Francisco Javier Pavón Molina
UB2638SIS6485

Principles of programming II

Object Oriented Programming

School of Science & Engineering

Major: Information Systems

Atlantic International University
Honolulu, Hawaii
March 2006

Contenido

1. Introducción

En este ensayo titulado “Principles of programming: Object Oriented Programming” pretendo dar un recorrido a este paradigma de programación actual que es utilizado por miles de programadores a nivel mundial, y que se ha convertido en un estándar en el desarrollo de aplicaciones tanto de escritorio, Web e Internet y dispositivos móviles.

A lo largo del presente texto se ira describiendo el origen, evolución, tendencias actuales y ciertos patrones de diseño, así como una vista aunque no muy profunda si lo mas fiel posible a un lenguaje de programación que ha revolucionado el presente y porque no decirlo también, el futuro de la programación actual, el lenguaje Java.

¿Por qué Java?, en lo que va de mi corta pero interesante carrera profesional en tecnologías de información he tenido contacto con un pequeño numero de lenguajes, desde Pascal (del cual desarrollé el ensayo de Principles of programming I), donde se trató la programación estructurada, dando una minúscula exploración a Delphi, lenguaje C y haciendo uso de herramientas propietarias en las cuales he tenido mas experiencia y profundización como ser la suite Microsoft Visual Studio 6.0 en el que comencé a dar mis primeros pasos con este estilo de programación de objetos y el cual en la actualidad sigo utilizando, haciendo un cambio a la plataforma .Net y lo que se ha aplica actualmente el concepto de framework[1] de desarrollo, existiendo varias tendencias o estilos; aunque las herramientas antes mencionadas proveen de cierta facilidad y productividad al programador y desarrollador de aplicaciones, tienen la limitación de la plataforma y ser propietarias[2] y no en muchas universidades se encuentran disponible para la enseñanza, es aquí donde entra Java.

Como se ira viendo, Java tiene la ventaja de ser multiplataforma y detallaré más a fondo estas características en los siguientes apartados, y si uno como profesional aplica para obtener un trabajo en el área de la programación, es casi una obligación desde un punto de vista personal conocer y desempeñarse en ambas plataformas y certificarse[3], para ofrecer y brindarle mayores soluciones a la empresa, organización o individuo que requiera de sus servicios.

También se pretende dar una pequeña crítica a este estilo de programación, porque aunque todo tiene sus ventajas y desventajas, y como dije al principio, la comunidad de desarrolladores es muy amplia, hay muchos de los cuales aun no están del todo conformes y prefieren todavía otros estilos y metodologías.

1 *En desarrollo de software, un* ***framework*** *es definido como una estructura de apoyo en la cual otro proyecto de software puede ser organizado y desarrollado.*
2 *El uso de herramientas propietarias, generalmente ligadas a tecnologías específicas bajo el control de determinadas empresas, contribuye a lograr una fuerte dependencia del futuro profesional en las mismas.*
3 *Actualmente, certificarse es un requisito en el mercado tecnológico. La competencia es cada vez mayor y debes estar preparado para los cambios del mercado.*

Las prácticas y desarrollos se harán con el JRE 5.0 Update 6 y como IDE me he decidido por Eclipse Project del cual también detallaré mas adelante.

Agradecer a Atlantic International University por brindarme la oportunidad de poder mostrar mis conocimientos y adquirir y profundizar los nuevos que he elegido en vista a un mejor crecimiento profesional.

2. Paradigma de la Programación

Un paradigma es una forma de representar y manipular el conocimiento. Representa un enfoque particular o filosofía para la construcción o ingeniería de software. No es mejor uno que otro sino que cada uno tiene sus ventajas y desventajas. También hay situaciones donde un paradigma es mejor que otro.

Un paradigma provee (y determina) la vista que el programador tiene de la ejecución del programa. Por ejemplo, en un programa orientado a objetos, los programadores pueden pensar en un programa como una colección de interacción de objetos, mientras en un programa funcional un programa puede ser pensado como una secuencia de evaluaciones de función sin estado.

Tal como diferentes grupos en ingeniería de software abogan por las diferentes metodologías, los distintos lenguajes de programación abogan por los diferentes paradigmas de programación. Algunos lenguajes son diseñados para apoyar un paradigma particular (Smalltalk[4] y Java apoyan la programación orientada a objetos mientras Haskell[5] y Scheme[6] apoyan el paradigma de programación funcional), otros lenguajes de programación apoyan múltiples paradigmas.

Algunos ejemplos de paradigmas de programación:

El paradigma imperativo es considerado el más común y está representado, por ejemplo, por el C[7] o por BASIC[8].

- El paradigma funcional está representado por la familia de lenguajes LISP[9], en particular Scheme.
- El paradigma lógico, un ejemplo es PROLOG[10].

Si bien puede seleccionarse la forma pura de estos paradigmas al momento de programar, en la práctica es habitual que se mezclen. Tal es el caso de lenguajes como C++, Delphi o Visual Basic, los cuales combinan el paradigma imperativo con el orientado a objetos.

Por ejemplo, C++ es diseñado para apoyar los elementos de programación procedural, programación basada a objetos, programación orientada a objetos, y programación genérica.

4-10 *Lenguajes de programación pertenecientes a distintos paradigmas.*

Uno puede escribir un programa puramente procedural en C ++, puede escribir un Sin embargo, los diseñadores y programadores deciden como construir un programa que usa aquellos elementos de un paradigma. programa puramente orientado a objetos en C ++, o puede escribir un programa que contiene los elementos de ambos paradigmas.

En estos estilos es donde entra la controversia por parte de muchos programadores en cuanto a cual es mejor o cumple sus necesidades, y con la evolución de los lenguajes, van aplicándose nuevos paradigmas, por ejemplo el de programación orientada a aspectos, el cual es reciente y su intención es permitir una adecuada modularización y posibilitar una mejor separación de conceptos. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA (Programación Orientada a Aspectos) es usado para referirse a varias tecnologías relacionadas.

Muchas veces nos encontramos, a la hora de programar, con problemas que no podemos resolver de una manera adecuada con las técnicas habituales usadas en la programación procedural o en la programación orientada a objetos. O que en otro lenguaje se resuelven de forma más sencilla y tratamos de evitar en la mayoría de las veces varias líneas de código. Con éstas, nos vemos forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades. Pero, como se repite en este apartado, todo tiene sus ventajas y se hará un énfasis a la programación orientada a objetos y sus fortalezas.

Algunos diseñadores de lenguajes han dado por sentado que la programación orientada a objetos, de por sí, no es adecuada para resolver de manera sencilla todos los problemas de programación, y hacen referencia al uso de lenguajes de programación *multiparadigma.*

La descripción de los otros paradigmas se dejara para un futuro apartado.

3. Programación orientada a objetos

Aunque existen distintas terminologías para nombrar los mismos conceptos y características importantes de la programación orientada a objetos, abreviada POO, es importante entender la mayoría de ellos tanto por quien la desconoce como por aquel que la conoce.

Las computadoras, más que máquinas, pueden considerarse como herramientas que permiten ampliar la mente ("bicicletas para la mente", como se enorgullece de decir Steve Jobs), además de un medio de expresión inherentemente diferente. Como resultado, las herramientas empiezan a parecerse menos a máquinas y más a partes de nuestra mente, al igual que ocurre con otros medios de expresión como la escritura, la pintura, la escultura, la animación o la filmación de películas.

La programación orientada a objetos (POO) es una parte de este movimiento dirigido a utilizar las computadoras como si de un medio de expresión se tratara.

La programación orientada a objetos asevera dar mayor flexibilidad, facilitando los cambios a los programas y es ampliamente popular en la gran escala de ingeniería de software. Además, los que proponen la POO argumentan que es mas fácil de aprender para quienes son nuevos en la programación de computadoras y que su abordaje a menudo es más fácil de desarrollar y mantener prestándose así mismo más análisis directo, codificación y entendimiento de situaciones complejas y procedimientos comparado a otros métodos de programación.

4. ¿Qué es la POO?

La programación orientada a objetos es una evolución lógica de la programación estructurada, en la que el concepto de variables locales a un procedimiento o función, que no son accesibles a otros procedimientos y funciones, se hace extensible a los propios subprogramas que acceden a estas variables. Pero la programación orientada a objetos va mucho más allá. En realidad, cambia la concepción de la metodología de diseño de los programas.

En la programación orientada a objetos, se definen objetos que conforman una aplicación. Estos objetos están formados por una serie de características y operaciones que se pueden realizar sobre los mismos. Estos objetos no están aislados en la aplicación, sino que se comunican entre ellos.

Los programadores que emplean lenguajes procedurales, escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciendo que realicen esos métodos en sí mismos.

4.1 Conceptos fundamentales

La programación orientada a objetos hace énfasis en los siguientes conceptos:

- **Abstracción**: Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando los están, una variedad de técnicas son requeridas para ampliar una abstracción. Es la habilidad de un programa de ignorar los detalles de un objeto. Por ejemplo “Rin Tin Tin” el perro puede ser tratado como perro la mayoría de las veces, menos cuando es abstraído apropiadamente al nivel de canido (superclase de perro) o carnívoro (superclase de canido) y así sucesivamente.

- **Encapsulamiento**: También llamada "ocultación de la información", esto asegura que los objetos no pueden cambiar el estado interno de otros objetos de maneras inesperadas; solamente los propios métodos internos del objeto pueden acceder a su estado. Cada tipo de objeto expone una *interfaz* a otros objetos que especifica cómo otros objetos pueden interactuar con él. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción.
- **Polimorfismo**: Las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.
- **Herencia**: Organiza y facilita el polimorfismo y la encapsulación permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en *clases* o Class y las clases en *árboles* o *enrejados* que reflejan un comportamiento común.

Cada uno de estos conceptos se irá detallando más a fondo en los siguientes apartados.

4.2 ¿Qué es un Objeto?

La respuesta a esta pregunta en términos ajenos a la programación parece simple. Un objeto es una persona, animal o cosa. Se distingue de otros objetos por tener unas determinadas características y “sirve” para algo, o dicho de otra forma, se pueden realizar distintas operaciones con/sobre ese objeto.

Por ejemplo: Una casa es un objeto.

CARACTERÍSTICAS: Número de pisos, altura total en metros, color de la fachada, número de ventanas, número de puertas, ciudad, calle y número donde está ubicada, etc.

OPERACIONES: Construir, destruir, pintar fachada, modificar alguna de las características, como por ejemplo, abrir una nueva ventana, etc.

Evidentemente, cada objeto puede definirse en función de multitud de características y se pueden realizar innumerables operaciones sobre él. Ya en términos de programación, será misión del programador determinar qué características y que operaciones interesa mantener sobre un objeto. Por ejemplo,

sobre el objeto casa puede no ser necesario conocer su ubicación y por lo tanto, dichas características no formarán parte del objeto definido por el programador. Lo mismo podría decirse sobre las operaciones.

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

En terminología de programación orientada a objetos, a las características del objeto se les denomina *ATRIBUTOS* y a las operaciones *MÉTODOS*. Cada uno de estos métodos es un procedimiento o una función perteneciente a un objeto.

TERMINOLOGÍA INFORMAL	TERMINOLOGÍA FORMAL
CARACTERÍSTICAS	**ATRIBUTOS**
OPERACIONES (PROCEDIMIENTOS Y FUNCIONES)	**MÉTODOS**

Un objeto está formado por una serie de características o datos (atributos) y una serie de operaciones (métodos). No puede concebirse únicamente en función de los datos o de las operaciones sino en su conjunto.

4.3 Abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede incluso afirmarse que la complejidad de los problemas a resolver es directamente proporcional a la clase (tipo) y calidad de las abstracciones a utilizar, entendiendo por tipo "clase", *aquello que se desea abstraer.* El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados "imperativos" desarrollados a continuación del antes mencionado ensamblador (como Fortran, BASIC y C) eran abstracciones a su vez del lenguaje citado. Estos lenguajes supusieron una gran mejora sobre el lenguaje ensamblador, pero su abstracción principal aún exigía pensar en términos de la estructura del computador más que en la del problema en sí a resolver. El programador que haga uso de estos lenguajes debe establecer una asociación entre el modelo de la máquina (dentro del "espacio de la solución", que es donde se modela el problema, por ejemplo, un computador) y el modelo del problema que de hecho trata de resolver (en el "espacio del problema", que es donde de hecho el problema existe).

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. El término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?". El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

Se sabe que los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: abstracción de datos (pertenecientes a los datos) y abstracción de control (perteneciente a las estructuras de control).

La abstracción encarada desde el punto de vista de la programación orientada a objetos expresa las características esenciales de un objeto, las mismas distinguen al objeto de los demás. Además de distinguir entre los objetos provee límites conceptuales. Entonces se puede decir que la encapsulación separa las características esenciales de las no dentro de un objeto. Si un objeto tiene mas características de las necesarias los mismos resultarán difíciles de usar, modificar, construir y comprender.

La misma genera una ilusión de simplicidad dado a que minimiza la cantidad de características que definen a un objeto.

Y todo es un objeto. Hay que pensar en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.

4.4 Métodos y mensajes

Un término que se utiliza más frecuentemente en Java y POO es método, al ser "una manera de hacer algo". Si se desea, es posible seguir pensando en funciones. Verdaderamente sólo hay una diferencia sintáctica, pero de ahora en adelante se usará el término "método" en lugar del término "función".

Los **métodos** son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

Las partes fundamentales de un método son su nombre, sus parámetros, el tipo de retorno y el cuerpo.

He aquí su forma básica:

```
tipoRetorno nombreMetodo ( /* lista de parámetros */ ) {
```

```
/ * Cuerpo del método */
}
```

El tipo de retorno es el tipo del valor que surge del método tras ser invocado. La lista de parámetros indica los tipos y nombres de las informaciones que es necesario pasar a ese método. Cada método se identifica unívocamente mediante el nombre del método y la lista de parámetros.
En Java los métodos pueden crearse como parte de una clase. Es posible que un método pueda ser invocado sólo por un objeto2, y ese objeto debe ser capaz de llevar a cabo esa llamada al método. Si se invoca erróneamente a un método de un objeto, se generará un error en tiempo de compilación. Se invoca a un método de un objeto escribiendo el nombre del objeto seguido de un punto y el nombre del método con su lista de argumentos, como: **nombreObjeto.nombreMetodo (**arg1**,** arg2, arg3). Por ejemplo, si se tiene un método **f** ( ) que no recibe ningún parámetro y devuelve un dato de tipo int, y si se tiene un objeto **a** para el que puede invocarse a **f** ( ), es posible escribir:

```
int x = a.f ();
```

El tipo del valor de retorno debe ser compatible con el tipo de x.

Este acto de invocar a un método suele denominarse *envío de un mensaje a un objeto.* En el ejemplo de arriba, el mensaje es **f** ( ) y el objeto es **a.** La programación orientada a objetos suele resumirse como un simple "envío de mensajes a objetos".

Un ejemplo claro sería el siguiente:

- Ejemplo de Clase **Mamífero**:

Datos	**Métodos**
Color	Desplazarse
Tamaño	Masticar
Peso	Digerir
Cantidad de dientes	Respirar
Cuadrúpedo/bípedo	Parpadear
Edad	Dormir

Ahora veamos un ejemplo más cercano a la informática: Una ventana. Clase **Ventana**:

Datos	Métodos
Tipo de letra encabezado	Minimizarse
Etiqueta de encabezado	Maximizarse
Color de Fondo	Cerrarse
Color de letra encabezado	Abrirse

Como puede verse, en esencia la **POO** lo que nos permite es ampliar el tipo de datos con los que se puede trabajar.

Todos los hombres comparten las características de la clase **hombre**. Sin embargo todos los hombres son distintos entre sí, en estatura, pigmentación, etc. A cada uno de los hombres particulares se le llama **"objetos de la clase hombre"**. Cada niño que nace es una ***instanciación*** de la clase hombre.

Entonces, los métodos los podemos considerar sinónimos de comportamientos, operaciones, procedimientos.

La aplicación de los ***métodos*** dará distintos resultados dependiendo de los ***valores*** con los que trabaje.

Para poder mandar mensajes a los objetos, necesitamos un operador, a este le llamamos el operador de envío. Cada lenguaje puede tener el suyo, pero es frecuente que se utilicen los dos puntos (':') o el punto ('.') (En C++ es el punto simple, igual que para referirnos a los elementos de los struct's).

Así, si queremos enviarle el mensaje **Caminar** al objeto **Juan** de la clase **hombre**, escribiríamos lo siguiente:

```
hmrJuan.Caminar()
+-++--+¦+-------+
 ¦   ¦  ¦     +---- Mensaje. Invoca el método de igual nombre[1].
 ¦   ¦  +---------- Operador de envío[2].
 ¦   +------------- Objeto de la clase hombre hmrJuan.
 +----------------- El prefijo hombre denota su clase.
```

1. En algunos lenguajes OOP se puede hacer que un mensaje invoque un método con nombre distinto.
2. Como hemos comentado, cada lenguaje puede utilizar su propio operador de envío.

El operador de envío hace que se ejecute la porción del código agrupada bajo el nombre del método, y el método trabajara con los datos propios de la instancia de la clase a la que se refiera.

Referencias a sí mismo

Un caso especial ocurre cuando estamos codificando un método de una clase y tenemos que referirnos a un dato o un método del propio objeto; estamos creando la clase, o por así decirlo, dentro del objeto mismo.

Entonces, para referirse al propio objeto, en OOP se hace uso de el mismo, si mismo, self (en ingles). Self (algunos lenguajes utilizan “self”, pero lo mas común es que utilicen “this”[11]: así lo hacen C++ y Java, por ejemplo.) en OOP se refiere al propio objeto con el que se esta trabajando. Por lo tanto, si estamos escribiendo un método de una clase y queremos enviar un mensaje al propio objeto, escribiríamos:

```
This.Ocultar ( );
```

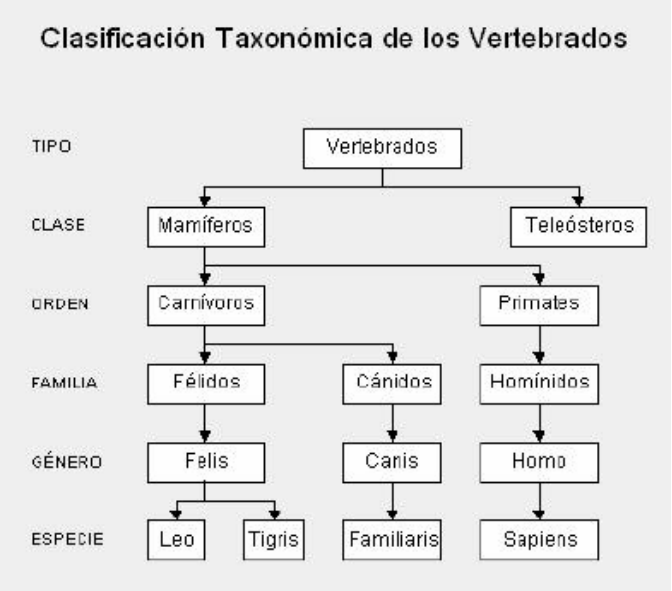
En este punto, puede parecer que un programa no es más que un montón de objetos con métodos que toman otros objetos como parámetros y envían mensajes a esos otros objetos. Esto es, sin duda, mucho de lo que está curriendo, pero en otros apartados se verá cómo hacer el trabajo de bajo nivel detallado, tomando decisiones dentro de un método.

4.5 Encapsulado y ocultación de la información: Clases

El concepto de clase, es simplemente una abstracción que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas –objetos- con características similares en grupos –clases-. Así, aun cuando existen por ejemplo multitud de vasos diferentes, podemos reconocer un vaso en cuanto lo vemos, incluso aun cuando ese modelo concreto de vaso no lo hayamos visto nunca. El concepto vaso es una abstracción de nuestra experiencia sensible. Al igual que ciertos ejemplos que pusimos en el apartado anterior.

Quizás el ejemplo mas claro para exponer esto lo tengamos en las taxonomías; los biólogos han dividido a todo ser (vivo o inerte) sobre la tierra en clases.

11 ver el apartado 12.5



Ellos, llaman a cada una de estas parcelas *reino, tipo, clase, especie, orden, familia, género*, etc.; sin embargo, nosotros a todas las llamaremos del mismo modo: clase. Así, hablaremos de la clase animal, clase vegetal y clase mineral, o de la clase félidos y de las clases leo (león) y tigres (tigre).

Cada clase posee unas cualidades que la diferencian de las otras. Así, por ejemplo, los vegetales se diferencian de los minerales –entre otras muchas cosas- en que los primeros son seres vivos y los minerales no. De los animales se diferencian en que las plantas son capaces de sintetizar clorofila a partir de la luz solar y los animales no.

Como vemos, el ser humano tiende, de un modo natural a clasificar los objetos del mundo que le rodean en clases; son definiciones estructuralistas de la naturaleza al estilo de la escuela francesa de Saussure[12].

Encapsulación

Un método de una clase puede llamar a otros métodos de su misma clase y pude cambiar los valores de los datos de su misma clase. Es esta la forma correcta de hacerse: los datos de una clase solo deben ser alterados por los métodos de su clase; y no de forma directa (que es como cambiamos los valores de las variables en un programa). Esta es una regla de oro que no debe de olvidarse: **todos los datos de una clase son privados y se accede a ellos mediante métodos públicos.**

12 *Ferdinand de Saussure. Lingüista suizo, considerado el fundador de la lingüística moderna.*

Según los dos modos comentados, tomemos como ejemplo un objeto perteneciente a la clase marco, modificaremos su dato nY1 (coordenada superior izquierda) de dos modos distintos: directamente y mediante el método PonerY1 ( ).

Cambio directo: oCajaGeneral.nY1 = 12;

Cambio mediante invocación de método: oCajaGeneral.PonerY1 (12);

Es más cómodo el primer método, ya que hay que escribir menos para cambiar el valor del dato y además, a la hora de construir la clase, no es necesario crear un método para cambiar cada uno de los datos del objeto. Sin embargo, y como se ha ido comentando, la OOP recomienda efusivamente que se utilice el segundo procedimiento. La razón es bien simple: una clase debe ser una estructura cerrada, no se debe poder acceder a ella si no es a través de los métodos definidos para ella. Si hacemos nY1 público (para que pueda ser accedido directamente), estamos violando el principio de encapsulación.

Encapsulación es la característica de la ***POO*** que hace que un objeto sea una caja negra. Se le envía un mensaje al objeto y este responde ejecutando el método apropiado.

Un método de una clase puede llamar a métodos de la misma clase.

Gracias a la encapsulación una clase que ha sido programada y probada puede usarse sin temor a que la programación de otros objetos basados en dichas clases tengan errores.

En lugar de considerar el programa como una enorme entidad, la ***encapsulación*** permite dividir un programa en ***componentes*** más pequeños e independientes.

Cada ***componente*** es autónomo y realiza su labor independientemente de los demás componentes.

La ***encapsulación*** mantiene esa independencia ocultando los detalles internos de cada componente, mediante una ***interfaz externa***.

Una ***interfaz*** lista los servicios proporcionados por un ***componente***. La ***interfaz*** actúa como un contrato con el mundo exterior que define exactamente lo que una entidad externa puede hacer con el objeto.

Una ***interfaz*** es un panel de control para el objeto. Al definir una interfaz puede especificarse cuales métodos serán definidos como ***públicos***, ***privados*** y ***protegidos***.

Los métodos ***públicos*** son aquellos que están disponibles para todo aquel que cree objetos basados en la clase.

Los métodos ***privados*** son aquellos que solo son invocados por otros métodos de la clase.

Cuando un ***comportamiento*** se desee poner a disposición del mundo exterior, debe tener ***acceso público***. Lo que se desee ***ocultar*** debe ***tener acceso protegido o privado***.

¿Porqué utilizar encapsulación?

- ***Independencia***: Se puede reutilizar el objeto en cualquier parte. Cuando se encapsulan de forma apropiada los objetos, estos no están limitados a un programa en particular. Para utilizarlo solo hay que poner en acción su interfaz.
- ***Transparencia***: La encapsulación le permite hacer cambios transparentes al objeto, en tanto no se altere la interfaz.
- ***Autonomía***: El uso de un objeto encapsulado no causará efectos secundarios inesperados entre el objeto y el resto del programa. Puesto que el objeto es autónomo, no tendrá ninguna interacción con el programa más allá de lo establecido por la interfaz.

Gracias a la encapsulación, una clase, cuando ha sido programada y probada hasta comprobar que no tiene fallos, podemos usarla sin miedo a que al programar otros objetos estos puedan interferir con los primeros produciendo efectos colaterales indeseables que arruinen nuestro trabajo; esto también permite depurar (eliminar errores de programación) con suma facilidad, ya que si un objeto falla, el error solo puede estar en esa clase y no en ninguna otra.

4.6 Modularidad

Un modulo puede definirse diversamente, pero generalmente debe ser un componente de un sistema mas grande, y opera independientemente dentro de las funciones de ese sistema con los otros componentes.

La modularidad es la propiedad que mide hasta que punto han estado compuestos de partes separadas llamadas módulos. Los programas tienen muchas relaciones mutuas directas entre cualquiera de dos partes aleatorias del código del programa.

4.7 Polimorfismo

Se denomina **polimorfismo** a la capacidad del código de un programa para ser utilizado con diferentes tipos de datos u objetos. También se puede aplicar a la propiedad que poseen algunas operaciones de tener un comportamiento diferente dependiendo del objeto (o tipo de dato) sobre el que se aplican.

El concepto de polimorfismo se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de *funciones polimórficas* y *tipos polimórficos*. Las primeras son aquellas funciones que pueden evaluarse y/o ser aplicadas a diferentes tipos de datos de forma indistinta; los *tipos polimórficos*, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

Se puede clasificar el polimorfismo en dos grandes clases:

- **Polimorfismo dinámico** (o **polimorfismo *ad hoc***) es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible.

- **Polimorfismo estático** (o **polimorfismo paramétrico**) es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizados.

El polimorfismo dinámico unido a la herencia es lo que en ocasiones se conoce como **programación genérica.**

Pongamos por ejemplo las clase hombre, vaca y perro, si todos les damos la orden –enviamos el mensaje- **Come,** cada uno de ellos sabe como hacerlo y realizara este comportamiento a su modo.

Veamos otro ejemplo algo más ilustrativo. Tomemos las clases *barco, avión y coche*, todas ellas derivadas de la clase padre vehículo; si les enviamos el mensaje **Desplázate**, cada una de ellas sabe como hacerlo.

Realmente, y para ser exactos, los mensajes no se envían a las clases, sino a todos o algunos de los objetos instaciados de las clases. Así, por ejemplo, podemos decirle a los objetos *Juan Sebastián el Cano* y *Kontiqui*, de la clase barco que se desplacen, con los que el resto de los objetos de esa clase permanecerán inmóviles.

Del mismo modo, si tenemos en pantalla cinco recuadros (marcos) y tres textos, podemos decirle a tres de los recuadros y a dos de los textos que cambien de color y no decírselo a los demás objetos. Todos estos sabrán como hacerlo porque hemos redefinido para cada uno de ellos su método **Pintarse** que bien podría estar en la clase padre ***Visual*** (conjunto de objetos que pueden visualizarse en pantalla).

En programación tradicional, debemos crear un nombre distinto para la acción de pintarse, si se trata de un texto o de un marco; en POO el mismo nombre nos sirve para todas las clases creadas si así lo queremos, lo que suele ser habitual. El mismo nombre suele usarse para realizar acciones similares en clases diferentes.

Si enviamos el mensaje **Imprímete** a objetos de distintas clases, cada uno se imprimirá como le corresponda, ya que todos saben como hacerlo.

El polimorfismo facilita el trabajo, ya que gracias a él, el número de nombre de métodos que tenemos que recordar disminuye ostensiblemente.

La mayor ventaja la obtendremos en métodos con igual nombre aplicados a las clases que se encuentran próximas a la raíz del árbol de clases, ya que estos métodos afectaran a todas las clases que de ellas se deriven.

4.8 Jerarquización: Herencia y Objetos compuestos

Herencia

Esta es la cualidad más importante de un sistema POO, la que nos dará mayor potencia y productividad, permitiéndonos ahorrar horas y horas de codificación y de depuración de errores.

Sería mejor si pudiéramos hacer uso de una clase ya existente, clonarla, y después hacer al "clon" las adiciones y modificaciones que sean necesarias. Efectivamente, esto se logra mediante la herencia, con la excepción de que si se cambia la clase original (denominada la *clase base, clase súper o clase padre*), el "clon" modificado (denominado *clase derivada, clase heredada, subclase o clase hijo*) también reflejaría esos cambios.

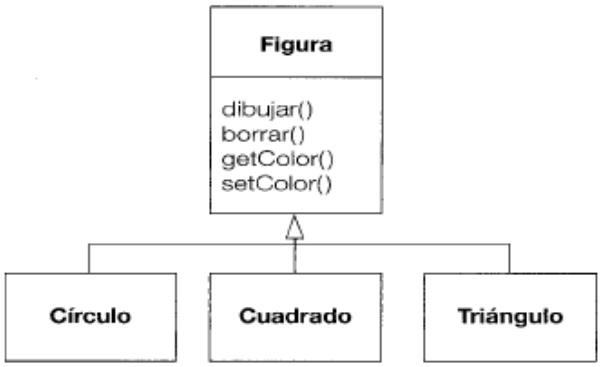
Puede haber más de una clase derivada. Un tipo hace más que definir los límites de un conjunto de objetos; también tiene relaciones con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que otro y también puede manipular más mensajes (o gestionarlos de manera distinta). La herencia expresa esta semejanza entre tipos haciendo uso del concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos que comparten los tipos que de él se derivan. A partir del tipo base, es posible derivar otros tipos para expresar las distintas maneras de llevar a cabo esta idea.

Por ejemplo, una máquina de reciclaje de basura clasifica los desperdicios. El tipo base es "basura", y cada desperdicio tiene su propio peso, valor, etc. y puede ser fragmentado, derretido o descompuesto.

Así, se derivan tipos de basura más específicos que pueden tener características adicionales (una botella tiene un color), o comportamientos (el aluminio se puede modelar, una lata de acero tiene capacidades magnéticas). Además, algunos comportamientos pueden ser distintos (el valor del papel depende de su tipo y condición). El uso de la herencia permite construir una jerarquía de tipos que expresa el problema que se trata de resolver en términos de los propios tipos.

Un segundo ejemplo es el clásico de la "figura geométrica" utilizada generalmente en sistemas de diseño por computador o en simulaciones de juegos. El tipo base es "figura" y cada una de ellas tiene un tamaño, color, posición, etc. Cada figura puede dibujarse, borrarse, moverse, colorearse, etc.

A partir de ésta, se pueden derivar (heredar) figuras específicas: círculos, cuadrados, triángulos, etc., pudiendo tener cada uno de los cuales características y comportamientos adicionales. Algunos comportamientos pueden ser distintos, como pudiera ser el cálculo del área de los distintos tipos de figuras. La jerarquía de tipos engloba tanto las similitudes como las diferencias entre las figuras.



Otro ejemplo, la clase león que se comenta en la figura de la clasificación taxonómica en el apartado 4.5, hereda cualidades –métodos- de todas las clases predecesoras –padres- y posee métodos propios, diferentes a los del resto de las clases. Es decir, las clases van especializándose según se avanza en el árbol taxonómico. Cada vez que creamos una clase heredada de otra (la padre) añadimos métodos a la clase padre o modificamos alguno de los métodos de la clase padre.

Veamos que hereda la clase león de sus clases padre:

Clase	Que hereda
Vertebrados	Espina dorsal
Mamíferos	Se alimenta con leche materna
Carnívoros	Al ser adultos se alimentan de carne

La clase león hereda todos los métodos de las clases padre y añade métodos nuevos que forman su clase distinguiéndola del resto de las clases: por ejemplo el color de su piel.

Pongamos ahora un ejemplo algo más informático: supongamos que usted ha construido una clase que le permite leer números enteros desde teclado con un formato determinado, calcular su IVA y almacenarlos en un archivo. Si desea poder hacer lo mismo con números reales (para que admitan decimales), solo deberá crear una nueva subclase para que herede de la clase padre todos sus

métodos y redefinirá solo el método de lectura de teclado. Esta nueva clase sabe almacenar y mostrar los números con formato porque lo sabe su clase padre.

Las cualidades comunes que comparten distintas clases, pueden y deben agruparse para formar una clase padre. Por ejemplo, usted podría derivar las clases presupuesto y factura de la superclase pedidos, ya que estas clases comparten características comunes. De este modo, la clase padre poseería los métodos comunes a todas ellas y solo tendríamos que añadir aquellos métodos propios de cada una de las subclases, pudiendo reutilizar el código escrito en la superclase desde cada una de las clases derivadas. Así, si enseñamos a la clase padre a imprimirse, cada uno de los objetos de las clases inferiores sabrán automáticamente y sin escribir ni una sola línea más de código imprimirse.

Es así, que la herencia es la cualidad mas importante de la POO ya que le permite reutilizar todo el código escrito para las superclases re-escribiendo solo aquellas diferencias que existan entre estas y las subclases.

Muchas veces las clases –especialmente aquellas que se encuentran próximas a la raíz en el árbol de la jerarquía de clases- son abstractas. Es decir, solo existen para proporcionar una base para la creación de clases mas especificas, y por lo tanto no puede instanciarse de ellas; son las clases virtuales.

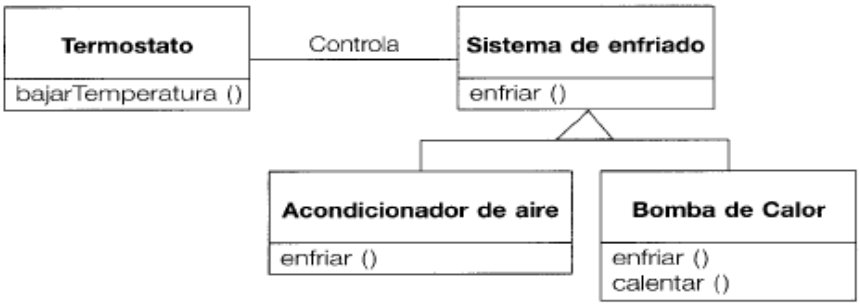
Una subclase hereda de su superclase solo aquellos miembros visibles desde la clase hija y por lo tanto solo puede redefinir estos.

Una subclase tiene forzosamente que redefinir aquellos métodos que han sido definidos como abstractos en la clase padre o padres.

Es habitual que la herencia suscite un pequeño debate: ¿debería la herencia superponer *sólo* las funciones de la clase base (sin añadir nuevas funciones miembro que no se encuentren en ésta)?. Esto significaría que el tipo derivado sea *exactamente* el mismo tipo que el de la clase base, puesto que tendría exactamente la misma interfaz. Consecuentemente, es posible sustituir un objeto de la clase derivada por otro de la clase base. A esto se le puede considerar *sustitución pura,* y a menudo se le llama el *principio de sustitución.* De cierta forma, ésta es la manera ideal de tratar la herencia. Habitualmente, a la relación entre la clase base y sus derivadas que sigue esta filosofía se le denomina relación *es-un,* pues es posible decir que "un círculo *es un* polígono". Una manera de probar la herencia es determinar si es posible aplicar la relación *es-un* a las clases en liza, y tiene sentido.

