INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Curso online: **PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA (TELEFORMACIÓN - ON LINE)**

Módulo 1: EL LENGUAJE JAVA

Autores

Jorge Molinero Muñoz

Miguel Sallent Sánchez

ÍNDICE

1. OBJETIVOS

En este primer capítulo aprenderemos qué es **Java** y cuáles son las tecnologías relacionadas con este lenguaje de programación. Aprenderemos sus principales características, y configuraremos el entorno de desarrollo para comenzar a desarrollar aplicaciones.

2. ¿QUÉ ES JAVA? CARACTERÍSTICAS

Java es un lenguaje de programación de alto nivel creado por la empresa **Sun MicroSystems** en 1995. Tomando como base el lenguaje C++, Java fue diseñado para ser pequeño, simple e independiente de la plataforma de ejecución.

En abril de 2010 Sun fue adquirida por Oracle, por lo que actualmente es esta empresa la que dirige la plataforma.

Las principales características de Java son:

Java es orientado a objetos

En Java los objetos son la base de todo el lenguaje, de forma que no es algo opcional para el programador, sino que todo gira alrededor de ellos.

La diferencia estriba en que, por ejemplo, en C++ se puede declarar una clase, pero no de forma obligatoria, ya que en su lugar puede emplearse una estructura o una unión; asimismo también pueden usarse grupos de variables con el estilo de programación procedimental. En Java no pueden definirse funciones ni procedimientos, ya que no existen, ni tampoco existen las estructuras, unions o las definiciones de tipos. En Java casi todo son clases u objetos, exceptuando las operaciones básicas (while, for, ...) y los tipos básicos (int, float, boolean, ...).

Por tanto Java ofrece todas las características de la programación orientada a objetos: jerarquía de clases, herencia, encapsulación y polimorfismo.

La principal razón para seguir la metodología de la programación orientada a objetos, aparte de la claridad y simplicidad, es la esperanza de poder reutilizar en el futuro los objetos que uno desarrolla, reduciendo así el coste de futuras aplicaciones, como veremos en el tema de *Programación Orientada a Objetos*.

Java es independiente de plataforma

Una de las principales características de Java es que el código fuente puede ser compilado en cualquier máquina independientemente del fabricante o el sistema operativo que utilice. Además,

una vez compilada una aplicación escrita en Java, ésta puede ser ejecutada, de nuevo, en cualquier máquina, sin necesidad de recompilar el código fuente.

Esto se debe a que la compilación de un programa Java crea una salida binaria que no es ejecutada directamente por la máquina, sino que es "ejecutada" por un programa intérprete: la Máquina Virtual de Java. Por ello se dice que Java es un lenguaje "interpretado".

Disponemos de numerosas distribuciones de la Máquina Virtual de Java para las distintas arquitecturas (Windows, Linux, Mac), de forma que un mismo binario java puede ser ejecutado en cualquiera de ellas. Este código binario independiente de plataforma se denomina *bytecode*.

Java es robusto y fiable

El objetivo de diseño de Java es no permitir al programador realizar las cosas que normalmente hace peor, y facilitarle el realizar las cosas que suele hacer bien.

Java no proporciona conversión automática de tipos. Para realizarla se tiene que convertir explícitamente de una clase a otra. En C++ la conversión se realiza automáticamente. Java, por tanto, no asume que el programador sabe lo que hace, se asegura que para hacer algo, ese algo debe estar reflejado en el código.

Los punteros no existen en Java para que el programador pueda utilizarlos de forma explícita (internamente los tipos de Java se implementan mediante punteros). No pueden accederse a los objetos indirectamente o por casualidad. Todo el acceso a los objetos se realiza declarándolos y mediante una referencia directa. Se ha eliminado por completo la compleja aritmética de punteros. El concepto de "la dirección de un objeto" se ha eliminado del modelo de programación, por lo que es más sencillo hacer las cosas correctamente. Por ejemplo, si se necesita usar un conjunto indexado de objetos, se utiliza un array de objetos.

