Introducción a la programación orientada a objetos

Vicent Moncho Mas

PID_00220485

UOC
Universitat Oberta de Catalunya

Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es

Índice

1. Introducción 5
1.1. La programación 5
1.2. Modelos de desarrollo 6
1.2.1. Programación imperativa 7
1.2.2. Programación funcional 8
1.2.3. Programación lógica 8
1.2.4. Programación orientada a objetos 9

2. Conceptos y principales características 11
2.1. El modelo orientado a objetos 11
2.1.1. Clases 12
2.1.2. Objetos 16
2.1.3. Herencia 19
2.2. Características de la programación orientada a objetos 22
2.2.1. Abstracción 22
2.2.2. Ocultamiento o encapsulamiento 23
2.2.3. Polimorfismo 25
2.2.4. Destrucción de objetos 29
2.2.5. Análisis y diseño orientado a objetos 30

3. POO en los distintos lenguajes de programación 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. Introducción

1.1. La programación

La programación es un proceso por el que se escribe, se prueba, se depura y se modifica el código. Los programas como elementos que forman el software son un conjunto de instrucciones que se ejecutan en el hardware con el objetivo de realizar una tarea determinada.

Para el desarrollo de programas de cierta envergadura o complejos, con ciertas garantías de calidad, es conveniente seguir alguno de los modelos de desarrollo de software existentes, en los que la programación es sólo una de las etapas del proceso de desarrollo del software.

Un **programa** es un conjunto de instrucciones que se ejecutan con el objetivo de resolver un cierto problema. Los programas se componen en varios algoritmos, y cada uno de éstos resuelve una parte del problema inicial. De esta manera, la complejidad de cada una de las partes es menor que la del programa completo (esta técnica es conocida como "divide y vencerás").

Según Niklaus Wirth, un programa está formado por algoritmos y una estructura de datos.

Niklaus Wirth (Winterthur, 1934)

Doctorado en 1963 en Berkeley, de 1963 a 1967 fue profesor de Informática en Stanford y en la Universidad de Zúrich. A partir de 1968, pasó a ser profesor de Informática en la ETH de Suiza y se tomó dos años sabáticos en la Xerox PARC de California.

Fue el jefe de diseño de los lenguajes de programación Euler, Algol W, Pascal, Modula, Modula-2 y Oyeron, y ocupó gran parte de su tiempo en el equipo de diseño e implementación de sistemas operativos Lilith y Oberon para el Lola en el diseño del hardware digital y el sistema de simulación.

Su artículo de desarrollo de un programa por refinamiento sucesivo ("Program Development by Stepwise Refinement") es considerado un texto clásico en la ingeniería del software, así como su libro *Algoritmos + Estructuras de datos = Programas*, que recibió un amplio reconocimiento, y que aún hoy es útil en la enseñanza de la programación. Recibió el Premio Turing por el desarrollo de estos lenguajes de programación en 1984.

La programación tiene como uno de sus principales **objetivos** la creación de software de calidad, y los principales factores que indican el **nivel de calidad de un programa** son los siguientes:

1) Corrección: un programa es correcto si hace lo que debe hacer tal y como se estableció. Para determinar si un programa actúa correctamente es muy importante disponer de una especificación de lo que debe resolver el programa antes de desarrollarlo, de esta manera, una vez implementado se ha de comprobar que realmente lo implementa.

2) Claridad: es fundamental que el código sea lo más claro y legible posible, para facilitar así su desarrollo y posterior mantenimiento. Se debe intentar que su estructura sea sencilla y coherente, así como cuidar el estilo en la edición; de esta manera, se facilita el trabajo del programador, tanto en la fase de creación como en las fases posteriores de corrección de errores, ampliaciones, modificaciones, etc. Fases que pueden ser realizadas incluso por otro programador, con lo que la claridad es aún más necesaria para que otros programadores puedan continuar el trabajo fácilmente.

3) Eficiencia: además de realizar aquello para lo que fue creado, debe realizarlo minimizando los recursos que utiliza. La eficiencia se basa en el tiempo que tarda en realizar la tarea para la que ha sido creado y a la cantidad de memoria que necesita. Hay otros recursos que también deben ser considerados: espacio en disco que utiliza, tráfico de red que genera, etc.

4) Portabilidad: se basa en la capacidad de poder ejecutarse en una plataforma, ya sea hardware o software, diferente a aquélla en la que se elaboró. La portabilidad es una característica muy deseable para un programa, ya que permite, por ejemplo, a un programa que se ha desarrollado para sistemas GNU/Linux ejecutarse también en la familia de sistemas operativos Windows, incluso en distintas máquinas virtuales de Java o en los distintos navegadores que existen en el mercado.

1.2. Modelos de desarrollo

Se entiende como modelos de desarrollo o paradigmas de programación a los distintos enfoques o filosofías que se han ido creando para la construcción del software. No existe un modelo mejor que otro, sino que, dependiendo del tipo de problema, un modelo puede ser más apropiado que otro.

Los modelos de desarrollo más comunes son los siguientes:

- Programación imperativa.
- Programación funcional.
- Programación lógica.
- Programación orientada a objetos.

1.2.1. Programación imperativa

La programación imperativa se basa en un conjunto de instrucciones que le indican al hardware cómo debe realizar una tarea. Está considerada la más común y la representan, por ejemplo, lenguajes como C o BASIC.

El hardware está diseñado para ejecutar código de máquina, escrito en una forma imperativa y secuencial. Desde esta perspectiva de bajo nivel, las sentencias son instrucciones en el lenguaje de máquina nativo del computador (por ejemplo, el lenguaje ensamblador).

Los primeros lenguajes imperativos fueron los lenguajes ensamblador, que están condicionados por el hardware que los debía implementar. En estos lenguajes, las instrucciones fueron muy simples, lo que provocó la implementación de hardware fácil, pero obstruyó la creación de programas complejos. Fortran, cuyo desarrollo fue iniciado en 1954 por John Backus en IBM, fue el primer gran lenguaje de programación que superó los obstáculos presentados por el código máquina en la creación de programas complejos.

Los siguientes lenguajes implementan la programación imperativa: BASIC, C, C#, C++, Fortran, Pascal, Java, Perl, PHP.

John Backus (Filadelfia, 1924-Oregón, 2007)

Informático estadounidense. John Backus ganó el Premio Turing en 1977 por sus trabajos en sistemas de programación de alto nivel, en especial por su trabajo con FORTRAN.

Para evitar las dificultades de programación de las calculadoras de su época, en 1954 Backus se encargó de la dirección de un proyecto de investigación en IBM para el proyecto y realización de un lenguaje de programación más cercano a la notación matemática normal. De ese proyecto surgió el lenguaje FORTRAN, el primero de los lenguajes de programación de alto nivel que tuvo un gran impacto, incluso comercial, en la emergente comunidad informática.

Tras la realización de FORTRAN, Backus fue un miembro muy activo del comité internacional que se encargó del proyecto de lenguaje ALGOL. En ese contexto propuso una notación para la representación de las gramáticas usadas en la definición de un lenguaje de programación (las llamadas gramáticas libres de contexto). Tal notación se conoce como BNF, o Forma de Naur y Backus (Backus-Naur Form), y une el nombre de Backus al de Peter Naur, un informático europeo del comité ALGOL que contribuyó a su definición.

En la década de los setenta, Backus se interesó sobre todo por la programación funcional, y proyectó el lenguaje de programación FP, descrito en el texto que le sirvió para ganar el Premio Turing: "Can Programming be Liberated from the Von Neumann Style?". Se trata de un lenguaje de uso fundamentalmente académico que, sin embargo, animó un gran número de investigaciones. El proyecto FP, transformado en FL, se terminó cuando Backus se jubiló en IBM, en 1991.

Fuente: Wikipedia.

1.2.2. Programación funcional

La programación funcional es un paradigma de programación que se basa en la utilización de funciones matemáticas. El máximo representante de este paradigma es el lenguaje LISP.

El objetivo es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa. Los programas están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración, lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas.

Existen dos grandes categorías de lenguajes funcionales: los **funcionales puros** (Haskell y Miranda) y los **híbridos** (Scala, Lisp, Scheme, Ocaml, SAP y Standard ML). La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos rígidos que los puros, ya que admiten conceptos de los lenguajes imperativos, como las secuencias de instrucciones o la asignación de variables.

Se podría incluir a Perl como lenguaje funcional híbrido, ya que aunque es un lenguaje de propósito general, se pueden implementar programas usando exclusivamente funciones definidas por el usuario.

1.2.3. Programación lógica

Generalmente, en los lenguajes de programación imperativos, cuando nos enfrentamos a un problema complejo, la implementación de la solución de éste puede ocultar o dificultar su comprensión. En este sentido, la lógica matemática es la manera más sencilla de expresar formalmente problemas complejos y de resolverlos mediante la aplicación de reglas, hipótesis y teoremas.

La programación lógica encuentra su hábitat natural en aplicaciones de inteligencia artificial, por ejemplo:

- Sistemas expertos, en los que el programa imita las recomendaciones de un experto sobre algún dominio de conocimiento.
- Demostración automática de teoremas. En este caso el programa genera nuevos teoremas sobre una teoría existente.

- Reconocimiento de lenguaje natural. El programa es capaz de comprender (con limitaciones) la información contenida en una expresión lingüística humana.

El principal lenguaje de programación lógica es Prolog.

1.2.4. Programación orientada a objetos

La programación orientada a objetos tiene su origen en el lenguaje Simula 67, creado por Ole-Johan Dahl y Kristen Nygaard en el Centro de Cómputo Noruego de Oslo.

El objetivo del lenguaje era la implementación de simulaciones de naves de manera simple y más óptima. Pensaron en agrupar los distintos tipos de naves en varias clases de objetos, considerando cada clase de objetos como responsable de definir sus propios datos y comportamiento.

Ole-Johan Dahl (Mandal, 1931-2002)

Es uno de los científicos de la computación más famosos en Noruega. Junto con Kristen Nygaard, Ole-Johan Dahl produjo las primeras ideas sobre programación orientada a objetos en los años sesenta en el Centro Noruego de Cómputo (NCC), como parte de los lenguajes de programación para simulación Simula I (1961-1965) y Simula 67 (1965-1968). Dahl y Nygaard fueron los primeros en desarrollar los conceptos de objeto, clase, herencia, creación dinámica de objetos, etc., todos aspectos importantes del paradigma de la OO.