Hay veces en las que es necesario añadir nuevos elementos a la interfaz del tipo derivado, extendiendo así la interfaz y creando un nuevo tipo. Éste puede ser también sustituido por el tipo base, pero la sustitución no es perfecta pues las nuevas funciones no serían accesibles desde el tipo base. Esta relación puede describirse como la relación *es-como-un"* el nuevo tipo tiene la interfaz del viejo

pero además contiene otras funciones, así que no se puede decir que sean exactamente iguales. Considérese por ejemplo un acondicionador de aire. Supongamos que una casa está cableada y tiene las botoneras para refrescarla, es decir, tiene una interfaz que permite controlar la temperatura. Imagínese que se estropea el acondicionador de aire y se reemplaza por una bomba de calor que puede tanto enfriar como calentar. La bomba de calor *es-como-un* acondicionador de aire, pero puede hacer más funciones. Dado que el sistema de control de la casa está diseñado exclusivamente para controlar el enfriado, se encuentra restringido a la comunicación con la parte "enfriadora" del nuevo objeto. Es necesario extender la interfaz del nuevo objeto, y el sistema existente únicamente conoce la interfaz original.



Por supuesto, una vez que uno ve este diseño, está claro que la clase base "sistema de enfriado" no es lo suficientemente general, y debería renombrarse a "sistema de control de temperatura" de manera que también pueda incluir calentamiento -punto en el que el principio de sustitución funcionará. Sin embargo, este diagrama es un ejemplo de lo que puede ocurrir en el diseño y en el mundo real.

Cuando se ve el principio de sustitución es fácil sentir que este principio (la sustitución pura) es la única manera de hacer las cosas, y de hecho, es bueno para los diseños que funcionen así. Pero hay veces que está claro que hay que añadir nuevas funciones a la interfaz de la clase derivada.

4.9 Ligadura Dinámica

La ligadura en las llamadas a métodos

La conexión de una llamada a un método se denomina *ligadura.* Cuando se lleva a cabo la *ligadura* antes de ejecutar el programa (por parte del compilador y el montador, cuando lo hay) se denomina *ligadura temprana.* Puede que este término parezca extraño pues nunca ha sido una opción con los lenguajes procedurales. Los compiladores de C tienen un único modo de invocar a un método utilizando la ligadura temprana.

La solución es la *ligadura tardía,* que implica que la correspondencia se da en tiempo de ejecución, basándose en el tipo de objeto. La *ligadura tardía* se denomina también *dinámica* o *en tiempo de ejecución.*

Cuando un lenguaje implementa la ligadura tardía, debe haber algún mecanismo para determinar el tipo de objeto en tiempo de ejecución e invocar al método adecuado. Es decir, el compilador sigue sin saber el tipo de objeto, pero el mecanismo de llamada a métodos averigua e invoca al cuerpo de método correcto. El mecanismo de *la ligadura tardía* varía de un lenguaje a otro, pero se puede imaginar que es necesario instalar algún tipo de información en los objetos.

Toda ligadura de métodos en Java se basa en la ligadura tardía a menos que se haya declarado un método como **constante.** Esto significa que ordinariamente no es necesario tomar decisiones sobre si se dará la ligadura tardía, sino que esta decisión se tomará automáticamente.

5. Diseño tradicional vrs. Diseño OO

Como hemos visto hasta ahora, el Diseño Orientado a Objetos ofrece muchas alternativas para la resolución de problemas al programador, y hemos mencionado también los distintos paradigmas de programación.

Ahora, repasemos puntos clave del diseño OO:

- El diseño OO se enfoca sobre objetos y clases basados en el mundo real.
- Hace un énfasis en el estado, comportamientos e interacciones de objetos
- Provee las siguientes ventajas:
    - Desarrollo rápido de aplicaciones (RAD)
    - Aumenta la calidad
    - Provee un fácil mantenimiento
    - Mejora la modificación
    - Incrementa el reutilizamiento de software

Pero como todo en este mundo, también tiene sus limitaciones e inconvenientes. De las primeras no cabe ni hablar, porque aun cuando sea el sistema mas apropiado del que se dispone para programar, todavía se enfrenta con ciertos problemas de diseño.

El mayor inconveniente real proviene de un “error” de planteamiento; y como casi siempre, estos son de mucha mayor dificultad a la hora de solucionarlos. El problema, según comenta uno de los mejores analistas de hoy en día, Jeff Duntemann[15], en un articulo de la revista Dr. Dobbs[16], es que “la encapsulación y la herencia se hallan en esquinas opuestas de la casa”.

15 *Escritor, Editor, Tecnólogo. Más en www.duntemann.com*
16 Dr. Dobbs Journal, más en www.ddj.com

Es decir, la encapsulación choca frontalmente con la herencia, y sin embargo, son dos piedras angulares de la POO.

Por un lado decimos que los objetos deben ser totalmente independientes y autónomos, y por otro, al heredar unas clases de otras, estamos dejando fuera de un objeto perteneciente a una clase hija gran parte de la información que esta necesita para poder comportarse.

Otro inconveniente que se puede observar es el que estriba en la imposibilidad de utilización conjunta de objetos de distintos programadores.
Un ejemplo simple, supongamos que se monta un coche en un garaje, puede comprar un carburador de cualquier marca y montarlo en un coche de otra marca, unos asientos de uno y montarlos en la carrocería de otro; esto mismo puede hacerse en POO.

Si Microsoft fabrica la clase *Menú*, que deriva de la clase *Visual* y Borland fabrica la clase *Ventana* aunque también derive de la clase *Visual*, no puede conseguir coger el menú de una y la ventana de la otra, a menos que todas las clases superiores (en este caso solo una) sean exactamente iguales: tengan el mismo nombre, contengan los mismos datos y los mismos métodos y en estos, todos los parámetros deben coincidir en orden y en tipo.

Este problema por ahora no tiene solución, y lo peor es que no se vislumbra que la tenga en un futuro próximo, ya que para ello seria necesario normalizar las clases (al menos las mas habituales), pero si no nos ponemos de acuerdo para utilizar un mismo HTML ¿Cómo nos vamos a poner de acuerdo para esto?

Critica
Los taxonomías jerárquicas no coinciden a menudo con el mundo real y los cambios del mundo real según algunos críticos, y debe evitarse. Sin embargo, muchos defensores de POO también hacen pensar en evitar las jerarquías y usar las técnicas de POO en cambio como la composición.

También, muchos sienten que POO corre lo opuesto a la filosofía de planeación correlativa y base de datos relacionales, mientras se vuelve a los arreglos de la base de datos de navegación de los años sesenta. No está claro que ésta es la falta de POO, desde que la planeación de bases de datos es fundamentalmente basada en las diferentes premisas que la planeación basada en objetos. En cualquier caso, las bases de datos relacionales trazan a las asociaciones en los modelos basados en objetos, y las diferencias parecen ser completamente debidas a las diferencias en el enfoque.

Hay una historia de desinterpretación de la relación entre planear basado en objetos y en la correlación a lo que puede enturbiar este problema. Hay también, variaciones en las opiniones sobre los papeles y definiciones de cada uno.

La desigualdad de impedancia entre las bases de datos y POO se causa por la diferencia de balanza entre funcionamientos realizados por los objetos y bases de datos; las transacciones de la base de datos, la unidad más pequeña de trabajo realizada por las bases de datos, son mucho más grandes que cualquier funcionamiento proporcionado por los objetos de POO.
Mientras se exige que POO es buena para "aplicaciones grandes", otros sienten que deben reducirse las aplicaciones grandes en cambio a muchas aplicaciones pequeñas, como procedimientos manejados a eventos que " se alimentan" fuera de una base de datos y armazones de interfaz de usuario basadas en declaratorias de programación.

Está reconocido que POO necesariamente no quiere decir falta de complejidad.

6. Evolución de los Lenguajes Orientados a Objetos

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Según se informa, la historia es que trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus *propios* datos y comportamientos. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC(Palo Alto Research Center) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como la metodología 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 (GUI por sus siglas en ingles), para los cuales la programación orientada a objetos está particularmente bien adaptada.

En el ETH de Zürich, Niklaus Wirth y sus colegas tuvieron también investigaciones como tópicos de abstracción de datos y programación modular. El lenguaje Modula-2 incluye ambos, y su diseño sucesor, Oberon incluye una aproximación distintiva a orientación a objetos, clases y semejantes. El acercamiento es diferente a Smalltalk, y muy diferente a C++.

Las características de Orientación a Objetos han sido añadidas a muchos lenguajes existentes durante el tiempo, incluyendo Ada, BASIC, Lisp, Fortran, Pascal, y otros. Agregando esos aspectos a lenguajes que no fueron inicialmente diseñados para ello condujo a problemas con la compatibilidad y mantenimiento de código. Los lenguajes Orientados a Objetos "Puros", por otra parte, carecían de

aspectos que muchos programadores han venido dependiendo. Para cruzar esta brecha, muchos intentos han sido hechos para crear nuevos lenguajes basados en métodos Orientados a Objetos pero permitiendo algunos aspectos procedurales en formas “seguras”.

El lenguaje Eiffel de Bertrand Meyer’s[17] fue un temprano y moderadamente exitoso lenguaje con esos objetivos.

En la década pasada Java ha emergido ampliamente en uso parcialmente por su similitud a C y C++, pero quizás lo más importante de su implementación es el uso de una maquina virtual que es proyectada a correr código inalterado sobre muchas plataformas distintas. Esta ultima característica lo ha hecho muy atractivo en amplias tiendas de desarrollo y ambientes heterogéneos.

La iniciativa de la plataforma .NET de Microsoft ha tenido un objetivo similar e incluye/soporta varios nuevos lenguajes, o variantes de algunos viejos.

Más recientemente, un número de lenguajes ha surgido para ser primariamente orientados a objetos aun compatibles con la metodología procedural, tales como Python y Ruby. Al lado de Java, probablemente el mas comercial lenguaje orientado a objetos reciente es Visual Basic .NET y C# diseñados para la plataforma .NET de Microsoft.

Tal como la programación procedural lleva a refinamientos de técnicas como la programación estructurada, el software orientado a objetos moderno diseña métodos que incluye refinamiento como el uso de diseño de patrones, diseño por convenios, y lenguajes de modelado (tal es UML, usado en ingeniería de software).

17 Bertrand Meyer, (nacido en Francia en 1950). Es uno de los primeros y vocales más proponentes de la POO. Sus libros sobre “Construcción de Software” son considerados los mejores trabajos presentes para la presentación de POO.

Para el aprendizaje

En el pasado, han existido muchas disputas como cual es el mejor lenguaje para comenzar con la programación orientada a objetos. Hay dos diferentes enfoques a tomar cuando se comienza con este estilo de programación. El primero es la idea que lo mejor es empezar con un lenguaje simple como Java, donde el aprendizaje se enfoca sobre la orientación a objetos y no es perturbado por semánticas complejas del lenguaje. Aun así, otra razón es que lo mejor es empezar con un lenguaje más complicado como C++, el cual soporta con más precisión todas las estructuras y capacidades prescritas por el lenguaje de modelado unificado (UML), aunque no hay un lenguaje conocido el cual esencialmente soporte totalmente las capacidades de UML. En la versión utilizada para el entorno de desarrollo Eclipse para este ensayo esta instalado la versión Eclipse UML 3.10 Free Edition de Omondo.

7. Lenguajes orientados a objetos

Entre los lenguajes orientados a objetos destacan los siguientes:

- Smalltalk
- Objective-C
- C++
- Ada 95
- Java
- Ocaml
- Python
- Delphi
- Lexico (en castellano)
- C#
- Eiffel
- Ruby
- ActionScript
- Visual Basic
- PHP
- PowerBuilder
- Clarion

Estos lenguajes de programación son muy avanzados en orientación a objetos.

Al igual que C++ otros lenguajes, como OOCOBOL, OOLISP, OOPROLOG y Object REXX, han sido creados añadiendo extensiones orientadas a objetos a un lenguaje de programación clásico.

Un nuevo paso en la abstracción de paradigmas de programación es la Programación Orientada a Aspectos (POA). Aunque es todavía una metodología

en estado de maduración, cada vez atrae a más investigadores e incluso proyectos comerciales en todo el mundo.

8. Java

8.1 ¿Qué es Java?

Java es un lenguaje de desarrollo de propósito general, y como tal es válido para realizar todo tipo de aplicaciones profesionales.

Entonces, ¿es simplemente otro lenguaje más? Definitivamente no. Incluye una combinación de características que lo hacen único y está siendo adoptado por multitud de fabricantes como herramienta básica para el desarrollo de aplicaciones comerciales de gran repercusión.

¿Qué lo hace distinto de los demás lenguajes?

Una de las características más importantes es que los programas "ejecutables", creados por el compilador de Java, son **independientes de la arquitectura**. Se ejecutan indistintamente en una gran variedad de equipos con diferentes microprocesadores y sistemas operativos.

- De momento, es público. Puede conseguirse un JDK (Java Developer's Kit) o Kit de desarrollo de aplicaciones Java gratis. No se sabe si en un futuro seguirá siéndolo.
- Permite escribir *Applets* (pequeños programas que se insertan en una página HTML) y se ejecutan en el ordenador local.
- Se pueden escribir aplicaciones para intrarredes, aplicaciones cliente/servidor, aplicaciones distribuidas en redes locales y en Internet.
- Es fácil de aprender y está bien estructurado.
- Las aplicaciones son fiables. Puede controlarse su seguridad frente al acceso a recursos del sistema y es capaz de gestionar permisos y criptografía. También, según Sun, la seguridad frente a virus a través de redes locales e Internet está garantizada. Aunque al igual que ha ocurrido con otras tecnologías y aplicaciones, se han descubierto, y posteriormente subsanado, "agujeros" en la seguridad de Java.

¿Qué se puede programar con Java?

Si tenía preconcebida la idea de que con Java sólo se programan applets para páginas Web, está completamente equivocado. Ya que Java es un lenguaje de propósito general, puede programarse en él cualquier cosa:

- **Aplicaciones independientes**. Como con cualquier otro lenguaje de propósito general.
- **Applets**. Pequeñas aplicaciones que se ejecutan en un documento HTML, siempre y cuando el navegador soporte Java, como ocurre con los

navegadores HotJava y las últimas versiones de Netscape y el explorador de Internet de Microsoft.

Para todo aquel que no conozca la programación orientada a objetos, este lenguaje es ideal para aprender todos sus conceptos, ya que en cada paso de su aprendizaje se va comprobando que las cosas se hacen en la forma natural de hacerlas, sin sorpresas ni comportamientos extraños de los programas. A medida que se va aprendiendo, se va fomentando en el programador, y sin esfuerzo, un buen estilo de programación orientada a objetos. En realidad, no puede ser de otra forma, ya que Java impide "hacer cosas extrañas" y, además, no permite "abandonar" la programación orientada a objetos, como ocurre con otros lenguajes de programación. Esto es bastante conveniente, de lo contrario, un programador que está aprendiendo puede sentir la tentación de "volver" a lo que conoce (la programación tradicional).

8.2 Breve Historia de Java

El lenguaje Java™ fue creado por Sun Microsystems Inc. en un proceso por etapas que arranca en 1990, año en el que Sun creó un grupo de trabajo, liderado por James Gosling, para desarrollar un sistema para controlar electrodomésticos e incluso PDAs o Asistentes Personales (pequeños ordenadores) que, además, permitiera la conexión a redes de ordenadores. Se pretendía crear un hardware polivalente, con un Sistema Operativo eficiente (SunOS) y un lenguaje de desarrollo denominado **Oak** (roble), el precursor de Java. El proyecto finalizó en 1992 y resultó un completo fracaso debido al excesivo coste del producto, con relación a alternativas similares, tras lo cual el grupo se disolvió.

Por entonces aparece Mosaic y la **World Wide Web.** Después de la disolución del grupo de trabajo, únicamente quedaba del proyecto el lenguaje Oak. Gracias a una acertada decisión de distribuir libremente el lenguaje por la Red de Redes y al auge y la facilidad de acceso a Internet, propiciado por la WWW, el lenguaje se popularizó y se consiguió que una gran cantidad de programadores lo depurasen y terminasen de perfilar la forma y usos del mismo. A partir de este momento, el lenguaje se difunde a una velocidad vertiginosa, añadiéndosele numerosas clases y funcionalidad para TCP/IP. El nombre del lenguaje tuvo que ser cambiado ya que existía otro llamado Oak. El nombre "Java" surgió en una de las sesiones de "brainstorming" celebradas por el equipo de desarrollo del lenguaje. Buscaban un nombre que evocara la esencia de la tecnología (viveza, animación, rapidez, interactividad…). Java fue elegido de entre muchísimas propuestas. No es un acrónimo, sino únicamente algo humeante, caliente y que a muchos programadores les gusta beber en grandes cantidades: una taza de café (Java en argot Inglés americano). De esta forma, Sun lanzó las primeras versiones de **Java** a principios de 1995.

Desde entonces, Sun ha sabido manejar inteligentemente el éxito obtenido por su lenguaje, concediéndose licencias a cualquiera sin ningún problema, fomentando

su uso entre la comunidad informática y extendiendo las especificaciones y funcionalidad del lenguaje.

8.3 Características de Java

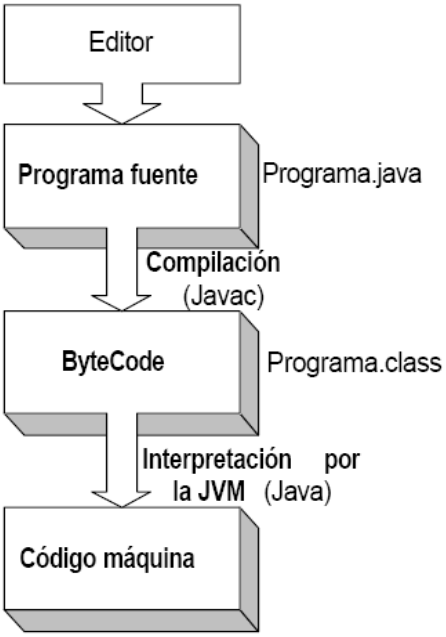
- Es intrínsecamente orientado a objetos.
- Funciona perfectamente en red.
- Aprovecha características de la mayoría de los lenguajes modernos evitando sus inconvenientes. En particular los del C++.
- Tiene una gran funcionalidad gracias a sus librerías (clases).
- NO tiene punteros manejables por el programador, aunque los maneja interna y transparentemente.
- El manejo de la memoria no es un problema, la gestiona el propio lenguaje y no el programador.
- Genera aplicaciones con pocos errores posibles.
- Incorpora *Multi-Threading* (para permitir la ejecución de tareas concurrentes dentro de un mismo programa).

El lenguaje Java puede considerarse como una evolución del C++. La sintaxis es parecida a la de este lenguaje, por lo que se hace referencia a dicho lenguaje. A pesar de que puede considerarse como una evolución del C++ no acarrea los inconvenientes del mismo, ya que Java fue diseñado “partiendo de cero”, es decir, un lenguaje “puro” y no necesitaba ser compatible con versiones anteriores de ningún lenguaje como ocurre con C++ y C.

Gracias a que fue diseñado “partiendo de cero” ha conseguido convertirse en un lenguaje orientado a objetos puro, limpio y práctico. No permite programar mediante otra técnica que no sea la programación orientada a objetos y, una vez superado el aprendizaje de la programación orientada a objetos, es realmente sencillo aprender Java.
¿El lenguaje es Compilado o Interpretado? Ni una cosa ni la otra. Aunque estrictamente hablando es interpretado, necesita de un proceso previo de compilación. Una vez “compilado” el programa, se crea un archivo que almacena lo que se denomina ***bytecodes*** o **j_code** (seudo código prácticamente al nivel de código máquina). Para ejecutarlo, es necesario un “intérprete”, la JVM (*Java Virtual Machine*) **máquina virtual Java**. De esta forma, es posible compilar el programa en una estación UNIX y ejecutarlo en otra con Windows utilizando la máquina virtual Java para Windows. Esta JVM se encarga de leer los ***bytecodes*** y traducirlos a instrucciones ejecutables directamente en un determinado microprocesador, de una forma bastante eficiente.

Que el programa deba ser “interpretado” no hace que sea poco eficiente en cuanto a velocidad, ya que la interpretación se hace prácticamente al nivel de código máquina. Por ejemplo, es mucho más rápido que cualquier otro programa interpretado como por ejemplo Visual Basic, aunque es más lento que el mismo

programa escrito en C++. Esta deficiencia en cuanto a la velocidad, puede ser aminorada por los compiladores *Just-In-Time* (JIT). Un compilador JIT transforma los *bytecodes* de un programa o un *applet* en código nativo de la plataforma donde se ejecuta, por lo que es más rápido. Suelen ser incorporados por los navegadores, como Netscape o Internet Explorer.

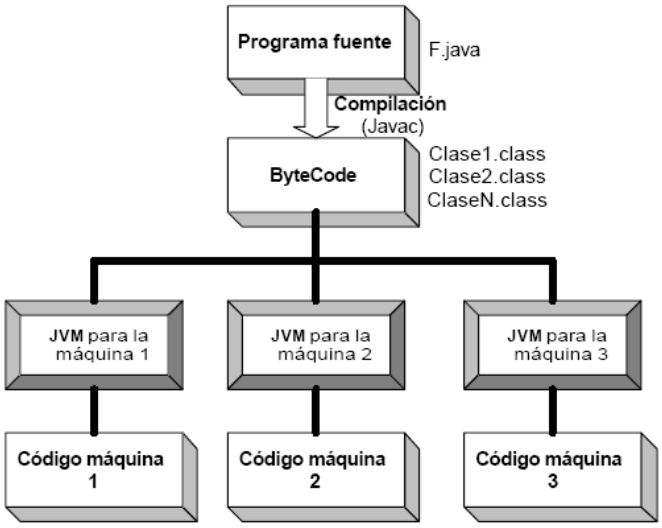


El lenguaje Java es robusto. Las aplicaciones creadas en este lenguaje son susceptibles de contener pocos errores, principalmente porque la gestión de memoria y punteros es realizada por el propio lenguaje y no por el programador. Bien es sabido que la mayoría de los errores en las aplicaciones vienen producidos por fallos en la gestión de punteros o la asignación y liberación de memoria. Además, el lenguaje contiene estructuras para la detección de excepciones (errores de ejecución previstos) y permite obligar al programador a escribir código fiable mediante la declaración de excepciones posibles para una determinada clase reutilizable.

La Máquina Virtual Java (JVM)

La máquina virtual Java es la idea revolucionaria[18] del lenguaje. Es la entidad que proporciona la independencia de plataforma para los programas Java "compilados" en *byte-code*.

18 Otros sistemas en el pasado, como por ejemplo el Pascal UCSD compilaban a un código intermedio (p-code) que luego era interpretado al ejecutar el programa.



Un mismo programa fuente compilado en distintas plataformas o sistemas operativos, genera el mismo archivo en byte-code. Esto es lógico, ya que se supone que el compilador de Java traduce el archivo fuente a código ejecutable por una máquina que únicamente existe en forma virtual (aunque se trabaja en la construcción de microprocesadores que ejecuten directamente el byte-code).

Evidentemente, si un mismo programa en byte-code puede ser ejecutado en distintas plataformas es porque existe un traductor de ese byte-code a código nativo de la máquina sobre la que se ejecuta. Esta tarea es realizada por la JVM.

Existe una versión distinta de esta JVM para cada plataforma. Esta JVM se carga en memoria y va traduciendo “al vuelo”, los byte-codes a código máquina. La JVM no ocupa mucho espacio en memoria, piénsese que fue diseñada para poder ejecutarse sobre pequeños electrodomésticos como teléfonos, televisores, etc.

8.4 Novedades en la versión 1.5

1. Introducción

En este documento se resumen las principales novedades que ofrece la versión 1.5 de Java, separándolas en diferentes áreas. Para una explicación más detallada, consultar la página de Sun:

http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html

2. Novedades en la máquina virtual

Autoajuste de memoria mejorado

La capacidad de autoajustar la cantidad de memoria necesaria (pila, recolector de basura, etc.) se ve mejorada en esta versión.

Compartir clases

Al instalar la máquina virtual, se cargan en memoria un conjunto de clases del sistema, en forma de representación interna, de forma que las siguientes llamadas a la máquina virtual ya encuentren estas clases mapeadas en memoria, y se permita que los datos de estas clases se compartan entre múltiples procesos dentro de la JVM.

Ajuste del recolector de basura

Relacionado con el autoajuste de memoria, el recolector de basura también se autoadapta a las necesidades de memoria de la aplicación, para evitar que el usuario tenga que ajustar su tamaño y características desde línea de comandos.

Tratamiento de errores fatales

El mecanismo de listado de errores fatales se ha mejorado de forma que se obtengan mejores diagnósticos a partir de las salidas generadas por dichos errores.

3. Novedades en el lenguaje

Tipos de datos parametrizados (*generics*)

Esta mejora permite tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos. Por ejemplo, podemos crear un *Vector* que sólo almacene *Strings*, o una *HashMap* que tome como claves *Integers* y como valores *Vectors.* Además, con esto nos ahorramos las conversiones *cast* al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

Ejemplo

```
// Vector de cadenas
Vector<String> v = new Vector<String>();
v.addElement("Hola");
String s = v.getElementAt(0);
v.addElement(new Integer(20));                    // Daría error!!

// HashMap con claves enteras y valores de vectores
```

```
HashMap<Integer, Vector> hm = new HashMap<Integer, Vector>();
hm.put(1, v);
Vector v2 = hm.get(1);
```

Autoboxing

Esta nueva característica evita al programador tener que establecer correspondencias manuales entre los tipos simples (*int, double*, etc.) y sus correspondientes *wrappers* o tipos complejos (*Integer, Double,* etc.). Podremos utilizar un *int* donde se espere un objeto complejo (*Integer*), y viceversa.

Ejemplo

```
Vector<Integer> v = new Vector<Integer>();
v.addElement(30);
Integer n = v.getElementAt(0);
n = n+1;
```

Mejoras en Ciclos

Se mejoran las posibilidades de recorrer colecciones y arrays, previniendo índices fuera de rango, y pudiendo recorrer colecciones sin necesidad de acceder a sus iteradores (*Iterator*).

Ejemplo

```
// Recorre e imprime todos los elementos de un array
int[] arrayInt = {1, 20, 30, 2, 3, 5};
for(int elemento: arrayInt)
    System.out.println (elemento);

// Recorre e imprime todos los elementos de un Vector
Vector<String> v = new Vector<String>();
for(String cadena: v)
    System.out.println (cadena);
```

Tipo *enum*

El tipo *enum* que se introduce permite definir un conjunto de posibles valores o estados, que luego podremos utilizar donde queramos:

Ejemplo

```
// Define una lista de 3 valores y luego comprueba en un switch
// cuál es el valor que tiene un objeto de ese tipo
enum EstadoCivil {soltero, casado, divorciado};
EstadoCivil ec = EstadoCivil.casado;
ec = EstadoCivil.soltero;
switch(ec)
{
```

```
    case soltero:   System.out.println("Es soltero");
                    break;
    case casado:    System.out.println("Es casado");
                    break;
    case divorciado:System.out.println("Es divorciado");
                    break;
}
```

Imports estáticos

Los *imports* estáticos permiten importar los elementos estáticos de una clase, de forma que para referenciarlos no tengamos que poner siempre como prefijo el nombre de la clase. Por ejemplo, podemos utilizar las constantes de color de la clase *java.awt.Color*, o bien los métodos matemáticos de la case *Math*.

Ejemplo

```
import static java.awt.Color;
import static java.lang.Math;

public class...
{
...
    JLabel lbl = new JLabel();
    lbl.setBackground(white);              // Antes sería
Color.white
...
    double raiz = sqrt(1252.2);            // Antes sería
Math.sqrt(...)
}
```

Argumentos variables

Ahora Java permite pasar un número variable de argumentos a una función (como sucede con funciones como *printf* en C). Esto se consigue mediante la expresión "..." a partir del momento en que queramos tener un número variable de argumentos.

Ejemplo

```
// Funcion que tiene un parámetro String obligatorio
// y n parámetros int opcionales

public void miFunc(String param, int... args)
{
...
    // Una forma de procesar n parametros variables
    for (int argumento: args)
    {
            ...
    }
...
```

```
}

...
miFunc("Hola", 1, 20, 30, 2);
miFunc("Adios");
```

Metainformación

Se tiene la posibilidad de añadir ciertas anotaciones en campos, métodos, clases y otros elementos, que permitan a las herramientas de desarrollo o de despliegue leerlas y realizar ciertas tareas. Por ejemplo, generar archivos fuentes, archivos XML, o un *Stub* de métodos para utilizar remotamente con RMI.

Un ejemplo más claro lo tenemos en las anotaciones que ya se utilizan para la herramienta Javadoc. Las marcas *@deprecated* no afectan al comportamiento de los métodos que las llevan, pero previenen al compilador para que muestre una advertencia indicando que el método que se utiliza está desaconsejado. También se tienen otras marcas *@param*, *@return*, *@see*, etc., que utiliza Javadoc para generar las páginas de documentación y las relaciones entre ellas.

Ejemplo

```
// Definición de una interfaz mediante metainformacion

public @interface MiInterfaz
{
    int metodo1();
    String metodo2();
}
```

4. Novedades en librerías principales

Red

Se han añadido cambios y mejoras para el trabajo en red, como:

- Soporte para IPv6 en Windows XP y 2003
- Establecimiento de *timeouts* para conectar y leer
- API para lanzar aplicaciones RMI a través de *inetd*
- La clase *InetAddress* permite testear si una URL es alcanzable (utilidad *ping*)
- Otras mejoras en el tratamiento de cookies, servidores proxy, tratamiento de URLs, etc.

Seguridad

Hay bastantes mejoras en seguridad. Se da soporte a más estándares de seguridad (SASL, OCSP, TSP, etc.), hay mejoras en la escalabilidad a través de SSLEngine, en criptografía, etc.

Internacionalización

Mediante la internacionalización podemos construir una aplicación que se adapte a varios idiomas, formatos monetarios, de fecha, etc., sin tener que reprogramarla. En este aspecto, se añaden mejoras en la versión 1.5, relacionadas con:

- La gestión de juegos de caracteres se basa en el formato Unicode 4.0, lo que afecta a las clases *Character* y *String*, entre otras.
- La clase *DecimalFormat* se ha modificado para poder procesar elementos de tipo *BigDecimal* o *BigInteger* sin perder precisión
- Se han añadido nuevos *Locales* soportados, como el vietnamita.

Formateador

La clase *Formatter* permite dar formato (justificación y alineamiento, formatos numéricos, de fecha, etc.) a las cadenas y otros tipos de datos, siguiendo un estilo parecido al *printf* de C. También se permite mediante la interfaz *Formattable* dar formatos (limitados) a tipos creados por el usuario.

Ejemplo

```
// Uso de formatter para construir cadenas formateadas
StringBuilder sb = new StringBuilder();
Formatter f = new Formatter(sb, Locale.US);
f.format("Hola, %1$2s, esta es tu visita numero %2$d", "Pepe",
20);
// Resultaría: "Hola, Pepe, esta es tu visita numero 20"

// También se tienen unos métodos predefinidos en ciertas clases
System.out.format("Hola, %1$2s, esta es tu visita numero %2$d",
"Pepe", 20);
System.err.printf("Hola, %1$2s, esta es tu visita numero %2$d",
"Pepe", 20);
String s = String.format("Hola, %1$2s", "Pepe");
```

Escaneador

La clase *Scanner* permite parsear un flujo de entrada (archivo, cadena de texto, stream de datos, etc.), y extraer tokens siguiendo un determinado patrón o tipo de datos. También se permite trabajar con expresiones regulares para indicar qué patrones se deben buscar.

Ejemplo

```
// Lectura de enteros de la entrada estándar
Scanner sc = Scanner.create(System.in);
int n = sc.nextInt();

// Lectura de todos los doubles de un archivo
Scanner sc = Scanner.create(new File("miFich.txt"));
while (sc.hasNextDouble())
    double d = sc.nextDouble();

// Uso de otros delimitadores
String s = "Esto hola es hola 1 hola ejemplo";
Scanner sc = Scanner.create(s).useDelimiter("\\s*hola\\s*");
System.out.println(sc.next());
System.out.println(sc.next());
System.out.println(sc.next());
// Sacaría Esto \n es \n 1
```

Arquitectura de JavaBeans

La arquitectura de JavaBeans se encuentra dentro del paquete *java.beans*. Se ha añadido una nueva clase, *IndexedPropertyChangeEvent*, subclase de *PropertyChangeEvent*, para dar soporte a eventos que respondan a cambios en propiedades indexadas (propiedades que utilicen un índice para identificar la parte del bean que haya cambiado). También se han añadido mejoras a la hora de crear editores de propiedades de beans (*PropertyEditor*), como el método *createPropertyEditor(...)*, y constructores públicos para la clase *PropertyEditorSupport*.

Arquitectura de colecciones

Como se ha comentado antes, las colecciones (estructura que contiene clases e interfaces como *Collection*, *Vector*, *ArrayList*, etc.) dan soporte a nuevas características del lenguaje, como el autoboxing, mejoras en los Ciclos (*for*) , y el uso de *generics*. Además, se han añadido interfaces nuevos, como *Queue, BlockingQueue* y *ConcurrentMap*, con algunas implementaciones concretas de dichas interfaces. Además, se tienen nuevas implementaciones de *List*, *Set* y *Map*.

Manipulación de bits

Los *wrappers* de tipos simples (es decir, clases como *Integer, Long, Double, Char*) soportan operaciones de bits, como *highestOneBit, lowestOneBit, rotateLeft, rotateRight, reverse*, etc.

Elementos matemáticos

El paquete *java.math* también ha sufrido modificaciones, como la adición en la clase *BigDecimal* de soporte para cálculo en coma flotante de precisión fija.

Clases como *Math* o *StrictMath* además incluyen soporte para senos y cosenos hiperbólicos, raíces cúbicas o logaritmos en base 10. Por último, también se da soporte al uso de números hexadecimales con coma flotante.

Serialización

Además de corregir errores previos en la serialización, se da soporte para serializar el nuevo tipo *enum*, aunque la forma de serializarlo difiere ligeramente de los tipos tradicionales.

Hilos

Se han añadido los paquetes *java.util.concurrent*, *java.util.concurrent.atomic* y *java.util.concurrent.locks*, que permiten crear diversas infraestructuras de hilos, como pooling de hilos o colas de bloqueos, liberando al programador de tener que controlar todas estas estructuras "a mano". En definitiva, se da soporte para automatizar la programación concurrente.

Además, a la clase *Thread* se le han añadido los métodos *getStackTrace* y *getAllStackTraces* para obtener la traza de los hilos en un momento dado.

Monitorización y gestión

Se tienen mejoras en el control o la monitorización tanto de las aplicaciones Java como de la máquina virtual (JVM), a través de la API de JMX.

5. Novedades en librerías adicionales

RMI

Se tienen dos mejoras fundamentales en RMI:

- Generación dinámica de Stubs: se generan en tiempo de ejecución, sin necesidad de utilizar *rmic* previamente. Sólo tendremos que utilizarlo para generar *Stubs* para clientes que funcionen con versiones anteriores de Java.
- Lanzamiento de *rmid* o un servidor RMI Java desde *inetd/xinetd*.

