado mediante ***SQL***. Se indica o que se quere facer, sen entrar nos detalles de como se fai. ***"di o que queres, pero non como o queres"***

Dentro de esta categoría se engloban a ***programación funcional*** con linguaxes coma **Haskell** ou a ***programación lóxica***, con **Prolog** coma linguaxe máis representativo.

Programación Funcional

Nas linguaxes funcionais puras como *Haskell*, todas as funcións son puras, e dicir, non teñen efectos fora da propia función e os cambios de estado (datos almacenados polo programa) se representan mediante funcións que se encargan de transformalo. Aínda que non son linguaxes imperativas, en xeral proporcionan mecanismos para describir as funcións coma unha serie de pasos.

8 A compilación JIT (Just In Time) consiste en ir compilando e almacenando o bytecode a código compilado nativo para posteriores execucións, de xeito que si se teñen que levar a cabo as mesmas instruccións xa se atopen compiladas

Na programación funcional non existen variables (ou ben en realidade son inmutables), xa que as funcións so necesitan procesar os datos e non se precisa almacenar o estado de nada. Cada función opera sobre os seus propios datos sen relación coas demais funcións.

Exemplo

Cálculo do factorial dun número en ***Haskell***. (Para probar o programa é necesario un compilador de Haskell (ghc), en Debian pode instalarse con ***apt-get install ghc)***

```
-- Inicio da aplicación realiza o factorial de 6
main = do
   putStrLn "Introduce un número enteiro >= 0: "
   inputjar <- getLine
   let n = read inputjar :: Int
   let r = fact n
   putStrLn("O resultado é " ++  show(n) ++ "!=" ++  show(r))

-- O factorial de 0 é 1. O factorial de n, e n multiplicado polo factorial de n-1
fact::Int->Int
fact 0=1
fact n=n*fact(n-1)
```

Programación Lóxica

A programación lóxica utiliza a lóxica para o análise dos problemas e o control das regras de dedución necesarias para alcanzar a solución. Na Programación Lóxica, se traballa de xeito descritivo, establecendo relacións entre entidades, indicando non como, se non que facer.

A programación lóxica intenta resolver problemas da seguinte natureza: *Dado un problema S, saber si a afirmación A é solución ou non do problema ou en que casos o é de xeito automático mediante un programa*. A programación lóxica constrúe a base de coñecementos mediante regras e feitos que permiten chegar á solución.

Pola súa natureza, este paradigma encaixa moi ben coa solución de problemas mediante intelixencia artificial, a minaría de datos ou a robótica.

Exemplo 1

Averiguar o factorial dun número. Para probar este programa en Debian será necesario un intérprete de prolog que podedes instalar con ***apt-get install swi-prolog***. Unha vez escrito o código fonte nun ficheiro **factorial.pl**, se poderán cargar as regras mediante ***swipl -f factorial.pl***. Nese momento, o intérprete solicitará o obxectivo a verificar. Podedes probar con ***"factorial(3,6)."***, que debe imprimir "true", con ***"factorial(3,8)."***, que debe imprimir "false", ou ***"factorial(4,W)."*** que debe imprimir W=24, que é o valor de W que satisface as regras. (tamén podedes utilizar o compilador GNU-prolog con apt-get install gprolog)

```
factorial(0,1).

factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

Este programa ten dúas cláusulas. A primeira non ten corpo, e é unha afirmación. A segunda é unha regra, porque ten un corpo. O corpo está a dereita do símbolo **:-** que indica o que se debe cumprir para que as regras sexan certas. As comas que separan as distintas afirmacións indican que todas elas deben cumprirse.

A primeira cláusula nos indica "*O factorial de 0 é 1*", a segunda indica que "O factorial de N é F , **SI** *N>0* é *si tomamos N1 coma o valor de N-1* e *F1 como o factorial de N1 o valor de F é igual a N * F1*".

O sistema será capaz de responder á pregunta obxectivo factorial(3,Resultado), obtendo o valor para Resultado.

Exemplo 2

Este exemplo é significativo do que é a programación lóxica. Indicamos ao sistema os nosos coñecementos en forma de cláusulas, e esas cláusulas nos permiten inferir resultados.