Dahl logró plaza de profesor en la Universidad de Oslo en 1968, en la que destacó como un privilegiado educador e investigador. Allí trabajó en *Estructuras Jerárquicas de Programas*, probablemente su publicación más influyente, escrita junto con C. A. R. Hoare y Edsger Dijkstra en el famoso libro *Structured Programming* de 1972, quizá el libro sobre software más conocido de esa década.

Al avanzar su carrera, Dahl se interesó en el uso de métodos formales, por ejemplo para razonar rigurosamente sobre orientación a objetos. Como todo buen científico de la computación, su experiencia alcanzaba desde la aplicación práctica de sus ideas a la demostración matemática de su corrección para asegurar la validez de su enfoque. Recibió el Premio Turing por su trabajo en el 2001, un año antes de fallecer.

Fuente: Wikipedia.

Kristen Nygaard (1926-2002)

Fue un matemático noruego, pionero de la informática y político. Kristen Nygaard obtuvo el título de máster de matemáticas en la Universidad de Oslo en 1956. Su tesis sobre teoría de probabilidad abstracta se tituló "Theoretical Aspects of Monte Carlo Methods" ("Aspectos teóricos de los métodos de Montecarlo"). Entre 1948 y 1960, Nygaard desarrolló varias tareas en el Departamento de Defensa noruego, incluyendo labores investigadoras. Fue cofundador y primer presidente de la Sociedad Noruega de Investigaciones Operacionales (1959-1964). En 1960, fue contratado por el Centro Noruego de Computación (NCC, por sus siglas en inglés), como responsable para establecer al NCC como un importante instituto de investigación en los años sesenta. Allí, junto con Ole-Johan Dahl, desarrolló los lenguajes SIMULA I (1961-1965) y SIMULA 67.

Trabajó para los sindicatos noruegos sobre planificación, control y procesamiento de datos, todo ello evaluado en función de los objetivos de la mano de obra organizada (1971-1973, junto con Olav Terje Bergo). Finalmente, también dedicó algo de esfuerzo al estudio del impacto social de las tecnologías de la computación, así como al lenguaje general de descripción de sistemas DELTA (1973-1975, junto con Erik Holbaek-Hanssen y Petter Haandlykken). Nygaard fue profesor en Aarhus (Dinamarca) en el curso 1975-1976, y fue nombrado profesor emérito en Oslo (a tiempo parcial desde 1977, y a tiempo completo entre 1984 y 1996).

> En noviembre del 2002, recibió, junto a Dahl, la Medalla John von Neumann del IEEE, "por la introducción de los conceptos subyacentes de la programación orientada a objetos, mediante el diseño e implementación de SIMULA 67".
>
> Fuente: Wikipedia.

Estos conceptos se perfeccionaron en el lenguaje Smalltalk, diseñado para ser un sistema completamente dinámico en el que los objetos se podrían crear y modificar "sobre la marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las interfaces gráficas de usuario, para las que la programación orientada a objetos está particularmente bien adaptada.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp y Pascal, entre otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y a la capacidad de mantenimiento del código.

Los lenguajes orientados a objetos "puros", por otra parte, carecían de las características que muchos programadores solían utilizar. Para evitar este problema, aparecieron una serie de iniciativas con el objeto de crear nuevos lenguajes basados en métodos, pero orientados a objetos, manteniendo de esta manera algunas de las características imperativas.

El lenguaje Eiffel fue el primero que cumplía estos objetivos, pero fue reemplazado por Java, en gran parte debido a la aparición de Internet y a la implementación de la máquina virtual de Java en la mayoría de navegadores.

El lenguaje PHP, a partir de la versión 5, soporta una orientación completa a objetos y cumple todas las características propias de la orientación a objetos.

En el caso de JavaScript, éste es un lenguaje orientado a objetos, en el que la herencia se implementa siguiendo el paradigma de programación basada en prototipos, esto es, las nuevas clases se generan clonando las clases base y ampliando sus métodos y propiedades.

2. Conceptos y principales características

La tecnología orientada a objetos ya no se aplica solamente en los lenguajes de programación, se aplica también en el análisis y diseño de software, al igual que en las bases de datos. Es uno de los modelos más productivos, que se debe a sus grandes capacidades y ventajas frente al resto de modelos de programación.

La orientación a objetos se basa en la división del programa en pequeñas unidades lógicas de código; estas unidades es lo que se conoce como objetos. Éstos son unidades independientes que se comunican entre sí mediante mensajes.

Las **principales ventajas** de un lenguaje orientado a objetos son:

- El fomento de la reutilización y extensión del código.
- La elaboración de sistemas más complejos.
- La relación del sistema con el mundo real.
- La creación de programas visuales.
- La construcción de prototipos.
- La agilización del desarrollo de software.
- La facilitación del trabajo en equipo.
- Un mantenimiento más sencillo del software.

Uno de los aspectos más interesantes de la programación orientada a objetos es que proporciona conceptos y herramientas con las que se modela y representa el mundo real tan fielmente como sea posible.

En los siguientes subapartados se mostrarán ejemplos utilizando el lenguaje de programación Java.

Se ha elegido el lenguaje Java por dos motivos principalmente: en primer lugar cumple todos los conceptos de la POO que vamos a explicar, y en segundo lugar su sintaxis es sencilla y por lo tanto, facilita la comprensión.

Los ejemplos no están para reproducirlos, sino para ayudar a comprender los conceptos que se explican. En el siguiente módulo el estudiante podrá practicar parte de estos conceptos con JavaScript, y solo es una parte, ya que este no cubre todo el paradigma de la programación orientada a objetos.

2.1. El modelo orientado a objetos

El modelo de programación orientada a objetos se basa en los siguientes conceptos fundamentales:

- Clases.

- Objetos.
- Herencia.
- Envío de mensajes.

2.1.1. Clases

La **estandarización** es una técnica que provoca que existan distintos objetos de un mismo tipo, que comparten el proceso de fabricación.

Ejemplos

Algunos ejemplos de estandarización podrían ser el teléfono móvil, el coche, el ordenador, la televisión, etc.

En el mundo de la programación orientada a objetos, nuestro teléfono móvil podría ser una instancia de una clase que se podría llamar "teléfono". Cada teléfono móvil tiene una serie de características, como la marca, el modelo, el sistema operativo, el tipo de pantalla y de teclado, etc.; y algunos de sus comportamientos son: realizar y recibir llamadas, enviar mensajes, transmitir datos, etc.

En el proceso de fabricación se aprovecha el hecho de que los móviles comparten características y se construyen modelos o plantillas comunes, a partir de las que se crean millares de teléfonos móviles del mismo modelo.

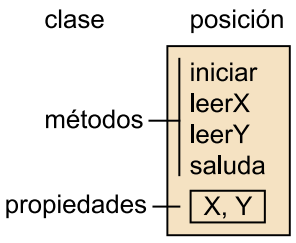
En el mundo de la programación orientada a objetos, la plantilla se conoce como clase y a los equipos que sacamos a partir de ella los llamamos objetos o instancia de clase. De esta manera, de una única clase se pueden crear muchos objetos del mismo tipo.

La **clase** es un modelo o prototipo, que define las variables y métodos comunes a todos los objetos, o una plantilla genérica para un conjunto de objetos de similares características.

La **instancia** es un objeto de una clase en particular.

La figura siguiente presenta un ejemplo de una clase "posición", en la que se tienen 4 métodos (iniciar, leerX, leerY y saluda) y dos propiedades X e Y:

Figura 1



A partir de esta clase se pueden crear tantos objetos como se necesiten y todos tendrán los métodos y las propiedades definidos en la clase.

En Java la creación de una clase se realiza utilizando la palabra clave reservada *class* seguida del nombre de la clase y finaliza con un bloque de código delimitado por dos llaves que definen el cuerpo de la clase. Por ejemplo:

```
class miPunto {
}
```

Aunque la clase miPunto es sintácticamente correcta, se la conoce como una clase vacía (ya que no dispone de métodos ni propiedades).

Pero una clase debe definir las propiedades y métodos de un objeto. La sintaxis de una definición de clase es:

```
class Nombre_De_Clase {
   tipo_de_variable nombre_de_propiedad1;
   tipo_de_variable nombre_de_propiedad2;
   // . . .
   tipo_devuelto nombre_de_metodo1( lista_de_parametros ) {
      cuerpo_del_metodo1;
   }
   tipo_devuelto nombre_de_metodo2( lista_de_parametros ) {
      cuerpo_del_metodo2;
   }
   // . . .
}
```

Los tipos tipo_de_variable y tipo_devuelto deben ser tipos simples Java o nombres de otras clases ya definidas anteriormente.

Las propiedades se definen en el interior de la clase declarando variables. Por ejemplo, a continuación se añaden a la clase miPunto dos propiedades, x e y:

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

Las propiedades se pueden declarar con dos clases de tipos de datos: un tipo simple del lenguaje o el nombre de una clase (será una referencia a un objeto).

Los métodos son funciones que definen la interfaz de una clase, sus capacidades y comportamiento. Éstos se declaran al mismo nivel que las variables dentro de la definición de clase.

En la declaración de los métodos se define el tipo de valor que devuelven y una lista de parámetros de entrada separados por comas. En el siguiente ejemplo, el método devuelve la suma de dos enteros:

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

En el caso de que no se desee devolver ningún valor, se deberá indicar como tipo la palabra reservada *void*. Asimismo, si no se desean parámetros, la declaración del método debería incluir un par de paréntesis vacíos:

```
void metodoVacio( ) { };
```

Los métodos son llamados indicando una instancia individual de la clase, que tendrá su propio conjunto único de variables de instancia, por lo que los métodos se pueden referir directamente a ellas. El siguiente método inicia() se utiliza para establecer los valores de las dos variables de instancia:

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

A continuación, vamos a presentar un ejemplo completo en el que se define una clase con métodos y propiedades. Las principales características de esta clase son las siguientes:

- La clase modela un punto de espacio, por lo tanto, dispone de dos propiedades que definen sus coordenadas.