JDBC

En Java 1.4 se introdujo la interfaz *RowSet*, que permitía pasar datos entre componentes. En Java 1.5 se han desarrollado 5 implementaciones de dicha interfaz, para cubrir 5 posibles casos de uso:

- *JdbcRowSet*: para encapsular un *ResultSet* o un driver que utilice tecnología JDBC

- *CachedRowSet*: desconecta de la fuente de datos y trabaja independientemente, salvo cuando esté obteniendo datos de dicha fuente o volcando datos en ella.
- *FilteredRowSet*: hereda de la anterior, y se utiliza para obtener un subconjunto de datos
- *JoinRowSet*: hereda de *CachedRowSet* y se emplea para unir múltiples objetos *RowSet*.
- *WebRowSet*: hereda de *CachedRowSet* para procesar datos XML.

JNDI

- Mejoras en la clase *javax.naming.NameClassPair* para poder acceder al nombre completo del servicio de directorio.
- Soporte para controles estándar de LDAP
- Soporte para manipular nombres de LDAP

6. Novedades en la interfaz de usuario

Internacionalización

Para permitir renderizar texto multilingüe utilizando fuentes lógicas, se tienen en cuenta tanto las fuentes del cliente como las del *locale* que tengamos instalado. Además, AWT utiliza APIs Unicode en Windows 2000 y XP, con lo que se puede manejar texto sin estar limitado a la configuración de *locale* de Windows.

Java Sound

Para el tratamiento del sonido desde Java, se tienen también los siguientes cambios, principalmente:

- Se permite el acceso a puertos en todas las plataformas
- También está disponible la entrada/salida MIDI

Java 2D

- Se cachean en memoria todas las imágenes construidas a partir de *BufferedImage*
- Métodos para controlar la aceleración hardware en imágenes
- Añadido soporte para aceleración hardware con OpenGL en Solaris y Linux.
- Creación de fuentes a partir de archivos y flujos de entrada
- Se ha mejorado la renderización de texto.

Flujos de imágenes

Se permite, además de los formatos de imágenes ya soportados (PNG, JPG, etc.), leer y escribir imágenes BMP y WBMP.

AWT

Las novedades más significativas son:

- La clase *MouseInfo* almacena información sobre la posición del ratón en el escritorio
- Se tienen nuevos métodos en la clase *Window* para especificar la posición para las nuevas ventanas que se creen. También se tienen métodos para asegurar que una ventana permanezca siempre en primer lugar (aunque esta característica no funciona bien en algunas versiones de Solaris o Linux)
- Nuevos métodos para acceder y notificar sobre el contenido del portapapeles.
- Se han corregido errores existentes en los *layout managers*: *GridBagLayout* y *FlowLayout*.
- Cambios y correcciones en eventos de teclado (nuevos mapeos de teclas, redefinición de métodos, etc.), y de acción (correcciones en cuanto al uso de Enter como evento de acción).

Swing

Entre otras, se tienen las novedades:

- Se proporcionan nuevos look & feels (uno de ellos, *Synth*, es *skinnable*, es decir, se puede personalizar el skin que muestra)
- Se añade soporte para imprimir *JTables*.
- Se permite definir menús contextuales (*popup menus*) en componentes, para que muestren las opciones que queramos. Para ello se tiene la clase *JPopupMenu*.
- Se han añadido mejoras en *JTextArea* para facilitar el scroll por su contenido, y la actualización cuando se modifique su texto.
- Podremos utilizar el método *JFrame.add(...)*, en lugar de *JFrame.getContentPane().add(...)*

```
// Antes
JFrame f = new JFrame();
f.getContentPane().add(new JButton("Hola"));

// Ahora
JFrame f = new JFrame();
f.add(new JButton("Hola"));
```

7. Novedades en despliegue de aplicaciones

Despliegue general

Se tienen, entre otras, las siguientes novedades en cuanto a despliegue de aplicaciones:

- Se ha unido gran parte de la funcionalidad entre el Java Plug-in y Java Web Start, de modo que ahora tienen un *Java Control Panel* común para ambos
- Las aplicaciones desplegadas se pueden configurar mediante un archivo de configuración, que puede ser accedido a través de una URL, y establecer ciertos parámetros de la aplicación.
- El formato de compresión Pack200 permite tener archivos JAR ultra-comprimidos (hasta 1/8 de su tamaño original), reduciendo los tiempos de descarga, o de despliegue mediante Java Web Start.
- Inclusión de marcas de tiempo (*timestamps*) en los archivos JAR firmados, de forma que se sabe cuándo se concedió el certificado para el archivo, y se previene así el uso de JARs cuyo permiso ya caducó.

Java Web Start

Hay también cambios específicos de Java Web Start, como son:

- Supresión del *Application Manager*, cuya mayor funcionalidad se ha encapsulado dentro del nuevo *Java Control Panel*, y el *JNLP Cache Viewer*, una nueva herramienta que permite gestionar las aplicaciones descargadas.
- Supersión del *Developer Bundle*, ya que Java Web Start está completamente integrado con JRE y SDK, y los elementos que contenía dicho *bundle* ahora están integrados en Java.
- Se tiene una caché de sistema, de forma que el administrador del sistema puede cargar al inicio programas en la caché de sistema, de forma que sean compartidas y accesibles por múltiples usuarios.
- Hay también una facilidad de importación, de forma que se facilita la instalación desde CDs, donde el código se carga primero desde un lugar y luego se actualiza desde otro. También permite preinstalar aplicaciones y librerías en caché, sin ejecutarlas.

8. Novedades en herramientas

JPDA

JPDA es un conjunto de interfaces utilizadas para depurar en entornos y sistemas de desarrollo. En la versión 1.5 se tienen nuevas mejoras y funcionalidades de esta herramienta. Muchas de estas nuevas funcionalidades se han añadido para

adaptar las nuevas características de la versión 1.5 de Java, como los argumentos variables, *imports* estáticos, etc.

JVMTI

JVMTI (*Java Virtual Machine Tool Interface*) es una nueva interfaz de programación nativa para utilizar en herramientas de desarrollo y monitorización. Forma parte de las interfaces contenidas en JPDA para depuración. Permite chequear el estado y controlar la ejecución de una aplicación.

Compilador (*javac*)

Se han añadido algunas características a la hora de utilizar el compilador, como son los parámetros:

- - *source 1.5*: habilita las nuevas características de la versión 1.5 de Java, para ser tenidas en cuenta por el compilador. Lleva implícito también un parámetro *-target 1.5*
- *-target 1.5*: permite al compilador utilizar características específicas de Java 1.5 en las librerías y la máquina virtual.
- *-Xlint*: permite producir *warnings* sobre código que, aunque es correcto, puede ser problemático debido a su construcción.

Herramienta *javadoc*

Se tienen nuevas funcionalidades en la herramienta *javadoc*. Como en otros elementos, muchos de los cambios dan soporte a características nuevas de Java 1.5 (*generics*, tipo *enum*, etc.). Se tienen también marcas nuevas, como *regression* para indicar funcionalidades que se desaconsejaron en versiones anteriores, pero que han sido corregidas en la nueva. También se da soporte a anotaciones para generar metainformación que utilizar en otros programas.

8.5 El Entorno de Desarrollo de Java

La herramienta básica para empezar a desarrollar aplicaciones o *applets* en Java es el JDK (*Java Developer's Kit*) o Kit de Desarrollo Java, que consiste, básicamente, en un compilador y un intérprete (JVM) para la línea de comandos.
No dispone de un entorno de desarrollo integrado (IDE), pero este puede ser descargado de varias fuentes en Internet (Netbeans, Eclipse en mi caso, etc.) suficiente para aprender el lenguaje y desarrollar pequeñas aplicaciones.

¿Dónde conseguirlo?

El Kit de desarrollo puede obtenerse en las direcciones siguientes:

· **http://www.sun.com**

· **http://www.javasoft.com**

El entorno para Windows9x/NT está formado por un archivo ejecutable que realiza la instalación, creando toda la estructura de directorios. El kit contiene básicamente:

· El compilador: **javac.exe**

· El depurador: **jdb.exe**

· El intérprete: **java.exe** y **javaw.exe**

· El visualizador de applets: **appletviewer.exe**

· El generador de documentación: **javadoc.exe**

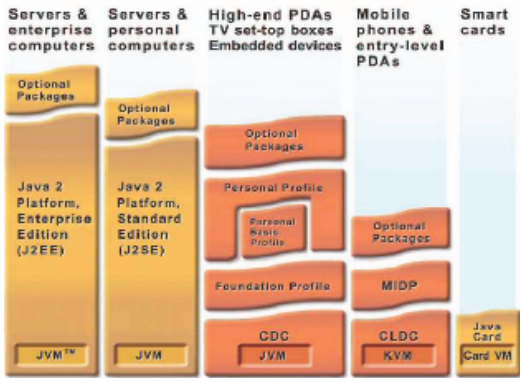
· Un desensamblador de clases: **javap.exe**

· El generador de archivos fuentes y de cabecera (.c y .h) para clases nativas en C: **javah.exe**

8.6 Herramientas para trabajar en Java

Existen distintas “ediciones” de Java para el desarrollo de aplicaciones en distintos ámbitos:

- Aplicaciones de propósito general (J2SE)
- Aplicaciones de gestión en entornos empresariales (J2EE)
- Aplicaciones para teléfonos móviles, PDAs y otros dispositivos electrónicos que permitan aplicaciones empotradas (J2ME)



La más utilizada es sin duda la edición estándar (J2SE).

J2SE incluye bibliotecas muy extensas y completas, que permiten la implementación de casi cualquier tipo de aplicación:

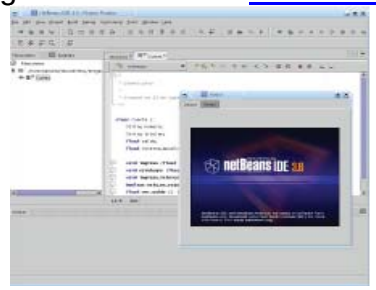
- Seguridad
- EEDDs

- Componentes (JavaBeans)
- Internacionalización
- E/S
- XML
- Redes y acceso a Internet
- Programación distribuida (RMI, CORBA)
- Matemática de precisión arbitraria
- Sonido
- Interfaz de usuario (AWT, SWING)
- Gráficos 2D
- Manipulación, carga y descarga de imágenes
- Impresión
- Acceso a bases de datos (JDBC)
- Gestión de preferencias y configuraciones

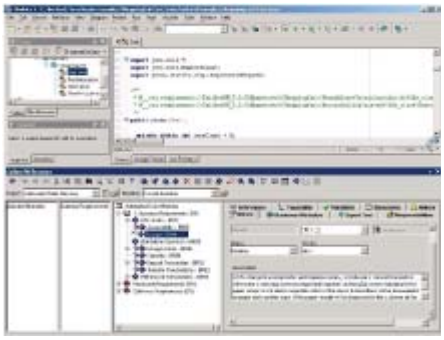
Varias de estas características se verifican en la sección de novedades, apartado 8.4.

Entonces, en los entornos de desarrollo podemos encontrar:

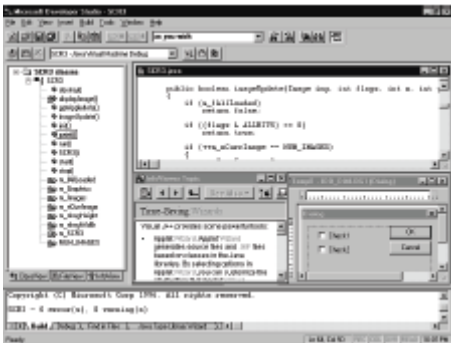
NetBeans. Entorno integrado de desarrollo Java de Sun, realizado íntegramente en Java (y por tanto multiplataforma). Consume bastantes recursos. Permite diseñar ventanas, escribir código, compilar, ejecutar etc. Requiere un JDK instalado. Puede obtenerse gratuitamente de www.netbeans.org



Borland JBuilder. Excelente entorno integrado de desarrollo Java de Borland. Al igual que Netbeans, también está realizado íntegramente en Java. Existen versiones limitadas que pueden bajarse de www.borland.com.



Microsoft Visual J++. Uno de los más populares, aunque las aplicaciones obtenidas pueden presentar problemas de compatibilidad con el SDK oficial de Java, por el uso de librerías especificas de Microsoft. Permite construir aplicaciones Java dentro de la plataforma .NET.



Otros IDE son Eclipse, el cual se detallará en el apartado 15, IBM WebSphere, Oracle JDeveloper, etc.

Java vrs otros lenguajes OO

	Java	C#	C++	Eiffel	Smalltalk
Año Aparición	1995	2000	1985	1985	1970
Sintaxis	ins. C++	ins. Java	ins. C	ins. Pascal	original
Difusión	Amplia	Amplia	Amplia	Limitada	Limitada
Librería de clases	Muy Amplia	Muy Amplia	Escasa	Amplia	Amplia
Recolector basura	Si	Si	No	Si	Si
Manejo objetos	Dinámico	Dinámico	Est./Din.	Est./Din.	Dinámico
Tipo ejecutable	Bytecode	IL Code	binario	binario	Byte codes
Ejecución	mediante JVM	mediante CLR	directa	directa	mediante SVM
Velocidad ejecutable	media	media	muy alta	alta	baja
Soporte excepciones	Si	Si	Si	Si	Si
Herencia múltiple	No	No	Si	Si	No
Soporte operadores	Muy Limitado	Limitado	Si	Si	No
Soporte plantillas	No	No	Si	Si	No

9. Tipos de datos, variables y matrices

9.1 Tipos de datos

En Java existen dos tipos principales de datos:

1) Tipos de datos simples.
2) Referencias a objetos.

Los tipos de datos simples son aquellos que pueden utilizarse directamente en un programa, sin necesidad del uso de clases (POO). Estos tipos son:

byte	**short**	**int**	**long**
float	**double**	**char**	**boolean**

El segundo tipo está formado por todos los demás. Se les llama **referencias** porque en realidad lo que se almacena en los mismos son punteros a zonas de memoria donde se encuentran almacenadas las estructuras de datos que los soportan. Dentro de este grupo se encuentran las clases (objetos) y también se incluyen las interfaces, los vectores y los Strings.

Pueden realizarse conversiones entre los distintos tipos de datos (incluso entre simples y referenciales), bien de forma implícita o de forma explícita.

9.2 Tipos de datos simples

Los tipos de datos simples soportados por Java son los siguientes:

TIPO	Descripción	Formato	long.	Rango
byte	byte	C-2*	1 byte	- 128 … 127
short	entero corto	C-2	2 bytes	- 32.768 … 32.767
int	entero	C-2	4 bytes	- 2.147.483.648 … 2.147.483.647
long	entero largo	C-2	8 bytes	- 9.223.372.036.854.775.808… 9.223.372.036.854.775.807
float	real en coma flotante de s.p. **	IEEE 754	32 bits	$\pm 3{,}4*10^{-38}$ … $\pm 3{,}4*10^{38}$
double	real en coma flotante de d.p.	IEEE 754	64 bits	$\pm 1{,}7*10^{-308}$ … $\pm 1{,}7*10^{308}$
char	carácter	Unicode	2 bytes	0 … 65.535
boolean	lógico		1 bit	`true` / `false`

C-2 = Complemento a dos. ** s.p. = Simple Precisión / d.p. = Doble Precisión

No existen más datos simples en Java. Incluso éstos que se enumeran son envueltos por clases equivalentes (java.lang.Integer, java.lang.Double, java.lang.Byte, etc.), que pueden tratarlos como si fueran objetos en lugar de datos simples.
A diferencia de otros lenguajes de programación como el C, en Java los tipos de datos simples no dependen de la plataforma ni del sistema operativo. Un entero de tipo int siempre tendrá 4 bytes, por lo que no tendremos sorpresas al migrar un programa de un sistema operativo a otro. Es más, ni siquiera hay que volverlo a compilar.
Eso sí, Java no realiza una comprobación de los rangos. Por ejemplo: si a una variable de tipo short con el valor 32.767 se le suma 1, el resultado será

-32.768 y no se producirá ningún error de ejecución.

Nota: A diferencia de otros lenguajes de programación, los Strings en Java no son un tipo simple de datos sino un objeto. Los valores de tipo **String** van entre comillas dobles (“Hola”), mientras que los de tipo **char** van entre comillas simples (‘K’).

Los valores que pueden asignarse a variables y que pueden ser utilizados en expresiones directamente reciben el nombre de literales.

9.3 Tipos de datos referenciales

El resto de tipos de datos que no son simples, son considerados referenciales. Estos tipos son básicamente las clases, en las que se basa la programación orientada a objetos.

Al declarar un objeto perteneciente a una determinada clase, se está reservando una zona de memoria[19] donde se almacenarán los atributos y otros datos pertenecientes a dicho objeto. Lo que se almacena en el objeto en sí, es un puntero (referencia) a dicha zona de memoria.

Dentro de estos tipos pueden considerarse las interfaces, los *Strings* y los vectores, que son unas clases un tanto especiales, y que se verán en detalle posteriormente.

Existe un tipo referencial especial nominado por la palabra reservada null que puede ser asignado a cualquier variable de cualquier clase y que indica que el puntero no tiene referencia a ninguna zona de memoria (el objeto no está inicializado).

Además, todos los tipos de datos simples vistos en el punto anterior pueden ser declarados como referenciales (objetos), ya que existen clases que los engloban.

Estas clases son:

19 En realidad, el momento en el que se realiza la reserva física del espacio de memoria es cuando se instancia el objeto realizando la llamada a su constructor, y no en el momento de la declaración.

Tipo de datos simple	Clase equivalente
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Clases que Engloban datos simples

9.4 Identificadores y variables

Los identificadores son los nombres que se les da a las variables, clases, interfaces, atributos y métodos de un programa.

Reglas para la creación de identificadores:
1. Java hace distinción entre mayúsculas y minúsculas, por lo tanto, nombres o identificadores como var1, Var1 y VAR1 son distintos.

2. Pueden estar formados por cualquiera de los caracteres del código Unicode, por lo tanto, se pueden declarar variables con el nombre: añoDeCreación, raïm, etc. (se acabó la época de los nombres de variable como ano_de_creacion), aunque eso sí, el primer carácter no puede ser un dígito numérico y no pueden utilizarse espacios en blanco ni símbolos coincidentes con operadores.

3. La longitud máxima de los identificadores es prácticamente ilimitada.

4. No puede ser una palabra reservada del lenguaje ni los valores lógicos true o false.

5. No pueden ser iguales a otro identificador declarado en el mismo ámbito.

6. Por convenio, los nombres de las variables y los métodos deberían empezar por una letra minúscula y los de las clases por mayúscula.

Además, si el identificador está formado por varias palabras la primera se escribe en minúsculas (excepto para las clases) y el resto de palabras se hace empezar

por mayúscula (por ejemplo: añoDeCreación). Estas reglas no son obligatorias, pero son convenientes ya que ayudan al proceso de codificación de un programa, así como a su legibilidad. Es más sencillo distinguir entre clases y métodos o variables.

Ejemplos válidos
añoDeNacimiento2
_otra_variable
NombreDeUnaVariableMuyLargoNoImporta
BotónPulsación

Ejemplos NO válidos	Razón
3valores	(número como primer carácter)
Dia&mes	&
Dos más	(espacio)
Dos-valores	-

Ya que el lenguaje permite identificadores todo lo largos que se desee, es aconsejable crearlos de forma que tengan sentido y faciliten su interpretación. El nombre ideal para un identificador es aquel que no se excede en longitud (lo más corto posible) siempre que califique claramente el concepto que intenta representar. Siempre dentro de unos límites; por ejemplo, no sería muy adecuado utilizar un identificador de un índice de un Ciclo como **indiceDeTalCiclo** en lugar de simplemente **i**.

Hay que evitar identificadores como a1, a2, a3, a4, va1, xc32, xc21, xsda, … y en general todos aquellos identificadores que no “signifiquen” nada.

Declaración de variables

La declaración de una variable se realiza de la misma forma que en C. Siempre contiene el nombre (identificador de la variable) y el tipo de dato al que pertenece. El ámbito de la variable depende de la localización en el programa donde es declarada.

Ejemplo:

```
int x;
```

Las variables pueden ser inicializadas en el momento de su declaración, siempre que el valor que se les asigne coincida con el tipo de dato de la variable.

Ejemplo:

```
int x = 0;
```

Ámbito de una variable

El ámbito de una variable es la porción de programa donde dicha variable es visible para el código del programa y, por tanto, referenciable.

El ámbito de una variable depende del lugar del programa donde es declarada, pudiendo pertenecer a cuatro categorías distintas.
· Variable local.
· Atributo.
· Parámetro de un método.
· Parámetro de un tratador de excepciones.

Como puede observarse, NO existen las variables globales. Esto no es un "defecto" del lenguaje sino todo lo contrario. La utilización de variables globales resulta peligrosa, ya que puede ser modificada en cualquier parte del programa y por cualquier procedimiento.

Además, a la hora de utilizarlas hay que buscar dónde están declaradas para conocerlas y dónde son modificadas para evitar sorpresas en los valores que pueden contener.

Los ámbitos de las variables u objetos en Java siguen los criterios "clásicos", al igual que en la mayoría de los lenguajes de programación como Pascal, C++, etc. No existen sorpresas.

Si una variable no ha sido inicializada, tiene un valor asignado por defecto. Este valor es, para las variables de tipo referencial (objetos), el valor null. Para las variables de tipo numérico, el valor por defecto es cero (0), las variables de tipo char, el valor '\u0000' y las variables de tipo boolean, el valor false.

Variables locales
Una variable local se declara dentro del cuerpo de un método de una clase y es visible únicamente dentro de dicho método.
Se puede declarar en cualquier lugar del cuerpo, incluso después de instrucciones ejecutables, aunque es una buena costumbre declararlas justo al principio.

También pueden declararse variables dentro de un bloque patentizado por llaves {…}. En ese caso, sólo serán "visibles" dentro de dicho bloque.

```
class Caracter {
	char ch;
	public Caracter(char c) {
	ch=c;
}
	public void repetir(int num) {
		int i;
		for (i=0;i<num;i++)
		System.out.println(ch);
	}
```

```
}
class Ej1 {
	public static void main(String argumentos[]) {
	Caracter caracter;
	caracter = new Caracter('H');
	caracter.repetir(20);
	}
}
```

En este ejemplo existe una variable local: `int i;` definida en el método `repetir` de la clase `Caracter`, por lo tanto, únicamente es visible dentro del método `repetir`.

También existe una variable local en el método main. En este caso, la variable local es un objeto:
`Caracter caracter;` → que sólo será visible dentro del método en el que está declarada (`main`).

Es importante hacer notar que una declaración como la anterior le indica al compilador el tipo de la variable caracter pero no crea un objeto. El operador que crea el objeto es new, que necesita como único parámetro el nombre del constructor (que será el procedimiento que asigna valor a ese objeto recién instanciado).

Cuando se pretende declarar múltiples variables del mismo tipo pueden declararse, en forma de lista, separadas por comas:

Ejemplo:

```
int x,y,z;
```

· Declara tres variables x, y, z de tipo entero.
· Podrían haberse inicializado en su declaración de la forma:

```
int x=0,y=0,z=3;
```

No es necesario que se declaren al principio del método. Puede hacerse en cualquier lugar del mismo, incluso de la siguiente forma:

```
public void repetir(int num) {
	for (int i=0;i<num;i++)
	System.out.println(ch);
}
```

En el caso de las variables locales, éstas no se inicializan con un valor por defecto, como se ha comentado en el punto anterior, sino que es necesario asignarles algún valor antes de poder utilizarlas en cualquier instrucción, de lo contrario el compilador genera un error, de tal forma que es **imposible** hacer uso de una variable local no inicializada sin que se percate de ello el compilador.

Las variables locales pueden ser antecedidas por la palabra reservada final. En ese caso, sólo permiten que se les asigne un valor una única vez.

Ejemplo:

```
final int x=0;
```

No permitirá que a x se le asigne ningún otro valor. Siempre contendrá 0.

No es necesario que el valor se le asigne en el momento de la declaración, podría haberse inicializado en cualquier otro lugar, pero una sola vez:

Ejemplo:

```
final int x;
…
x=y+2;
```

Después de la asignación x=y+2, no se permitirá asignar ningún otro valor a x.

Atributos

Los atributos de una clase son las características que se van a tener en cuenta sobre un objeto y por lo tanto su ámbito está circunscrito, en principio, dentro de la clase a la cual caracterizan. Se declaran de la misma forma que las variables locales pero, además, pueden tener algunos modificadores que afectan al ámbito de los mismos.

En el ejemplo anterior, ch es un atributo de la clase Caracter y por lo tanto es "manipulable" en cualquier método de dicha clase, como de hecho ocurre en los método repetir () y Carácter ().

Para acceder a un atributo de un objeto desde algún método perteneciente a otra clase u objeto se antepone el nombre del objeto y un punto al nombre de dicho atributo. Por ejemplo:

caracter.ch

Con los nombres de los métodos se hace lo mismo.

Ejemplo:

caracter.repetir(20);

Parámetros de un método

Los parámetros se declaran en la cabecera del método de la siguiente forma:

```
[Modificadores_de_método] Tipo_devuelto Nombre_de_método (lista_de_parámetros)
{
        ...
}
```

La *lista_de_parámetros* consiste en una serie de variables, separadas por comas y declarando el tipo al que pertenecen.

Ejemplo:

```
public static void miMétodo(int v1, int v2, float v3, String
v4, ClaseObjeto v5);
```

Nótese que aunque existan varios parámetros pertenecientes al mismo tipo o clase, no pueden declararse abreviadamente, como ocurre con las variables locales y los atributos, indicando el tipo y a continuación la lista de parámetros separados por comas. Así, es ilegal la siguiente declaración del método anterior:

```
public static void miMétodo(int v1, v2, float v3, String v4,
ClaseObjeto v5); → (ILEGAL)
```

La declaración de un parámetro puede ir antecedida, como ocurre con las variables locales, por la palabra reservada final. En ese caso, el valor de dicho parámetro no podrá ser modificado en el cuerpo del método.

Los parámetros de un método pueden ser de dos tipos:

- **Variables de tipo simple de datos:** En este caso, el paso de parámetros se realiza siempre por valor. Es decir, el valor del parámetro de llamada no puede ser modificado en el cuerpo del método (El método trabaja con una copia del valor utilizado en la llamada).
- **Variables de tipo objeto (referencias):** En este caso, lo que realmente se pasa al método es un puntero al objeto y, por lo tanto, el valor del parámetro de llamada sí que puede ser modificado dentro del método (El método trabaja directamente con el valor utilizado en la llamada), a no ser que se anteponga la palabra reservada final.

Tipo del parámetro	Método de pase de parámetro
Tipo simple de datos (ejemplo: int, char, boolean, double, etc.)	POR VALOR
Tipo referencial (Objetos de una determinada clase, vectores y Strings)	POR REFERENCIA

```
class Objeto {
     int variable;
     public Objeto(int var){
            variable=var;
     }
}

class Parametros {
     public static void main (String argumentos[]) {
            int var1;
            Objeto obj;
            obj = new Objeto(1);
            var1 = 2;
            System.out.println("Valor del objeto = "+obj.variable);
            System.out.println("Valor de la variable = "+var1);
            modifica(var1,obj);
            System.out.println("-Después de llamar a modifica()-");
            System.out.println("Valor del objeto = "+obj.variable);
            System.out.println("Valor de la variable = "+var1);
     }
     static void modifica (int vv, Objeto oo) {
            vv++;
            oo.variable++;
     }
}
```

La salida del programa sería la siguiente:

Valor del objeto = 1

Valor de la variable = 2

-Después de llamar a modifica ( )-

Valor del objeto = 2

Valor de la variable = 2

Como puede verse, después de la llamada, el valor del objeto obj sí ha sido modificado (se ha realizado un pase de parámetro por referencia), mientras que el valor de la variable var1 no ha sido modificado (se ha realizado un paso de parámetro por valor).

9.5 Conversión de Tipos

La palabra *conversión* se utiliza con el sentido de "convertir a un molde". Java convertirá automáticamente un tipo de datos en otro cuando sea adecuado. Por ejemplo, si se asigna un valor entero a una variable de coma flotante, el compilador convertirá automáticamente el int en float. La conversión permite llevar a cabo estas conversiones de tipos de forma explícita, o forzarlas cuando no se diesen por defecto.

Para llevar a cabo una conversión, se pone el tipo de datos deseado (incluidos todos los modificadores) entre paréntesis a la izquierda de cualquier valor. He aquí un ejemplo:

```
        void conversiones ( ) {
        int i = 200;
        long 1 = (long) i;
        long 12 = (lon9)2OO;
}
```

Como puede verse, es posible llevar a cabo una conversión, tanto con un valor numérico, como con una variable. En las dos conversiones mostradas, la conversión es innecesaria, dado que el compilador convertirá un valor int en long cuando sea necesario. No obstante, se permite usar conversiones innecesarias para hacer el código más limpio. En otras situaciones, puede ser esencial una conversión para lograr que el código compile.

En C y C++, las conversiones pueden conllevar quebraderos de cabeza. En Java, la conversión de tipos es segura, con la excepción de que al llevar a cabo una de las denominadas *conversiones reductoras*(es decir, cuando se va de un tipo de datos que puede mantener más información a otro que no puede contener tanta) se corre el riesgo de perder información. En estos casos, el compilador fuerza a hacer una conversión explícita, diciendo, de hecho, "esto puede ser algo peligroso de hacer -si se quiere que lo haga de todas formas, tiene que hacer la conversión de forma explícita". Con una ***conversión extensora*** no es necesaria una conversión explícita porque el nuevo tipo es capaz de albergar la información del viejo tipo sin que se pierda nunca ningún bit.

Java permite convertir cualquier tipo primitivo en cualquier otro tipo, excepto boolean, que no permite ninguna conversión. Los tipos clase no permiten ninguna conversión. Para convertir una a otra debe utilizar métodos especiales **(String** es un caso especial y se verá más adelante en este libro que los objetos pueden convertirse en una ***familia*** de tipos; un Roble puede convertirse en Árbol y viceversa, pero esto no puede hacerse con un tipo foráneo como Roca.)

Toda expresión en Java tiene un tipo de dato asociado, ya sea simple o referencial, y puede ser deducido de la estructura de la expresión y los literales que pudiera contener.

Las conversiones de tipos pueden ser apropiadas para cambiar el tipo de una determinada expresión que no se corresponda con el tipo necesario.

Tipos de conversiones de tipos:

Conversión por ampliación: Consiste en cambiar el tipo de dato por otro cuyo rango es mayor y por lo tanto, contiene al primero. En este tipo de

conversión no se pierde información (aunque puede perderse precisión en la conversión de datos enteros a reales).

Conversión por reducción: En este tipo de conversión sí que puede perderse información debido a que se pretende almacenar más información de la que "cabe" en el tipo de destino. Al ser una conversión "semidestructiva", es necesario indicar que se pretende realizar la misma explícitamente para evitar pérdidas de información por conversiones por reducción inadvertidas.

Conversión por ampliación de tipos de datos simples

En estos casos no se pierde información sobre la magnitud de los valores numéricos.

Tipo a convertir	**Tipo resultante**
byte	short int long float double
short	int long float double
char	int long float double
int	long float double
long	float double
float	double

Para los casos de asignación no es necesario realizar ninguna conversión explícita de tipos. Por ejemplo, se puede asignar un valor de tipo byte a una variable de tipo short, int, long, float o double.

También se puede asignar un carácter (char) a una variable de tipo int, long, float o double.

```
int i = 'A';
```

Los únicos casos en los que no se preserva necesariamente la exactitud entre los valores del tipo a convertir y el tipo resultante, ya que **puede perderse precisión** son:

Tipo a convertir	**Tipo resultante**
int	float
long	float double

En estos casos de pérdida de precisión, se redondea al valor más cercano.
Ejemplo de pérdida de precisión:

```
class Conversion1 {
      public static void main(String arg[]) {
            int i = 1234567890;
            float f = i;
            System.out.println("f="+f);
            // (int)f convierte f, de tipo float, a entero
            System.out.println(i-(int)f);
      }
}
```

Produce la siguiente salida:

```
f=1.23456794E9
-46
```

Esto es debido a que el tipo float sólo es preciso hasta 8 dígitos significativos.

Conversión por reducción de tipos de datos simples
En estos casos puede perderse información ya que los rangos de representación son distintos, no estando el del tipo de dato a convertir incluido en el rango del tipo de destino.

La forma de convertir de un tipo a otro es trasladando los *n* bits menos significativos a la zona de memoria de destino con capacidad para esos *n* bits. De esta forma puede perderse incluso el signo de valores numéricos (representados en complemento a dos).

Tipo a convertir	Tipo resultante	Tipo a convertir	Tipo resultante
byte	char	double	byte short char Int Long float
short	byte char		
char	byte short		
int	byte short char		
long	byte short char int		
float	byte short char int long		

Reducción de datos simples

En estos casos, en que puede perderse información en el proceso de conversión de tipos, es necesario indicarlo explícitamente para evitar que esta pérdida se produzca de forma accidental, o por lo menos de forma inadvertida por el programador.

La forma de realizar la conversión de tipo por reducción es mediante la anteposición a la expresión a convertir, el tipo de destino encerrado entre paréntesis.

```
class Conversion2 {
      public static void main(String arg[]) {
            float f1 = 1.234567f;
            float f2 = 7.654321f;
            int i1 = (int)f1;
            int i2 = (int)f2;
            System.out.println("f1="+f1+" i1="+i1);
            System.out.println("f2="+f2+" i2="+i2);
            int i = 1234567;
            short s = (short)i;
            System.out.println("i="+i+" s="+s);
      }
}
```

Salida por pantalla:

f1=1.234567 i1=1
f2=7.654321 i2=7
i=1234567 s=-10617

Conversión por ampliación de tipos de datos referenciales

Al igual que ocurre con los tipos de datos simples, también para los objetos pueden realizarse conversiones tanto por ampliación como por reducción.

Las conversiones por ampliación permitidas son:

Tipo a convertir	**Tipo resultante**
La clase null	- Cualquier clase, interfase o vector
Cualquier clase C	- Cualquier clase R siempre que C sea subclase de R - Cualquier interfase I si C implementa I - La clase Object
Cualquier interfase I	- Cualquier interfase K siempre que I sea subinterfaz de K - La clase Object
Cualquier vector A	- La clase Object - La clase Cloneable - Cualquier vector R siempre que el tipo de los elementos de A y R sean datos referenciales y exista una conversión por ampliación entre ellos.

En estos casos no es necesaria ninguna conversión explícita.

```
class C1 {
	public String clase;
	public C1() {
		clase = "clase c1";
	}
}
class C2 extends C1{
	public C2() {
		clase = "clase c2";

	}
}

class Conversion3 {
	public static void main(String arg[]) {
		C1 c1 = new C2();
		System.out.println(c1.clase);
	}
}
```

Produce la siguiente salda por pantalla:

Clase c2

En el ejemplo, se ha declarado un objeto c1 de la clase C1, que es subclase de C2. A c1 se le asigna un objeto de la clase C2 (al realizar la llamada al constructor de C2). Esto puede hacerse sin ninguna conversión explícita ya que C2 es subclase de C1. Casi gráficamente puede entenderse mediante un ejemplo de la siguiente forma: imaginemos una clase Ave y una subclase Pato; si se dispone de una variable de la clase Ave, a dicha variable se le puede asignar un "Pato", ya que un pato es, efectivamente, un ave. Por lo tanto, a una variable de la clase Ave, se le puede asignar un objeto "Pato" sin ninguna conversión explícita ya que es un proceso "natural". El caso contrario no siempre es cierto. Si se dispone de una variable de la clase Pato, no se le puede asignar un objeto de la clase "Ave", ya que cualquier ave no es un pato.