```
chove(X):-
        nublado(X),
        frio(X).
frio(X):-
        temperatura(X,baja).
nublado(X):-
        sin_sol(X).

temperatura(xoves,baja).
sen_sol(xoves).
```

Este programa ten 3 cláusulas e 2 afirmacións. Na primeira se establece que o día X chove **SI** o día X está nubrado **e** o día X fai frío. A segunda cláusula se establece que o día X fai frío **SI** o día X a temperatura "baixa". A terceira cláusula establece que o día X está nubrado **SI** o día X é sin_sol. Por último se establecen as afirmacións de que a temperatura o xoves 'baja' e que o xoves estaremos 'sen_sol'. O sistema será capaz de responder á pregunta obxectivo *"chove(xoves)"* a partir de estas regras, chegando á conclusión de que si, que chove.

Programación Imperativa

A diferencia da programación declarativa, na programación imperativa se describen un conxunto de sentencias que van alterando o estado (os datos) dun programa. A programación imperativa é a programación natural para as CPU, xa que é o modo no que traballan de xeito básico.

Neste paradigma as solucións aos problemas se expresan indicando unha secuencia de accións a realizar para chegar á solución, o que se coñece como *algoritmo*.

Dentro de esta categoría se engloban a ***programación estruturada***, e a ***programación orientada a obxectos***, sendo estes os paradigmas de uso máis xeral e estendido na actualidade pola súa gran versatilidade, e utilizados polas linguaxes más habituais como JavaScript, PHP, C/C++, Python, C#, Java ... etc., aínda que cada vez é máis común que incorporen características de outros paradigmas que os converten en ***linguaxes multiparadigma,*** como poden ser as expresións lambda[9].

Programación Estruturada

A programación estruturada recibe o seu nome do "Teorema da estrutura", que establece que toda función resoluble cun programa de ordenador pode ser escrita nunha linguaxe de programación que combine unicamente tres estruturas lóxicas chamadas estruturas de control:

- **Secuencia**: Execución dunha sentencia detrás de outra. As sentencias poden ser asignacións, expresións aritméticas ou lóxicas ou chamadas a funcións.
- **Selección**: Execución dun grupo ou outro de instrucións dependendo dun valor booleano, que normalmente se obtén avaliando unha expresión lóxica. Utilizando pseudocódigo se representa:

```
Si (condicion) enton
        secuencia
se non
        secuencia
```

9 As ***expresións lambda*** basicamente son funcións sen nome que se definen no mesmo lugar no que son chamadas. Tamén se coñecen como funcións anónimas.

Fin-Si

- **Iteración**: Execución dun grupo de instrucións mentres unha variable booleana sexa certa (ou ata que sexa certa....). Esta estrutura tamén se coñece como ciclo ou bucle. Utilizando pseudocódigo se representa:

Mentras(condicion)
secuencia
Fin-Mentras

Este teorema elimina o uso de certas instrucións de uso común en linguaxes de moi baixo nivel como son as instrucións de salto (***goto***) aínda que a veces se poidan empregar por motivos de optimización da velocidade.

Polo tanto, a programación estruturada e aquela que unicamente utiliza estas tres estruturas.

Deseño Descendente ou Programación Modular

Dentro da programación estruturada, a hora de pensar na solución a un problema a mellor aproximación é a de "divide e vencerás", tamén chamada deseño descendente ou "Top-Down", porque consiste en ir descompoñendo os problemas complexos (Top) cara a problemas máis sinxelos (Down). Para esta aproximación é moi importante o concepto de ***función***.

Unha función, e un conxunto de operacións para resolver un problema concreto a partir dunha serie de datos de entrada e producindo (ou non) uns resultados de saída.

As distintas linguaxes de programación proporcionan sempre un conxunto de funcións xa programadas para a realización de tarefas básicas, como entrada e saída de datos, cálculos numéricos... etc. Que se coñece como ***libraría de funcións da linguaxe (ou biblioteca de funcións..)***. Tamén e común o uso nas aplicacións de funcións de propósito xeral realizadas por terceiros para aumentar a funcionalidade dispoñible dun modo simple, coñecidas como **librarías "de terceiros"** . Sexa como fora, estas librarías nos permiten realizar actividades que caen fora da propia linguaxe de programación.

O máis importante, sen embargo, e que as distintas linguaxes de programación nos permiten definir as nosas propias funcións, que logo podemos empregar nos nosos programas. Aquí comenza a aproximación do deseño descendente.

O deseño descendente consiste en examinar o problema non como un todo "monolítico" a resolver cun conxunto de instrucións simples unha detrás de outra, se non como un conxunto de problemas que é necesario resolver para solucionar o problema global. Cada un de estes "subproblemas" corresponderá normalmente cunha función que o resolve. Utilizando esas novas funcións será posible resolver e implementar a solución.

Posteriormente aplicamos a mesma técnica para resolver cada un dos subproblemas (funcións) empregadas.

No momento de analizar un "subproblema" caben dúas posibilidades:

- **O "subproblema" é trivial**, e se pode solucionar cunha combinación simple das instrucións da linguaxe e das funcións xa dispoñibles. Si este "subproblema" non é común, e dicir, non se prevé que poida darse en outras partes do programa, podemos codificar a solución directamente escribindo a secuencia de instrucións apropiada. Si pensamos que o

"subproblema" é común, e que se pode dar máis veces en este ou outro programa, crearemos unha función coa implementación. Deste xeito iremos creando a nosa propia biblioteca de funcións facendo máis fácil a realización de programas segundo imos resolvendo problemas.

- **O "subproblema" non é trivial**, e precisa dunha combinación de instrucións máis complexas. Nese caso, é necesario dividilo en problemas máis sinxelos, de xeito análogo ao problema inicial.

Vexamos un efecto claro de esta aproximación, resolvendo primeiro o problema como "un todo", e logo aplicando o deseño descendente:

Escribir un programa que calcule o número e cunha resolución a elixir polo usuario. A resolución indicará o número de términos que se teñen que utilizar para o cálculo, xa que se calculará utilizando a sucesión de Mac Laurin: e=1+1/1!+1/2!+1/3!+1/4!....

a) Aproximación "non descendente", solucionamos o problema paso a paso:

```
Visualizar "Introduce Precisión >= 1"
Ler de teclado o valor da precisión e gardala en pr
e=1
n=1
Mentres n<pr facer
        # Calculamos o factorial do número n
        f=1;
        Mentres n>1 facer
                f=f*n
        Fin Mentres
        # Acumulamos en e o sumando. Supoñemos división decimal
        e = e + 1 / f
        # O seguinte número a calcular o seu factorial....
        n = n + 1
Fin Mentres
Visualizar " e = " e
```

b) Aproximación "descendente"

```
Visualizar "Introduce Precisión >= 1"
Ler de teclado o valor da precisión e gardala en pr
e=1
n=1
Mentres n<pr facer
        # Acumulamos en e o sumando. Supoñemos división decimal. O calculo do factorial é un problema "diferente"
        e = e + 1 / factorial(n);
        # Seguinte sumando....
        n = n + 1
Fin Mentres
Visualizar "e = " e
```

Como podemos observar, dividimos o problema en dúas partes. Unha é calcular o número **e** mediante a sucesión, outro é o cálculo do factorial dun número. Aínda que é un caso sinxelo se observan as seguintes vantaxes:

1. Dispoñeremos nun futuro dunha nova función ***factorial*** que nos permitirá calcular o factorial dun número cando o precisemos, pasando a formar parte da nosa propia libraría de funcións.
2. O programa é máis lexible, e reflexa mais fielmente o enunciado.
3. O programa é máis fácil de modificar e corrixir. Si o factorial se calcula mal, corriximos a función do factorial, se non, corriximos o corpo principal. Tamén é mais fácil facer probas para verificar en que parte falla.
4. A división do problema permite repartir o traballo entre diversos programadores. A función factorial pódese "encargar" a outra persoa que coñeza "que é iso de un factorial" simplemente dicíndolle :

“Escríbeme unha función que reciba un número enteiro como argumento e me devolva o factorial de ese número. A función debe chamarse ***factorial***”.

5.

Por suposto, para que o programa sexa completo, precisamos da función factorial:

```
# A función factorial calcula o factorial dun número.

función factorial: Recibe un número enteiro N e devolve outro número enteiro que e o resultado de calcular N!
Si N == 0  retornar o valor 1  # Por convenio, o factorial de 0 é 1
r = 1
Mentres N > 1
        r = r * N
        N = N – 1
Fin Mentres
Retornar o valor de r
```

A programación modular actualmente é a base da programación imperativa non orientada a obxectos, xa que como veremos, a orientación a obxectos se pode ver como “programación modular levada ao extremo”.

Para resolver calquera problema minimamente complexo é necesario dividilo en outros máis sinxelos. Moitas veces será posible utilizar solucións xa existentes a eses problemas máis simples, e se non é o caso, ao resolvelos crearemos solucións que poderemos utilizar de novo no futuro, dando lugar a distintos “módulos”, que podemos agrupar en ficheiros denominados “librarías”. Estes módulos se poden incorporar ao programa (librarías estáticas), ou ser cargados cando sexan necesarios durante a execución do programa (librarías dinámicas, en Windows DLL’s).

As vantaxes máis salientables do deseño descendente son:

- **Facilita a solución dos problemas**, evitando que a dificultade dun aspecto do problema impida chegar á solución global.
- **Permite a reutilización do código**, xa que unha vez implementadas as funcións necesarias para o problema que estamos a resolver poderán ser utilizadas en novos problemas.
- **Permite a colaboración**, xa que varios programadores poden resover simultáneamente os distintos subproblemas.
- **Facilita a modificación, optimización e arranxo de erros**, xa que o traballo quedará dividido en distintas partes con funcionalidades moi concretas.

Linguaxes Salientables

Calquera linguaxe de programación imperativo se pode utilizar como linguaxe non orientado a obxectos, incluso as linguaxes puramente orientadas a obxectos como Java.

A orientación a obxectos, do mesmo modo que o deseño descendente, e un xeito de deseñar as aplicacións, mais alá de que a linguaxe teña mecanismos que faciliten ese deseño (linguaxes orientadas a obxectos).

Linguaxes que normalmente se empregan sen facer un uso explícito de orientación a obxectos ou que simplemente non dispoñen de mecanismos que a faciliten son as linguaxes de ***Shell***, ***C***, ***Pascal***, ***PHP*** (a partir de PHP5 xa se dispón de mecanismos de orientación a obxecto "serios") ou ***JavaScript*** (aínda que ten mecanismos de orientación a obxectos, están baseados en prototipos[10]).

10 tamén chamada programación baseada en instancias ou "***classless***", é un estilo de programación orientada a obxectos na que os obxectos non son creados a partir de clases se non mediante a clonación de outros obxectos ou mediante a escritura de código por parte do programador. Dese xeito os obxectos xa existentes poden servir de prototipos para os que o programador necesite crear.

Programación Orientada a Obxectos

A programación orientada a obxectos supón un xeito distinto de analizar os problemas para a súa solución. Aínda que finalmente se fai uso da programación estruturada, a visión do problema non é a da descompoñer o problema xeral en outros máis sinxelos (top-down).

Na programación orientada a obxectos, o problema a resolver se ve como unha interacción entre o conxunto de obxectos que interveñen no problema. Os obxectos estarán compostos de un conxunto de características (***atributos***, representados por variables), e un conxunto de "capacidades" ou "cousas que é capaz de facer" o obxecto (**métodos**, representados con funcións), de xeito análogo aos obxectos da vida real.