- La clase dispone de dos métodos. El primero realiza la suma de las dos coordenadas y devuelve el valor; el segundo calcula la distancia entre el propio punto y las coordenadas del punto pasado como parámetro.

```
//Definimos una clase miPunto con:
//un método metodoSuma, que nos retorna la suma de las dos coordenadas que se pasan como parámetros
//un método distancia que calcula la distancia entre el punto pasado como parámetro y el punto
//del objeto
//un método metodoVacio ¡que no hace nada!

class miPunto {
 //Definimos dos propiedades x e y, que definen las coordenadas del objeto punto
  int x, y;

 //Definimos un método que retorna el valor de la suma de los dos valores enteros que se le pasan
 //como parámetros
```

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

 //Definimos una función que nos retorna la distancia entre el punto objeto y el que se pasa
 //por parámetro
    doble distancia( int x, int y ) {
              //Asignamos a la variable dx la diferencia entre la coordenada x del objeto
              (this.x) y la coordenada x pasada por el parámetro
     int dx= this.x - x;
     //Asignamos a la variable dy la diferencia entre la coordenada y del objeto (this.y)
     //y la coordenada y pasada por el parámetro
              int dy = this.y - y;
              //Retornamos el resultado de la raíz cuadrada de la suma de dx y dy, por lo
              //tanto, la distancia entre dos puntos
     return Math.sqrt( dx*dx + dy*dy );
    }
    //Definimos un método vacío que no hace nada.
    void metodoVacio( ) { }

 //Definimos un método que inicializa el objeto con el valor que pasamos como parámetro
 //de forma que ya tenemos el primer objeto miPunto con las coordenadas definidas
    void inicia( int paramX, int paramY ) {
             x = paramX;
             y = paramY;
 }

 //Definimos un nuevo método inicializador, pero al no poner la palabra this, no se está
 //asignando a la variable de instancia, sino a la misma
 //que tenemos en el método, por lo que en este método realmente solo se modifica la variable y
    void inicia2( int x, int y ) {
x = x;
this.y = y;
    }

 //Definimos un método constructor miPunto, donde tenemos dos parámetros que se asignan a las
 //propiedades de la instancia x e y
    miPunto( int paramX, int paramY ) {
             this.x = paramX;   // En esta línea this se puede obviar, ya que x corresponde a
             //la variable de la función, pero como se trata
        // del constructor, es la propia instancia de la clase
             y = paramY;   // Esta línea es equivalente a la anterior
    }

 //Definimos un método constructor miPunto, donde como no recibimos ningún parámetro se
 inicializa con las coordenadas (-1,-1)
    miPunto() {
```

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

2.1.2. Objetos

Existen muchas definiciones del concepto *objeto*. En primer lugar, en el mundo real un objeto es cualquier cosa que vemos a nuestro alrededor. En este mismo momento estás delante de unos materiales en formato papel o digital, ambos son objetos, como lo son el teléfono móvil, la televisión o un vehículo.

Si analizamos un objeto del mundo real como un ordenador portátil, observamos que éste está formado por los siguientes componentes: la placa base, el procesador, el disco duro y los módulos de memoria. Es el trabajo coordinado de todos los componentes lo que provoca que el ordenador realice sus funciones.

Cada uno de estos componentes es sumamente complejo, está fabricado por diferentes compañías con distintos materiales e incluso con distintos diseños. Pero no es necesario conocer cómo funcionan internamente cada uno de ellos, cada uno es una unidad autónoma y todo lo que se necesita saber de su funcionamiento es cómo interactúan entre sí cada uno de los componentes. Un procesador y un módulo de memoria son compatibles con la placa base si su interacción es correcta.

La programación orientada a objetos funciona de la misma manera, los programas están construidos en función de diferentes componentes, cada uno de ellos desempeña un papel específico y todos los componentes pueden comunicarse entre ellos de maneras definidas.

Un **objeto** es una unidad de código formado por propiedades que definen su estado y métodos que definen su comportamiento.

En el mundo real, todo objeto tiene dos componentes: **características** y **comportamiento**. En el caso del ejemplo del vehículo, tenemos las siguientes características o propiedades: marca, modelo, color, velocidad máxima, etc.; y el comportamiento se basaría en las siguientes acciones: frenar, acelerar, retroceder, llenar combustible, cambiar llantas, etc.

En el mundo de la programación, los objetos también tienen características y comportamientos; de esta manera, los objetos en la programación almacenan sus características en variables e implementan su comportamiento mediante funciones.

Ejemplo

Imaginemos estacionado en el garaje un Ford Focus de color azul capaz de llegar a 180 km/h; si se pasa al mundo de la programación, podemos considerar un objeto Automóvil con las siguientes propiedades:

- Marca = Ford.
- Modelo = Focus.
- Color = Azul.
- Velocidad = 180 km/h.

Cuando a las propiedades del objeto se le asignan valores, el objeto adquiere un estado; de este modo, las variables almacenan los estados de un objeto en un determinado momento.

Siguiendo con la clase planteada en el subapartado anterior, a continuación se puede observar el proceso de creación de un objeto o instanciación de la clase.

Cada vez que se crea una clase, se añade otro tipo de dato que se puede utilizar igual que uno de los tipos simples de datos, por este motivo, al declarar una nueva variable se puede utilizar un nombre de clase como indicador de tipo:

```
miPunto p;
```

Es una declaración de una variable *p*, que es una referencia a un objeto de la clase miPunto, de momento con el valor por defecto null.

Las clases implementan un método especial llamado constructor, este método inicia el objeto inmediatamente después de su creación y tiene exactamente el mismo nombre de la clase que lo implementa; es decir, no puede haber ningún otro método que comparta su nombre con el de su clase. Una vez definido, se llama automáticamente al constructor cuando se crea un objeto de esa clase (al utilizar el operador new).

El constructor no devuelve ningún tipo, ni siquiera void. Su misión es iniciar todo estado interno de un objeto (sus propiedades), haciendo que el objeto sea utilizable inmediatamente; reservando memoria para sus variables, iniciando sus valores...

Por ejemplo:

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

Este constructor denominado "constructor por defecto", al no tener parámetros, establece el valor –1 a las variables de instancia x e y de los objetos que construya.

Cuando no encuentre un constructor de la clase, el compilador llamará al constructor de la superclase que, en caso de no existir, será el de la clase Object().

Este otro constructor, sin embargo, recibe dos parámetros:

```
miPunto( int paraX, int paraY ) {
   inicia( paraX, paraY );
}
```

La lista de parámetros especificada después del nombre de una clase en una sentencia new se utiliza para pasar parámetros al constructor. Se llama al método constructor justo después de crear la instancia y antes de que new devuelva el control al punto de la llamada.

El operador new crea una instancia de una clase (un objeto) y devuelve una referencia a ese objeto. Por ejemplo:

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

Este ejemplo crea una instancia de miPunto() inicializada por los valores (2,3) y es referenciado por la variable *p2*.

Las referencias a objetos realmente no contienen los objetos a los que referencian, sino la dirección en la que el objeto es almacenado; de esta manera se pueden crear múltiples referencias a un mismo objeto, por ejemplo:

```
miPunto p3 = p2;
```

Sólo se ha creado un objeto miPunto, pero existen dos variables (*p2* y *p3*) que lo referencian. Cualquier cambio realizado en el objeto referenciado por *p2* afectará al objeto referenciado por *p3*. La asignación de *p2* a *p3* no reserva memoria ni modifica al objeto.

La siguiente asignación al valor null de p2 tiene como efecto que p2 ya no apuntará al objeto, pero este último aún existe y además p3 aún hace referencia a él:

```
p2 = null; // p3 todavía apunta al objeto creado con new
```

Cuando no exista ninguna variable que haga referencia a un objeto, el intérprete de Java libera automáticamente la memoria utilizada por el objeto (este proceso se conoce como **Garbage collector**).

Cuando se realiza una instancia de una clase (mediante new) se reserva en la memoria un espacio para el conjunto de datos que definen los atributos de la clase que se indica en la instanciación. A este conjunto de variables se le denomina **variables de instancia**.

La potencia de las variables de instancia se basa en que se crea un conjunto distinto de ellas para cada uno de los objetos nuevos creados, es decir, cada objeto tiene su propia copia de las variables de instancia de su clase, por lo que los cambios sobre las variables de instancia de un objeto no tienen efecto sobre las variables de instancia de otro objeto.

Acceso al objeto

El operador punto (.) se utiliza para acceder a las propiedades de la instancia y a los métodos, mediante su referencia a objeto:

```
referencia_a_objeto.nombre_de_variable_de_instancia;
referencia_a_objeto.nombre_de_metodo( lista-de-parametros );
```

En el siguiente ejemplo se puede observar el uso de la sintaxis anterior:

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

De esta manera se crea un objeto miPunto con las coordenadas (100,200), pero a continuación se llama al método inicia que actualiza las anteriores coordenadas a (300,400).

2.1.3. Herencia

La herencia es uno de los conceptos más importantes en la programación orientada a objetos y consiste en la posibilidad de creación de clases a partir de otras existentes. Lo que hace tan potente la herencia es que la nueva clase puede heredar de la primera sus propiedades y sus métodos (aparecen así los conceptos de clase padre o superclase y clase hija o subclase).

De esta manera, una subclase puede tener incorporados las propiedades y métodos heredados de la clase padre y, además, puede añadir propiedades y métodos propios a los heredados.

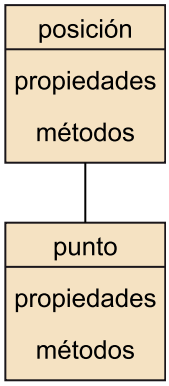
En el proceso de fabricación de modelos de vehículos se utiliza de manera intensiva la herencia; por ejemplo, a partir de una base de chasis un mismo fabricante construye modelos distintos que comparten propiedades y comporta-

mientos, con la particularización de que se añaden nuevas propiedades y comportamientos que los definen como un nuevo modelo, aunque, en realidad, gran parte de su estructura interna es común y compartida por otros modelos.

La herencia es un mecanismo utilizado para definir similitud entre clases, simplificando definiciones de clases similares previamente detenidas. Existen **dos tipos de herencia**:

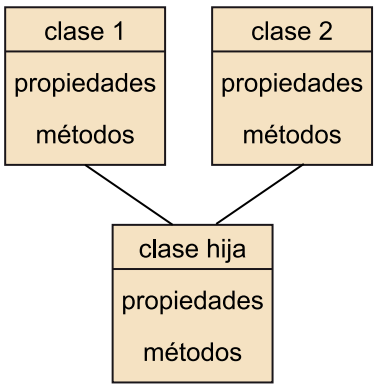
- La **herencia simple** se produce cuando el lenguaje sólo permite que una clase derive de una clase. En el siguiente ejemplo se observa cómo la clase punto hereda de la clase posición:

Figura 2



- La **herencia múltiple** se produce cuando una clase puede ser derivada de más de una clase. El siguiente ejemplo muestra cómo a partir de dos clases que definen objetos de sonido e imagen se puede crear una nueva clase que herede de las dos anteriores:

Figura 3



La **herencia** permite la reutilización de código de forma muy simple para el programador.

Para indicar que una clase hereda de otra, heredando tanto sus propiedades como sus métodos, en Java se usa el término *extends*, como en el siguiente ejemplo:

```
public class SubClase extends SuperClase {
   // Contenido de la clase
}
```

Por ejemplo, a continuación se crea una clase miPunto3D, que va a heredar de la clase miPunto:

```
class miPunto3D extends miPunto {
   int z;
   miPunto3D( ) {
      x = 0; // Heredado de miPunto
      y = 0; // Heredado de miPunto
      z = 0; // Nuevo atributo
   }
}
```

La palabra clave *extends* se utiliza para indicar que se va a crear una subclase de la clase que es nombrada a continuación; en el ejemplo anterior, miPunto3D es hija de miPunto.

La clase Object

Es la superclase de todas las clases de Java. Todas las clases derivan, directa o indirectamente de ella. Si al definir una nueva clase no aparece la cláusula extends, Java considera que dicha clase desciende directamente de Object.

La clase Object aporta una serie de métodos básicos comunes a todas las clases:

- public boolean equals(Object obj): se utiliza para comparar, en valor, dos objetos. Devuelve true si el objeto que recibe por parámetro es igual, en valor, que el objeto desde el que se llama al método. Si se desean comparar dos referencias a objeto se pueden utilizar los operadores de comparación == y !=.
- public int hashCode(): devuelve un código hash para ese objeto, para poder almacenarlo en una Hashtable.
- protected Object clone() throws CloneNotSupportedException: devuelve una copia de ese objeto.
- public final Class getClass(): devuelve el objeto concreto, de tipo Class, que representa la clase de ese objeto.
- protected void finalize() throws Trowable: realiza acciones durante la recogida de basura.

2.2. Características de la programación orientada a objetos

2.2.1. Abstracción

La abstracción es un ejercicio mental por el cual, a partir de un objeto complejo, somos capaces de extraer las propiedades y el comportamiento esencial, dejando de lado aquellos aspectos que no son relevantes.

Ejemplo

Al describir el cuerpo humano, se refiere a la cabeza, brazo(s), pierna(s), etc.

Gracias a la abstracción se pueden representar las características esenciales de un objeto sin preocuparse de las características restantes (no esenciales). La abstracción se centra en la vista externa de un objeto, de modo que sirva para separar el comportamiento esencial de un objeto de su implementación.

En los lenguajes de programación orientada a objetos, el concepto de Clase es la representación y el mecanismo por el que se gestionan las abstracciones.

En POO, se puede considerar a una Persona, por ejemplo, como un objeto que tiene propiedades (como nombre, altura, peso, color de pelo, color de ojos, etc.) y métodos (como hablar, mirar, andar, correr, parar, etc.).

Con la abstracción, un objeto Tren puede manipular objetos Persona sin tener en cuenta sus propiedades ni métodos, ya que sólo le interesa, por ejemplo, calcular la cantidad de personas que están viajando en él en ese momento, sin tener en cuenta ninguna otra información relacionada con dichas personas, tales como la altura, el nombre, el color de ojos, etc.

Hay situaciones en las que se necesita definir una clase que represente un concepto abstracto, y por lo tanto no se pueda proporcionar una implementación completa de algunos de sus métodos.

Cuando creamos una clase, podemos declarar un determinado método como abstracto. Haciéndolo, forzamos a que cualquier subclase haya de implementar obligatoriamente este método, o bien declararlo igualmente abstracto.

Cualquier clase que contenga métodos declarados como abstract también se debe declarar como abstract, y no se podrán crear instancias de dicha clase.

A continuación, se presenta un ejemplo de clases abstractas:

```
abstract class claseA {
   abstract void metodoAbstracto();
   void metodoConcreto() {
      //El método concreto de claseA;
   }
}
class claseB extends claseA {
```

```
    void metodoAbstracto(){
       //El método abstracto de claseB;
    }
}
```

La clase abstracta claseA ha implementado el método concreto metodoConcreto(), pero el método metodoAbstracto() era abstracto y por eso ha tenido que ser redefinido en la clase hija claseB.

2.2.2. Ocultamiento o encapsulamiento

Es la capacidad de ocultar los detalles internos del comportamiento de una clase y exponer públicamente únicamente los detalles que son necesarios para el resto del sistema.

El ocultamiento permite dos características fundamentales:

- **Restringir el uso de la clase**: existirá cierto comportamiento privado de la clase que no podrá ser accedido por otras clases.

- **Controlar el uso de la clase**: existirán ciertos mecanismos para modificar el estado de la clase y es mediante estos mecanismos como se validará que algunas condiciones se cumplan.

De esta manera, cada objeto está aislado del exterior y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo se puede interactuar. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente son los propios métodos internos del objeto los que pueden acceder a su estado.

Esto asegura que no se pueda cambiar el estado interno de un objeto de maneras inesperadas, al eliminar efectos secundarios e interacciones inesperadas.

Si se continúa el ejemplo anterior de la clase Persona, donde se tiene una propiedad que se llama DNI, que contiene el número de DNI, se define un método NIF que te retorna el valor del NIF. Dentro de la clase habrá definido un método que a partir de un número de DNI retorna la letra del NIF. Ahora bien, este método no es accesible, no se puede llamar directamente para obtener solo la letra. Se ha de llamar al método NIF (que además, la operación que lo llama tampoco conoce si se trata de un método o una propiedad) y obtener directamente el valor del NIF entero.

En el lenguaje Java, todas las propiedades y métodos de una clase son accesibles desde el código de la misma clase, el control del acceso desde otras clases y de la herencia por las subclases; los miembros (atributos y métodos) de las clases tienen **tres modificadores** posibles que definen el tipo de control de acceso:

- Public: las propiedades o métodos declarados con public son accesibles en cualquier lugar en el que sea accesible la clase, y son heredados por las subclases.

- Private: las propiedades o métodos declarados private son accesibles sólo en la propia clase.

- Protected: las propiedades o métodos declarados protected son sólo accesibles para sus subclases.

En caso de que no se especifique ningún modificador, las propiedades y métodos son accesibles desde cualquier clase de las que esta forma parte.

Package

Un *package* en el lenguaje Java es un mecanismo que agrupa clases similares en un tipo de librería. Una característica interesante es que no puede haber dos clases con el mismo nombre en un mismo *package*, pero sí que puede ser en el caso de que pertenezcan a *packages* distintos.

Por ejemplo:

```
class Padre { // Hereda de Object
   // Atributos
   private int numeroFavorito, nacidoHace, dineroDisponible;
   // Métodos
   public int getApuesta() {
      return numeroFavorito;
   }
   protected int getEdad() {
      return nacidoHace;
   }
   private int getSaldo() {
      return dineroDisponible;
   }
}
class Hija extends Padre {
   // Definición
}
class Visita {
   // Definición
}
```

En el anterior ejemplo, un objeto de la clase Hija hereda los tres atributos (numeroFavorito, nacidoHace y dineroDisponible) y los tres métodos (getApuesta(), getEdad() y getSaldo()) de la clase Padre, y podrá invocarlos. Cuando se llame al método getEdad() de un objeto de la clase Hija, se devolverá el valor de la variable de instancia nacidoHace de ese objeto, y no de uno de la clase Padre.

Sin embargo, un objeto de la clase Hija no podrá invocar al método getSaldo() de un objeto de la clase Padre, con lo que se evita que el Hijo conozca el estado de la cuenta corriente de un Padre.

La clase Visita sólo podrá acceder al método getApuesta() para averiguar el número favorito de un Padre, pero de ninguna manera podrá conocer ni su saldo, ni su edad.

2.2.3. Polimorfismo

La palabra *polimorfismo* proviene del griego y significa 'que posee formas diferentes'. Éste es uno de los conceptos esenciales de una programación orientada a objetos. Así como la herencia está relacionada con las clases y su jerarquía, el polimorfismo se relaciona con los métodos.

El **polimorfismo** permite modificar el comportamiento de un operador dependiendo de los operandos.

En general, existen **tres tipos de polimorfismo**:

1) Polimorfismo de sobrecarga: ocurre cuando funciones con el mismo nombre existen con funcionalidad similar, pero en clases que son completamente independientes unas de otras (no deben ser clases hijas de la clase objeto).

Ejemplo

La clase Punto, la clase Image y la clase Link pueden todas tener la función "display". Esto significa que no es necesario preocuparse por el tipo de objeto con el que se está trabajando si todo lo que se desea es visualizarlo en la pantalla.

De esta manera, el polimorfismo de sobrecarga permite definir operadores cuyo comportamiento variará de acuerdo con los parámetros que se les apliquen. Así es posible agregar el operador + y hacer que se comporte de manera distinta, o bien cuando está haciendo referencia a una operación entre dos números enteros (suma) o bien cuando se encuentra entre dos cadenas de caracteres (concatenación).

2) Polimorfismo paramétrico: es la capacidad para definir varias funciones utilizando el mismo nombre, pero usando parámetros con diferente tipo, diferente número de parámetros o las dos cosas. De esta manera se selecciona automáticamente el método correcto que aplicar en función del tipo de datos pasados en el parámetro. Por lo tanto, se pueden definir varios métodos

homónimos de suma() efectuando una suma de valores, de modo que si los parámetros son numéricos, devuelva el valor de la suma de éstos, y si son cadenas de texto, devuelva la concatenación de las cadenas.

3) Polimorfismo de subtipado: se basa en la habilidad de redefinir un método en clases que se heredan de una clase base. Por lo tanto, se puede llamar a un método de objeto sin la necesidad de conocer su tipo intrínseco; así se permite no tomar en cuenta detalles de las clases especializadas de una familia de objetos y enmascararlos con una interfaz común (siendo ésta la clase básica).

Ejemplo

Un ejemplo podría ser un juego de ajedrez con los objetos Rey, Reina, Alfil, Caballo, Torre y Peón, cada uno heredando del objeto Pieza. El método Movimiento podría, usando polimorfismo de subtipado, hacer el movimiento correspondiente de acuerdo a la clase objeto que se llama.

Así pues, se trata de una característica que permite que una clase tenga varios procedimientos con el mismo nombre, pero con distinto tipo y/o número de argumentos. Se obtienen de esta manera comportamientos diferentes, asociados a objetos distintos, pero que comparten el mismo nombre, y al llamarlos por ese nombre se utiliza el comportamiento correspondiente al objeto que se esté usando.

Por ejemplo, una compañía paga a sus empleados semanalmente, y éstos se clasifican en cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que reciben un sueldo por hora y pago por tiempo extra, por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que reciben un porcentaje de sus ventas, y empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas.

Para este período de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, y agrega un 10% a sus salarios base. En este ejemplo, se definiría el método Pago utilizando polimorfismo de subtipado.

En el lenguaje de programación Java, el polimorfismo se puede implementar con las siguientes técnicas:

1) Selección dinámica de método

Las dos clases implementadas a continuación tienen una relación subclase/superclase simple con un único método que se sobrescribe en la subclase:

```
class claseA {
   void metodoDinamico() {
      // El método dinámico de claseA;
   }
}
```

```
class claseB extends claseA {
   void metodoDinamico() {
      // El método dinámico de claseB;
   }
}
```

Por lo tanto, si se ejecutan las siguientes sentencias:

```
claseA referenciaA = new claseB();
referenciaA.metodoDinamico();
```

se estará ejecutando el metodoDinamico definido en la claseB.

Se declara la variable de tipo claseA, y después se almacena una referencia a una instancia de la clase claseB en ella. Al llamar al método metodoDinamico() de claseA, el compilador de Java verifica que claseA tiene un método llamado metodoDinamico(), pero el intérprete de Java observa que la referencia es realmente una instancia de claseB, por lo que llama al método metodoDinamico() de claseB en vez de al de claseA.

Esta manera de polimorfismo dinámico en tiempo de ejecución es uno de los mecanismos más poderosos que ofrece el diseño orientado a objetos para soportar la reutilización del código y su robustez.

2) Sobrescritura de un método

Durante una jerarquía de herencia puede interesar volver a escribir el cuerpo de un método, para realizar una funcionalidad de diferente manera dependiendo del nivel de abstracción en el que nos encontremos. A esta modificación de funcionalidad se le llama sobrescritura de un método.

Por ejemplo, en una herencia entre una clase SerVivo y una clase hija Persona, si la clase SerVivo tuviese un método Alimentarse(), debería volver a escribirse en el nivel Persona, puesto que una persona no se alimenta ni como un Animal ni como una Planta.

La mejor manera de observar la diferencia entre sobrescritura y sobrecarga es mediante un ejemplo. A continuación, se puede observar la implementación de la sobrecarga de la distancia en 3D y la sobrescritura de la distancia en 2D.

```
class miPunto3D extends miPunto {
   int x,y,z;
   double distancia(int pX, int pY) { // Sobrescritura
      int retorno=0;
      retorno += ((x/z)-pX)*((x/z)-pX);
      retorno += ((y/z)-pY)*((y/z)-pY);
      return Math.sqrt( retorno );
```

```
    }
}
```

Se inician los objetos mediante las sentencias:

```
miPunto p3 = new miPunto(1,1);
miPunto p4 = new miPunto3D(2,2);
```

y llamando a los métodos de la siguiente manera:

```
p3.distancia(3,3); //Método miPunto.distancia(pX,pY)
p4.distancia(4,4); //Método miPunto3D.distancia(pX,pY)
```

Los métodos se seleccionan en función del tipo de la instancia en tiempo de ejecución, no de la clase en la que se está ejecutando el método actual. A esto se le llama **selección dinámica de método**.

3) Sobrecarga de método

Es posible que se necesite crear más de un método con el mismo nombre, pero con listas de parámetros distintas. A esto se le llama sobrecarga del método. La sobrecarga de método se utiliza para proporcionar a Java un comportamiento polimórfico.

Un ejemplo de uso de la sobrecarga es, por ejemplo, crear constructores alternativos en función de las coordenadas, tal y como se hacía en la clase miPunto:

```
miPunto( ) { //Constructor por defecto
   inicia( -1, -1 );
}
miPunto( int paramX, int paramY ) { // Parametrizado
   this.x = paramX;
   y = paramY;
}
```

Se llama al constructor basándose en el número y tipo de parámetros que se les pase. Al número de parámetros con tipo de una secuencia específica se le llama **signatura de tipo**. Java utiliza estas signaturas de tipo para decidir a qué método llamar. Para distinguir entre dos métodos, no se consideran los nombres de los parámetros formales sino sus tipos y su número.

```
miPunto p1 = new miPunto(); // Constructor por defecto
miPunto p2 = new miPunto( 5, 6 ); // Constructor parametrizado
```

2.2.4. Destrucción de objetos

Un **destructor** es un método de la clase que realiza la tarea opuesta a su constructor, libera la memoria que fue asignada al objeto que fue creado por el constructor. Es deseable que el destructor se invoque implícitamente cuando el objeto abandone el bloque en el que fue declarado.

El destructor le permite al programador despreocuparse de liberar la memoria que deja de utilizar y correr el riesgo de que ésta se sature.

En Java, la destrucción se puede realizar de manera automática o de manera personalizada, en función de las características del objeto.

La destrucción por defecto: recogida de basura

El intérprete de Java posee un sistema de recogida de basura que, por lo general, permite que el programador no se preocupe de liberar la memoria asignada explícitamente.

El recolector de basura es el encargado de liberar una zona de memoria dinámica que había sido reservada mediante el operador new, cuando el objeto ya no va a ser utilizado durante el programa (por ejemplo, se sale del ámbito de utilización, o no es referenciado nuevamente).

El sistema de recogida de basura se ejecuta periódicamente, buscando objetos que ya no estén referenciados.

La destrucción personalizada: finalize

En ocasiones, una clase mantiene un recurso que no es de Java como un descriptor de archivo o un tipo de letra del sistema de ventanas. En este caso, es recomendable utilizar la finalización explícita, para asegurar que dicho recurso se libera. Para especificar una destrucción personalizada se añade un método a la clase con el nombre finalize:

```
class ClaseFinalizada{
   ClaseFinalizada() { // Constructor
      // Reserva del recurso no Java o recurso compartido
   }
   protected void finalize() {
      // Liberación del recurso no Java o recurso compartido
   }
}
```

El intérprete de Java llama al método finalize() cuando ha de destruir el objeto.

2.2.5. Análisis y diseño orientado a objetos

Para el desarrollo de software orientado a objetos no basta usar un lenguaje orientado a objetos, también es necesario realizar un análisis y diseño orientado a objetos.

El modelado visual es la clave para realizar el análisis orientado a objetos, desde los inicios del desarrollo de software orientado a objetos han existido diferentes metodologías para implementar este modelado, pero, sin lugar a duda, el Lenguaje de Modelado Unificado (UML) puso fin a la guerra de metodologías.

Según los mismos diseñadores del lenguaje UML, éste tiene como fin modelar cualquier tipo de sistemas (no solamente de software) usando los conceptos de la orientación a objetos. Además, este lenguaje debe ser entendible tanto para los programadores como para las máquinas.

Actualmente, en la industria del desarrollo de software, el UML es un estándar de facto en el modelado de sistemas. Fue la compañía Rational quien creó estas definiciones y especificaciones del estándar UML, y posteriormente lo publicó en el mercado.

La misma empresa creó uno de los programas más populares para este fin: el Rational Rose; pero también existen otros programas, como Poseidon, que dispone de licencias del tipo community edition que permiten su uso libremente.

El UML consta de todos los elementos y diagramas que permiten modelar los sistemas desde el paradigma orientado a objetos. Cuando se construyen de manera correcta, los modelos orientados a objetos son fáciles de comunicar, cambiar, expandir, validar y verificar.

Este modelado en UML es flexible al cambio y permite crear componentes plenamente reutilizables.

3. POO en los distintos lenguajes de programación

El objetivo de este apartado es presentar los principales lenguajes de programación que han incorporado la POO en su implementación. Para ello, en cada uno de estos lenguajes se realizará una pequeña revisión histórica y se presentarán sus principales características.

3.1. Smalltalk

Smalltalk fue desarrollado en Xerox Parc (Palo Alto Research Center) bajo el impulso de Alan Kay durante la década de los setenta. Inicialmente debía ser un lenguaje para un ordenador personal llamado Dynabook dirigido a todo tipo de usuarios (incluidos niños). Debía ser, por lo tanto, un sistema con un entorno intuitivo y fácil de programar. Aunque el proyecto Dynabook nunca se completó, el lenguaje adquirió vida propia y continuó su camino.

Alan Kay (1940)

Informático estadounidense pionero en la programación orientada a objetos y el diseño de sistemas de interfaces de usuario. Es profesor adjunto de en la Universidad de California en Los Ángeles. Una de sus frases mas célebres es: "La mejor manera de predecir el futuro es inventarlo".

Es poco conocida la gran importancia que tuvo este desarrollo en la evolución posterior de la informática. De él parten muchas de las ideas que hoy son la base de las interfaces de usuario, como el uso de gráficos, ratón, ventanas y menús desplegables.

Smalltalk es un lenguaje orientado a objetos puro (el mismo término, si no el concepto, fue inventado por Alan Kay) e incluye todos los conceptos clave, como clases, métodos, mensajes y herencia. Todo el programa es una cadena de mensajes enviados a objetos.

Las principales características del lenguaje son:

- Orientación a objetos pura.
- Tipos dinámicos.
- Herencia simple.
- Compilación en tiempo de ejecución o interpretado.

Smalltalk es un modelo puro orientado a objetos, lo que significa que, en el entorno, todo es tratado como un objeto. Entre los lenguajes orientados a objetos, Smalltalk es el más consistente en cuanto al manejo de las definiciones y propiedades del paradigma orientado a objetos.

Se puede afirmar que es más que un lenguaje, es un entorno de desarrollo con más de doscientas clases y varios miles de métodos. Smalltalk contiene los siguientes componentes:

- Un lenguaje.

- Un modelo de objeto, que define cómo actúan los objetos e implementa la herencia, el comportamiento de clases e instancias, la asociación dinámica, el manejo de mensajes y las colecciones.

- Un conjunto de clases reutilizables, que dispone de una gran cantidad de clases que pueden ser reutilizadas en cualquier programa. Estas clases proveen las funciones básicas en el lenguaje, además del soporte para la portabilidad a diferentes plataformas, incluida la portabilidad de las interfaces gráficas de usuario.

- Un conjunto de herramientas de desarrollo, que habilitan a los programadores a mirar y modificar las clases existentes, así como a renombrar, agregar y borrar clases. También proveen de detección de errores, incluida la habilidad de agregar paradas en la ejecución, observar los valores de las variables, modificar el valor de variables en ejecución y realizar cambios al código en tiempo de ejecución de un programa.

- Un entorno en tiempo de ejecución, que permite a los usuarios terminar con el ciclo "compilado-linkeado-ejecución" de un programa. Esto permite a los usuarios ejecutar un programa en Smalltalk mientras se cambia el código fuente, de manera que los cambios realizados en el código fuente son reflejados instantáneamente en la aplicación que se está ejecutando.

Una de las mejores características de Smalltalk es el **alto grado de reutilización** del código, ya que contiene un gran conjunto de objetos que pueden ser utilizados directamente o modificados de un modo sencillo para satisfacer la necesidad de una aplicación en general.

Smalltalk **no posee una notación explícita** para describir un programa entero. Sí que se emplea una sintaxis explícita para definir ciertos elementos de un programa, tales como métodos, pero la manera en la que esos elementos están estructurados dentro de un programa entero generalmente es definida por las múltiples implementaciones.

La **sintaxis** de Smalltalk tiende a ser minimalista, lo que implica que existe un grupo reducido de palabras reservadas y declaraciones en comparación con la mayoría de los lenguajes populares. Smalltalk posee un grupo de 5 palabras reservadas: *self*, *super*, *nil*, *true* y *false*.

Las **implementaciones** utilizan técnicas de recolección de basura para detectar y reclamar espacio en memoria asociado con objetos que ya no se utilizarán más en el sistema. La manera de ejecución del recolector de basura es en background, es decir, como un proceso de baja prioridad no interactivo, aunque en algunas implementaciones es posible ejecutarlo a demanda. La frecuencia y las características de la recolección dependen de la técnica utilizada por la implementación.

Web recomendada

En la web de Smalltalk se puede encontrar información ampliada sobre el desarrollo y las características de este lenguaje.

3.2. Eiffel

Es un lenguaje de programación escrito por Bertrand Meyer. A diferencia de Smalltalk, incluye un preprocesador que permite la traducción de código Eiffel al lenguaje C. Es popular en el campo de la ingeniería de software, ya que permite la encapsulación, el control de acceso y el ámbito de las modificaciones. Por sus capacidades técnicas, es, presumiblemente, el mejor lenguaje orientado a objetos puro.

Bertrand Meyer (1950)

Estudió en la Escuela Politécnica de París, obtuvo un máster en la Universidad de Stanford y se doctoró en Filosofía en la Universidad de Nancy. Su principal vía se basa en que los lenguajes de programación deben ser simples, elegantes y fáciles de usar. Fue el diseñador inicial del lenguaje y del método Eiffel. Una de sus frases más celebres es "un elemento de software no es correcto ni incorrecto de por sí: es correcto si se comporta de acuerdo a su especificación".

En este lenguaje los programas consisten en la declaración de colecciones de clases que incluyen métodos y en los que se asocian los atributos. De esta manera, el punto primordial de un programa en Eiffel es la declaración de clases. Las clases y los atributos son accesibles a partir de la implementación de un concepto llamado característica, que es, a la vez, una agrupación de datos y una manera típica de tratarlos.

En Eiffel, una declaración de clase puede incluir:

- Una lista de características exportables.
- Una lista de las clases antecesora: clases de la que ésta hereda.
- Una lista de declaraciones de características.

Las principales **características** del lenguaje son:

- Se trata de un lenguaje orientado a objetos puro.
- Es un lenguaje de programación orientado hacia el diseño de grandes aplicaciones. Las propiedades anteriores lo hacen ideal para el diseño de aplicaciones en grupos de trabajo.
- El paso intermedio a código C se puede considerar una ventaja y no un inconveniente, ya que aquellas secciones que sean difíciles de tratar con Eiffel pueden elaborarse a partir de código C. Su compatibilidad con C asegura también su portabilidad hacia otros sistemas operativos
- El manejo de la memoria, un punto delicado en todos los lenguajes orientados a objetos, no es transparente como en el caso de Smalltalk.
- Las librerías de clases son reducidas.

Web recomendada

En la web de la compañía Eiffel Software, en la que se ofrece un entorno de desarrollo bajo la licencia GPL, se puede encontrar información ampliada sobre el desarrollo y las características de dicho lenguaje.

- Su rendimiento es mayor que el de Smalltalk, pero ante la necesidad de incluir un módulo Run-time dentro del ejecutable, su tamaño crece y su rendimiento baja.

3.3. C++

Es un lenguaje de programación diseñado a mediados de los años ochenta por Bjarne Stroustrup, de manera que amplía el lenguaje de programación C con mecanismos que permiten la manipulación de objetos. Por este motivo, el lenguaje C++ es considerado un lenguaje híbrido.

Bjarne Stroustrup (1950)

Es un científico de la computación y catedrático de Ciencias de la Computación en la Universidad A&M de Texas. Ha destacado por desarrollar el lenguaje de programación C++ y escribir el manual de referencia del lenguaje *The C++ Programming Language*.

Posteriormente, se añadieron facilidades de programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y programación orientada a objetos). Por ello se suele decir que el C++ es un **lenguaje multiparadigma**. Actualmente, existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos.

La mayor contribución que realiza C++ al C es la introducción del tipo clase, ya que las clases permiten definir conjuntos de propiedades y los métodos que las manipulan.

C++ dispone de **tres tipos de métodos constructores** que son ejecutados cuando una instancia de una clase es creada con el objetivo de inicializar o definir el estado del objeto:

- **Constructor predeterminado**. Es el constructor que no recibe ningún parámetro en la función. Si no se define ningún constructor, el sistema proporciona uno predeterminado.

- **Constructor de copia**. Es un constructor que recibe un objeto de la misma clase y realiza una copia de sus atributos. Al igual que el predeterminado, si no se define, el sistema proporciona uno.

- **Constructor de conversión**. Este constructor, recibe como único parámetro, un objeto o variable de otro tipo distinto al suyo propio. Es decir, convierte un objeto de un tipo determinado a otro objeto del tipo que se está generando.

De la misma manera que son necesarios métodos constructores, son necesarios los destructores, que no son más que funciones miembro especiales cuyo cometido es liberar los recursos que el objeto de dicha clase haya adquirido en tiempo de ejecución al expirar éste. Los destructores son invocados automáticamente al alcanzar el flujo del programa el fin del ámbito en el que está declarado el objeto.

Existen **dos tipos de destructores**:

- **Públicos**: se pueden llamar desde cualquier parte del programa.
- **Privados**: cuando no se permite la destrucción del objeto por parte del usuario.

En C++ es posible definir clases abstractas, que están diseñadas sólo como clase padre de las que se deben derivar clases hija. Una clase abstracta se usa para representar aquellas entidades o métodos que después se implementarán en las clases derivadas, pero la clase abstracta en sí no contiene ningún código específico, sólo representa los métodos que se deben implementar. Por ello, no es posible instanciar una clase abstracta, pero sí una clase concreta que implemente los métodos definidos en ella.