Si se dispone de una variable de la clase Pato y se le desea asignar un objeto de la clase Ave, habrá que realizar una conversión explícita (ver punto siguiente) para "asegurar" que el ave que asignamos es "realmente" un pato.

Conversión por reducción de tipos de datos referenciales

Las conversiones por reducción permitidas son:

Tipo a convertir	**Tipo resultante**
La clase Object	- Cualquier vector - Cualquier interfase - Cualquier clase
Cualquier clase C	- Cualquier clase R siempre que C sea superclase de R - Cualquier interfase I si C no es final y no implementa I
Cualquier interfase I	- Cualquier clase C que no sea final - Cualquier clase C que sea final siempre que C implemente I - Cualquier interfase K siempre que I no sea subinterface de K y no tengan métodos con el mismo nombre pero distinto tipo de dato de retorno.
Cualquier vector A	- Cualquier vector R siempre que el tipo de los elementos de A y R sean datos referenciales y exista una conversión por reducción entre ellos.

En estos casos es necesario indicarlo explícitamente.

La forma de realizar la conversión de tipo por reducción es mediante la anteposición a la expresión a convertir, el tipo de destino encerrado entre paréntesis.

El siguiente ejemplo muestra una conversión no permitida y genera un mensaje de error por el compilador:

```
class C1 {
      public String clase;
      public C1() {
            clase = "clase c1";
      }
}
class C2 extends C1{
      public C2() {
            clase = "clase c2";
      }
}

class Conversion4 {
      public static void main(String arg[]) {
            C1 c1 = new C1();
            C2 c2 = c1;
            System.out.println(c2.clase);
      }
}
```

```
Conversion4.java:16: Incompatible type for declaration. Explicit
cast needed to convert C1 to C2.
      C2 c2 = c1;
      ^
      1 error
```

La conversión de un objeto de la clase C1 a la clase C2 necesita del programador la indicación de que se sabe que se está asignando un valor de la clase C1 a una variable de la clase C2, pero que en realidad c1 no contiene un objeto de la clase C1 sino un objeto de la clase C2.

Para realizar la conversión en el ejemplo anterior se antepone la clase a convertir encerrada entre paréntesis de la siguiente manera:

```
class Conversion4 {
      public static void main(String arg[]) {
            C1 c1 = new C1();
            // Conversión por reducción de tipo de
            // dato referencial
            C2 c2 = (C2) c1;
            System.out.println(c2.clase);
      }
}
```

Con esta conversión explícita se indica que el objeto almacenado en c1 es, en realidad, de la clase C2 (lo cuál no es cierto). Ahora el compilador no genera ningún error, pero al ejecutar el programa se genera una excepción:

```
java.lang.ClassCastException:
at Conversion5.main(Conversion5.java:17)
```

No se habría generado ninguna excepción si en c1 hubiera un objeto de la clase C2 de la siguiente forma:

```
class Conversion4 {
      public static void main(String arg[]) {
            C1 c1 = new C2(); // Conversión por ampliación
            // Conversión por reducción de tipo de dato
            // referencial
            C2 c2 = (C2) c1;
            System.out.println(c2.clase);
      }
}
```

La salida por pantalla sería:

Clase c2

Conversiones de tipos no permitidas

Tipo a convertir	**Tipo resultante**
Tipo referencial	Tipo simple
Tipo simple	Tipo referencial (Excepto al tipo String)
Null	Tipo simple
Cualquier tipo	Tipo boolean
Tipo boolean	Cualquier otro tipo (Excepto al tipo String)
Cualquier clase C	- Cualquier clase R siempre que C no sea subclase de R y R no sea subclase de C (Excepto a la clase String) - Cualquier interfase I si C es “final” y no implementa I - Cualquier vector (Excepto si C es de la clase Object)
Cualquier interfase I	- Cualquier clase C (Excepto si C es String) si es “final” y no implementa I - Cualquier interfase K si I y K declaran métodos con la misma cabecera y distinto tipo de dato de retorno.
Cualquier vector	- Cualquier interfase excepto al interfase

	Cloneable. - Cualquier otro vector si no hay conversión entre los tipos de los elementos (excepto la conversión a String).

9.6 Vectores o Matrices

Para manejar colecciones de objetos del mismo tipo estructurados en una sola variable se utilizan los vectores[20]. En Java, los vectores son en realidad objetos y por lo tanto se puede llamar a sus métodos (como se verá en el capítulo siguiente).

Existen dos formas equivalentes de declarar vectores en Java:

1) tipo nombreDelVector [ ];
2) tipo [ ] nombreDelVector;

Ejemplo:

```
int vector1[], vector2[], entero; //entero no es un vector
int[] otroVector;
```

También pueden utilizarse vectores de más de una dimensión:

Ejemplo:

```
int matriz [] [];
int [] [] otraMatriz;
```

Los vectores, al igual que las demás variables pueden ser inicializados en el momento de su declaración. En este caso, no es necesario especificar el número de elementos máximo reservado. Se reserva el espacio justo para almacenar los elementos añadidos en la declaración.

Ejemplo:

```
String Días []= {"Lunes","Martes","Miércoles","Jueves",
"Viernes","Sábado","Domingo"};
```

Una simple declaración de un vector no reserva espacio en memoria, a excepción del caso anterior, en el que sus elementos obtienen la memoria necesaria para ser almacenados. Para reservar la memoria hay que llamar explícitamente a un constructor new[21] de la siguiente forma:

new tipoElemento[numElementos] **;**

20 También llamados arrays, matrices o tablas.
21 Véase la sección 12.4

Ejemplo:

```
int matriz[][];
matriz = new int[4][7];
```

También se puede indicar el número de elementos durante su declaración:

Ejemplo:

```
int vector[]= new int[5];
```

Para hacer referencia a los elementos particulares del vector, se utiliza el identificador del vector junto con el índice del elemento entre corchetes. El índice del primer elemento es el cero (0) y el del último, el número de elementos menos uno.

Ejemplo:

```
j = vector[0]; vector[4]= matriz[2][3];
```

El intento de acceder a un elemento fuera del rango de la matriz, a diferencia de lo que ocurre en C, provoca una excepción (error) que, de no ser manejado por el programa, será el compilador quien aborte la operación.

Para obtener el número de elementos de un vector en tiempo de ejecución se accede al atributo de la clase vector length. No olvidemos que los vectores en Java son tratados como un objeto.

```
class Array1 {
      public static void main (String argumentos[]) {
            String colores[] = {"Rojo","Verde","Azul",
                                    "Amarillo","Negro"};
            int i;
            for (i=0;i<colores.length;i++)
                  System.out.println(colores[i]);
      }
}
```

El ejemplo anterior produce la siguiente salida por pantalla:

```
Rojo
Verde
Azul
Amarillo
Negro
```

Arrays multidimensionales

Java permite crear fácilmente arrays multidimensionales:

```
/ / : c04:ArrayMultidimensional.java
/ / creando arrays multidimensionales.
import java.util.*;
      public class ArrayMultidimensional {
            static Random aleatorio = new Random();
```

```
          static int pAleatorio (int modulo) {
          return Math. abs (aleatorio. nextInt ( ) ) % modulo + 1;
          }
     static void visualizar (String S) {
          System.out.println (S);
     }
     public static void main (String[] args) {
          int[l [ ] al = {
          { 1, 2, 3}
          { 4, 5 , 6}
     };
     for(int i = O; i < al.length; i++)
          for (int j = O; j < al [i] . length; j++)
          visualizar ("al [ " + i + "1 [ " + j +
          "j = " + al[i] [j]);

/ / array 3-D de longitud fija:
     int [] [] [] a2 = new int [2] [2] [4];
          for (int i = O; i < a2.length; i+t)
               for (int j = O; j < a2 [i] . length; jt+)
                    for(int k = O; k < a2[i] [j] .length;
                    k++)
               visualizar ("a2 [ " + i + "1 [ " + j + "][" + k +
               "J = "+ a2 [i][j][k]);
/ / array 3-D con vectores de longitud variable:
     int [] [ ] [] a3 = new int [pAleatorio (7) ] [] [] ;
          for (int i = O; i < a3.length; i++) {
               a3 [i] = new int [pAleatorio (5) ] [] ;
               for(int j = O; j < a3[i] .length; j++)
               a3[il [jl = new int[pAleatorio(5)1;
          }
     for (int i = O; i < a3.length; i++)
          for (int j = O; j < a3 [i] .length; j++)
               for(int k = O; k < a3[i] [j] .length;
                    k++)
          visualizar ("a3 [ " + i + "1 [ " + j + "][" + k +
                    "J = " + a3 [il [jl [kl ) ;
/ / Array de objetos no primitivos:
     Integer [] [] a4 = {
          { new Integer (1) , new Integer (2) } ,
               { new Integer (3) , new Integer ( 4 ) },
               { new Integer (S), new Integer (6) } ,
          };
     for(int i = O; i < a4.length; i++)
          for (int j = O; j < a4 [i] . length; j++)
          visualizar("a4[" + i + "1 [ " + j +
                    " J = " + a4 [i] [j]);
          Integer [] [ ] a5;
          a5 = new Integer [3] [] ;
          for(int i = O; i < a5.length; i++) {
               a5[i] = new Integer[3];
```

```
                for (int j = 0; j < a5 [i] .length; j++)
                a5 [i] [ j ] = new Integer (i* j) ;
     }
     for(int i = 0; i < a5.length; i++)
          for(int j = 0; j < a5[i] .length; j++)
          visualizar ("a5 [ " + i + "1 [ " + j +
                    " J = "J + a5[i] [j]);
     }
} / / / : -
```

El código utilizado para imprimir utiliza el método **length,** de forma que no depende de tamaños fijos de array.

El primer ejemplo muestra un array multidimensional de tipos primitivos. Se puede delimitar cada vector del array por llaves:

```
int[][] a1 = {
        { 1, 2, 3, },
        { 4, 5, 6, },
     };
```

Cada conjunto de corchetes nos introduce en el siguiente nivel del array.

El segundo ejemplo muestra un array de tres dimensiones asignado con **new.** Aquí, se asigna de una sola vez todo el array:

```
int [] [] [] a2 = new int [2] [2] [4] ;
```

Pero el tercer ejemplo muestra que cada vector en los arrays que conforman la matriz puede ser de cualquier longitud:

```
int[][][] a3 = new int[pAleatorio(7)][][];
     for(int i = 0; i < a3.length; i++) {
       a3[i] = new int[pAleatorio(5)][];
       for(int j = 0; j < a3[i].length; j++)
         a3[i][j] = new int[pAleatorio(5)];
     }
```

El primer **new** crea un array con un primer elemento de longitud aleatoria, y el resto, indeterminados. El segundo **new** de dentro del Ciclo **for** rellena los elementos pero deja el tercer índice indeterminado hasta que se acometa el tercer **new.**

Se verá en la salida que los valores del array que se inicializan automáticamente a cero si no se les da un valor de inicialización explícito.

Se puede tratar con arrays de objetos no primitivos de forma similar, lo que se muestra en el cuarto ejemplo, que demuestra la habilidad de englobar muchas expresiones **new** entre llaves:

```
Integer[][] a4 = {
        { new Integer(1), new Integer(2)},
        { new Integer(3), new Integer(4)},
        { new Integer(5), new Integer(6)},
      };
```

El quinto ejemplo muestra cómo se puede construir pieza a pieza un array de objetos no primitivos:

```
Integer[][] a5;
      a5 = new Integer[3][];
      for(int i = 0; i < a5.length; i++) {
         a5[i] = new Integer[3];
         for(int j = 0; j < a5[i].length; j++)
            a5[i][j] = new Integer(i*j);
      }
```

El **i*j** es simplemente para poner algún valor interesante en el **Integer.**

10 Operadores

Los operadores son partes indispensables en la construcción de expresiones.

Existen muchas definiciones técnicas para el término expresión. Puede decirse que una expresión es una combinación de operandos ligados mediante operadores.

Los operandos pueden ser variables, constantes, funciones, literales, etc. y los operadores todos los enumerados en este punto.

10.1 Operadores aritméticos:

Operador	**Formato**	**Descripción**
+	op1 + op2	Suma aritmética de dos operandos
-	op1 - op2 -op1	Resta aritmética de dos operandos Cambio de signo
*	op1 * op2	Multiplicación de dos operandos
/	op1 / op2	División entera de dos operandos
%	op1 % op2	Resto de la división

		entera (o módulo)
++	++op1 op1++	Incremento unitario
--	-- op1 op1 --	Decremento unitario

El operador - puede utilizarse en su versión unaria (- op1) y la operación que realiza es la de invertir el signo del operando.

Como en C, los operadores unarios ++ y -- realizan un incremento y un decremento respectivamente. Estos operadores admiten notación prefija y postfija.

- ++op1: En primer lugar realiza un incremento (en una unidad) de op1 y después ejecuta la instrucción en la cual está inmerso.
- op1++: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un incremento (en una unidad) de op1.
- --op1: En primer lugar realiza un decremento (en una unidad) de op1 y después ejecuta la instrucción en la cuál está inmerso.
- op1--: En primer lugar ejecuta la instrucción en la cual está inmerso y después realiza un decremento (en una unidad) de op1.

La diferencia entre la notación prefija y la postfija no tiene importancia en expresiones en las que únicamente existe dicha operación:

++contador; es equivalente a: **contador++;**
--contador; **contador--;**

Es importante no emplear estos operadores en expresiones que contengan más de una referencia a la variable incrementada, puesto que esta práctica puede generar resultados erróneos fácilmente.

La diferencia es apreciable en instrucciones en las cuáles están incluidas otras operaciones.

Por ejemplo:

```
cont = 1;                cont = 1;
do {                     do {
     …}                       …}
while ( cont++ < 3 ); while ( ++cont < 3 );
```

En el primer caso, el Ciclo se ejecutará 3 veces, mientras que en el segundo se ejecutará 2 veces.

Otro ejemplo:

```
a = 1;          a = 1;
b = 2 + a++;    b = 2 + ++a;
```

- En el primer caso, después de las operaciones, b tendrá el valor 3 y a el valor 2.
- En el segundo caso, después de las operaciones, b tendrá el valor 4 y a el valor 2.

10.2 Operadores relacionales

Operador	Formato	Descripción
>	op1 > op2	Devuelve true (cierto) si op1 es mayor que op2
<	op1 < op2	Devuelve true (cierto) si op1 es menor que op2
>=	op1 >= op2	Devuelve true (cierto) si op1 es mayor o igual que op2
<=	op1<= op2	Devuelve true (cierto) si op1 es menor o igual que op2
==	op1 == op2	Devuelve true (cierto) si op1 es igual a op2
!=	op1 != op2	Devuelve true (cierto) si op1 es distinto de op2

Los operadores relacionales actúan sobre valores enteros, reales y caracteres (char); y devuelven un valor del tipo boolean (true o false).

```
Class Relacional {
      public static void main(String arg[]) {
            double op1,op2;
            op1=1.34;
            op2=1.35;
            System.out.println("op1="+op1+" op2="+op2);
            System.out.println("op1>op2 = "+(op1>op2));
            System.out.println("op1<op2 = "+(op1<op2));
            System.out.println("op1==op2 = "+(op1==op2));
            System.out.println("op1!=op2 = "+(op1!=op2));
            char op3,op4;
            op3='a'; op4='b';
            System.out.println("'a'>'b' = "+(op3>op4));
            }
      }
```

Salida por pantalla:

```
      op1=1.34 op2=1.35
      op1>op2 = false
      op1<op2 = true
      op1==op2 = false
      op1!=op2 = true
      'a'>'b' = false
```

Los operadores == y != actúan también sobre valores lógicos (boolean).

10.3 Operadores lógicos

Operador	Formato	Descripción
&&	op1 && op2	Y lógico. Devuelve true (cierto) si son ciertos op1 y op2
½½	op1 ½½ op2	O lógico. Devuelve true (cierto) si son ciertos op1 o op2
!	! op1	Negación lógica. Devuelve true (cierto) si es false op1.

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso (true / false).

```
class Bool {
      public static void main ( String argumentos[] ) {
            boolean a=true;
            boolean b=true;
            boolean c=false;
            boolean d=false;
            System.out.println("true Y true = " + (a && b) );
            System.out.println("true Y false = " + (a && c) );
            System.out.println("false Y false = " + (c && d) );
            System.out.println("true O true = " + (a || b) );
            System.out.println("true O false = " + (a || c) );
            System.out.println("false O false = " + (c || d) );
            System.out.println("NO true = " + !a);
            System.out.println("NO false = " + !c);
            System.out.println("(3 > 4) Y true = " + ((3 >4) && a) );
            }
      }
```

Produciría la siguiente salida por pantalla:

```
true Y true = true
true Y false = false
false Y false = false
true O true = true
true O false = true
false O false = false
NO true = false
NO false = true
(3 > 4) Y true = false
```

10.4 Operadores de asignación

El operador de asignación es el símbolo igual (=).

op1 = Expresión

Asigna el resultado de evaluar la expresión de la derecha a op1.

Además del operador de asignación existen unas abreviaturas cuando el operando que aparece a la izquierda del símbolo de asignación también aparece a la derecha del mismo:

Operador	Formato	Equivalencia
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
>>=	op1 >>= op2	op1 = op1 >> op2
<<=	op1 <<= op2	op1 = op1 << op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

10.5 Precedencia de operadores en Java

Operadores postfijos	[ ] . (paréntesis)
Operadores unarios	++expr --expr -expr ~ !
Creación o conversión de tipo	new (tipo)expr
Multiplicación y división	* / %
Suma y resta	+ -
Desplazamiento de bits	<< >> >>>
Relacionales	< > <= >=
Igualdad y desigualdad	== !=
AND a nivel de bits	&
XOR a nivel de bits	^
OR a nivel de bits	|
AND lógico	&&
OR lógico	||
Condicional al estilo C	? :
Asignación	= += -= *= /= %= ^= &= |= >>= <<= >>>=

11. Sentencias de Control

Las estructuras de control son construcciones hechas a partir de palabras reservadas del lenguaje que permiten modificar el flujo de ejecución de un programa. De este modo, pueden crearse construcciones de alternativas y Ciclos de repetición de bloques de instrucciones.

Hay que señalar que un bloque de instrucciones se encontrará encerrado mediante llaves {……..} si existe más de una instrucción.

11.1 Sentencias de Selección

Las sentencias de selección o estructuras alternativas son construcciones que permiten alterar el flujo secuencial de un programa, de forma que en función de una condición o el valor de una expresión, el mismo pueda ser desviado en una u otra alternativa de código.

Las estructuras alternativas disponibles en Java son:

- Alternativa if-else.
- Alternativa switch.

If-else

Forma simple:

if (expresión**)**
    Bloque instrucciones

El bloque de instrucciones se ejecuta si, y sólo si, la *expresión* (que debe ser lógica) se evalúa a true, es decir, se cumple una determinada condición.

```
if (cont == 0)
     System.out.println("he llegado a cero");
```

La instrucción System.out.println("he llegado a cero"); sólo se ejecuta en el caso de que cont contenga el valor cero.

Con cláusula else:

if (expresión**)**
    Bloque instrucciones 1
else
    Bloque instrucciones 2

El bloque de instrucciones 1 se ejecuta si, y sólo si, la *expresión* se evalúa a true.
Y en caso contrario, si la expresión se evalúa a false, se ejecuta el bloque de instrucciones 2.

```
if (cont == 0)
     System.out.println("he llegado a cero");
else
     System.out.println("no he llegado a cero");
```

Si cont vale cero, se mostrará en el mensaje "he llegado a cero". Si cont contiene cualquier otro valor distinto de cero, se mostrará el mensaje "no he llegado a cero".

If-else anidados:

En muchas ocasiones, se anidan estructuras alternativas if-else, de forma que se pregunte por una condición si anteriormente no se ha cumplido otra sucesivamente. Por ejemplo: supongamos que realizamos un programa que muestra la nota de un alumno en la forma (insuficiente, suficiente, bien, notable o sobresaliente) en función de su nota numérica. Podría codificarse de la siguiente forma:

```
class Nota {
	public static void main (String argumentos[]) {
		int nota;
		if (argumentos.length<1) {
			// argumentos.length contiene el número de elementos
			// del array argumentos, que contiene los parámetros
			// suministrados en la línea de comandos.
			System.out.println("Uso: Nota num");
			System.out.println("Donde num = nº entre 0 y 10");
			}
		else
			{
			nota=Integer.valueOf(argumentos[0]).intValue();
			// la instrucción anterior convierte un
			// String a entero.
			if (nota<5)
				System.out.println("Insuficiente");
			else
				if (nota<6)
					System.out.println("Suficiente");
				else
					if (nota<7)
						System.out.println("Bien");
					else
						if (nota<9)
							System.out.println("Notable");
						else
							System.out.println("Sobresaliente")
							;
				}
			}
		}
```

En Java, como en C y a diferencia de otros lenguajes de programación, en el caso de que el bloque de instrucciones conste de una sola instrucción (no necesita ser parentizado), se pone un punto y coma (**;**) justo antes de la cláusula else.

Switch

Forma simple:

switch (expresión**) {**
	case valor1**:** instrucciones1**;**
	case valor2**:** instrucciones2**;**
	...
	case valorN**:** instruccionesN**;**
}

En este caso, a diferencia del anterior, si instrucciones1 ó instrucciones2 ó *instruccionesN* están formados por un bloque de instrucciones sencillas, no es necesario parentizarlas mediante las llaves ({**…}**).

En primer lugar se evalúa la *expresión* cuyo resultado puede ser un valor de cualquier tipo. El programa comprueba el primer valor (*valor1*). En el caso de que el valor resultado de la expresión coincida con *valor1,* se ejecutaran las *instrucciones1.* Pero ¡ojo! También se ejecutarían las instrucciones *instrucciones2. instruccionesN* hasta encontrarse con la palabra reservada break. Si el resultado de la expresión no coincide con *valor1,* evidentemente no se ejecutarían *instrucciones1*, se comprobaría la coincidencia con *valor2* y así sucesivamente hasta encontrar un valor que coincida o llegar al final de la construcción switch.

En caso de que no exista ningún valor que coincida con el de la *expresión*, no se ejecuta ninguna acción.

Si lo que se desea es que únicamente se ejecuten las instrucciones asociadas a cada valor y no todas las de los demás case que quedan por debajo, la construcción switch sería la siguiente:

```
Switch (expresión) {
        case valor1: instrucciones1;
                        break;
        case valor2: instrucciones2;
                        break;
        …
        case valorN: instruccionesN;
}
```

```
class DiaSemana {
      public static void main(String argumentos[]) {
            int día;
            if (argumentos.length<1) {
                  System.out.println("Uso: DiaSemana num");
                  System.out.println("Donde num = nº entre 1 y 7");
            }
            else {
                  día=Integer.valueOf(argumentos[0]).intValue();
                  switch (día) {
                  case 1: System.out.println("Lunes");
                              break;
                  case 2: System.out.println("Martes");
                              break;
                  case 3: System.out.println("Miércoles");
                              break;
                  case 4: System.out.println("Jueves");
                              break;
                  case 5: System.out.println("Viernes");
                              break;
                  case 6: System.out.println("Sábado");
                              break;
                  case 7: System.out.println("Domingo");
```

```
                }
            }
        }
}
```

Nótese que en el caso de que se introduzca un valor no comprendido entre 1 y 7, no se realizará ninguna acción. Esto puede corregirse utilizando el siguiente formato:

Con cláusula por defecto:

Switch (expresión**) {**
    Case valor1**:** instrucciones1**;**
    Case valor2**:** instrucciones2**;**
    …
    Case valorN**:** instruccionesN**;**
    Default: instruccionesPorDefecto**;**
}

En este caso, *instruccionesPorDefecto* se ejecutarán en el caso de que ningún valor case coincida con el valor de *expresión*. O en caso de ejecutar algunas instrucciones en alguno de los case, que no haya ninguna instrucción break desde ese punto hasta la cláusula default.

```
class DiasMes {
      Public static void main (String argumentos []) {
            int mes;
            if (argumentos.length<1) {
                  System.out.println("Uso: DiasMes num");
                  System.out.println("Donde num = nº del mes");
                  }
            else {
                  mes=Integer.valueOf(argumentos[0]).intValue();
                  switch (mes) {
                        case 1:
                        case 3:
                        case 5:
                        case 7:
                        case 8:
                        case 10:
                        case 12: System.out.println("El mes "+mes +
                                          " Tiene 31 días");
                              break;
                        case 4:
                        case 6:
                        case 9:
                        case 11: System.out.println("El mes "+mes +
                                          " Tiene 30 días");
                              break;
                        case 2: System.out.println("El mes "+mes +
                                    " Tiene 28 ó 29 días");
                              break;
                        default: System.out.println("El mes "+mes +
                                          " no existe");
```

```
                    }
                }
            }
        }
```

En este ejemplo, únicamente se ejecutará la cláusula default en el caso en que el valor de mes sea distinto a un número comprendido entre 1 y 12, ya que todas estas posibilidades se encuentran contempladas en las respectivas cláusulas case y, además, existe un break justo antes de la cláusula default, por lo que en ningún otro caso se ejecutará la misma.

11.2 Sentencias de iteración

Los Ciclos o iteraciones son estructuras de repetición. Bloques de instrucciones que se repiten un número de veces mientras se cumpla una condición o hasta que se cumpla una condición.

Existen tres construcciones para estas estructuras de repetición:

- Ciclo for.
- Ciclo do-while.
- Ciclo while.

Como regla general puede decirse que se utilizará el Ciclo for cuando se conozca de antemano el número exacto de veces que ha de repetirse un determinado bloque de instrucciones. Se utilizará el Ciclo do-while cuando no se conoce exactamente el número de veces que se ejecutará el Ciclo pero se sabe que por lo menos se ha de ejecutar una. Se utilizará el Ciclo while-do cuando es posible que no deba ejecutarse ninguna vez.

Estas reglas son generales y algunos programadores se sienten más cómodos utilizando principalmente una de ellas. Con mayor o menor esfuerzo, puede utilizarse cualquiera de ellas indistintamente.

Ciclo For

for (inicialización **;** condición **;** incremento**)**
bloque instrucciones

- La cláusula *inicialización* es una instrucción que se ejecuta una sola vez al inicio del Ciclo, normalmente para inicializar un contador.
- La cláusula *condición* es una expresión lógica, que se evalúa al inicio de cada nueva iteración del Ciclo. En el momento en que dicha expresión se evalúe a false, se dejará de ejecutar el Ciclo y el control del programa pasará a la siguiente instrucción (a continuación del Ciclo for).

- La cláusula *incremento* es una instrucción que se ejecuta en cada iteración del Ciclo como si fuera la última instrucción dentro del bloque de instrucciones. Generalmente se trata de una instrucción de incremento o decremento de alguna variable.

Cualquiera de estas tres cláusulas puede estar vacía, aunque SIEMPRE hay que poner los puntos y coma (**;**).

El siguiente programa muestra en pantalla la serie de Fibonacci[22] hasta el término que se indique al programa como argumento en la línea de comandos. (NOTA: siempre se mostrarán, por lo menos, los dos primeros términos).

```
Class Fibonacci {
      public static void main(String argumentos[]) {
            int numTerm,v1=1,v2=1,aux,cont;
            if (argumentos.length<1) {
                  System.out.println("Uso: Fibonacci num");
                  System.out.println("Donde num = nº de términos");
                  }
            else {
                  numTerm=Integer.valueOf(argumentos[0]).intValue();
                  System.out.print("1,1");
                  for (cont=2;cont<numTerm;cont++) {
                        aux=v2;
                        v2+=v1;
                        v1=aux;
                        System.out.print(","+v2);
                        }
                  System.out.println();
                  }
            }
      }
```

Ciclo do-while

do
bloque instrucciones
while (Expresión**);**

En este tipo de Ciclo, *bloque instrucciones* se ejecuta siempre una vez por lo menos, y ese bloque de instrucciones se ejecutará mientras *Expresión* se evalúe a true. Por lo tanto, entre las instrucciones que se repiten deberá existir alguna que, en algún momento, haga que la *Expresión* se evalúe a false, de lo contrario el Ciclo sería infinito.

22 La serie de Fibonacci es una serie de números enteros que comienza con 1, 1. A partir del tercer término, el siguiente término de la serie se calcula como la suma de los dos anteriores. De esta forma, la serie comienza con los siguientes enteros: 1,1,2,3,5,8,13,21,34,55, Visto en Principles of Programming I.

Ejemplo: El mismo que antes (Fibonacci).

```
class Fibonacci2 {
      public static void main(String argumentos[]) {
            int numTerm,v1=0,v2=1,aux,cont=1;
            if (argumentos.length<1) {
                  System.out.println("Uso: Fibonacci num");
                  System.out.println("Donde num = nº de términos");
                  }
            else {
                  numTerm=Integer.valueOf(argumentos[0]).intValue();
                  System.out.print("1");


                  do {
                        aux=v2;
                        v2+=v1;
                        v1=aux;
                        System.out.print(","+v2);
                  } while (++cont<numTerm);
                  System.out.println();
            }
      }
}
```

En este caso únicamente se muestra el primer término de la serie antes de iniciar el Ciclo, ya que el segundo siempre se mostrará, porque el Ciclo do-while siempre se ejecuta una vez por lo menos.

Ciclo while

while (Expresión**)**
bloque instrucciones

Al igual que en el Ciclo do-while del apartado anterior, el bloque de instrucciones se ejecuta mientras se cumple una condición (mientras *Expresión* se evalúe a true), pero en este caso, la condición se comprueba ANTES de empezar a ejecutar por primera vez el Ciclo, por lo que si *Expresión* se evalúa a false en la primera iteración, entonces el bloque de instrucciones no se ejecutará ninguna vez.

Ejemplo: Fibonacci:

```
class Fibonacci3 {
      public static void main(String argumentos[]) {
            int numTerm,v1=1,v2=1,aux,cont=2;
            if (argumentos.length<1) {
                  System.out.println("Uso: Fibonacci num");
                  System.out.println("Donde num = nº de términos");
                  }
            else {
                  numTerm=Integer.valueOf(argumentos[0]).intValue();
                  System.out.print("1,1");
```

```
                while (cont++<numTerm) {
                      aux=v2;
                      v2+=v1;
                      v1=aux;
                      System.out.print(","+v2);
                      }
                System.out.println();
                }
          }
    }
```

Como puede comprobarse, las tres construcciones de ciclo (for, do-while y while) pueden utilizarse indistintamente realizando unas pequeñas variaciones en el programa.

11.3 Sentencias de Salto

En Java existen dos formas de realizar un salto incondicional en el flujo "normal" de un programa. A saber, las instrucciones break y continue.

Dentro del cuerpo de cualquier sentencia de iteración también se puede controlar el flujo del Ciclo utilizando **break** y **continue. Break** sale del Ciclo sin ejecutar el resto de las sentencias del Ciclo. **Continue** detiene la ejecución de la iteración actual y vuelve al principio del Ciclo para comenzar la siguiente iteración.

Break

La instrucción break sirve para abandonar una estructura de control, tanto de las alternativas (if-else y switch) como de las repetitivas o Ciclos (for, dowhile y while). En el momento que se ejecuta la instrucción break, el control del programa sale de la estructura en la que se encuentra.

```
class Break {
      public static void main(String argumentos[]) {
            int i;
            for (i=1; i<=4; i++) {
                  if (i==3) break;
                  System.out.println("Itereación: "+i);
                  }
            }
      }
```

Este ejemplo produciría la siguiente salida por pantalla:

Iteración: 1
Iteración: 2

Aunque el bucle, en principio indica que se ejecute 4 veces, en la tercera iteración, *i* contiene el valor 3, se cumple la condición del if (i==3) y por lo tanto se ejecuta el break y se sale del bucle for.

Continue

La instrucción continue sirve para transferir el control del programa desde la instrucción continue directamente a la cabecera del bucle (for, do-while o while) donde se encuentra.

```
class Continue {
      public static void main(String argumentos[]) {
            int i;
            for (i=1; i<=4; i++) {
                  if (i==3) continue;
                  System.out.println("Itereación: "+i);
            }
      }
}
```

Este programa es muy similar al anterior, pero en lugar de utilizar la instrucción break, se ha utilizado continue. El resultado es el siguiente:

Iteración: 1
Iteración: 2
Iteración: 4

Puede comprobarse la diferencia con respecto al resultado del ejemplo del apartado anterior. En este caso no se abandona el bucle, sino que se transfiere el control a la cabecera del bucle donde se continúa con la siguiente iteración.

Tanto el salto break como en el salto continue, pueden ser evitados mediante distintas construcciones pero en ocasiones esto puede empeorar la legibilidad del código. De todas formas existen programadores que odian este tipo de saltos y no los utilizan en ningún caso.

12. Clases

12.1 Definición de Clase

Como se vio en un principio, todo en Java son clases (objetos). Si no se tienen claros los conceptos básicos de la programación orientada a objetos, volver al apartado 4.5.

Desde un punto de vista simplista, una clase es un conjunto de valores (atributos) junto con las funciones y procedimientos que operan sobre los mismos (métodos), todo ello tratado como una entidad. Estas clases constituyen los bloques principales en los cuales se encuentra contenido el código.

12.2 Formato general de una clase

En un archivo fuente puede declararse una o más clases:

class Clase1 {

```
        ...
}
class Clase2 {
        ...
}
        ...
class ClaseN {
        ...
}
```

El cuerpo de las clases comienza con una llave abierta (**{**) y termina con una llave cerrada (**}**).

Declaración de clase **{**
    Cuerpo de clase
}

La declaración de una clase define un tipo de dato referencial.
Dentro del cuerpo de la clase se declaran los atributos de la clase y los métodos.

12.3 Declaración de referencias a objeto y creación de objetos

La declaración mínima para una clase es la siguiente:

Class NombreClase

Una declaración de este tipo indica que la clase no desciende de ninguna otra, aunque en realidad, todas las clases declaradas en un programa escrito en Java son descendientes, directa o indirectamente, de la clase Object que es la raíz de toda la jerarquía de clases en Java.

```
class ObjetoSimpleCreado {
      String variable = "Una variable";
      int entero = 14;
      public String obtnerString() {
            return variable;
      }
}

class ObjetoSimple {
      public static void main(String arumentos[]) {
            ObjetoSimpleCreado varObj = new ObjetoSimpleCreado();
            System.out.println(varObj.toString());
      }
}
```

Muestra en pantalla la siguiente línea de texto:

ObjetoSimpleCreado@13937d8

En este caso, la clase ObjetoSimpleCreado ha sido declarada como no descendiente de ninguna otra clase, pero a pesar de ello, hereda de la superclase Object (java.lang.Object) todos sus métodos, entre los que se encuentran el método toString() que, en este caso, devuelve el siguiente valor: "ObjetoSimpleCreado@13937d8" (el nombre de la clase junto con el puntero al objeto). Este método, que heredan todas las clases que puedan declararse, debería ser redefinido por el programador para mostrar un valor más significativo.