A definición dos atributos e métodos que compoñen un tipo concreto de obxectos se coñece como ***Clase***. Os obxectos son elementos concretos pertencentes a unha Clase, creados mediante o que se coñece como *instanciación* (en case todas as linguaxes mediante o operador ***new***). ***Os obxectos polo tanto, son instancias dunha Clase.***

Para resolver un problema mediante orientación a obxectos é necesario polo tanto identificar en primeiro lugar os obxectos que interveñen no problema, e definir as distintas clases ás que pertencen. O programa consistirá na creación dos distintos obxectos e a invocación apropiada dos seus métodos.

Si partimos do enunciado do problema, podemos axudarnos do seguinte:

- Os nomes normalmente coincidirán ou ben con obxectos ou con atributos dos obxectos.
- Os verbos normalmente coincidirán con accións que realiza algún obxecto, polo tanto deberían ser métodos.

As linguaxes orientadas a obxectos non se limitan á proporcionar xeitos de definir clases e crear obxectos, se non que proporcionan varios mecanismos interesantes:

- **Herdanza**

  Permite definir novas clases partindo dunha definición previa. A nova clase incorporará de xeito automático os atributos e comportamento (métodos) da clase de orixen.

  Exemplo:

  Si definimos unha clase "Poligono", éste podería ter como atributos por exemplo o número e a posición dos seus vértices, e como métodos un método que permita pintar o polígono, e outro que permita desplazalo. A partir de ahí, poderíamos definir a clase "Rectangulo", que heredando de Polígono conserva eses métodos e atributos de xeito automático, e pode incorporar algúns novos. **Se podería decir en ese caso que un Rectángulo é un Polígono, xa que ten os mesmos atributos e métodos** (o contrario non é certo, os Polígonos non teñen por qué ser Rectángulos)

- **Sobreposición**

  Cando definimos unha nova clase partindo dunha definición previa, poderemos modificar o comportamento dos métodos na nova clase "sobrepoñendo" o código da clase de orixen.

Exemplo:

Si observamos o exemplo indicado na herencia, poderíamos modificar o comportamento do método para pintar na clase Rectangulo de xeito que se realice dun xeito máis eficiente que o que realiza o método Polígono. Esta nova versión do método de pintado “sobrepon” a función definida en Poligono.

- **Polimorfismo**

O Polimorfismo se encarga de asegurar que cando chamamos a un método dunha clase, se executa a versión correcta do mesmo. Ser verá con detalle máis adiante.

- **Sobrecarga**

A sobrecarga permite dispoñer de distintas versións do mesmo método, sempre que teñan número ou tipos de parámetros distintos. Esto facilita a claridade dos programas.

```
class Animal {  // Clase Base
	String name;
	public Animal(String name) {
		this.name=name;
	}
	public String talk() {
		return "";
	}
	public String getName() {
		return this.name;
	}
}

class Cat extends Animal { // Clase Heredada
	public Cat() {
		super("Cat");
	}
	public String talk() { // Sobreposición do método talk de Animal
		return "miauuuuu";
	}
}

class Dog extends Animal {
	public Dog() {
		super("Dog");
	}
	public String talk() { // Sobreposicion do método talk de Animal
		return "guau-guau";
	}
}

public class Example {
	public static void main(String[] args) {
		Animal an1=new Cat();
		Animal an2=new Dog();
		Cat c=new Cat();
		Dog d=new Dog();
		System.out.println("The "+an1.getName()+" says "+an1.talk());
		System.out.println("The "+an2.getName()+" says "+an2.talk());
		System.out.println("The "+c.getName()+" says "+an1.talk());
		System.out.println("The "+d.getName()+" says "+d.talk());
	}
}
```

Para probar este programa se deberán crear un ficheiro para cada clase (Animal.java, Cat.java, Dog.java e Example.java), a continuación se compilará Example.java (**javac Example.java**) e logo se poderá executar con **java Example**