Existen en C++ **tres modos de herencia simple** que se diferencian en el modo de manejar la visibilidad de los componentes de la clase resultante:

- **Herencia pública**: con este tipo de herencia se respetan los comportamientos originales de las visibilidades de la clase padre en la clase hija.

- **Herencia privada**: con este tipo de herencia todo componente de la clase padre será privado en la clase hija.

- **Herencia protegida**: con este tipo de herencia, todo componente publico y protegido de la clase base será protegido en la clase derivada, y los componentes privados siguen siendo privados.

La sobrecarga de operadores en C++ es una manera de implementar polimorfismo, de este modo, se puede definir el comportamiento de un operador del lenguaje para que trabaje con tipos de datos definidos por el usuario. No todos los operadores de C++ son factibles de sobrecargar, y, entre aquellos que pueden ser sobrecargados, se deben cumplir ciertas condiciones.

Lectura recomendada

Bruce Eckel es autor de libros y artículos sobre programación. Sus obras más conocidas son *Thinking in Java* y *Thinking in C++*, dirigidas a programadores con poca experiencia en la programación orientada a objetos.

Existe una página oficial en la que se está realizando la traducción al castellano del libro de Bruce Eckel *Thinking in C++*.

3.4. ActionScript 3.0

ActionScript fue desarrollado con la finalidad en sus inicios de agregarle interactividad al formato de animación vectorial Flash. A partir de ese momento, los diseñadores Flash se vieron en la necesidad de convertirse en programadores para añadir todas las posibilidades que el lenguaje podía proporcionar.

ActionScript es un lenguaje de programación orientado a objetos, utilizado en las aplicaciones web animadas que son creadas en el entorno Flash. El lenguaje fue introducido a partir de la versión 4 de Flash y desde entonces ha evolucionado en cada una de las nuevas versiones.

Se trata de lenguaje Script basado en especificaciones estándar de industria ECMA-262, un estándar para JavaScript, de ahí que ActionScript se parezca tanto a JavaScript. La versión más extendida actualmente es ActionScript 3.0, que significó una mejora en el manejo de programación orientada a objetos al ajustarse mejor al estándar ECMA-262.

Las principales **características** del lenguaje se comentan a continuación:

- Dispone de una máquina virtual que es la encargada de la interpretación del código, independientemente de la plataforma en la que éste se ejecute.
- La sintaxis del lenguaje se ajusta al estándar ECMAScript (ECMA 262).
- Dispone de una interfaz de programación que permite un control de bajo nivel de los objetos que componen las películas Flash.
- Una API XML basada en la especificación de ECMAScript para XML (E4X) (ECMA-357 edición 2). E4X es una extensión del lenguaje ECMAScript que añade XML como un tipo de datos nativo del lenguaje.
- Un modelo de eventos basado en la especificación de eventos DOM (modelo de objetos de documento) de nivel 3.

Por lo tanto, se trata de un lenguaje de programación empotrado en las películas creadas con Flash, pero que cumple los estándares ECMAScript, como JavaScript, por lo que ambos lenguajes comparten gran parte de la sintaxis.

Web recomendada

Podéis encontrar manuales, tutoriales, artículos, etc. en la web de la comunidad de programadores de ActionScript: www.actionscript.org.

3.5. Ada

En los años setenta, el departamento de Defensa de Estados Unidos tenía proyectos que se desarrollaban en un conjunto variado de lenguajes de programación. Esta variedad suponía un cierto problema y su solución se basó en la búsqueda de un único lenguaje que cumpliese ciertas normas obligatorias. Se lanzó un concurso y, de las diferentes propuestas planteadas, en mayo de 1979 se seleccionó la propuesta planteada por Honeywell Bull, y se le dio el nombre de Ada.

Origen de Ada

El nombre se eligió en conmemoración de *lady* Ada Augusta Byron (1815-1852), condesa de Lovelace, hija del poeta lord George Byron, a quien se considera la primera programadora de la historia, por su colaboración y relación con Charles Babbage, creador de la máquina analítica.

El Departamento de Defensa y los ministerios equivalentes de varios países de la OTAN exigían el uso de este lenguaje en los proyectos que contrataban (esta obligación se conocía como el "Ada mandate"). La obligatoriedad en el caso de Estados Unidos finalizó en 1997.

La sintaxis del lenguaje esta inspirada en Pascal, por lo que es legible incluso para programadores que no conozcan el lenguaje. Se trata de un lenguaje que no escatima en la longitud de las palabras clave, ya que uno de sus principios es que un programa se escribe una vez, se modifica decenas de veces y se lee miles de veces (la legibilidad es más importante que la rapidez de escritura).

Fue diseñado con el propósito principal de generar programas con la mayor calidad posible, con el objetivo de obtener confianza por parte de los usuarios. Es posible implementar cualquier tipo de software en Ada, pero su principal uso ha sido en software de control en tiempo real y de misión crítica.

Por otro lado, Ada, como lenguaje que promueve las buenas prácticas en ingeniería del software, es muy usado en la enseñanza de la programación en muchas universidades de todo el mundo.

Se trata de un lenguaje de programación imperativo, orientado a objetos, concurrente y distribuido. Sus principales **características** son:

- Su sintaxis inspirada en Pascal es fácilmente legible.
- Es un lenguaje *case insensitive*, es decir, sus identificadores y palabras clave son equivalentes sea cual sea el uso de mayúsculas y minúsculas.
- Es un lenguaje con tipado fuerte: asigna en cada objeto un conjunto de valores claramente definido, lo que impide la confusión entre conceptos lógicamente distintos. Esto hace que el compilador detecte más errores que en otros lenguajes.
- Está preparado para la construcción de grandes programas. Para crear programas sostenibles y transportables, de cualquier tamaño, se necesitan mecanismos de encapsulado para compilar por separado y para gestionar bibliotecas.
- Dispone de mecanismos que permiten el manejo de excepciones. De esta manera, los programas son construidos por capas y se limitan las consecuencias de los errores en cualquiera de sus partes.
- Con la abstracción de datos se separan los detalles de la representación de los datos y las especificaciones de las operaciones lógicas sobre ellos para obtener mayor transportabilidad y mejor mantenimiento.
- Dispone de la capacidad de procesamiento paralelo, y así evita la necesidad de añadir estos mecanismos por medio de llamadas al sistema operativo, con lo que consigue mayor fiabilidad y transportabilidad.
- Dispone de la posibilidad de crear unidades genéricas; éstas son necesarias, ya que parte de un programa puede ser independiente del tipo de valores

Web recomendada

En GRB ADA95 se dispone de una guía básica de aprendizaje del lenguaje de programación: http://www.gedlc.ulpgc.es/docencia/NGA/index.html.

que manipular. Para ello, necesitamos que se utilice este mecanismo que permite crear partes de un programa similares a partir de una plantilla.

3.6. Perl

El creador de Perl, Larry Wall, anunció la versión 1.0 del lenguaje el 18 de diciembre de 1987. En los siguientes años, éste se expandió de manera muy rápida, aunque hasta 1991 la única documentación de Perl era una simple (y cada vez más larga) página de manual Unix.

Larry Wall (1954)

Programador, lingüista y autor, Larry Wall es conocido por ser el creador del lenguaje de programación Perl. Recibió en 1998 el primer premio de la Free Software Foundation para el avance del software libre. Es el coautor del libro *Programming Perl* (comúnmente llamado el "libro del camello"), que es el recurso básico de los programadores de Perl.

El 26 de octubre de 1995, se creó el Comprehensive Perl Archive Network (CPAN). CPAN es una colección de sitios web que almacenan y distribuyen fuentes en Perl, binarios, documentación, scripts y módulos. Originalmente, cada sitio CPAN debía ser accedido mediante su propio URL; actualmente, www.cpan.org redirecciona automáticamente a uno de los cientos de repositorios espejo de CPAN.

Perl es un lenguaje de propósito general originalmente desarrollado para la manipulación de texto y ahora es utilizado para un amplio rango de tareas que incluyen administración de sistemas, desarrollo web, programación en red y desarrollo de GUI. Se previó que fuera práctico (facilidad de uso, eficiente, completo) en lugar de hermoso (pequeño, elegante, mínimo).

Sus principales características son que es fácil de usar, que soporta tanto la programación estructurada como la programación orientada a objetos y la programación funcional y que tiene incorporado un poderoso sistema de procesamiento de texto y una enorme colección de módulos disponibles.

La estructura completa de Perl deriva ampliamente del lenguaje C, por lo tanto, es un lenguaje imperativo, con variables, expresiones, asignaciones, bloques de código delimitados por llaves, estructuras de control y subrutinas.

Perl también toma características de la programación shell, por lo que dispone de muchas funciones integradas para tareas comunes y para acceder a los recursos del sistema.

En la versión 5 de Perl se añadieron características para soportar estructuras de datos complejas, funciones de primer orden (por ejemplo, clausuras como valores) y un modelo de programación orientada a objetos. Éstos incluyen referencias, paquetes y una ejecución de métodos basada en clases y la introducción de variables de ámbito léxico, que hizo más fácil escribir código robusto. Una característica importante introducida en Perl 5 fue la habilidad de empaquetar código reutilizable en estructuras de módulos.

Todas las versiones de Perl implementan tipificado automático de datos y gestión de memoria. De esta manera, el intérprete conoce el tipo y los requerimientos de almacenamiento de cada objeto en el programa; reserva y libera espacio para ellos según sea necesario. Las conversiones legales de tipo se realizan de manera automática en tiempo de ejecución; las conversiones ilegales son consideradas errores fatales.

Se ha usado desde los primeros días de la Web para escribir guiones (scripts) CGI. Es una de las "tres Pes" (Perl, Python y PHP), que son los lenguajes más populares para la creación de aplicaciones web, y es un componente integral de la popular solución LAMP para el desarrollo web.

Proyectos importantes escritos en Perl son Slash, IMDb y UseModWiki; además, sitios web con un nivel alto de tráfico, como Amazon.com y Ticketmaster.com utilizan el lenguaje. Además, es ampliamente usado en finanzas y bioinformática, donde es apreciado por su desarrollo rápido, tanto de aplicaciones como de despliegue, así como la habilidad de manejar grandes volúmenes de datos.

Lectura recomendada

Tenéis a vuestra disposición en línea un pdf que introduce al programador en el modelo de programación orientada a objetos del lenguaje Perl: www.gwolf.org/files/poo_perl.pdf.

3.7. PHP

PHP fue originalmente diseñado en Perl, basándose en la escritura de un grupo de CGI binarios escritos en el lenguaje C por el programador danés-canadiense Rasmus Lerdorf en el año 1994 para mostrar su *curriculum vitae* y guardar ciertos datos, como la cantidad de tráfico que su página web recibía. El 8 de junio de 1995 fue publicado "Personal Home Page Tools" después de que Lerdorf lo combinara con su propio Form Interpreter para crear PHP/FI.

Rasmus Lerdorf (1968)

Programador nacido en Groenlandia al que se considera uno de los más importantes creadores de PHP. En 1995, Lerdorf creo un CGI en Perl que mostraba el número de visitas que había obtenido su página web personal, este script lo llamó PHP (Personal Home Page) y fue este pequeño script el detonante del nuevo lenguaje script. Creó una lista de correo para intercambiar opiniones, sugerencias y correcciones que provocó la base que acabó formalizando a PHP como una herramienta de software libre donde el aporte de la comunidad mundial ha provocado que sea una de los lenguajes de programación web más utilizados.

PHP es un lenguaje de programación diseñado especialmente para desarrollo web que también puede ser incrustado en las propias páginas de la Red; por otra parte, es interpretado por el servidor, que normalmente devolverá como resultado una nueva página web. La sintaxis de PHP es similar a la de los lenguajes Perl y C, esto provoca que la curva de aprendizaje del lenguaje sea muy corta.