Si en lugar de la instrucción System.out.println(varObj.toString()); se hubiera utilizado la siguiente: System.out.println(varObj.obtenerString()) la salida por pantalla habría sido:

Una variable

Declaración de la superclase (herencia)

La superclase es la clase de la cual hereda otra clase todos sus atributos y métodos. La forma de declarar que una clase hereda de otra es:

Class NombreClase **extends** NombreSuperclase

Ejemplo:

Class Nif extends Dni

Declara una clase Nif que hereda todos los atributos y los métodos de la clase Dni.

Lista de interfaces

Una interfase es un conjunto de constantes y métodos, pero de éstos últimos únicamente el formato, no su implementación. Cuando una clase declara una lista de interfaces, asume que se van a redefinir todos los métodos definidos en la interfase. Esta posibilidad corrige el inconveniente de que, en Java, no pueden declararse clases descendientes de más de una superclase.

class *NombreClase* ***implements*** *Interface1*, *Interface2*, ... , *InterfaceN*

Por ejemplo:

```
Class Nif extends Dni implements OperacionesAritméticas,
OperacionesLógicas
```

En la interface OperacionesAritméticas pueden estar definidos, por ejemplo, los métodos suma(), resta(), etc. Mediante esta declaración, comprometo a la clase Nif (más bien a su programador) a redefinir los métodos suma(), resta(), etc., de lo contrario el compilador mostrará el correspondiente mensaje de error. También deberán redefinirse los métodos de la interfaz OperacionesLógicas.

Modificadores de clase

Los modificadores de clase son palabras reservadas que se anteponen a la declaración de clase. Los modificadores posibles son los siguientes:

- public
- abstract
- final

La sintaxis general es la siguiente:

> modificador **class** NombreClase [**extends** NombreSuperclase]
> [**implements** listaDeInterfaces]

Si no se especifica ningún modificador de clase, la clase será visible en todas las declaradas en el mismo paquete[23]. Si no se especifica ningún paquete, se considera que la clase pertenece a un paquete por defecto al cual pertenecen todas las clases que no declaran explícitamente el paquete al que pertenecen.

Public

Cuando se crean varias clases que se agrupan formando un paquete (*package*), únicamente las clases declaradas public pueden ser accedidas desde otro paquete.

Toda clase public debe ser declarada en un Archivo fuente con el nombre de esa clase pública: NombreClase.java. De esta afirmación se deduce que en un Archivo fuente puede haber más de una clase, pero sólo una con el modificador public.

Abstract

Las clases abstractas no pueden ser instanciadas[24]. Sirven únicamente para declarar subclases que deben redefinir aquellos métodos que han sido declarados abstract**.** Esto no quiere decir que todos los métodos de una clase abstracta deban ser abstractos18, incluso es posible que ninguno de ellos lo sea. Aún en este último caso, la clase será considerada como abstracta y no podrán declararse objetos de esta clase.

Cuando alguno de los métodos de una clase es declarado abstracto, la clase debe ser obligatoriamente abstracta, de lo contrario, el compilador genera un mensaje de error.

23 Ver el apartado 14
24 Instanciar una clase significa declarar un objeto cuyo tipo sea esa clase.

```
abstract class Animal {
      String nombre;
      int patas;
      public Animal(String n, int p) {
            nombre=n;
            patas=p;
      }
      abstract void habla();
      // método abstracto que debe ser redefinido por las subclases
      }
      class Perro extends Animal {
      // La clase perro es una subclase de la clase abstracta Animal
            String raza;
            public Perro(String n, int p, String r) {
                  super(n,p);
                  raza=r;
                  }
            public void habla() {
            // Este método es necesario redefinirlo para poder instanciar
            // objetos de la clase Perro
                  System.out.println("Me llamo "+nombre+": GUAU, GUAU");
                  System.out.println("mi raza es "+raza);
                  }
            }
      class Gallo extends Animal {
      // La clase Gallo es una subclase de la clase abstracta Animal
            public Gallo(String n, int p) {


                  super(n,p);
            }
            public void habla() {
            // Este método es necesario redefinirlo para poder instanciar
            // objetos de la clase Gallo
                  System.out.println("Soy un Gallo, Me llamo "+nombre);
                  System.out.println("Kikirikiiii");
            }
      }
      class Abstracta {
            public static void main(String argumentos[]) {
                  Perro toby = new Perro("Toby",4,"San Bernardo");
                  Gallo kiko = new Gallo("Kiko",2);
                  kiko.habla();
                  System.out.println();
                  toby.habla();
            }
}
```

Salida por pantalla del programa:

```
Soy un Gallo, Me llamo Kiko
Kikirikiiii
Me llamo Toby: GUAU, GUAU
mi raza es San Bernardo
```

El intento de declarar un objeto del tipo Animal, que es abstract, habría generado un mensaje de error por el compilador.

Las clases abstractas se crean para ser superclases de otras clases. En este ejemplo, se ha declarado el método habla() como abstracto porque queremos que todos los animales puedan hablar, pero no sabemos qué es lo que van a decir (qué acciones se van a realizar), por lo que es declarada de tipo abstract**.** Las clases que heredan de Animal deben implementar un método habla() para poder heredar las características de Animal.

Final

Una clase declarada final impide que pueda ser superclase de otras clases.
Dicho de otra forma, ninguna clase puede heredar de una clase final.

Esto es importante cuando se crean clases que acceden a recursos del sistema operativo o realizan operaciones de seguridad en el sistema. Si estas clases no se declaran como final, cualquiera podría redefinirlas y aprovecharse para realizar operaciones sólo permitidas a dichas clases pero con nuevas intenciones, posiblemente oscuras.

A diferencia del modificador abstract, pueden existir en la clase métodos final sin que la clase que los contiene sea final (sólo se protegen algunos métodos de la clase que no pueden ser redefinidos).

Una clase no puede ser a la vez abstract y final ya que no tiene sentido, pero sí que puede ser public abstract o public final.

El cuerpo de la clase

Una vez declarada la clase, se declaran los atributos y los métodos de la misma dentro del cuerpo.

```
Declaración de clase {
Declaración de atributos
Declaración de clases interiores
Declaración de Métodos
}
```

La declaración de clases interiores (también conocidas como clases anidadas) no es imprescindible para programar en Java.
Declaración de atributos

Los atributos sirven, en principio, para almacenar valores de los objetos que se instancian a partir de una clase.

La sintaxis general es la siguiente:

[modificadorDeÁmbito**] [static] [final] [transient] [volatile]** tipo nombreAtributo

Existen dos tipos generales de atributos:

- Atributos de objeto.
- Atributos de clase.

Los atributos de objetos son variables u objetos que almacenan valores distintos para instancias distintas de la clase (para objetos distintos).

Los atributos de clase son variables u objetos que almacenan el mismo valor para todos los objetos instanciados a partir de esa clase.

Dicho de otra forma: mientras que a partir de un atributo de objeto se crean tantas copias de ese atributo como objetos se instancien, a partir de un atributo de clase sólo se crea una copia de ese atributo que será compartido por todos los objetos que se instancien.

Si no se especifica lo contrario, los atributos son de objeto y no de clase. Para declarar un atributo de clase se utiliza la palabra reservada static.

La declaración mínima de los atributos es:

tipo nombreAtributo

Si existen varios atributos del mismo tipo (en la misma clase), se separan sus nombres mediante comas (**,**):

```
class Punto {
      int x, y;
      String nombre;
      …
}
```

Atributos static

Mediante la palabra reservada static se declaran atributos de clase.

```
class Persona {
      static int numPersonas=0; // atributo de clase
      String nombre; // atributo de objeto
      public Persona (String n) {
            nombre = n;
            numPersonas++;
      }
      public void muestra() {
            System.out.print("Soy "+nombre);
            System.out.println(" pero hay "+ (numPersonas-1) +
                                    " personas más.");
```

```
            }
      }
      class Static {
            public static void main(String argumentos[]) {
            Persona p1,p2,p3;
            // se crean tres instancias del atributo nombre
            // sólo se crea una instancia del atributo numPersonas
            p1 = new Persona("Pedro");
            p2 = new Persona("Juan");
            p3 = new Persona("Susana");
            p2.muestra();
            p1.muestra();
      }
}
```

Salida por pantalla:

```
Soy Juan pero hay 2 personas más.
Soy Pedro pero hay 2 personas más.
```

En este caso, numPersonas es un atributo de clase y por lo tanto es compartido por todos los objetos que se crean a partir de la clase Persona. Todos los objetos de esta clase pueden acceder al mismo atributo y manipularlo. El atributo nombre es un atributo de objeto y se crean tantas instancias como objetos se declaren del tipo Persona. Cada variable declarada de tipo Persona tiene un atributo nombre y cada objeto puede manipular su propio atributo de objeto.
En el ejemplo, se crea un atributo numPersonas y tres atributos nombre (tantos como objetos de tipo Persona).

Atributos final

La palabra reservada final calificando a un atributo o variable sirve para declarar constantes, no se permite la modificación de su valor. Si además es static, se puede acceder a dicha constante simplemente anteponiendo el nombre de la clase, sin necesidad de instanciarla creando un objeto de la misma.

El valor de un atributo final debe ser asignado en la declaración del mismo. Cualquier intento de modificar su valor generará el consiguiente error por parte del compilador.

```
class Circulo {
      final double PI=3.14159265;
      int radio;
      Circulo(int r) {
            radio=r;
      }
      public double area() {
            return PI*radio*radio;
      }
}
class Final {
      public static void main(String argumentos[]) {
```

```
      Circulo c = new Circulo(15);
      System.out.println(c.area());
      }
}
```

Podría ser útil en algunos casos declarar una clase con constantes:

```
class Constantes {
      static final double PI=3.14159265;
      static final String NOMBREEMPRESA = "Ficticia S.A.";
      static final int MAXP = 3456;
      static final byte CODIGO = 1;
}
```

Para acceder a estas constantes, no es necesario instanciar la clase Constantes, ya que los atributos se han declarado static**.** Simplemente hay que anteponer el nombre de la clase: Constantes.PI, Constantes.CODIGO, etc. Para utilizarlas.

Atributos transient

Los atributos de un objeto se consideran, por defecto, persistentes. Esto significa que a la hora de, por ejemplo, almacenar objetos en un archivo, los valores de dichos atributos deben también almacenarse.

Aquellos atributos que no forman parte del estado persistente del objeto porque almacenan estados transitorios o puntuales del objeto, se declaran como transient (podemos denominarlos transitorios).

```
class Transient {
      int var1, var2;
      transient int numVecesModificado=0;
      void modifica(int v1, int v2) {
            var1=v1;
            var2=v2;
            numVecesModificado++;
      }
}
```

En este caso, la variable numVecesModificado almacena el número de veces que se modifica en el objeto los valores de los atributos var1 y var2. A la hora de almacenar el objeto en un fichero para su posterior recuperación puede que no interese el número de veces que ha sido modificado. Declarando dicho atributo como transient este valor no se almacenará y será reinicializado al recuperarlo.

Atributos volatile

Si una clase contiene atributos de objeto que son modificados asíncronamente por distintos *threads*[25] que se ejecutan concurrentemente, se pueden utilizar atributos volatile para indicarle a la máquina virtual Java este hecho, y así cargar el atributo desde memoria antes de utilizarlo y volver a almacenarlo en memoria después, para que cada *thread* puede "verlo" en un estado coherente. Esto nos ayudará a mantener la coherencia de las variables que puedan ser utilizadas concurrentemente.

```
class Volatil {
      volatile int contador;
      . . .
}
```

Los atributos volatile son ignorados por la versión 1.0 del compilador del JDK.

Modificadores de ámbito de atributos

Los modificadores de ámbito de atributo especifican la forma en que puede accederse a los mismos desde otras clases. Estos modificadores de ámbito son:

- private.
- public.
- protected.
- El ámbito por defecto.

Atributos private

El modificador de ámbito private es el más restrictivo de todos. Todo atributo private es visible únicamente dentro de la clase en la que se declara. No existe ninguna forma de acceder al mismo si no es a través de algún método (no private) que devuelva o modifique su valor.

Una buena metodología de diseño de clases es declarar los atributos private siempre que sea posible, ya que esto evita que algún objeto pueda modificar su valor si no es a través de alguno de sus métodos diseñados para ello.

Atributos public

El modificador de ámbito public es el menos restrictivo de todos. Un atributo public será visible en cualquier clase que desee acceder a él, simplemente anteponiendo el nombre de la clase.

25 Un thread (o hilo de ejecución) funciona de forma paralela a otros theads del mismo proceso. Los threads se ejecutan asíncronamente cuando pueden ejecutarse "paralelamente", sus instrucciones se entremezclan en el tiempo unas con otras, sin que se pueda predecir el orden de ejecución de las instrucciones de uno de los threads respecto de los otros.

Las aplicaciones bien diseñadas minimizan el uso de los atributos public y maximizan el uso de atributos private. La forma apropiada de acceder y modificar atributos de objetos es a través de métodos que accedan a los mismos, aunque en ocasiones, para acelerar el proceso de programación, se declaran de tipo public y se modifican sus valores desde otras clases.

```
final class Empleado {
      public String nombre;
      public String dirección;
      private int sueldo;
}
```

En este ejemplo existen dos atributos public (nombre y dirección) y uno private (sueldo). Los atributos nombre y dirección podrán ser modificados por cualquier clase, por ejemplo de la siguiente forma:

```
      emple1.nombre=”Pedro López”;
```

Mientras que el atributo sueldo no puede ser modificado directamente por ninguna clase que no sea Empleado. En realidad, para que la clase estuviera bien diseñada, se deberían haber declarado private los tres atributos y declarar métodos para modificar los atributos. De estos métodos, el que modifica el atributo sueldo podría declararse de tipo private para que no pudiera ser utilizado por otra clase distinta de Empleado.

Atributos protected

Los atributos protected pueden ser accedidos por las clases del mismo paquete (*package*) y por las subclases del mismo paquete, pero no pueden ser accedidas por subclases de otro paquete, aunque sí pueden ser accedidas las variables protected heredadas de la primera clase.

El ámbito por defecto de los atributos

Los atributos que no llevan ningún modificador de ámbito pueden ser accedidos desde las clases del mismo paquete, pero no desde otros paquetes.

Resumen de ámbitos de atributos

	Acceso desde:			
Modificador	**misma clase**	**subclases**	**mismo paquete**	**todo el mundo**
private	SI	NO	NO	NO
public	SI	SI	SI	SI
protected	SI	SEGÚN	SI	NO
por defecto	SI	NO	SI	NO

12.4 Métodos y constructores

Sintaxis general de los métodos:

Declaración de método **{**
Cuerpo del método
}

Declaración de método

La declaración mínima sin modificadores de un método es:

TipoDevuelto NombreMétodo **(**ListaParámetros**)**

Donde:

- *TipoDevuelto* es el tipo de dato devuelto por el método (función). Si el método no devuelve ningún valor, en su lugar se indica la palabra reservada void. Por ejemplo:
  Void noDevuelveNada
- *NombreMétodo* es un identificador válido en Java.
- *ListaParámetros* si tiene parámetros, es una sucesión de pares **tipo – valor** separado por comas. Ejemplo: int mayor (int x, int y). Los parámetros pueden ser también objetos. **Los tipos simples de datos se pasan siempre por valor y los objetos y vectores por referencia**.

Cuando se declara una subclase, esa subclase hereda, en principio, todos los atributos y métodos de la superclase (clase padre). Estos métodos pueden ser redefinidos en la clase hija simplemente declarando métodos con los mismos identificadores, parámetros y tipo devuelto que los de la superclase. Si desde uno de estos métodos redefinidos se desea realizar una llamada al método de la superclase, se utiliza el identificador de la superclase y se le pasan los parámetros.

Ejemplo: suponiendo que se ha declarado una clase como heredera de otra (SuperC) en la que existe el método int mayor (int x, int y), se puede redefinir este método simplemente declarando un método con el mismo identificativo y parámetros int mayor (int x , int y). Si desde dentro del método redefinido se desea hacer referencia al método original se podría utilizar: var = SuperC(x,y);

También se pueden declarar métodos para una misma clase con los mismos identificadores pero con parámetros distintos.

```
class Mayor {
// Declara dos métodos con el mismo identificativo
// uno de ellos acepta dos enteros y el otro dos
// enteros largos.
// ambos métodos devuelven el valor mayor de los
// dos enteros que se pasan como parámetros.
      static int mayor(int x, int y) {
```

```
            if (x>y)
                  return x;
            else
                  return y;
      }
      static long mayor(long x, long y) {
            if (x>y)
                  return x;
            else
                  return y;
      }
}
class ConstantesMayor {
      static final int INT = 15;
      static final long LONG = 15;
}
Class SubMayor extends Mayor {
// modifica la clase Mayor de la siguiente forma:
// los métodos devuelven el valor mayor de entre
// los dos parámetros que se le pasan, pero
// siempre, como mínimo devuelve el valor
// de las constantes INT y LONG
      static int mayor(int x, int y) {
      // llama al método mayor de la superclase
            int m = Mayor.mayor(x,y);
            return Mayor.mayor(m,ConstantesMayor.INT);
      }
      static long mayor(long x, long y) {
      // llama al método mayor de la superclase
            long m = Mayor.mayor(x,y);
            return Mayor.mayor(m,ConstantesMayor.LONG);
      }
}
class EjecutaMayor {
      public static void main(String argumentos[]) {
            int int1=12,int2=14;
            long long1=20, long2=10;
            System.out.println("ENTEROS:");
            System.out.println("mayor de 12 y 14 = " +
                               Mayor.mayor(int1,int2));
            System.out.println("mayor de 12 y 14 y 15 = " +
                               SubMayor.mayor(int1,int2));
            System.out.println("ENTEROS LARGOS:");
            System.out.println("mayor de 20 y 10 = " +
                               Mayor.mayor(long1,long2));
            System.out.println("mayor de 20 y 10 y 15 = " +
                               SubMayor.mayor(long1,long2));
            }
}
```

Declaración completa de métodos. Sintaxis general:

> [**ModificadorDeÁmbito**] [**static**] [**abstract**] [**final**] [**native**] [**synchronized**] **TipoDevuelto NombreMétodo (** [**ListaParámetros**] **)** [**throws ListaExcepciones**]

Métodos static

Los métodos static son métodos de clase (no de objeto) y por tanto, no necesita instanciarse la clase (crear un objeto de esa clase) para poder llamar a ese método. Se ha estado utilizando hasta ahora siempre que se declaraba una clase ejecutable, ya que para poder ejecutar el método main() no se declara ningún objeto de esa clase.

Los métodos de clase (static) únicamente pueden acceder a sus atributos de clase (static) y nunca a los atributos de objeto (no static).

```
class EnteroX {
      static int x;
      static int x() {
            return x;
      }
      static void setX(int nuevaX) {
            x = nuevaX;
      }
}
```

Métodos abstract

Los métodos abstract se declaran en las clases abstract. Es decir, si se declara algún método de tipo abstract, entonces, la clase debe declararse obligatoriamente como abstract.

Cuando se declara un método abstract, no se implementa el cuerpo del método, sólo su signatura. Las clases que se declaran como subclases de una clase abstract deben implementar los métodos abstract. Una clase abstract no puede ser instanciada, únicamente sirve para ser utilizada como superclase de otras clases.

Métodos final

Los métodos de una clase que se declaran de tipo final no pueden ser redefinidos por las subclases. Esta opción puede adoptarse por razones de seguridad, para que nuestras clases no puedan ser extendidas por otros.

Modificadores de ámbito de los métodos

Los modificadores de ámbito de los métodos son exactamente iguales que los de los atributos, especifican la forma en que puede accederse a los mismos desde otras clases. Estos modificadores de ámbito son:

- private
- public
- protected

- El ámbito por defecto

	Acceso desde:			
Modificador	**misma clase**	**subclases**	**mismo paquete**	**todo el mundo**
Private	SI	NO	NO	NO
Public	SI	SI	SI	SI
Protected	SI	SEGÚN	SI	NO
por defecto	SI	NO	SI	NO

Lista de excepciones potenciales

Todos los métodos pueden generar excepciones (condiciones de error). Estas excepciones pueden ser "lanzadas" por el método para que sean tratadas por el método que realizó la llamada. Todas las posibles excepciones que pueda generar el método (y que no sean tratadas por el propio método) deben ser declaradas como una lista separada por comas en la declaración del mismo detrás de la declaración del identificador del método y la lista de parámetros:

TipoDevuelto NombreMétodo **(** [ListaParámetros] **)** [**throws** ListaExcepciones]

Ejemplo:

```
class Nif {
      int dni;
      char letra;
      static char tabla[]={'T','R','W','A','G','M','Y','F','P','D',
                           'X', B','N','J','Z','S','Q','V','H','L',
                           'C','K','E'};
      public Nif(int ndni,char nletra) throws NifException{
            if (Character.toUpperCase(nletra)==tabla[ndni%23]) {
                  dni=ndni;
                  letra=Character.toUpperCase(nletra);
            } else throw new LetraNifException();
      }
      …
}
```

En la declaración del método Nif (constructor) indicamos que puede "lanzar" la excepción NifException. Luego, en la cláusula else se procede a lanzar efectivamente la excepción cuando la letra calculada no coincide con la que hemos pasado como parámetro.

Cuerpo del método

Sintaxis general de los métodos:

Declaración de método **{**
Cuerpo del método
}

El cuerpo del método contiene la implementación del mismo y se codifica entre las llaves del método. Dentro del mismo pueden declararse variables locales al mismo.

El único caso en el que no se implementa el cuerpo del método es en la declaración de clases abstractas.

La forma de declarar la salida del método es mediante la palabra reservada return. Si el tipo de dato devuelto del método ha sido declarado void, entonces únicamente se sale del cuerpo del método mediante la instrucción return**;** Si en la declaración de la cabecera del método se ha declarado un tipo de dato devuelto por el mismo, entonces se devolverá el valor correspondiente al método mediante return seguido de una expresión cuyo resultado es del mismo tipo que el declarado.

Constructores

Un constructor es un método especial de las clases que sirve para inicializar los objetos que se instancian como miembros de una clase.

Para declarar un constructor basta con declarar un método con el mismo nombre que la clase. **No se declara el tipo devuelto por el constructor** (ni siquiera void), aunque sí que se pueden utilizar los modificadores de ámbito de los métodos: public, protected, private.

Los constructores tienen el mismo nombre que la clase y todas las clases tienen uno por defecto (que no es necesario declarar), aunque es posible sobrescribirlo e incluso declarar distintos constructores (sobrecarga de métodos) al igual que los demás métodos de una clase.

Para inicializar un objeto de una determinada clase se llama a su constructor después de la palabra reservada new**.**

El constructor, mecanismo aparentemente elaborado de inicialización, proporciona un importante mecanismo para realizar la inicialización. Los constructores permiten *garantizar* la inicialización correcta y la limpieza (el compilador no permitirá que un objeto se cree sin los constructores pertinentes), se logra un control y seguridad completos.

Entonces, Existe en cada clase una función miembro que tiene el mismo nombre que la clase. A esta se le llama: ***constructor de la clase.***
Es común tener varios constructores para una clase; esto se lleva a cabo a través de la sobrecarga de funciones.

12.5 Sobrecarga de métodos

Surge un problema cuando se trata de establecer una correspondencia entre el concepto de matiz del lenguaje humano y un lenguaje de programación. A menudo, la misma palabra expresa varios significados -se ha ***sobrecargado.*** Esto es útil, especialmente cuando incluye diferencias triviales. Se dice "lava la camisa", "lava el coche" y "lava el perro". Sería estúpido tener que decir "IavaCamisas la camisa", "lavacoche el coche" y "lavaperro el perro" simplemente para que el que lo escuche no tenga necesidad de intentar distinguir entre las acciones que se llevan a cabo. La mayoría de los lenguajes humanos son redundantes, por lo que incluso aunque se te olviden unas pocas palabras, se sigue pudiendo entender. No son necesarios identificadores únicos -se puede deducir el significado del contexto.

En Java (y C++) otros factores fuerzan la sobrecarga de los nombres de método: el constructor. Dado que el nombre del constructor está predeterminado por el nombre de la clase, sólo puede haber un nombre de constructor. Pero ¿qué ocurre si se desea crear un objeto de más de una manera? Por ejemplo, suponga que se construye una clase que puede inicializarse a sí misma de manera estándar o leyendo información de un archivo. Se necesitan dos constructores, uno que no tome argumentos (el constructor por defecto, llamado también constructor sin parámetros), y otro que tome como parámetro un **String,** que es el nombre del archivo con el cual inicializar el objeto. Ambos son constructores, por lo que deben tener el mismo nombre -el nombre de la clase. Por consiguiente, la sobrecarga de métodos es esencial para permitir que se use el mismo nombre de métodos con distintos tipos de parámetros. Y aunque la sobrecarga de métodos es una necesidad para los constructores, es bastante conveniente y se puede usar con cualquier método.

He aquí un ejemplo que muestra métodos sobrecargados, tanto constructores como ordinarios:

```
//: c04:Sobrecarga.java
// Muestra de sobrecarga de métodos
// tanto constructores como ordinarios.
import java.util.*;

class Arbol {
       int altura;
       Arbol 0
              visualizar ("Plantando un retoño") ;
              altura = 0;
       }
       Arbol (int i) {
```

```
            visualizar("Creando un nuevo arbol que tiene "
            + i + metros de alto");
            altura = i;
      }
      void info ( ) {
            visualizar("E1 arbol tiene " + altura
                  t " metros de alto");
      }
      void info(String S) {
            visualizar(s + ": El arbol tiene "
                  + altura t " metros de alto");
      }
      static void visualizar (String S) {
            System.out.println (S) ;
      }
public class sobrecarga {
      public static void main (String[] args) {
            for(int i =0; i <5; i++) {
            Arbol t =new Arbol (i) ;
            t.info();
            t. info ("metodo sobrecargado") ;
      }
      // Constructor sobrecargado:
      new Arbol() ;
      }
}///:-
```

Se puede crear un objeto **Arbol,** bien como un retoño, sin argumentos, o como una planta que crece en un criadero, con una altura ya existente. Para dar soporte a esto, hay dos constructores, uno que no toma argumentos (a los constructores sin argumentos se les llama *constructores por defecto*), y uno que toma la altura existente.

Podríamos también querer invocar al método **info**( ) de más de una manera. Por ejemplo, con un parámetro **String** si se tiene un mensaje extra para imprimir, y sin él si no se tiene nada más que decir. Parecería extraño dar dos nombres separados a lo que es obviamente el mismo concepto.

Afortunadamente, la sobrecarga de métodos permite usar el mismo nombre para ambos.

12.6 Utilización de this

Invocando a constructores desde constructores

Cuando se escriben varios constructores para una clase, hay veces en las que uno quisiera invocar a un constructor desde otro para evitar la duplicación de código. Esto se puede lograr utilizando la palabra clave **this.**

Normalmente, cuando se dice **this,** tiene el sentido de "este objeto" o "el objeto actual", y por sí mismo produce la referencia al objeto actual. En un constructor, la palabra clave **this** toma un significado diferente cuando se le da una lista de

parámetros: hace una llamada explícita al constructor que coincida con la lista de parámetros. Por consiguiente, hay una manera directa de llamar a otros constructores:

```
//: c04:Flor. java
// Invocación a constructores con "this".
public class Flor
       int numeropetalos = 0;
       String S = new String("nullW);
       Flor (int petalos) {
              numeroPetalos = petalos;
              System.out.println("Constructor w/ parametro entero solo, Numero
                                  de petalos = " + numeroPetalos) ;
       }
       Flor(String SS) {
              System.out.println("Constructor w/ parametro cadera solo, S=" +
                                  SS);
              S = SS;
       }
       Flor(String S, int petalos) {
              this (petalos) ;
//!this(s); // ¡No se puede invocar dos!
       this.s = S; // Otro uso de "this"
       System.out.println("cadena y entero Parámetros");
       }
Flor () }
       this ("Hola", 47) ;
       System.out.println( "constructor por defecto (sin parametros) ");
       }
       void print (){
//!this(11); // ;No dentro de un no-constructor!
              System.out.println( "Numero de Petalos = " + numeroPetalos + " S = "t
              S);

       public static void main (String[] args) {
              Flor x = new Flor ();
              x.print ();
       }
}
```

El constructor **Flor (String S, int petalos)** muestra que se puede invocar a un constructor utilizando **this,** pero no a dos. Además, la llamada al constructor debe ser la primera cosa que se haga o se obtendrá un mensaje de error del compilador.

El ejemplo también muestra otra manera de ver el uso de **this.** Dado que el nombre del parámetro **s** y el nombre del atributo **S** son el mismo, hay cierta ambigüedad. Se puede resolver diciendo **this.s** para referirse al dato miembro.

En **print (**) se puede ver que el compilador no permite invocar a un constructor desde dentro de otro método que no sea un constructor.

El significado de estático (static)

Teniendo en cuenta la palabra clave this, uno puede comprender completamente qué significa hacer un método **estático.** Significa que no hay un **this** para ese método en particular. No se puede invocar a métodos no **estático** desde dentro de métodos estáticos" aunque al revés sí que es posible), y se puede invocar al método **estático** de la propia clase, sin objetos. De hecho, esto es principalmente el fin de un método **estático.** Es como si se estuviera creando el equivalente a una función global (de C).

La diferencia es que las funciones globales están prohibidas en Java, y poner un método **estático** dentro de una clase permite que ésta acceda a otros métodos **estáticos** y a campos **estáticos.**

Hay quien discute que los métodos **estáticos** no son orientados a objetos, puesto que tienen la semántica de una función global; con un método **estático** no se envía un mensaje a un objeto, puesto que no hay **this.** Esto probablemente es un argumento justo, y si uno acaba usando ***un*** *montón* de métodos estáticos, seguro que tendrá que replantearse su estrategia.

Sin embargo, los métodos **estáticos** son pragmáticos y hay veces en las que son genuinamente necesarios, por lo que el hecho de que sean o no "PO0 pura" se deja para los teóricos. Si duda, incluso Smalltalk tiene un equivalente en sus "métodos de clase".

12.7 Argumentos de La línea de comandos

Al igual que ocurre en C, a un programa escrito en Java también se le pueden pasar parámetros en la línea de comandos. Estos parámetros se especifican en el método main() del programa, donde siempre hay que declarar un vector de Strings.

```
class MiClase {
      …
      public static void main( String argumentos[]) {
      …
```

Recordemos que en Java, los vectores son objetos, con todas las ventajas que ello implica. Así, se puede acceder en tiempo de ejecución al número de elementos del vector (número de argumentos) mediante el atributo público length**.** Si se intenta acceder a un elemento del vector más allá del límite, se genera una excepción.

```
class Comandos {
      public static void main (String argumentos[] ) {
      int i;
      for (i=0 ; i<argumentos.length ; i++)
            System.out.println("Parámetro "+i+": "+
            argumentos[i]);
```

```
        }
}
```

Si se ejecuta el programa de la siguiente forma:

Java Comandos primero segundo tercero

Se produce la siguiente salida por pantalla:

```
Parámetro 0: primero
Parámetro 1: segundo
Parámetro 2: tercero
```

Todos los parámetros son considerados Strings, por lo que si se desea pasar un parámetro de otro tipo, hay que hacer la conversión adecuada.

Los espacios en blanco son considerados como separadores de parámetros. Si se desea pasar como parámetro una cadena de caracteres que incluya espacios en blanco, habrá que hacerlo encerrándola entre comillas dobles.

Si se ejecuta el programa de la siguiente forma:

Java Comandos “Esto es un parámetro” “esto es otro”

Se produce la siguiente salida por pantalla:
Parámetro 0: Esto es un parámetro
Parámetro 1: esto es otro

13. Paquetes

13.1 ¿Qué es un paquete?

Un paquete (*package*) es una librería de clases que se agrupan para evitar problemas de nomenclatura o identificadores repetidos y por razones de organización.

Los paquetes están organizados jerárquicamente y agrupan clases e interfaces que tienen algo que ver conceptualmente unas con otras.

Se obtiene al utilizar la palabra clave **import** para importar una biblioteca completa, como en:

```
import java.uti1. *;
```

Esto trae la biblioteca de utilidades entera, que es parte de la distribución estándar de Java. Dado que, por ejemplo, la clase **ArrayList** se encuentra en **java.uti1** es posible especificar el nombre completo **java.uti1. ArrayList** (lo cual se puede hacer sin la sentencia **import)** o bien se puede simplemente decir **ArrayList** (gracias a la sentencia **import).**

Si se desea incorporar una única clase, es posible nombrarla sin la sentencia **import:**

```
import java.util.ArrayList;
```

Ahora es posible hacer uso de **ArrayList,** aunque no estarán disponibles ninguna de las otras clases de **java-util.**

13.2 Declaración de paquetes

Para declarar un paquete al cual pertenecerán las clases e interfaces declaradas, la primera línea del fichero fuente debe ser esa declaración de paquete:

Sintaxis:

```
package NombrePaquete;
interface Interface1 {
        .
}
class Clase1 {
        .
}
class Clase2 {
        .
}
```

NombrePaquete: puede estar compuesto por varios identificadores separados por puntos. Por ejemplo:

victor.graf
victor.mate

Únicamente las clases e interfaces declaradas como public podrán ser accedidas desde otro paquete.
Pueden existir varios ficheros fuente (.java) con la misma declaración de paquete, de forma que todas las clases e interfaces declaradas en ficheros con el mismo nombre de paquete se incluirán en la misma librería (paquete).

Las clases e interfaces declaradas como miembros del paquete deberán estar en un directorio con el mismo nombre que el del paquete para poder ser utilizadas por otros paquetes, teniendo en cuenta que cada uno de los distintos identificadores separados por puntos representa un directorio diferente. Así, las clases declaradas en el paquete **victor.graf** deberán estar almacenadas en el directorio **victor\graf**.

Además, para hacer uso de estas clases, el directorio en el cual está el subdirectorio **victor** tendrá que estar incluido en la variable CLASSPATH.

13.3 Los especificadores de acceso public y private

Cuando se usa la palabra clave public, significa que la declaración de miembro que continúe inmediatamente a public estará disponible a todo el mundo, y en especial al programador cliente que hace uso de la biblioteca. Supóngase que se define un paquete postre, que contiene la siguiente unidad de compilación:

```
//: c05:postre:Galleta.java
// Crea una biblioteca.
package c05.postre;

public class Galleta {
      public Galleta (){
            System.out.println ("Constructor de Galleta");
      }
void morder (){System.out.println ("morder") ;}
}///:-
```

Galleta.java debe residir en un subdirectorio de nombre postre, en un directorio bajo c05 (que se corresponde con el Capítulo 5 del libro *Thinking in Java*, 2da. Edición, del cual se usan varios ejemplos en este material) que debe estar bajo uno de los directorios de CLASSPATH. No hay que cometer el error de pensar que Java siempre buscará en el directorio actual como uno de los directorios de partida para la búsqueda. Si no se tiene un '.' como una de las rutas del CLASSPATH, Java no buscará ahí.

La palabra clave private significa que nadie puede acceder a ese miembro excepto a través de los métodos de esa clase. Otras clases del mismo paquete no pueden acceder a miembros privados, de forma que es como si se estuviera incluso aislando la clase contra uno mismo. Por otro lado, no es improbable que un paquete esté construido por varias personas que colaboran juntas, de forma que privado permite cambiar libremente ese miembro sin necesidad de preocuparse de si el cambio influirá a otras clases del mismo paquete.

Esto está bien, dado que el acceso por defecto es el que se usa normalmente (y el que se lograría si se olvida añadir algún control de acceso). Por consiguiente, uno generalmente pensaría en lo referente al acceso a los miembros de un programa, que habría que hacer éstos explícitamente **públicos,** y como resultado, puede que inicialmente no se piense en usar la palabra clave **private** a menudo. (Lo cual es distinto en C++.) Sin embargo, resulta que el uso consistente de **private** es muy importante, especialmente cuando está involucrada la ejecución multihilo.

He aquí un ejemplo del uso de **private:**

```
//: cO5 :Helado. java
// Demuestra el uso de la palabra clave "private".
class Vainilla {
      private Vainilla (){}
            static Vainilla prepararvainilla0 {
            return new Vainilla ();
      }
}
```

```
public class Helado {
      public static void main(String[] args) {
      //!Vainilla x =new Vainilla() ;
      Vainilla x =Vainilla.prepararVainilla0;
      }
} ///:-
```

Esto muestra un ejemplo de cómo el modificador **privado** resulta útil: se podría querer controlar cómo se crea un objeto y evitar que alguien pueda acceder directamente a un constructor particular (o a todos ellos). En el ejemplo de arriba, no se puede crear un objeto **Vainilla** a través de su constructor; por el contrario, debe invocarse al método **prepararvainilla** ( ).

13.4 Como hacer uso de los paquetes

Para hacer uso de alguna clase perteneciente a un paquete existen dos fórmulas:

1) Importando la clase mediante la palabra reservada import:

Sintaxis:

import NombrePaquete**.**NombreClase**;**
class Clase1 **{**
.
}
....

Donde ***NombreClase*** es el nombre de la clase que va a utilizarse o un asterisco (*) para importar todas las clases pertenecientes al paquete.

Después de importar un paquete, puede accederse a las clases de tipo public y a sus métodos.

2) Anteponiendo el nombre del paquete a la clase:

Ejemplo:

```
objeto = new victor.graf.Clase1();
…
```

En cualquier caso, siempre que haya ambigüedad porque haya clases con el mismo nombre en distintos paquetes que se importen, habrá que distinguirlas utilizando el nombre completo (paquete + clase).

13.5 Paquetes pertenecientes a Java

En el JDK se suministran varios paquetes. Van a distinguirse los que acompañan a las versiones del JDK del 1.0 en adelante, estando atentos a los cambios realizados en las nuevas versiones. Ver apartado 8.4.

java.lang: Paquete que contiene las clases principales del lenguaje Java. Es el único paquete que se importa (import) automáticamente sin tener que hacerlo explícitamente.

java.io: Contiene clases para manejar *streams* de entrada y salida para leer y escribir datos en ficheros, *sockets*, etc.

java.util: Contiene varias clases para diversas utilidades, tales como: generación de número aleatorios, manipulación de Strings, fechas, propiedades del sistema, etc.

java.net**:** Paquete que contiene clases para la gestión de redes, *sockets*, IP, URLs, etc.

java.awt: (AWT = *Abstract Window Toolkit*) proporciona clases para el menejo de GUIs (*Graphic User Interface*) como ventanas, botones, menús, listas, texto, etiquetas, etc.

java.awt.image: Paquete perteneciente al paquete awt que proporciona clases para la manipulación de imágenes.

java.awt.peer: Es un paquete que conecta el AWT de Java con implementaciones específicas del sistema en el que se ejecuta. Estas clases son utilizadas por Java y el programador normal no debería tener que utilizarlas.

java.applet: Contiene clases para la creación de applets.

sun.tools.debug: Contiene clases para la depuración de programas.

java.lang.reflect: Proporciona una pequeña API que soporta la inspección de clases y objetos en la máquina virtual Java

java.math: Proporciona dos nuevas clases: BigDecimal y BigInteger.

java.rmi: Proporciona acceso a invocación de métodos remotos.
- java.rmi.dgc:
- java.rmi.registry:
- java.rmi.server:

java.security: Proporciona algoritmos de encriptación y seguridad.
- java.security.acl:
- java.security.interfaces:

java.sql: Proporciona acceso a bases de datos mediante SQL (JDBC).

java.text: Proporciona internacionalización de textos.

14. Herencia

14.1 Introducción

La verdadera potencia de la programación orientada a objetos radica en su capacidad para reflejar la abstracción que el cerebro humano realiza automáticamente durante el proceso de aprendizaje y el proceso de análisis de información.

Las personas percibimos la realidad como un conjunto de objetos interrelacionados. Dichas interrelaciones, pueden verse como un conjunto de abstracciones y generalizaciones que se han ido asimilando desde la niñez. Así, los defensores de la programación orientada a objetos afirman que esta técnica se adecua mejor al funcionamiento del cerebro humano, al permitir descomponer un problema de cierta magnitud en un conjunto de problemas menores subordinados del primero.

La capacidad de descomponer un problema o concepto en un conjunto de objetos relacionados entre sí, y cuyo comportamiento es fácilmente identificable, puede ser muy útil para el desarrollo de programas informáticos.

Es por eso que la herencia es una de las principales ventajas de la POO. Esta propiedad permite definir clases descendientes de otras, de forma que la nueva clase (la clase descendiente) hereda de la clase antecesora todos sus ATRIBUTOS y MÉTODOS. La nueva clase puede definir nuevos atributos y métodos o incluso puede redefinir atributos y métodos ya existentes (por ejemplo: cambiar el tipo de un atributo o las operaciones que realiza un determinado método).

La herencia, Es la forma natural de definir objetos en la vida real. La mayoría de la gente diría, por ejemplo, que un chalet es una casa con jardín. Tiene las mismas características y propiedades u operaciones que pueden realizarse sobre una casa y, además, incorpora una nueva característica, el **jardín**. En otras ocasiones, se añadirá funcionalidad (métodos) y no atributos. Por ejemplo: un pato es un ave que nada. Mantiene las mismas características que las aves y únicamente habría que declarar un método sobre la nueva clase (el método **nadar**).

Una de las características más atractivas de Java es la reutilización de código. Pero para ser revolucionario, es necesario poder hacer muchísimo más que copiar código y cambiarlo.

Se reutiliza código creando nuevas clases, pero en vez de crearlas de la nada, se utilizan clases ya existentes que otra persona ya ha construido y depurado.

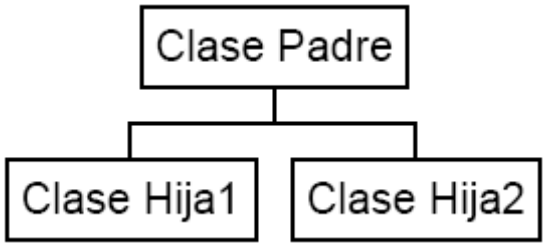
El truco es usar clases sin manchar el código existente. En este capítulo se verán dos formas de lograrlo.

La primera es bastante directa: simplemente se crean objetos de las clase existentes dentro de la nueva clase. A esto se le llama composición, porque la clase nueva está compuesta de objetos de clases existentes. Simplemente se está reutilizando la funcionalidad del código, no su forma.

El segundo enfoque es más sutil. Crea una nueva clase como un tipo de una clase ya existente. Literalmente se toma la forma de la clase existente y se le añade código sin modificar a la clase ya existente. Este acto mágico se denomina herencia, y el compilador hace la mayoría del trabajo.

14.2 Jerarquía

La herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica; una clase *padre* o superclase sobre otras clases hijas o subclases.



Los descendientes de una clase heredan todas las variables y métodos que sus ascendientes hayan especificado como *heredables*, además de crear los suyos propios.

La característica de herencia, nos permite definir nuevas clases derivadas de otra ya existente, que la especializan de alguna manera. Así logramos definir una jerarquía de clases, que se puede mostrar mediante un árbol de herencia.

En todo lenguaje orientado a objetos existe una jerarquía, mediante la que las clases se relacionan en términos de herencia. En Java, el punto más alto de la jerarquía es la clase
Object de la cual derivan todas las demás clases.

14.3 Herencia múltiple

En la orientación a objetos, se consideran dos tipos de herencia, simple y múltiple. En el caso de la primera, una clase sólo puede derivar de una única superclase. Para el segundo tipo, una clase puede descender de varias superclases.

En Java sólo se dispone de herencia simple, para una mayor sencillez del lenguaje, si bien se compensa de cierta manera la inexistencia de herencia múltiple con un concepto denominado interfases.

14.4 Declaración

Para indicar que una clase deriva de otra, heredando sus propiedades (métodos y atributos), se usa el término *extends*, como en el siguiente ejemplo:

```
Public class SubClase extends SuperClase {
// Contenido de la clase_
}
```

Por ejemplo, creamos una clase *MiPuntos3D*, hija de la clase ya mostrada *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 decir que deseamos crear una subclase de la clase que es nombrada a continuación, en nuestro caso MiPunto3D es hija de MiPunto.

14.5 Limitaciones de la Herencia

Todos los campos y métodos de una clase son siempre accesibles para el código de la misma clase.

Para controlar el acceso desde otras clases, y para controlar la herencia por las subclases, los miembros (atributos y métodos) de las clases tienen tres modificadores posibles de control de acceso:

. ***Public***: Los miembros declarados *public* son accesibles en cualquier lugar en que sea accesible la clase, y son heredados por las subclases.
. ***Private***: Los miembros declarados private son accesibles sólo en la propia clase.
.***Protected***: Los miembros declarados protected son accesibles sólo para sus subclases
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 este 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*, solo 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 (sería una indiscreción, ¿no?).

14.6 La clase Object

La clase *Object* es la superclase de todas las clases da 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 funciones básicas comunes a todas las clases:

- *public booleans 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.

14.7 Clases y métodos abstractos

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.

Se puede declarar que ciertos métodos han de ser sobrescritos en las subclases, utilizando el modificador de tipo abstract. A estos métodos también se les llama responsabilidad de subclase. Cualquier subclase de una clase *abstract* debe implementar todos los métodos abstract de la superclase o bien ser declarada también como *abstract.*

Cualquier clase que contenga métodos declarados como abstract también se tiene que declarar como abstract, y no se podrán crear instancias de dicha clase (operador *new*).

Por último se pueden declarar constructores abstract o métodos abstract static.

Veamos un ejemplo de clases abstractas:

```
abstract class claseA {
      abstract void metodoAbstracto();
      void metodoConcreto() {
            System.out.println("En el metodo concreto de claseA");
      }
}
class claseB extends claseA {
      void metodoAbstracto(){
            System.out.println("En el metodo 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.*

```
claseA referenciaA = new claseB();
referenciaA.metodoAbstracto();
referenciaA.metodoConcreto();
```

La salida de la ejecución del programa es:

```
En el metodo abstracto de claseB
En el metodo concreto de claseA
```

15. El IDE Eclipse

15.1 ¿Qué es Eclipse?

En la Web oficial de Eclipse (www.eclipse.org), se define como "An IDE for everything and nothing in particular" (un IDE para todo y para nada en particular). Eclipse es, en el fondo, únicamente un armazón (*workbench*) sobre el que se pueden montar herramientas de desarrollo para cualquier lenguaje, mediante la implementación de los plugins adecuados.

La arquitectura de plugins de Eclipse permite, demás de integrar diversos lenguajes sobre un mismo IDE, introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo como: herramientas UML, editores visuales de interfaces, ayuda en línea para librerías, etc.

15.2 El Proyecto Eclipse

El IDE Eclipse es, únicamente, una de las herramientas que se engloban bajo el denominado *Proyecto Eclipse*. El *Proyecto Eclipse* aúna tanto el desarrollo del IDE Eclipse como de algunos de los plugins mas importantes (como el JDT, plugin para el lenguaje Java, o el CDT, plugin para el lenguaje C/C++).

Este proyecto también alcanza a las librerías que sirven como base para la construcción del IDE Eclipse (pero pueden ser utilizadas de forma completamente independiente), como por ejemplo, la librería de widgets[26] SWT.

26 Símbolo grafico de interfase que permite interacción entre el usuario y la computadora (símbolo, icono, ventana, etc.)

15.3 El Consorcio Eclipse

En su origen, el *Proyecto Eclipse* era un proyecto de desarrollo *OpenSource*, soportado y mantenido en su totalidad por IBM. Bajo la dirección de IBM, se fundó el *Consorcio Eclipse* al cual se unieron algunas empresas importantes como Rational, HP o Borland.

Desde el día 2 de febrero de 2004, el *Consorcio Eclipse* es independiente de IBM y entre otras, está formado por las empresas:

HP, QNX, IBM, Intel, SAP, Fujitsu, Hitachi, Novell, Oracle, Palm, Ericsson y RedHat, además de algunas universidades e institutos tecnológicos.

15.4 La librería SWT

El entorno de desarrollo Eclipse, incluyendo sus plugins, está desarrollado por completo en el lenguaje Java. Un problema habitual en herramientas Java (como NetBeans) es que son demasiado "pesadas". Es decir, necesitan una máquina muy potente para poder ejecutarse de forma satisfactoria. En gran medida, estas necesidades vienen determinadas por el uso del API Swing para su interfaz gráfico.

Swing es una librería de widgets portable a cualquier plataforma que disponga de una máquina virtual Java pero a costa de no aprovechar las capacidades nativas del sistema donde se ejecuta, lo cual supone una ejecución sensiblemente más lenta que la de las aplicaciones nativas.

SWT es una librería de widgets equivalente a Swing en la cual, se aprovechan los widgets nativos del sistema sobre el que se ejecuta.

El hecho de aprovechar los widgets nativos, permite que la ejecución de interfaces de usuario sea mucho más rápida y fluida que si se utilizase Swing y, además, siempre dispone del "Look and Feel" del sistema, sin necesidad de "emularlo".

La contrapartida es que la librería SWT es nativa, es decir, es necesario disponer de una librería SWT específica para cada sistema operativo.

Existen versiones de SWT para los S.O. más habituales, incluyendo Windows, Linux, HP-UX, MacOS, etc.

15.5 Obtener, instalar y ejecutar Eclipse IDE

El IDE Eclipse se puede obtener bajándolo directamente del sitio Web oficial del *Proyecto Eclipse* - www.eclipse.org - o de cualquier "mirror" autorizado. Existen versiones instalables para cualquier plataforma que soporte la librería SWT, descargas que incluyen el código fuente y descargas que incluyen los plugins más habituales. Además, de este mismo sitio, se puede descargar la librería SWT independientemente y su SDK.

Como Eclipse está escrito en Java, en necesario, para su ejecución, que exista un JRE (Java Runtime Environment) instalado previamente en el sistema.

La instalación de Eclipse, es tan sencilla como descomprimir el archivo descargado en el directorio que se estime conveniente. Como ser el directorio archivos de programas en Windows.

15.6 Obtener e instalar Plugins

La descarga básica del entorno Eclipse incluye algunos de los plugins más básicos, pero siempre es deseable obtener alguna funcionalidad extra. Para ello, es necesario instalar nuevos plugins.
En el apartado *Community* del sitio Web oficial de Eclipse se pueden encontrar enlaces a cientos de plugins.

> **Advertencia**
> Es importante escoger cuidadosamente los plugins que se van a instalar pues, la cantidad de plugins instalados, influye en el rendimiento del IDE Eclipse, en especial, en el tiempo de arranque inicial de la aplicación.

Para añadir un nuevo plugin, basta con descomprimir el archivo descargado en el subdirectorio “Plugins” de la carpeta donde está instalado Eclipse. La próxima vez que se ejecute Eclipse, automáticamente, se reconocerán y añadirán los nuevos plugins instalados.

15.7 Ejecutar Eclipse

Las versiones que se pueden descargar del sitio Web de Eclipse vienen con un ejecutable que permite lanzar directamente el IDE Eclipse. Antes de ejecutar Eclipse es importante verificar que se tienen permisos de escritura en el directorio, ya que, la primera vez que se ejecuta, Eclipse tiene que crear las carpetas en las que guardará información sobre workspaces, logs, etc.

15.8 Un vistazo general al IDE

La primera vez que se ejecuta Eclipse se puede ver una pantalla muy similar a la que se muestra en la Figura 1. Antes de enfrentarse a la dura tarea del programador, es interesante echar un primer vistazo al entorno para conocer sus características particulares, la forma que tiene de organizar el trabajo, las herramientas adicionales que ofrece, etc.

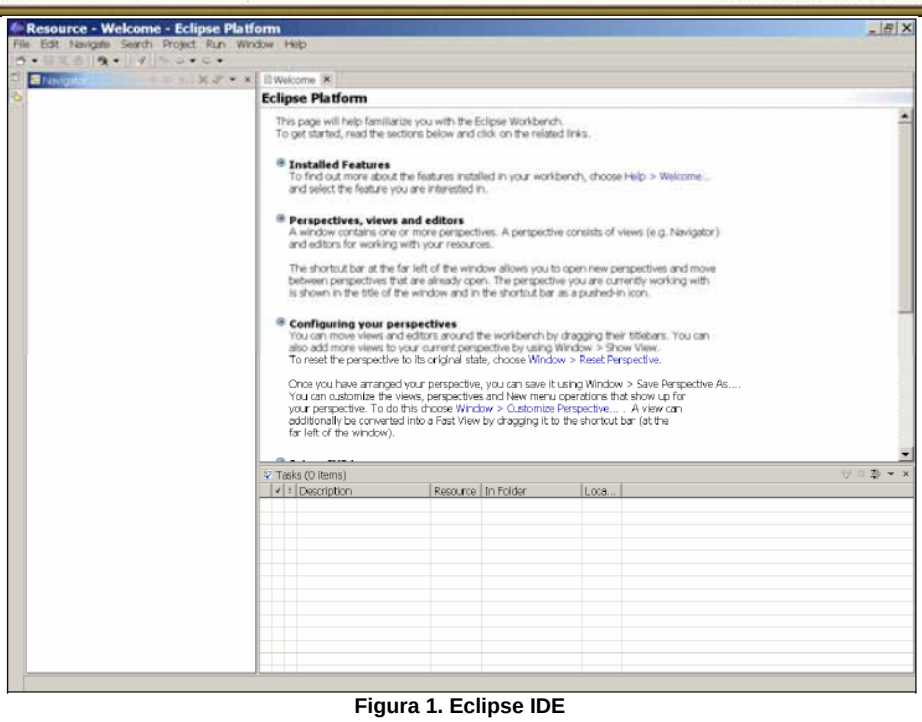


Figura 1. Eclipse IDE

Editores

La ventana principal (la mas grande en la Figura 1), se llama “Editor”. Los Editores son el lugar donde se escribirán los programas. Es posible tener varios Editores abiertos a la vez, apilados uno encima de otro. En la parte superior de la ventana de Editores, se mostrarán solapas que permiten acceder a cada uno de los Editores abiertos (o bien cerrarlos directamente).

Vistas

Además de los Editores, existen un segundo tipo de ventanas “secundarias”, que se llaman Vistas. Las Vistas sirven para cualquier cosa, desde navegar por un árbol de directorios, hasta mostrar el contenido de una consulta SQL. Se puede decir que las Vistas son ventanas auxiliares para mostrar información, requerir datos, etc.

Cada plugin puede definir Editores propios y todas las Vistas que sean necesarias. En la Figura 1, están abiertas dos ventanas de Vistas.
La Vista vertical de la izquierda, mostrará el árbol de directorios de los proyectos (cuando los haya). La Vista horizontal inferior muestra una pequeña “agenda” de tareas pendientes que pueden ser introducidas por el usuario, de forma directa, o por Eclipse, en función de determinados eventos.

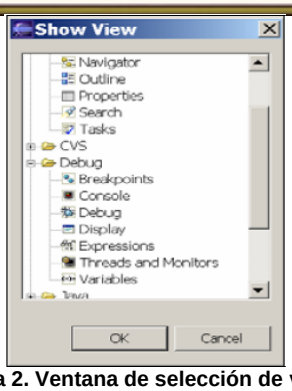


Figura 2. Ventana de selección de vistas

Para seleccionar qué Vistas se deben mostrar, se utiliza la opción "*Show View*" en el menú "*Window*" (ver Figura 2).

Barras de Herramientas

El tercero de los componentes del entorno son las barras de herramientas. Existen dos barras de herramientas: la barra de herramientas principal y la barra de Perspectivas.

La barra de herramientas principal contiene accesos directos a las operaciones mas usuales (guardar, abrir, etc.), botones que permiten lanzar la ejecución de herramientas externas y tareas relacionadas con el Editor activo (ejecutar un programa, depurar, etc.).

La barra de Perspectivas contiene accesos directos a las Perspectivas que se están utilizando en el proyecto. Una Perspectiva es un conjunto de ventanas (Editores y Vistas) relacionadas entre sí. Por ejemplo, existe una Perspectiva Java que facilita el desarrollo de aplicaciones Java y que incluye, además del Editor, Vistas para navegar por las clases, los paquetes, etc.

La Perspectiva que está abierta en la Figura 1, es la llamada "*Resource Perspective*" y su función es navegar por el árbol de directorios de un proyecto y editar los ficheros que contiene utilizando el Editor mas adecuado.
Se puede seleccionar las perspectivas activas – las que se muestran en la Barra de Perspectivas – utilizando la opción "*Open Perspective*" del menú *Window*. Desde este mismo menú también es posible definir Perspectivas personalizadas.
Además de las barras de herramientas principales, cada Vista puede tener su propia barra de herramientas.

Programar con Eclipse

Eclipse es un IDE que no está orientado específicamente hacia ningún lenguaje de programación en concreto. El uso de un determinado lenguaje, estará supeditado a la existencia de un plugin que le de soporte.

Con la versión estándar del entorno Eclipse se distribuye el plugin necesario para programar en lenguaje Java, su nombre es JDT.
Del sitio oficial de Eclipse se puede bajar también el plugin CDT para los lenguajes C/C++. Buscando un poco más en las bases de datos de plugins se pueden encontrar extensiones para lenguajes como Pascal o Python.

Nuevo Proyecto Java

Para poder realizar un programa en Eclipse es necesario crear un proyecto. Un Proyecto agrupa a un conjunto de recursos relacionados entre sí (código fuente, diagramas de clases o documentación).
Se puede crear un nuevo proyecto desde el menú (*File/New/Project*), desde la barra de herramientas principal o desde la vista "*Navigator*" (abriendo el menú pop-up con el botón derecho del ratón y la opción *New*/*Project*).
Cualquiera de estas tres opciones lanzará el *wizard* de creación de proyectos. Para iniciar un proyecto Java se debe seleccionar la opción *Java*/*Java Project*.
Después de indicar un nombre y una ubicación para el nuevo Proyecto se puede, opcionalmente, realizar algunas configuraciones como son:

- Crear un subdirectorio para almacenar el código y un subdirectorio diferente para almacenar las clases compiladas.
- Indicar las dependencias del nuevo proyecto respecto a proyectos anteriores (existentes en el mismo workspace).
- Indicar la ubicación de librerías (.jar) que necesita el proyecto y/o definir variables de entorno.
- Definir el orden de búsqueda de los *classpaths* que se manejan, principalmente para solucionar conflictos en caso de que haya clases con el mismo nombre cualificado.

Es siempre recomendable definir una carpeta (por ejemplo, de nombre *src*) para contener el código y otra (de nombre *bin*, por ejemplo) donde se dejarán los *.class* generados.

En la solapa *Libraries*, se pueden añadir todos los .jar que sean necesarios (con el botón "*Add External Jars…*").

Todas estas configuraciones pueden modificarse en cualquier momento a través del menú contextual de la vista *Navigator*, en la opción *Properties Java Build Path*.
Al crear el proyecto Java, Eclipse, de forma automática, abre la Perspectiva Java, que es la colección de vistas que define el plugin JDT para programar con Java. Esta Perspectiva está compuesta de las vistas: *Package Explorer* (que permite navegar por todos los paquetes –classpaths- a los que puede acceder el proyecto) y *Outline* (que muestra un esquema de la clase cuyo código se está visualizando en el Editor activo).

Además, si la Perspectiva Java está activa, se añaden a la barra de herramientas principal algunos botones extra que permiten acceder con rapidez a las funciones más usuales (ejecutar, depurar, crear clases, etc.)

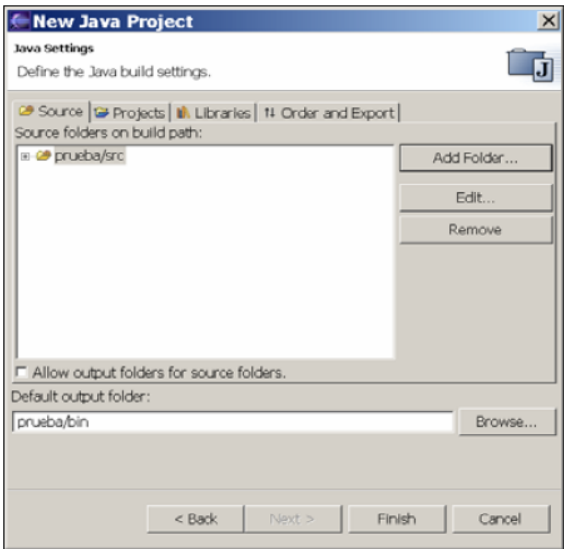


Figura 3. Nuevo proyecto Java.

Nuevas Clases

Ejemplo.

A modo de ejemplo, crearemos un Proyecto Java, cuyo nombre será *Prueba*. Su ubicación será la carpeta por defecto y no requerirá ningún *.jar* extra en el classpath.
Además, se configurará como carpeta para el código, el directorio *prueba/src* y para las clases compiladas, se utilizará la carpeta *prueba/bin*.

El modo más directo de crear una nueva clase (o interfase) es utilizar el "*wizard de creación de clases*" que se puede lanzar, teniendo la Perspectiva Java activa, a través del botón correspondiente (Figura 4) en la barra de herramientas.

Figura 4. Botón creación clases.

El *wizard de creación de clases*, se compone de un único formulario en el que se indicarán las características de la nueva clase (o interfase) que se quiere crear: nombre, superclase, interfaces que implementa, etc.

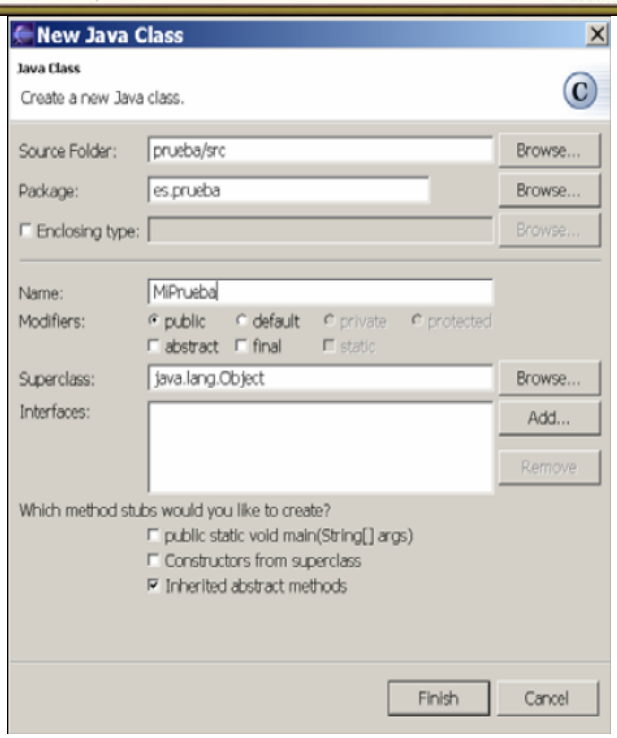


Figura 5. Nueva clase.

En ocasiones, especialmente cuando se trata de proyectos grandes, con muchos paquetes de nombres largos, el wizard de creación de clases puede ser una herramienta lenta y pesada.

Existen otros medios para crear clases. El más sencillo es, en la vista *Navigator*, crear un nuevo fichero java. Para ello, basta con seleccionar la carpeta donde se va a guardar

Ejemplo
Aprovechando el proyecto Prueba que hemos creado en el apartado anterior, podemos ahora crear una nueva clase, utilizando el wizard. Esta clase se llamará *MiPrueba* y estará pertenecerá al paquete *es.prueba*.

La nueva clase y, en el menú pop-up seleccionar la opción *New/File*. Es importante que el fichero que se cree tenga la extensión .java.
En este apartado se ha hablado de cómo crear nuevas clases. La creación de nuevos interfaces es prácticamente igual a la de las clases, en todos los sentidos.

Programar con Eclipse

Cuando se crea una nueva clase se puede ver, en la ventana Editor, que algunas palabras están coloreadas de forma diferente. Este marcado de palabras es debido a que los Editores Java que implementa el plugin JDT, incluyen capacidad para realizar *syntax highlighting* (o reconocimiento sintáctico de palabras reservadas del lenguaje).

De esta forma, las palabras reservadas del lenguaje aparecerán escritas en negrita y en color Burdeos, los comentarios en verde y los comentarios de documentación (javadoc) en azul.

Corrector de Errores

Aparte de identificar las palabras reservadas del lenguaje, JDT puede detectar, y marcar sobre el código de un programa, los lugares donde se pueden producir errores de compilación. Esta característica funciona de forma muy parecida a los correctores ortográficos que tienen los procesadores de textos (ver Figura 5).
Cuando Eclipse detecta un error de compilación, se marcará la sentencia errónea, subrayándola con una línea ondulada roja (o amarilla, si en lugar de un error se trata de un *warning*).

Si el programador posiciona el puntero del ratón sobre la instrucción que produjo el fallo, se mostrará una breve explicación de por qué dicha instrucción se ha marcado como errónea.

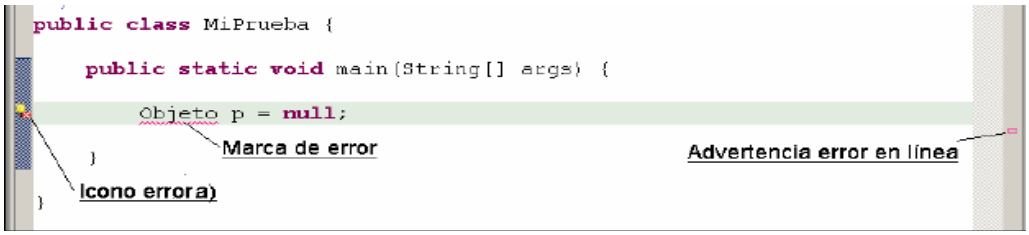


Figura 6. Corrector integrado.

Cada línea de código que contenga un error (o un warning), será también marcada con un icono de error (o warning) que aparecerá en la barra de desplazamiento izquierda del editor (identificado, en la Figura 6 como “a)”). Si el programador pulsa una vez sobre dicha marca (ojo, una sola vez, no doble click) se desplegará un menú pop-up mediante el cuál, Eclipse mostrará posibles soluciones para los errores detectados. Si se acepta alguna de las sugerencias del menú pop-up, Eclipse se encargará de llevar a cabo dicha solución, de forma completamente automática.

Code Completion

Un entorno de desarrollo no puede considerarse como útil, en la práctica, si no dispone de la capacidad de completar automáticamente las sentencias que está escribiendo el programador. El mecanismo de “*code completion*” en Eclipse es muy similar al que implementan otros IDEs: cuando se deja de escribir durante un determinado intervalo de tiempo se muestran, si los hay, todos los términos (palabras reservadas, nombres de funciones, de variables, de campos, etc.) que empiecen por los caracteres escritos.

Si se escriben determinados caracteres (como el punto, por ejemplo) se puede provocar la ejecución del mecanismo de “*code completion*” sin necesidad de esperar a que pase el tiempo establecido.

Otra característica relacionada con el *code completion* es la asistencia a la escritura en llamadas a funciones. Automáticamente, cuando se van a escribir los parámetros que se pasan a un método, se muestra una caja de texto indicando los tipos que éstos pueden tener.

Templates

De forma similar a muchos otros entornos de desarrollo, Eclipse permite definir y utilizar *templates*. Los *templates* son plantillas de código (generalmente porciones de código de uso habitual y muy repetitivo) que se escriben automáticamente.

Los *templates* están compuestos de dos partes: un bloque de código (o de comentario), de uso frecuente, que se escribe automáticamente y una cadena que provoca la escritura del *template*. Las cadenas que disparan templates serán reconocidas por el sistema de *code completion*, con la diferencia de que, en lugar de terminar la escritura de la cadena, ésta será sustituida por el *template* que tiene asociado.

Ejemplo.

En el ejemplo inferior, se pretende escribir un bucle *for* que itere un *array*. Se trata de un tipo de construcción muy común, por ello, es firme candidata a ser asociada a un *template*.