Cuando el cliente realiza una petición al servidor para que le envíe una página web, el servidor ejecuta el intérprete de PHP. Éste procesa el script y genera el contenido de manera dinámica (por ejemplo, obteniendo información de

una base de datos). El resultado es enviado por el intérprete al servidor, quien, a su vez, se lo envía al cliente. Mediante extensiones es también posible la generación de archivos PDF, Flash, así como imágenes en diferentes formatos.

PHP es portable y, por lo tanto, se puede ejecutar en la mayoría de los sistemas operativos: UNIX, Linux, Mac OS X y Windows.

A continuación, presentamos las principales **características** del lenguaje:

- Se trata de un lenguaje multiplataforma.
- Está completamente orientado a la web.
- Tiene capacidad de conexión con la mayoría de los motores de base de datos que se utilizan en la actualidad; en este sentido, destaca su conectividad con MySQL y PostgreSQL.
- Posee capacidad de expandir su potencial utilizando la enorme cantidad de módulos de los que dispone.
- Es software libre, por lo que se presenta como una alternativa de fácil acceso.
- Implementa el paradigma de la programación orientado a objetos.
- Dispone de una biblioteca nativa de funciones sumamente amplia e incluida.
- No requiere definición de tipos de variables, aunque sus variables se pueden evaluar también por el tipo que estén manejando en tiempo de ejecución.
- Implementa la capacidad de manejo de excepciones.
- La ofuscación de código es el único modo de ocultar las fuentes.

Web recomendada

En la página oficial de PHP encontraréis más información sobre dicho lenguaje: www.php.net.

3.8. C#

Los primeros rumores de que Microsoft estaba desarrollando un nuevo lenguaje de programación surgieron en 1998, en referencia a un lenguaje que entonces llamaban COOL y que decían que era muy similar a Java. En junio del 2000, Microsoft despejó todas las dudas liberando la especificación de un nuevo lenguaje llamado C#. A esto le siguió rápidamente la primera versión de prueba del entorno de desarrollo estándar .NET, que incluía un compilador de C#.

C# es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, que es similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

C#, como parte de la plataforma .NET, está normalizado por ECMA desde diciembre del 2001 (ECMA-334 "Especificación del lenguaje C#").

El 7 de noviembre del 2005 se publicó la versión 2.0 del lenguaje, que incluía mejoras tales como tipos genéricos, métodos anónimos, iteradores, tipos parciales y tipos anulables. El 19 de noviembre del 2007 se publicó la versión 3.0 de C#, entre cuyas mejoras destacaban los tipos implícitos, los tipos anónimos y el LINQ (*Language Integrated Query*; consulta integrada en el lenguaje).

Aunque C# forma parte de la plataforma .NET (ésta es una interfaz de programación de aplicaciones), se trata de un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma.

Microsoft

.NET es la propuesta de Microsoft a la programación en entornos web y nace con el objetivo de competir con la plataforma Java. Como la mayoría de productos de Microsoft su gran baza es que proporciona una manera rápida y económica de desarrollar aplicaciones.

Actualmente, existe un compilador GNU de C# (Mono) que genera programas para distintas plataformas como Win32, UNIX y Linux.

Web recomendada

Encontraréis más información sobre el lenguaje C# en la página Centro de desarrolladores en C#, dirigida por Microsoft: http://msdn.microsoft.com/es-es/vcsharp/default.aspx.

3.9. Java

Los inicios de Java se remontan al año 1991, cuando un grupo de ingenieros dirigido por Patrick Naughton y James Gosling quería diseñar un pequeño lenguaje de programación que pudiera ser usado para dispositivos de consumo como los equipos de televisión por cable (este proyecto se llamó Green Project); dado que estos dispositivos no tienen una gran capacidad de memoria, el lenguaje debía ser simple y generar código muy reducido.

James Gosling (1956)

Doctorado por la Universidad de Carnegie Mellon y conocido como el creador del lenguaje de programación Java; realizó el diseño original, la implementación del compilador original y la máquina virtual Java, por lo que fue elegido miembro de la Academia Nacional de Ingeniería de Estados Unidos (NAE).

Por otro lado, los fabricantes de este tipo de dispositivos electrónicos suelen cambiar los chips con bastante frecuencia. La aparición de un nuevo chip más barato y, generalmente, más eficiente conduce a dichos fabricantes a incluirlo en las nuevas series de sus cadenas de producción, ya que esta diferencia de precio, por pequeña que sea, puede generar un ahorro considerable en dispositivos de tirada masiva.

Si se usaban lenguajes como C o C++, debían compilar todos los programas con el compilador de ese nuevo chip y esto encarecía los desarrollos. Por lo tanto, dado que los fabricantes podían elegir diferentes CPU, era importante no estar atado a una sola arquitectura: era muy importante la portabilidad. Si se conseguía un lenguaje que produjese un código independiente de la CPU, se evitaba la necesidad de compilar todos los programas que existían (para diferentes aparatos electrónicos) cuando apareciese una nueva CPU. Simplemente deberían contar con un intérprete de ese código para la nueva CPU (que podía darse a los fabricantes si el lenguaje se popularizaba suficientemente).

En abril de 1991, Gosling empezó a trabajar en el nuevo lenguaje de Green Project, decidió que las ventajas aportadas por la eficiencia de C++ no compensaban los grandes costes de pruebas y depuración del código. Como dijo Gosling, "el lenguaje era una herramienta, no el fin", así que desarrolló un lenguaje de programación que, aun partiendo de la sintaxis de C++, intentaba remediar los aspectos de C++, que eran la causa de la mayoría de los problemas.

Gosling decidió llamar a su lenguaje Oak (se supone que refiriéndose a un roble que estaba enfrente de la ventana de su lugar de trabajo en Sun). Los primeros programas en Oak se ejecutaron en agosto de 1991. Más tarde, en Sun, se dieron cuenta de que ya había un lenguaje llamado Oak y lo rebautizaron como Java (en inglés norteamericano significa *café*; el equipo que desarrollaba este lenguaje se reunía en una cafetería cercana a las instalaciones de Sun para discutir distendidamente el proyecto).

En 1992, el Green Project lanzó su primer producto, llamado "*7" (Star Seven). Era una especie de mezcla entre PDA y control remoto extremadamente inteligente, diseñado para realizar un control integrado de un hogar con todos los aparatos electrónicos que lo componen. El sistema presentaba una interfaz basada en la representación de la casa y el control se llevaba a cabo mediante una pantalla táctil.

En el sistema aparecía Duke, la actual mascota de Java. Desafortunadamente, no hubo mucho interés por él y en ese momento el equipo de Green Project se embarcó en un concurso convocado por la Time Warner para diseñar un equipo para la televisión por cable que fuera capaz de ocuparse de nuevos servicios de cable como el vídeo bajo demanda. Es decir, se aplicaba Java a la interfaz de la televisión interactiva.

Ninguno de estos dos proyectos se convirtió en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo y le sirvieron como bautismo de fuego.

A mediados de 1994, la popularidad de la web atrajo la atención de los directivos de Sun. Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podría llegar a ser el campo de juego adecuado para disputar a Microsoft su supremacía casi absoluta en el terreno

del software, y vio en Oak/Java el instrumento idóneo para llevar a cabo estos planes. Se dio cuenta de que los requisitos para el software de los dispositivos electrónicos y los equipos de televisión (*set top boxes*) eran los mismos que los requisitos para la web (código sencillo, independiente de plataforma, seguro y fiable).

Decidieron programar un navegador empleando la tecnología Java. Aquel primer programa, llamado WebRunner, quedó listo en mayo de 1995 (Patrick Naughton escribió un prototipo de este navegador en un fin de semana de inspiración) y, al ver sus enormes posibilidades, decidieron mejorar el navegador. El 23 de mayo de 1995, en la Sun World 95 de San Francisco, Sun presenta su nuevo navegador HotJava y Netscape anuncia su intención de integrar Java en su navegador.

A partir de aquel verano, los acontecimientos se desarrollan vertiginosamente para el mundo Java, sobre todo después del lanzamiento y distribución libre del Java Development Kit (JDK) 1.0. A finales de 1995 (¡sólo seis meses después del lanzamiento de JDK!), Java había firmado acuerdos con las principales firmas de software para que pudieran utilizar Java en sus productos, entre otras, Netscape, Borland, Mitsubishi Electronics, Dimension X, Adobe, IBM, Lotus, Macromedia, Oracle, SpyGlass, etc., pero lo más espectacular fue el anuncio por parte de Bill Gates, presidente y director ejecutivo de Microsoft, el 7 de diciembre de 1995, de la voluntad por parte de Microsoft de obtener la licencia de utilización de Java.

Aquel anuncio mostraba claramente que Microsoft consideraba a Java como una parte importante en la estrategia global de Internet. Este anuncio es significativo si se considera el desprecio que meses antes Bill Gates había mostrado hacia Java, cuando se refirió a éste como "un lenguaje más". El propio director general de Microsoft en España había calificado a Java como "un lenguaje para tostadoras".

Durante 1996, se planteó el debate de crear un "terminal tonto" llamado NC (Network Computer) que únicamente sirviera para conectarse a la World Wide Web. En un principio, se proyectó que este terminal estaría gobernado por un sistema operativo Java. La idea del NC no cuajó como se esperaba y apareció otra intermedia entre el NC y el PC: el NetPC, que tampoco tuvo demasiado éxito. Sin embargo, la idea de producir un sistema operativo sí que se ha desarrollado, se llama JavaOS.

El lema de Java es "escribe una vez y ejecútalo en cualquier sitio". La idea principal que transmite este lema es la portabilidad de Java: una vez escrito el código fuente y traducido a bytecode, puede ejecutarse en cualquier máquina con cualquier sistema operativo (incluso aunque no sea un ordenador), sin necesidad de recompilarlo.

Para conseguir la portabilidad, el código fuente Java se "compila" para una máquina ficticia llamada Máquina Virtual Java (JVM o *Java Virtual Machine*), lo que genera un código llamado código de octetos o bytecode. El código fuente también es portable, pero usar ese código intermedio llamado código de octetos tiene varias ventajas:

- Permite un mayor rendimiento, ya que gran parte del proceso de traducción del código fuente a unas instrucciones de una CPU específica ya está realizado.

- Permite mantener en secreto el código fuente original, lo que puede ser importante en cierto tipo de programas, en los que se desea que sean portables y a la vez difíciles de manipular, curiosear, copiar, etc. También puede extrapolarse otra idea importante: en cualquier sitio (navegadores) se puede ejecutar código Java (applets) sin importar de dónde provenga, ya que no hay grandes peligros de seguridad (como pueden ser virus o programas que atenten contra la privacidad).

Básicamente, la idea que propone es que el usuario sólo necesita tener a su lado un mero elemento de interacción (a veces llamado "terminal tonto") sin demasiada potencia de procesamiento o capacidad de almacenamiento (por ejemplo, carente de disco duro) y que el verdadero ordenador (elementos de computación y almacenamiento) puede estar distribuido en una red.

Desde el principio Java fue diseñado como un lenguaje orientado a objetos. Los objetos agrupan en estructuras encapsuladas tanto sus datos como los métodos (o funciones) que manipulan esos datos. La tendencia del futuro, a la que Java se suma, apunta hacia la programación orientada a objetos, especialmente en entornos cada vez más complejos y basados en una red.

Web recomendada

En la web oficial de Sun podéis obtener más información sobre el lenguaje de programación Java: java.sun.com.

3.10. JavaScript

JavaScript es un lenguaje que se introdujo en la versión 2.0 del Netscape Navigator, a principios de 1996, y que Microsoft aceptó más tarde para que su Internet Explorer ganara cuota de mercado, aunque ambas versiones tienen características incompatibles.

Ved también

Las características de la orientación a objetos en JavaScript se estudian en detalle en el módulo "Orientación a objetos en JavaScript".