```
public static void main(String[] args) {

    for

}
```

Si en el código anterior se pulsa la combinación ctrl.+ espacio, y se selecciona la opción "*for – iterate over array*", el resultado que se obtiene es el siguiente:

```
public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {

    }

}
```

El plugin JDT, por defecto, define una buena cantidad de *templates*, tanto para construcciones de código, como para la escritura de javadoc pero, de todas formas, es posible definir nuevos *templates* personalizados (o modificar los existentes).

A la ventana de configuración de templates se accede a través del menú principal en la opción *Window/Preferences/Java/Editor/Templates*.

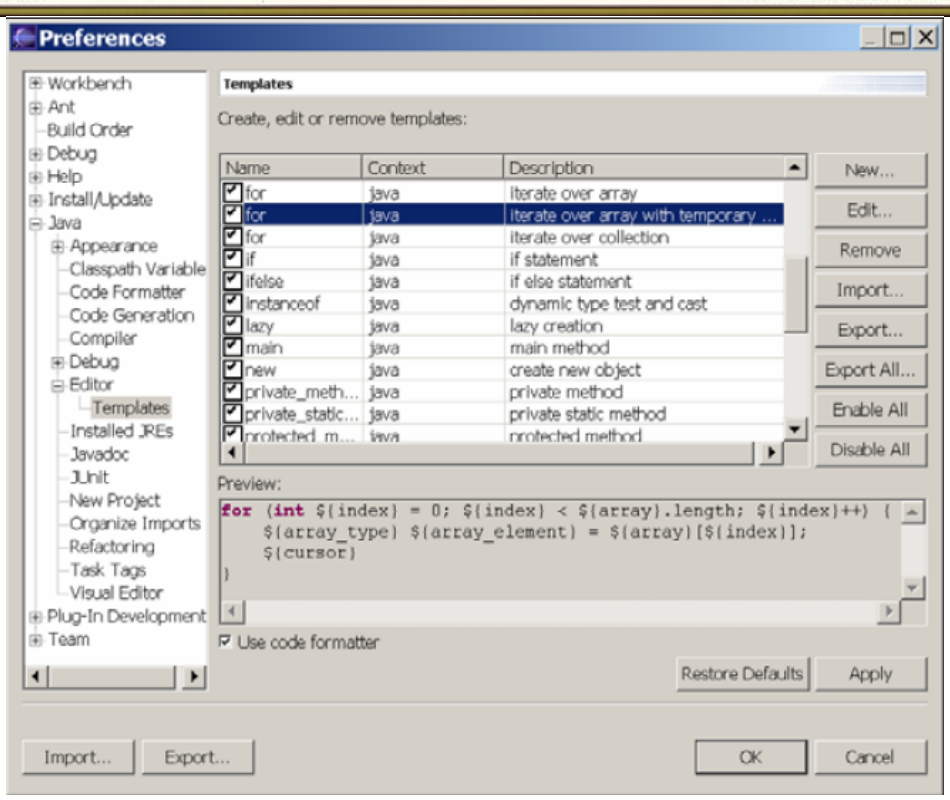


Figura 7. Configuración de templates.

Code Formatting

Todos los programadores sabemos lo importante que es disponer de un código ordenado, legible y fácil de entender. Al menos en teoría. En la práctica no suele encontrarse nunca tiempo, ni ganas, para conservar el aspecto del código.

Eclipse incorpora una herramienta para realizar automáticamente el formateo del código de acuerdo a unos criterios preestablecidos.

Para formatear el código que muestra el Editor activo, basta con seleccionar la entrada *Source/Format* del menú contextual que aparece al pulsar con el botón derecho del ratón sobre el propio Editor.

Ejemplo.

Veamos cómo funciona el *formateador de código* para el ejemplo que se muestra a continuación:

```
public static void main(String[] args) {

    System.out.println("Esta es una línea muuuy larga y además con varias instrucciones"); int i =0; i++;


}
```

Si se ejecuta el formateo automático, el resultado obtenido es el siguiente:

```
public static void main(String[] args) {

    System.out.println(
        "Esta es una línea muuuy larga y además con varias instrucciones");
    int i = 0;
    i++;

}
```

El *formateador de código* puede configurarse en la ventana de *Preferences* del entorno (opción *Window/Preferences* del menú principal), en el apartado *Java/Code Formatter*. En esta ventana, entre otras cosas, se puede indicar la longitud de las líneas de texto, la identación a aplicar, el formato para las asignaciones, etc.…

Manipulación del Código

La capacidad de formato automáticamente a los programas, es sólo una de las posibilidades de manejo de la estructura del código que soporta Eclipse.

Otras posibilidades, englobadas bajo la entrada *Source* del menú contextual del Editor son:

- *Comment y Uncomment*. Estas dos opciones permiten seleccionar un trozo de código y comentarlo (o descomentarlo) de una vez. Los comentarios que se establecen de esta forma, son comentario de "tipo línea" (//…) por lo tanto, no se ven afectados en caso de que existan previamente bloques de comentarios en el código seleccionado.
- *Add Javadoc Comment*. Escribe un bloque de comentarios javadoc para el elemento seleccionado. Por ejemplo, si se coloca el cursor sobre un método y se ejecuta la operación *Source/Add Javadoc Comment*, se creará, sobre ese método, el esqueleto predefinido para la documentación javadoc, que contendrá etiquetas para cada uno de los parámetros (@param), para el resultado (@return), para las excepciones (@throws), etc…
- *Add import*. Escribe las sentencias **import** para la clase sobre la que esté posicionado el cursor (o sobre la más próxima si no está sobre ninguna).
- *Organize Imports*. Agrupa las sentencias import en función de la ubicación de las clases (o paquetes) referenciados, en la jerarquía global de paquetes del proyecto.
- Opciones de generación automática de "esqueletos" de código. Estas opciones permiten generar, automáticamente, el código necesario para definir métodos get y set (*Source/Generate Setter and Setter…*) para los atributos de la clase, extender constructores y otros métodos definidos en una superclase o en un interfaz, etc.

Todas estas opciones de manipulación de código pueden configurarse en las entradas del menú principal *Window/Preferences/Java/Organize/Imports* y *Window/ Preferences/Java/Code Generation*.

Refactoring

En el punto anterior, se comentaban algunas de las facilidades que ofrece Eclipse para crear y manipular bloques de código de una forma fácil y cómoda, evitando el tedio de tener que realizar todo el trabajo a mano.
Todas las operaciones de manejo de código explicadas trabajan, únicamente, con código escrito sobre un mismo fichero (o perteneciente a una misma clase). Si las modificaciones que se quieren realizar deben involucrar a varias clases, escritas en varios ficheros diferentes, todos ellos pertenecientes al mismo proyecto, entonces se pueden utilizar las herramientas de *Refactorización*.
Las herramientas de *Refactoring* son especialmente útiles cuando se trata de realizar modificaciones, o actualizaciones, en el código, que afectan a varios elementos del diseño.

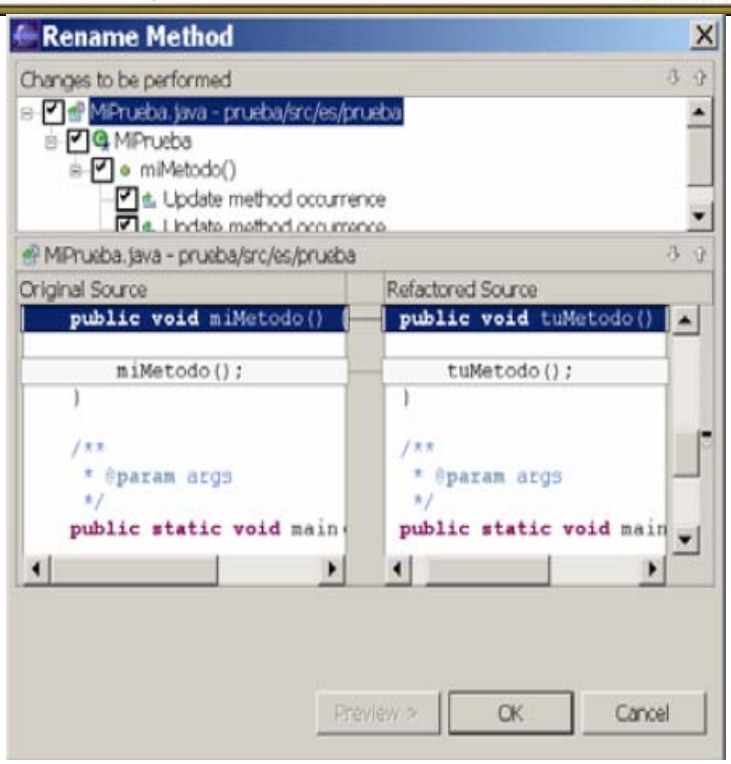
En Eclipse, se puede acceder a las operaciones de *Refactoring* a través de la opción *Refactor* en el menú principal o en el menú pop-up del Editor.

Ejemplo.
En este ejemplo, se utilizarán las operaciones de *Refactoring* para cambiar el nombre de un método. Esta modificación no solamente afecta a la clase que lo implementa. También afectará a todas las clases que realicen llamadas a dicho método, a las subclases que lo redefinan, etc.

```
public void miMetodo() {

    miMetodo();
}
```

Para realizar el cambio de nombre, habrá que seleccionar el nombre del método a modificar y lanzar la operación *Refactor*/*Rename…* del menú contextual. Aparece el diálogo que solicita un nuevo nombre y pide confirmación para actualizar también las referencias que se hagan, en el proyecto, al método que se modifica.



Si se pulsa el botón "*Preview >*", se muestra una comparación del antes y el después de cada porción de código que se va a modificar.
Una vez aceptados los cambios, el resultado final es:

```
public void tuMetodo() {

    tuMetodo();
}
```

La herramienta de *Refactoring* que incluye Eclipse permite realizar muchas otras refactorizaciones (aparte de los cambios de nombre). Entre otras están: cambiar los parámetros de un método, mover un método a una subclase o superclase, extraer un interfaz, convertir una variable local en un atributo, etc.

Compilar

Una de las características más curiosas del IDE Eclipse es el modo en que se compilan los proyectos. No existe en Eclipse ningún botón que permita compilar individualmente un fichero concreto. La compilación es una tarea que se lanza automáticamente al guardar los cambios realizados en el código. Por esta razón es prácticamente innecesario controlar manualmente la compilación de los proyectos.
En caso de necesidad, existe una opción en la entrada *Project* del menú principal, llamada "*Rebuild Project*" que permite lanzar todo el proceso de compilación completo (también existe la entrada "*Rebuild All*" para re-compilar todos los proyectos abiertos).
Cuando se compila un proyecto completo, Eclipse utiliza una secuencia de instrucciones de construcción (build) predefinidas en función de la configuración del proyecto (carpetas de código, carpeta destino, JDK a utilizar, classpaths definidos, etc.)

Sin embargo, en muchas ocasiones, el proceso de construcción de un proyecto incluye muchas otras operaciones además de la mera compilación del código (crear una base de datos, inicializar determinados ficheros, crear archivos de despliegue, etc.)

Es posible configurar el compilador de Eclipse para que realice cualquier proceso de construcción del proyecto de forma automática. Para ello, se utilizan scripts Ant [http://ant.apache.org].
Ant es una evolución de los clásicos Makefiles, que utiliza scripts escritos sobre XML. No es el objetivo de este documento explicar la herramienta Ant, pero sí cómo integrarla en el entorno Eclipse.

Si se quiere utilizar una herramienta de compilación (en este caso Ant) diferente del constructor (Builder) definido por defecto por la herramienta Eclipse, debe ejecutarse la opción "*Project/Properties/External Tool Builders*". En esta ventana, se pueden añadir, modificar o eliminar los diferentes mecanismos de compilación y construcción disponibles para el proyecto.

Para añadir un nuevo script Ant, habrá que pulsar el botón "*New…*" y seleccionar la opción "*Ant Build*" que, nos llevará directamente al wizard de configuración de scripts Ant.

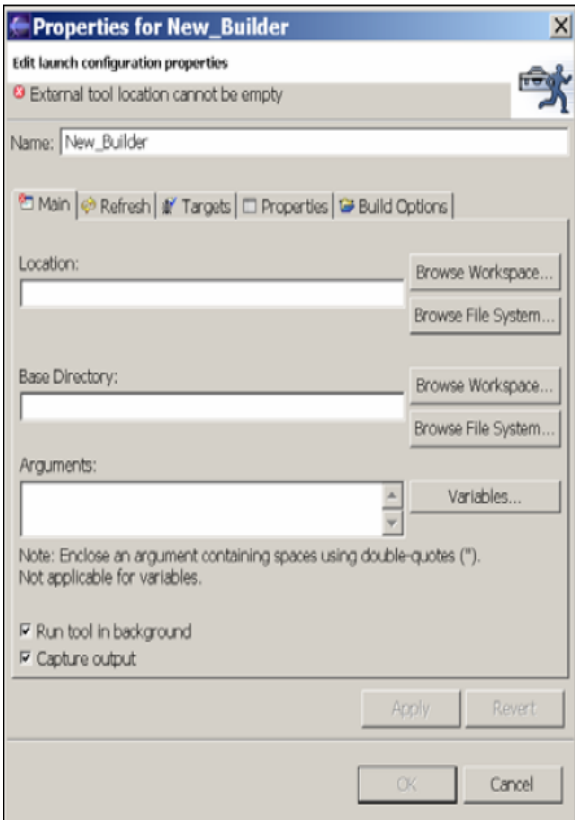


Figura 8. Wizard Ant.

Como se puede ver en la Figura 8, el wizard Ant es una herramienta sencilla. En la solapa principal (la que se muestra en la Figura 8) es necesario indicar la localización del script Ant que se quiere utilizar para compilar el proyecto, así como el directorio base y los argumentos. El resto de las solapas sirven para definir el modo en que Eclipse va a gestionar las llamadas al script.

Ejecutar

Una vez compilado correctamente, ejecutar el proyecto es la parte más sencilla (si el proyecto está correctamente programado claro). Prácticamente todas las opciones de ejecución se pueden manejar desde el botón *Run* de la barra de herramientas principal (ver Figura 9).

Figura 9. Botón ejecución.

El botón *Run* puede utilizarse de dos formas: bien pinchando el propio botón, en este caso, se repetirá la última ejecución realizada, o bien pinchado sobre la flecha a su lado lo cual permitirá ver el menú de ejecución.

El menú de ejecución, a su vez tiene dos partes. La entrada "*Run As*" permite ejecutar directamente la clase que se está mostrando en la ventana del Editor activo, utilizando la configuración de ejecución por defecto.
La entrada "*Run…*", permitirá definir nuevas configuraciones de ejecución.

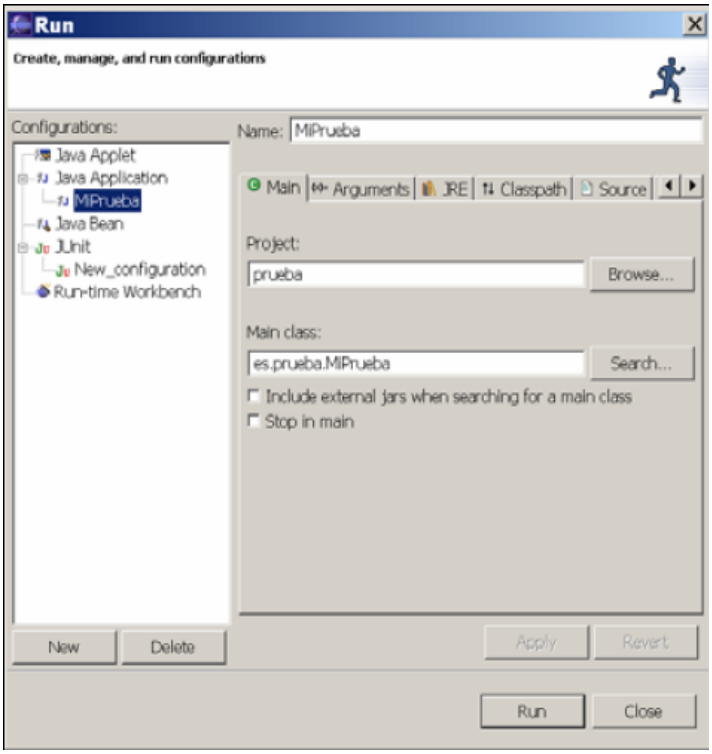


Figura 10. Ventana de configuraciones de ejecución.

Una configuración de ejecución es un conjunto de parámetros que se tendrán en cuenta a la hora de lanzar una ejecución de un programa. Algunos de estos parámetros pueden ser: un classpath determinado, la versión del JRE que se utilizará o los propios parámetros que se pasarán a la clase que se va a ejecutar.
Los parámetros que se pueden definir, y sus valores por defecto, vendrán determinados por el tipo de programa que se va a ejecutar. No tendrá los mismos parámetros una aplicación Java que un applet, por ejemplo.

Depurar Aplicaciones

La principal diferencia entre un simple editor y un buen entorno de desarrollo es que éste integre, o no, una buena herramienta visual para depurar los programas escritos.
Eclipse incluye un depurador potente, sencillo y muy cómodo de utilizar.

Figura 11. Botón depurador.

Lanzar el depurador es una tarea exactamente igual que ejecutar un programa, solo que en lugar de utilizar el botón de ejecución, se utiliza el botón de depuración (ver Figura 11). Estos dos botones, y los menús que despliegan, tienen un comportamiento exactamente idéntico (salvo por el hecho de que el botón de depuración provoca la ejecución paso a paso de los programas).

Cuando el depurador entra en acción, de forma automática, se abre la *Perspectiva Depuración* (Figura 12), en la que se muestra toda la información relativa al programa que se está depurando.

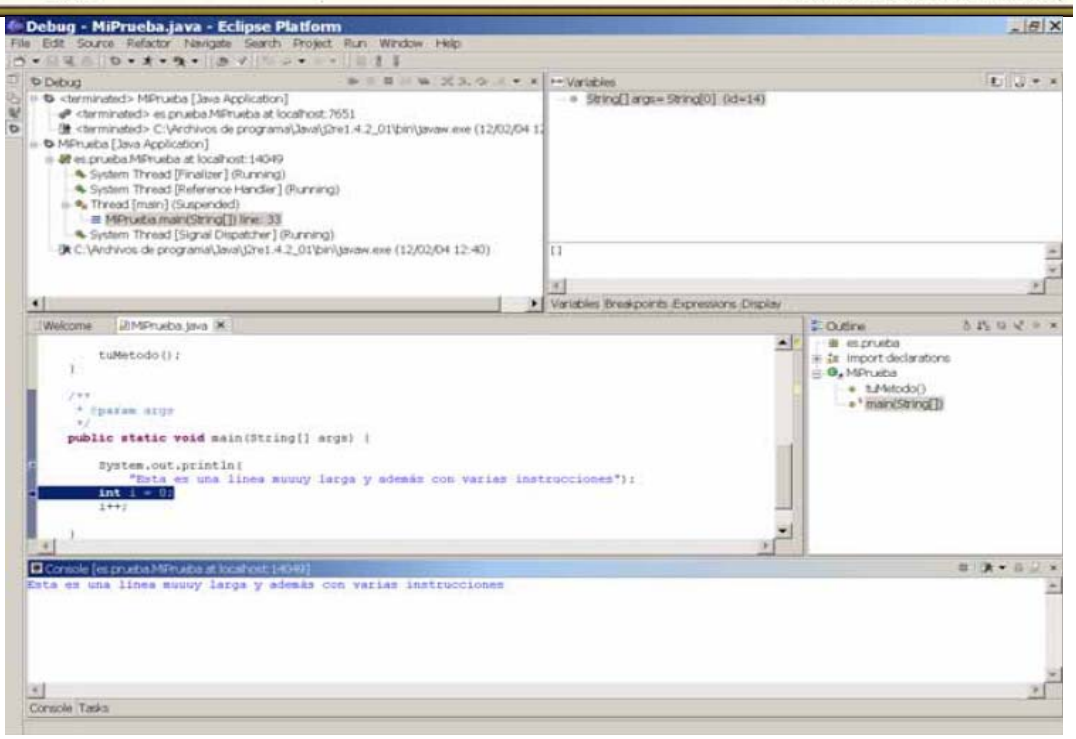


Figura 12. Perspectiva Debug.

Vista Editor

La ventana más importante, situada en el centro de la pantalla, sigue siendo el Editor. Sobre el Editor se irá marcando la traza de ejecución del programa con una pequeña flecha azul, situada sobre el margen izquierdo. La línea sobre la que esté dicha flecha, será la próxima en ejecutarse.
Cuando la Perspectiva Debug está activa, el menú contextual del Editor cambia para mostrar opciones de depuración, por ejemplo, ejecutar el programa hasta la línea que tiene el cursor, inspeccionar una variable (o una expresión seleccionada), etc.

Vistas de Inspección

En la parte superior izquierda, se puede ver la ventana *Debug*. En esta ventana es donde se controla la ejecución del programa que se está depurando ya que contiene la barra de botones de ejecución. En esta barra están los clásicos botones para detener la depuración, ejecutar hasta el final, ejecutar paso a paso, etc.

Además, esta ventana también muestra información a cerca de los hilos (threads) activos y de los procesos de depuración realizados con anterioridad.
La vista de inspección (a la derecha de la vista Debug), permite ver los valores de los diferentes elementos (variables, breakpoints, expresiones…) que intervienen en el programa, en un instante de ejecución determinado.

Las dos vistas más interesantes son la vista de inspección de variables, que muestra los valores que toman todas las variables (atributos, campos, etc.) cuyo

ámbito alcanza a la línea que se está ejecutando en un momento dado y la lista de inspección de breakpoints.
Establecer un breakpoint es tan sencillo como hacer doble clic en el margen izquierdo del Editor del código, a la altura de la línea sobre la que se quiere detener la ejecución.
El breakpoint creado quedará identificado por un punto azul (ver Figura 12) sobre la línea.

La vista de inspección de breakpoints permite, además de ver todos los breakpoints definidos, configurar sus propiedades. A través del menú pop-up de la vista se puede activar o desactivar un breakpoint, eliminarlo, configurarlo para que detenga la ejecución cuando se pase por él un determinado número de veces o cuando lo haga un hilo en concreto, etc.

Atención
Muchas veces, por prisa o por descuido, se lanzan nuevos procesos de depuración sin detener los anteriores (sobre todo, cuando se entra en una vorágine de cambios con el objetivo de solucionar un bug que se resiste más de lo esperado). Si los procesos abiertos se apilan demasiado, se puede agotar la memoria. Para evitarlo, de vez en cuando, se puede echar una ojeada a la perspectiva Debug y comprobar que todos los hilos tengan el estado [terminated], y si no lo tienen, finalizarlos (con el botón que detiene la depuración y el hilo previamente seleccionado en la vista Debug).

Vista Consola
Por último, en la parte inferior de la Perspectiva Debug, se muestra la consola. La consola es la vista sobre la cual se redirecciona tanto la entrada como la salida estándar, del programa que se está depurando (o ejecutando).

Documentación
En este apartado se hablará de todos los aspectos relativos a la documentación en el entorno Eclipse, tanto de la incorporación de archivos de documentación para ser utilizados durante la programación, como de la generación de la documentación de la aplicación.

Configurar el acceso a JavaDocs
El primer valor que se debe configurar es la ubicación de la documentación de las librerías Java estándar. Esta configuración se define para todo el entorno y será accesible para cualquier proyecto con el que se trabaje.
La configuración de la documentación estándar se realiza en la ventana de configuración de los posibles JRE que puede utilizar Eclipse, a la cual se accede desde la opción "*Window/Preferences/Java/Installed JREs*" del menú principal.

En esta ventana, se puede ver una lista de los JREs disponibles, para configurar la documentación de alguno de ellos, basta con seleccionarlo y utilizar el botón

"*Edit…*". En el wizard que aparece (Figura 13) habrá una caja de texto donde se puede introducir la dirección de la carpeta de documentación correspondiente.

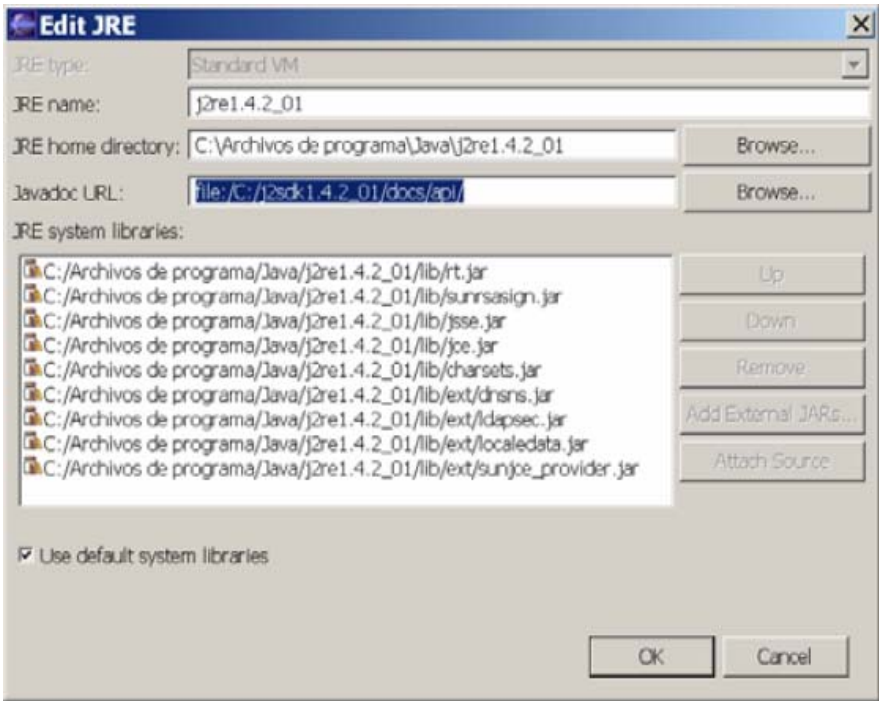


Figura 13. Configuración de un JRE.

Además de las librerías estándar de Java, es frecuente que en los proyectos se utilicen otras muchas librerías, que tengan su propia documentación. Eclipse permite integrar en su sistema de ayuda cualquier JavaDoc relacionado con las librerías que se utilicen en un proyecto.

La configuración de la documentación de las librerías que utiliza el proyecto se realiza desde la ventana de configuración de las propias librerías *en Project/Properties/Java Build Path/Libraries*.

En esta ventana, cada librería (.jar) en el classpath del proyecto, aparece como una entrada que tiene dos propiedades (se muestran pulsando el signo "+" al lado de su nombre): la ubicación de su documentación y la ubicación de su código.
Para poder utilizar la documentación de una librería, basta con escribir su ubicación en el lugar correspondiente.

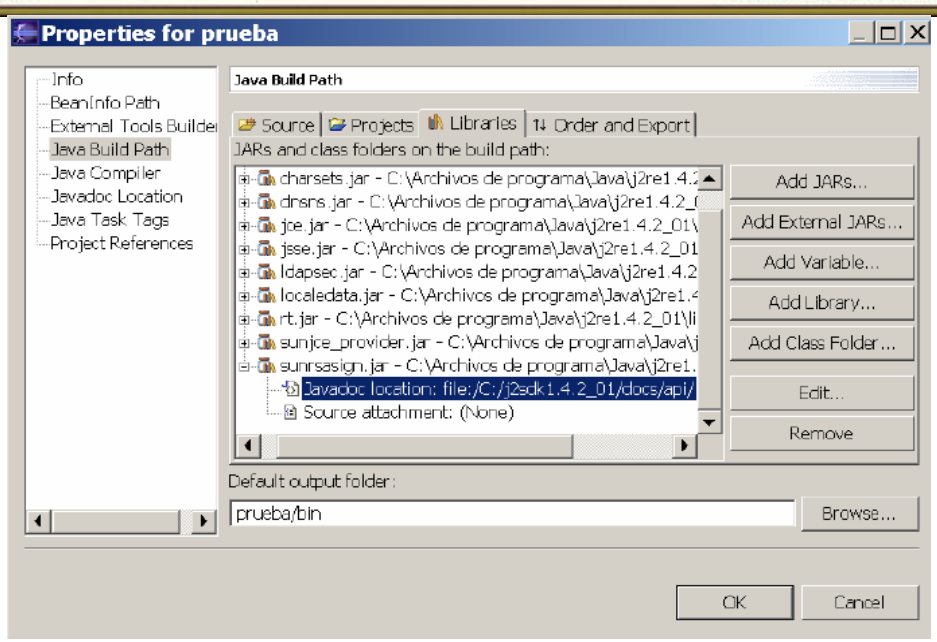


Figura 14. Configuración de la documentación de una librería.

En la Figura 14 se muestra el lugar, en la ventana de Propiedades del proyecto, donde se puede configurar al documentación de una librería concreta.

Por último, también es interesante poder acceder y consultar la propia documentación JavaDoc del proyecto que se está implementando. La configuración de este tipo de documentación también se realiza desde la ventana de propiedades del proyecto (*Project/Properties*), en el apartado "*Javadoc Location*" (Figura 15).

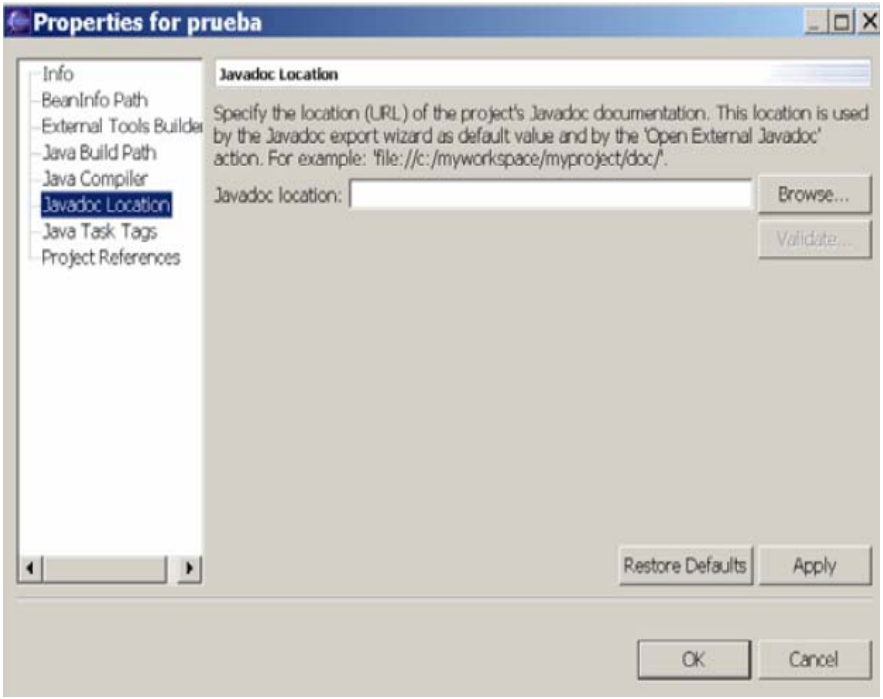


Figura 15. Configuración del Javadoc del proyecto.

Utilizar la documentación

Una vez que todas las posibles fuentes de documentación del proyecto han sido configuradas acceder a ellas es lo más sencillo, basta con seleccionar, en el Editor, el elemento que se quiere consultar y pulsar F1. La ayuda se desplegará, en formato HTML, en el navegador integrado de Eclipse (Figura 16).

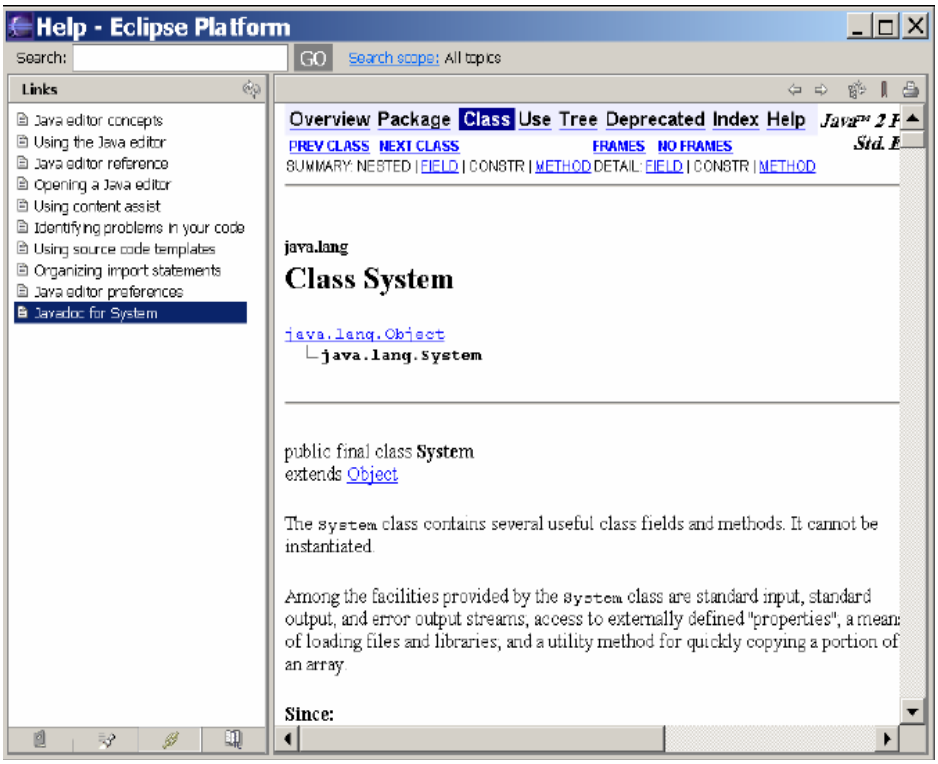


Figura 16. Ayuda en línea de Eclipse.

Generar JavaDoc del proyecto

Además de poder consultar las diferentes fuentes de documentación javadoc que maneja el proyecto, Eclipse permite, de una forma muy sencilla generar, automáticamente, la documentación del propio proyecto.

Antes de poder crear los ficheros de documentación, es necesario configurar la herramienta JavaDoc que Eclipse debe utilizar. Para ello basta con escribir la ubicación del ejecutable javadoc en la opción "*Window/Preferences/Java/Javadoc*" accesible desde el menú principal (Figura 17).

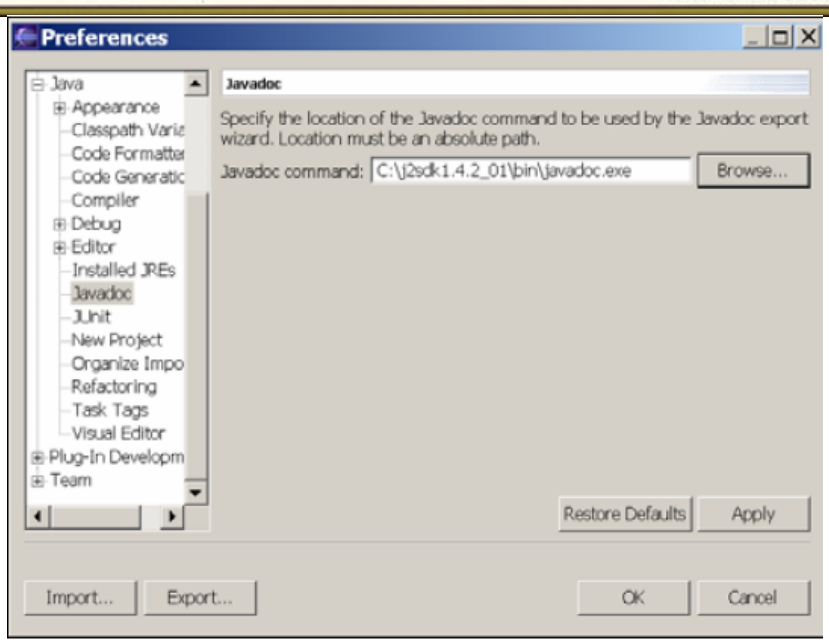


Figura 17. Configuración del ejecutable javadoc.

Para generar la documentación del proyecto, en Eclipse, se utiliza el wizard de "*exportación de javadoc*". Se puede acceder a este wizard seleccionado, en la Vista *Resources*, el proyecto y, a través de la entrada "*Export…*" del menú pop-up, escogiendo la opción "*Javadoc*".

El wizard está compuesto de tres ventanas cuya utilización es bien sencilla. En la primera de ellas se indicarán las clases y paquetes para los cuales se va a generar documentación, el nivel de protección de los miembros cuya documentación se escribirá y el Doclet que se va a utilizar (Figura 18).

Figura 18. Exportar Javadoc (I)

En la segunda de las pantallas del wizard se configura el título de la documentación, las opciones de generación, el las referencias (enlaces) que se deben generar y la hoja de estilos a aplicar (Figura 19).

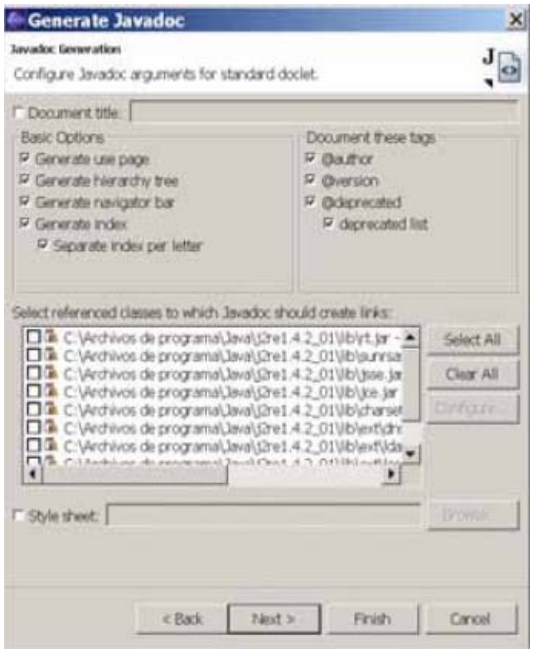


Figura 19. Exportar Javadoc (II)

La tercera, y última, pantalla se destina a la configuración de la página web que se utilizará como portada y a la adición de cualquier otro parámetro que se le quiera pasar al Doclet que generará la documentación (Figura 20).

Figura 20. Exportar Javadoc (III)

Pruebas

Eclipse facilita la tarea de crear y ejecutar pruebas unitarias utilizando el framework JUnit [www.junit.org].
Para poder utilizar el framework JUnit, es necesario colocar la librería *junit.jar* en el classpath del proyecto.

El plugin JDT incluye un wizard para la creación de casos de prueba (JUnit Test Cases) muy similar al propio wizard de creación de clases, explicado en este mismo documento.

Crear un nuevo TestCase, como decía, es muy similar a crear una nueva clase. De igual forma, se puede utilizar el botón de creación de clases (en la barra de herramientas principal, con la Perspectiva Java activa). En el menú desplegable que se muestra, en lugar de seleccionar Class o Interface, como se había explicado, se debe seleccionar la opción TestCase, lo cual mostrará el wizard JUnit.
Este wizard se compone de dos pantallas. La primera de ellas (Figura 21) sirve para realizar una configuración general de la clase de prueba (TestCase), indicando, entre otras cosas: su nombre, el paquete al que pertenecerá, su superclase, los métodos que debe sobrescribir (por ejemplo, SetUp( ), el main(String[ ]) , etc.) o la clase sobre la cuál se va a realizar las pruebas.

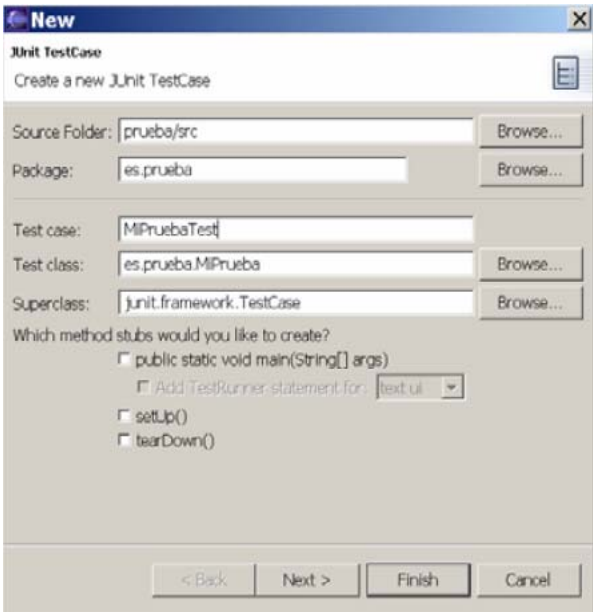


Figura 21. Wizard JUnit (I)

En la segunda pantalla (Figura 22) del wizard JUnit se seleccionarán los métodos para los cuáles se deben generar casos de prueba.

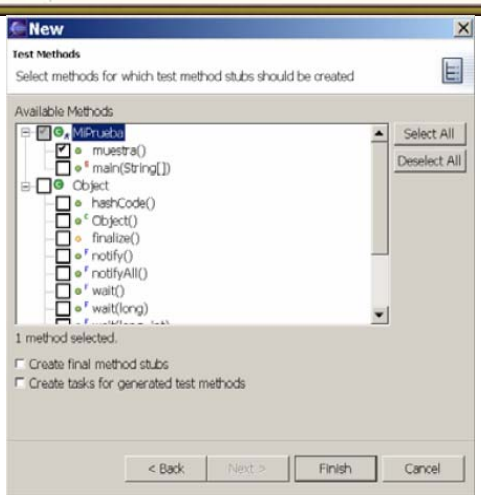


Figura 22. Wizard JUnit (II)

Ejemplo.

A modo de ejemplo, se generará la clase de prueba (TestCase) correspondiente a la clase MiPrueba ( ) que se ha ido definiendo en los ejemplos anteriores.

El código de esta clase será el siguiente:

```
public class MiPrueba {

    public void muestra() {

        System.out.println("Mostrando");
    }

    /**
     * @param args
     */
    public static void main(String[] args) {

        System.out.println(
            "Esta es una linea muuuy larga y además con varias instrucciones");
        int i = 0;
        i++;

    }

}
```

Para crear el caso de prueba, se utilizará el wizard JUnit tal cual se acaba de explicar. En las Figuras 21 y 22, se pueden ver los datos de configuración del caso de prueba de este ejemplo.

El código del Caso de Prueba generado es el siguiente:

```
public class MiPruebaTest extends TestCase {

    /**
     * Constructor for MiPruebaTest.
     * @param arg0
     */
    public MiPruebaTest(String arg0) {
        super(arg0);
    }

    public void testMuestra() {
    }

}
```

Trabajo en equipo con Eclipse.

El IDE Eclipse integra un cliente para el sistema de gestión de versiones CVS. Tanto el acceso a repositorios compartidos de código, como la navegación por los mismos, en Eclipse, se realizan a través de la Perspectiva CVS (Figura 23), a la cual se puede acceder mediante el menú principal (*Window/Open Perspective/Other/CVS*).

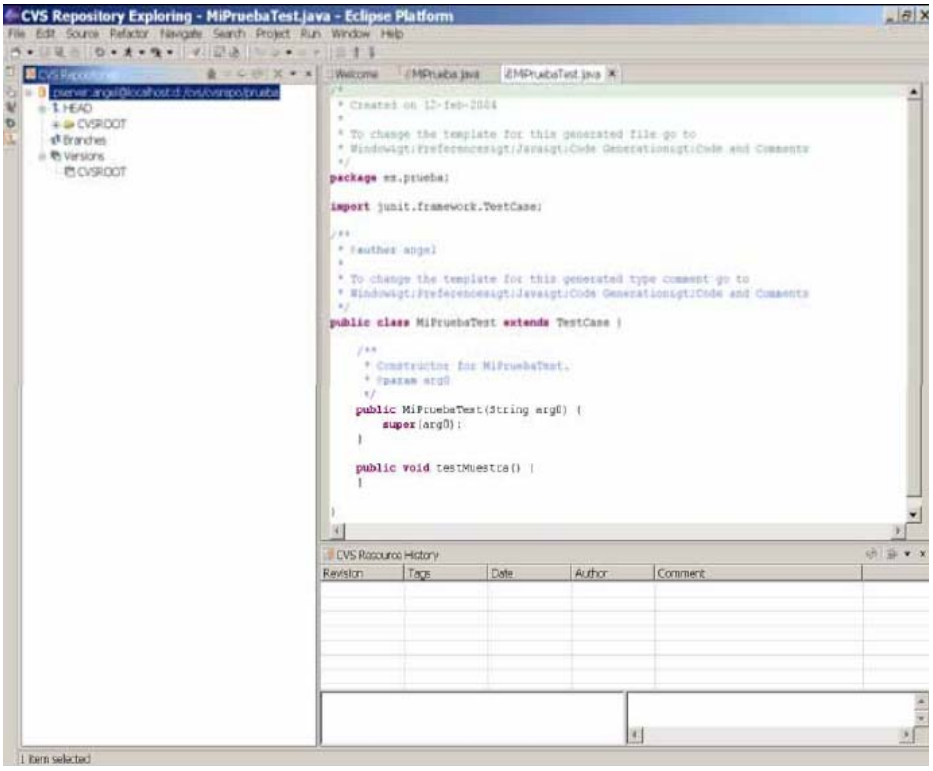


Figura 23. Perspectiva CVS.

Al igual que en las perspectivas que se han tratado en este documento, en la Perspectiva CVS, la ventana principal es el Editor. De forma análoga al resto, el Editor, en la Perspectiva CVS permite ver el código de los archivos almacenados en el repositorio.

La vista "*CVS Repositories*" (a la izquierda en la Figura 23) es la ventana fundamental para el acceso a repositorios de código. Sobre esta vista se podrán establecer (y romper) conexiones con los diferentes repositorios, además de servir como navegador para cada uno de ellos.

Para establecer (o eliminar) conexiones con repositorios, se utiliza el menú contextual (pop-up) de la vista. En este menú, se puede acceder a opciones que permitirán configurar conexiones, descartar conexiones activas, modificar sus parámetros, etc.
La tercera vista (en la parte inferior derecha) de la Perspectiva CVS (vista "*CVS Resurce History*") muestra el registro histórico de cambios aplicados sobre el archivo que se está mostrando en el Editor.

Crear conexión CVS

Para conectar con un repositorio CVS se utiliza el wizard de *localización de repositorios*. Como se comentaba en el párrafo anterior, este wizard es accesible desde el menú contextual de la vista *CVS Repositories*, en la entrada"*New/Repository Location…*".

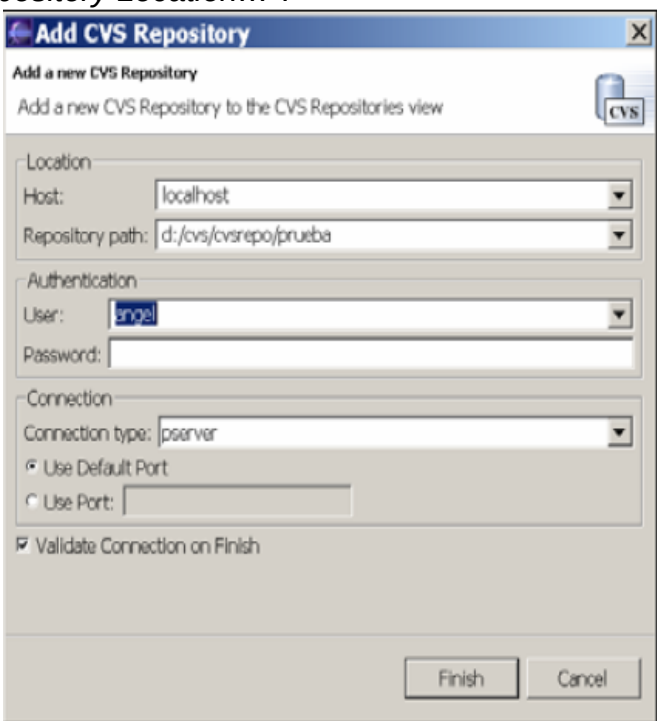


Figura 24. Conexión con Repositorio CVS.

Establecer una conexión CVS es tan sencillo como rellenar los campos del wizard, proporcionando la información necesaria. Se puede ver en la Figura 24 que la información necesaria es: localización del repositorio, autentificación del usuario en dicho repositorio y método de conexión.

Compartir Proyecto

Antes de empezar a trabajar con un repositorio compartido, es necesario asociar el proyecto CVS almacenado, con el proyecto Eclipse con el que se va a trabajar. Existen dos vías para realizar esta asociación:

a) El proyecto Eclipse no existe en el repositorio y se quiere introducir en el mismo. Para ello es necesario, en cualquier vista (por ejemplo, en la vista *Navigator*) seleccionar el proyecto y, a través del menú pop-up, escoger la opción "*Team/Share Project*…". Se mostrará, entonces, una ventana donde se debe especificar el repositorio en el que se va a almacenar el proyecto, o bien, indicar que se debe crear uno nuevo.
Si se opta por esta última opción, se pasará a la ventana de creación de conexiones con repositorios CVS (Figura 24).

b) Se quiere extraer un proyecto del repositorio CVS. En este caso, la compartición del proyecto se realiza desde la vista "*CVS Repositories*" de la Perspectiva CVS. Se seleccionará la carpeta correspondiente al proyecto con el que se quiere trabajar y, a través del menú pop-up, se escogerá la opción "*Check Out As Project*", si se quiere extraer como un

proyecto genérico, o la opción "*Check Out As…*", si lo que se pretende es trabajar con un tipo de proyecto concreto (como un proyecto Java por ejemplo).

Ejecutar comandos CVS

Todas las órdenes que puede recibir el programa de línea de comandos cvs son accesibles desde el cliente implementado por Eclipse. Estas órdenes se agrupan en el submenú "*Team*" del menú contextual (pop-up) de cualquier vista que permita navegar por los ficheros que contiene el proyecto (vistas "*Navigator"* y "*Package Explorer"*, por ejemplo).

Estas órdenes son las conocidas: commit, update, branch, merge, tag as version …
Su función es la misma que tienen sus equivalentes en el programa cvs. Por ejemplo, la orden commit, actualiza el contenido del repositorio guardando los cambios. La orden update, realizará la tarea contraria, actualizando el proyecto local con los datos del repositorio.

Control Cambios
Eclipse mantiene, de forma local, un registro de los cambios realizados en cualquier archivo del proyecto. Es posible comprobar los cambios que se han ido realizando y, en caso de error, volver a versiones anteriores.
El acceso a la pantalla de control de versiones local (Figura 25) se realiza a través del menú contextual (pop up) en cualquier vista que permita ver los ficheros del proyecto (*Navigator* o *Package Explorer*) en la entrada "*Replace UIT/ Local History…*", teniendo seleccionado el fichero cuyo historial se quiere comparar.

Esta última opción permite ver los cambios realizados y, en caso de necesidad, volver a una situación anterior. Si la intención es únicamente comprobar el código en versiones anteriores, y no hacer ningún cambio, se puede utilizar la orden "*Compare With*" del menú pop up.

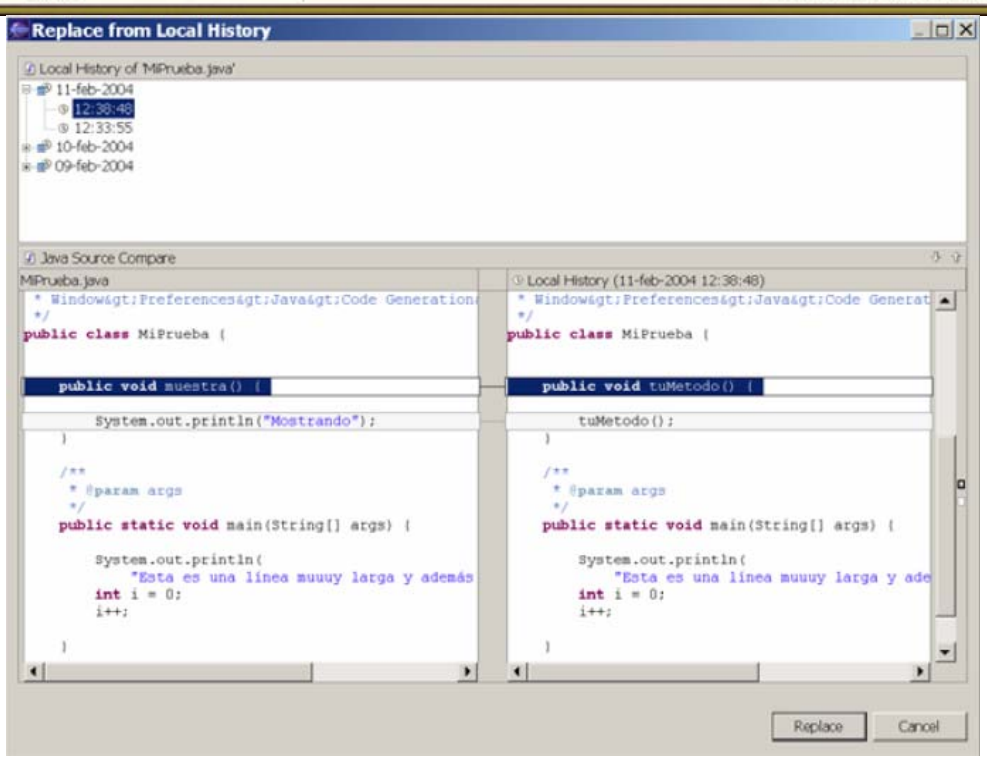


Figura 25. Historial local de cambios.

El interfaz de comparación y recuperación de versiones (Figura 25), puede utilizarse también sobre los ficheros del proyecto que están contenidos en el repositorio CVS, es decir, que no sólo es útil para el control de cambios en las versiones locales del proyecto. Para lanzar la ventana de control de cambios (Figura 25) para un fichero del repositorio CVS, desde la vista "*CVS Repositories*", con el fichero a comparar seleccionado, se debe llamar a la opción "*Compare With…*" del menú contextual.

15.9 Otras herramientas de interés.

En la distribución del IDE Eclipse, además de las herramientas de programación otras utilidades estándar para realizar tareas habituales tales como gestionar tareas, realizar búsquedas sobre el código fuente, simplificar la navegación, etc.

Gestor de Tareas.

El entorno Eclipse incluye una vista, de nombre *Tasks*, que sirve para gestionar las tareas pendientes en un proyecto.
La vista *Tasks* (Figura 26), actúa también como B.O.E (Boletín Oficial de Eclipse), ya que es el medio a través del cual, Eclipse notifica cualquier error, advertencia, etc. que detecte a la hora de compilar, generar código o de realizar cualquier tarea de forma automática.

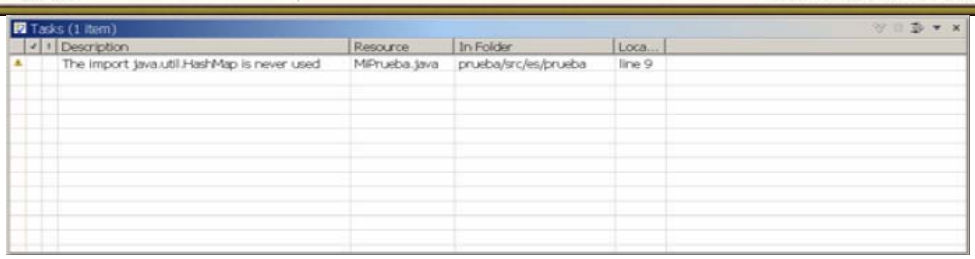


Figura 26

Cuando Eclipse compile una clase, si detecta algún error de compilación, o algún warning, este evento se notificará, en la vista *Tasks*, como una tarea más. Lo mismo ocurre cuando se escribe, de forma automática, esqueletos de código o documentación que deben ser rellenados por el programador.
Además de las tareas introducidas automáticamente por Eclipse, es posible que el programador introduzca las suyas propias. Existen dos formas de introducir tareas:

a) Mediante el menú contextual de la vista *Tasks*, seleccionando la opción "*New Task*".
b) Utilizando etiquetas (*Tags*) que Eclipse reconocerá como entradas para el gestor de tareas (Figura 27). Las *Tags* son etiquetas, palabras, que cuando se escriben dentro de un comentario, Eclipse las asocia directamente con una tarea. Por defecto se define la etiqueta *TODO* (To Do, "Por hacer" en inglés), pero es posible definir etiquetas personalizadas en la ventana de configuración del proyecto (*Project/Properties*), en la opción "Java *Task Tags*".

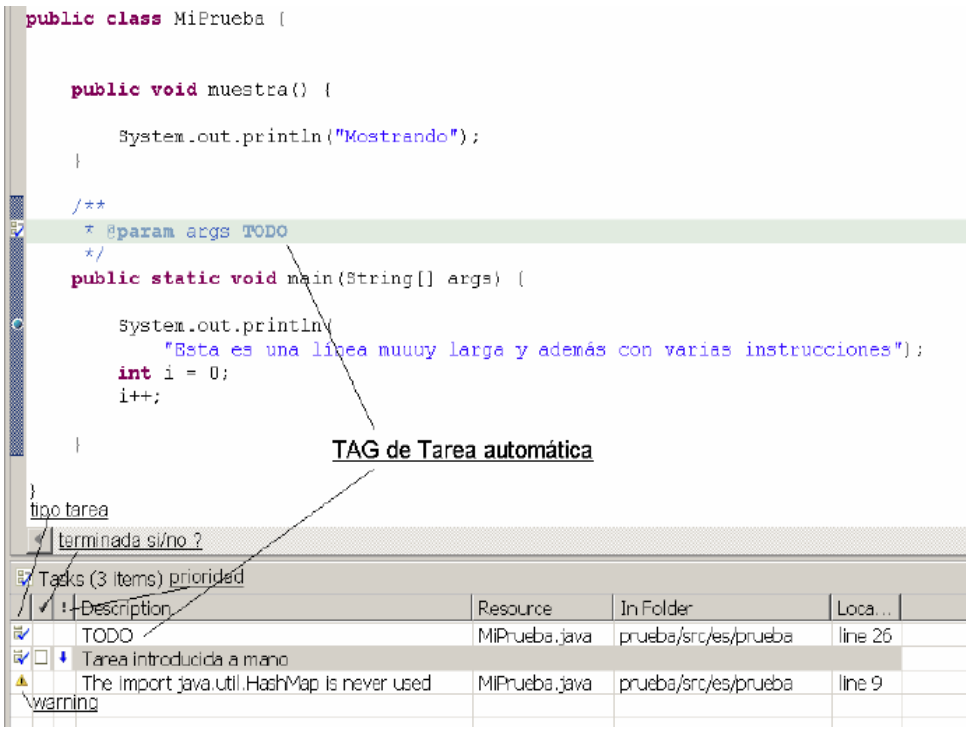


Figura 27

En el menú contextual (pop up) de la vista *Tasks*, además de introducir nuevas tareas, es posible purgar las tareas completadas, establecer filtros para que solamente se muestren las apropiadas, etc.

Búsqueda semántica.

Prácticamente todos (por no decir todos) los editores y entornos de desarrollo disponen de un mecanismo de búsqueda más o menos completo. La diferencia del motor de búsquedas que implementa Eclipse, respecto al de otros entornos, es que éste no se limita únicamente a buscar coincidencias sintácticas en los archivos del proyecto (como hacen la mayoría) sino que dispone de opciones de búsqueda semántica.
Las opciones de búsqueda semántica son las siguientes:

- Buscar los lugares del código donde se hace referencia a un tipo, o clase, determinados.
- Buscar la definición de una clase concreta.
- Buscar las clases que implementan un interfaz definido.
- Buscar los lugares en los cuales se accede para lectura, o escritura, a una variable, o atributo, determinados.

Todas estas opciones están accesibles en el menú principal, en la entrada *Search*.

Algunos Plugins interesantes.

Todas las opciones que se han comentado hasta ahora, en este documento, están implementadas en la versión estándar del IDE Eclipse.
Eclipse permite extender esta funcionalidad básica gracias a su arquitectura de plugins.

Instalar un plugin es muy sencillo, basta con descomprimir el archivo del plugin (generalmente un .zip) en la carpeta *Eclipse/Plugins*. Una vez "instalado" el plugin de esta manera, la siguiente vez que se ejecute Eclipse, éste se encargará de realizar la carga de todos los nuevos plugins añadidos.
Es posible saber, qué plugins están activos en un momento determinado., para ello, es necesario lanzar la ayuda de Eclipse (desde el menú principal, seleccionado la entrada "*Help/About Eclipse Platform*"). Desde la ventana que se despliega, se puede ver la lista de plugins (Figura 28) pulsando el botón "*Plug-in Details*".

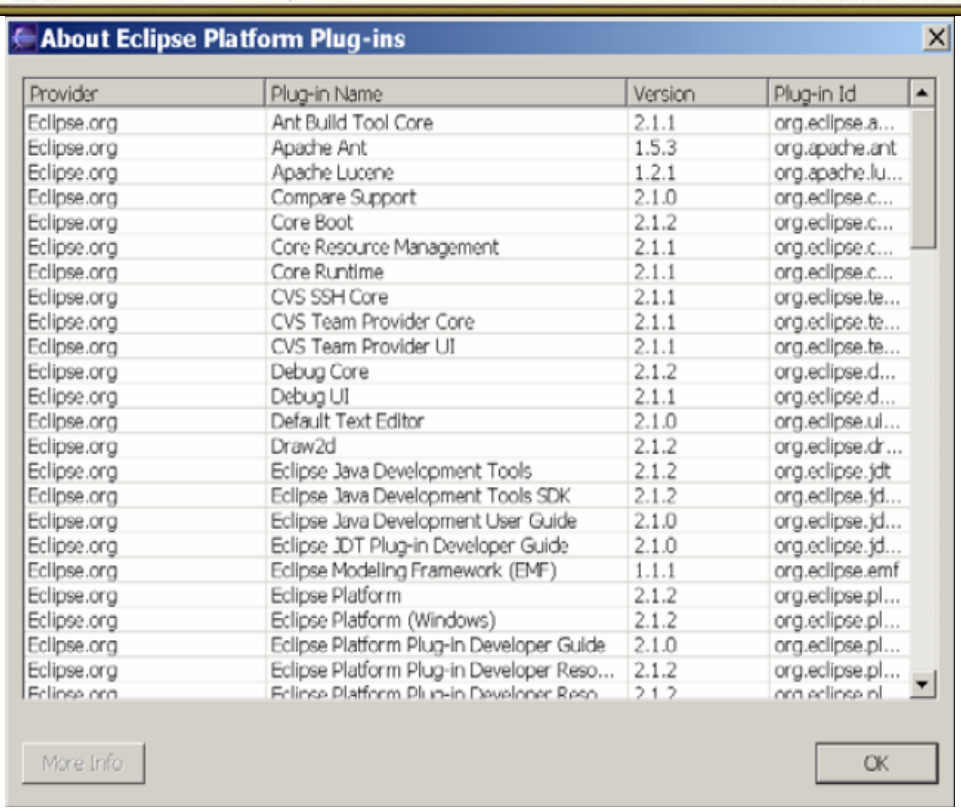


Figura 28

Plugins

Para terminar, se comentan algunos de los plugins más populares de la plataforma Eclipse:

CDT

CDT es el equivalente, para los lenguajes C y C++, al plugin JDT. Prácticamente todo lo escrito, en este documento, referido JDT es aplicable a CDT. Las diferencias más importantes están en la gestión de la documentación Javadoc (que es específica de la plataforma Java) y en el uso de librerías JUnit (también específicas de Java). Otra diferencia importante es que CDT no puede compilar automáticamente el código, es necesario indicar un fichero Makefile para ello (si no se indica ningún Makefile y existe alguno en el directorio principal del proyecto, lo utilizará). CDT es un plugin oficial de la plataforma Eclipse y puede ser obtenido de la dirección www.eclipse.org.

VisualEditor

Hasta hace muy poco, la mayor pega que se le achacaba al IDE Eclipse era que no disponía de un editor gráfico de interfaces de usuario. Pues bien, esta pega ya no existe. El Proyecto Eclipse dispone de un plugin, llamado Visual Editor, que permite, de forma sencilla y completamente visual, diseñar interfaces gráficos de usuario (Figura 29).
VE es un plugin oficial de Eclipse y puede obtenerse de la dirección www.eclipse.org.

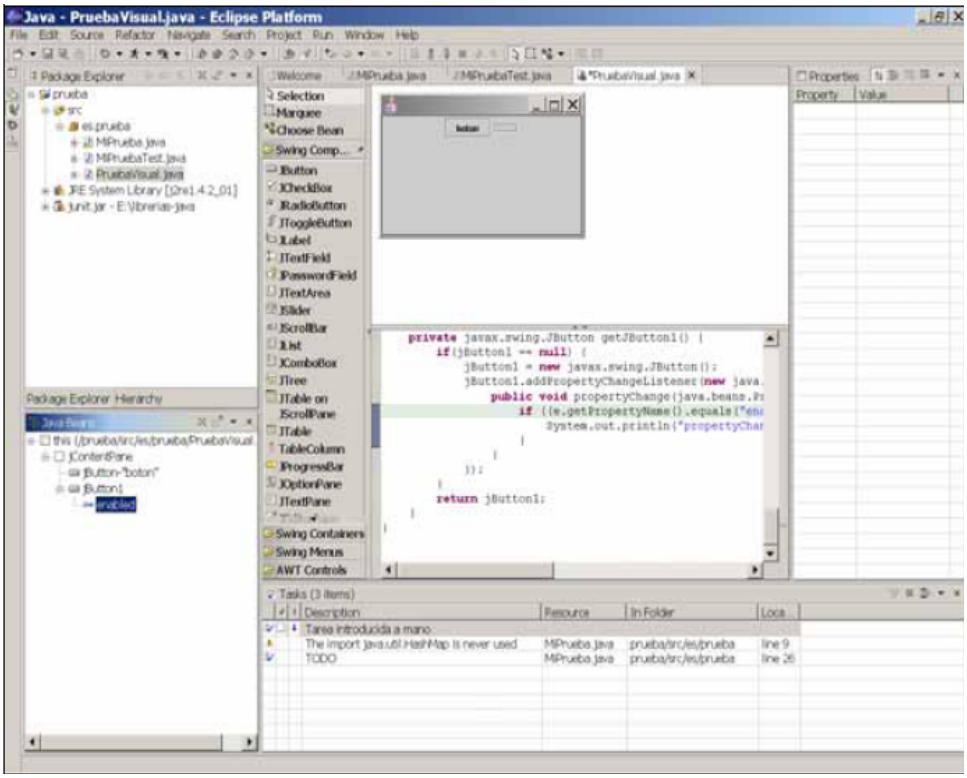


Figura 29. Plugin VE desplegado.

Omondo UML

Los plugins disponibles para Eclipse no se tienen por qué limitar obligatoriamente a la programación. Existen plugins que permiten integrar otras partes del proceso de desarrollo de aplicaciones como, por ejemplo, el diseño. El plugin UML (Figura 30) permite unificar diseño e implementación en una sola herramienta. Cualquier actualización realizada sobre el modelo UML, se verá reflejada en el código fuente de todas las clases a las que dicha modificación afecte, y a la inversa, cualquier cambio en el código se verá reflejado en los diagramas de clases UML.

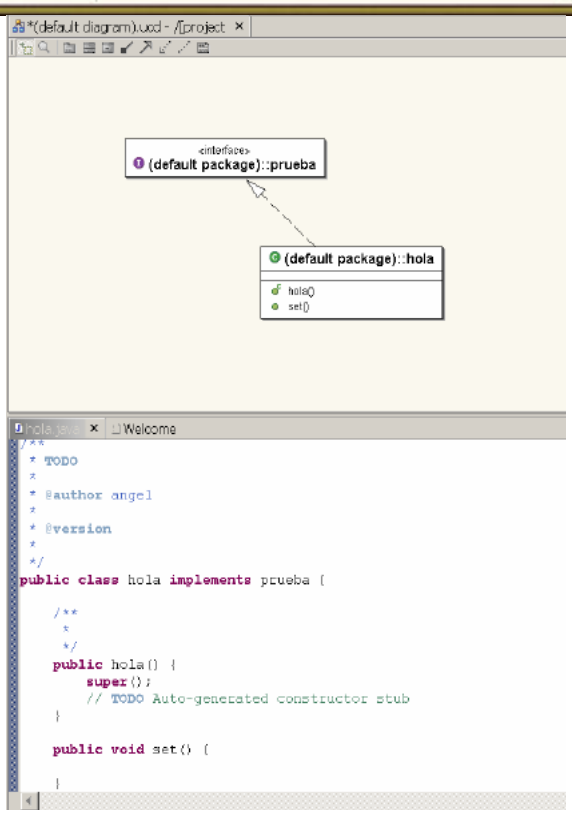


Figura 30. Plugin Omondo UML.

Jalopy

A pesar de que Eclipse incluye una herramienta para formatear el código fuente, es posible, que sus opciones estén algo limitadas para determinados proyectos. Jalopy es un plugin que permite formatear el código, añadir comentarios, etc. pero es mucho más flexible que la herramienta de formateo estándar.

Lomboz

Lomboz es un plugin que facilita el desarrollo de aplicaciones J2EE. Incluye reconocimiento sintáctico para páginas JSP y HTML, capacidad de depurar aplicaciones Web (incluyendo páginas JSP), lanzamiento automático de los servidores de aplicaciones más habituales, wizards para crear EJBs y un largo etcétera.

El plugin Lomboz, que es OpenSource desde hace relativamente poco tiempo, se puede obtener en www.objectlearn.com.

16. Conclusiones

Como se ha visto a lo largo de este trabajo, la programación orientada a objetos y Java proveen al programador de una potente plataforma y metodología de desarrollo para el programador principiante de este paradigma como al que cuenta con experiencia en el campo. Y sin duda alguna, ha sido una muy buena elección para seguir adentrándose en este amplio ramo de la computación.

Sin ser un partidario de una u otra herramienta para el desarrollo de software y aplicaciones, ha quedado de manifiesto por lo dicho en las secciones precedentes, que el uso de Software Propietario en la educación favorece la dependencia tecnológica, no sólo de los futuros profesionales sino también de sus futuros clientes (o alumnos, en el caso en que dicho profesional incursione en la docencia, como la ha sido mi caso). Y aunque el uso de “Software Libre” y ciertas licencias de software sólo data de unas décadas atrás, el concepto de libertad ha estado asociado desde siempre al desarrollo de las ciencias. Es así como han podido evolucionar la matemática, la física, y otras disciplinas.

Recordemos la célebre frase de Isaac Newton: *"Si he visto más lejos que otros, es porque estaba parado sobre el hombro de gigantes"*.

El Software Libre, al permitir el uso irrestricto, el análisis y la reutilización de los programas impulsa fuertemente el intercambio de conocimientos y la colaboración entre programadores de todo el mundo.

Hay ciertas dudas que siempre se plantean con respecto al desarrollo y crecimiento profesional de un programador, si certificarse en una plataforma de programación o estudiar muy bien las plataformas que se imparten en la universidad, es aquí donde tenemos ha Visual Studio y a C++/Java, siendo la segunda opción una herramienta muy utilizada en el ámbito de la educación superior formando parte de las clases troncales del futuro profesional de las ciencias de la computación, en mi opinión personal y de muchos con amplia experiencia se puede decir lo siguiente:

El mercado laboral estima lo que son atributos que se quieran poner en una hoja de vida y ello agradará o no a quien lo lea según el criterio que este utilizando, lo que se llama perfil. En muchas empresas de Latinoamérica, como ser la pequeña y mediana empresa, valoran mucho la experiencia del candidato. En las empresas de línea (grandes o productoras de software) valúan mucho la experiencia pero siempre el marco de conocimientos que da un titulo es importante.

Con la educación superior se obtiene u obtendrá un marco teórico de lo que se tiene que hacer, como y porque. Tal vez no se obtenga experiencia en el momento, pero el cursar y aprobar ciertas materias se le permitirá al individuo donde buscar y como implementar soluciones que sean factibles.

La experiencia sin formación es como tener “alas” pero no “pies”. Si uno adquiere los conocimientos, se aplica en ellos y los mejora siempre será un

firme candidato a bolsas de trabajo, siempre y cuando este dentro de las posibilidades de brindarse. La formación teórica siempre establece un margen a favor. Y la certificación hoy en día también es muy importante.

Hay un dicho por ahí de un autor francés que dice:

"En teoría, no hay diferencia entre teoría y práctica. En la práctica, si la hay"

Tiene siempre que existir un compromiso por la excelencia y brindar lo mejor de si para ser un mejor hijo, hermano, compañero, amigo y servir a DIOS, a la patria y al mundo.

17. Bibliografía

- Eckel, Bruce. "Thinking in Java. Third Edition". Prentice Hall. Disponible en www.mindview.net/Books/TIJ
- Sierra, Kathy; Bates, Bert. "Head First Java. 2nd Edition. Chap. 2". O'reilly
- Balagtas Tiu, Florence. "Introduction to Programming in Java". JEDI. Version 1.0 Mayo 2005.
- Introducción a la OOP. Referencia del Programador. Grupo EIDOS
- Guía de inicialización al lenguaje Java. Versión 2.0 octubre de 1999. Universidad de Burgos.
- Apuntes y lecturas de la asignatura programación orientada a objetos, Universidad Castilla-La Mancha en su sitio de Internet.
- Apuntes y lecturas de la asignatura POO. Universidad Tecnológica de Honduras. www.uth.edu
- Java Course, disponible en www.javapassion.com
- Programming Paradigm. Wikipedia, the Free Encyclopedia.
- Object Oriented Design Tips. www.eventhelix.com
- Artículo: Instalar el entorno Eclipse. En www.retratoensepia.com
- 1 Hora con Eclipse. Eclipse Newbies. En eclipse_newbies@yahoo.com.ar
- www.wikipedia.org