El programador tampoco tiene que preocuparse por la *gestión de memoria*: no hay que reservar memoria o liberarla ya que la máquina virtual se encarga de ello automáticamente, evitando los típicos errores que se producen en otros lenguajes por ejemplo cuando no se libera correctamente la memoria que ya no va a ser utilizada.

La fiabilidad de Java se extiende desde el lenguaje hasta el compilador y el sistema de ejecución. En tiempo de compilación se identifican múltiples errores de programación, no solo sintácticos, sino también semánticos. En tiempo de ejecución las comprobaciones son más amplias y efectivas.

Java es seguro

En otros lenguajes de programación, como C y C++, existen un gran número de condiciones potenciales que afectan a la seguridad: obtener acceso al sistema operativo, causar un retorno inesperado del control del programa, sobrescribir áreas críticas de memoria, adquirir la habilidad de

inspeccionar y/o modificar otros programas, buscar información de seguridad, obtener acceso no autorizado al sistema de ficheros...

La flexibilidad de estos lenguajes es lo que desencadena estas consecuencias no deseadas. Java es auto-defensivo por naturaleza, realizando un elevado número de comprobaciones para evitar los diferentes ataques que pueden producirse sobre la seguridad.

Java es concurrente

Java, como Ada, proporciona soporte interno para la programación concurrente. Esta característica permite la ejecución de múltiples hilos de ejecución o *threads* en un único programa, permitiéndose la realización de múltiples tareas simultáneamente. Por ejemplo, permitir la ejecución de una animación gráfica, reproducir a la vez un fichero de sonido relacionado y permitir que el usuario obtenga de forma interactiva información de aquello que está visualizando. La concurrencia permite a su vez liberar ciclos de ejecución de la CPU para realizar tareas de mantenimiento, como por ejemplo la recolección automática de memoria, permitiendo que el impacto de estas funciones sobre el rendimiento del programa sea menor.

La programación concurrente es más compleja cuando los diferentes *threads* deben interactuar entre sí, por lo que Java provee los mecanismos necesarios para que la ejecución se realice de forma segura y correcta mediante mecanismos de sincronización.

3. ENTORNO DE DESARROLLO

3.1. JDK: JAVA DEVELOPMENT KIT

El **JDK** es el entorno de desarrollo ofrecido por Oracle de forma gratuita que permite desarrollar aplicaciones Java.

En general cuando decimos que un ordenador tiene Java instalado, a lo que nos referimos es a que tiene instalado un **JRE** (*Java Runtime Environment*), es decir, una máquina virtual que nos permite ejecutar aplicaciones Java.

El JRE no incluye herramientas de desarrollo, para lo que necesitaremos instalar el JDK, que sí incluye herramientas como un compilador, un depurador, etc. El JDK también incorpora un JRE para ejecutar las aplicaciones, así pues podemos decir que JDK es un JRE ampliado.

Podemos descargar el JDK desde la página web de Oracle:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

En la plataforma Java hay que tener en cuenta que lo que antes se llamaba JDK, pasó a llamarse más tarde SDK. En la versión actual, el kit de desarrollo se vuelve a llamar JDK, pero en general son

términos que se refieren a lo mismo. Por este motivo, pueden verse indistintamente los dos términos en este curso.

Java ofrece distintas distribuciones, en función del tipo de aplicación que vayamos a desarrollar. Las principales son:

- Java SE (Standard Edition): permite desarrollar aplicaciones de escritorio, así como applets y librerías genéricas.
- Java EE (Enterprise Edition): permite desarrollar aplicaciones web, y además introduce otra serie de utilidades usadas en entornos profesionales.
- Java ME (Micro Edition): permite desarrollar aplicaciones para dispositivos móviles (teléfonos, pdas, tablets, etc).

En este curso trabajaremos con la versión más básica (Java SE).

Nótese que las distintas ediciones no se diferencian en los términos de la licencia con que son distribuidas, sino en las características que incluyen.

3.2. ENTORNOS DISPONIBLES

Para desarrollar aplicaciones en Java lo único que necesitamos es instalar el JDK, y con un simple editor de textos podremos crear aplicaciones, compilándolas y ejecutándolas mediante un intérprete de comandos, como se verá más adelante (por ejemplo la consola MS-DOS de Windows).

Sin embargo los diferentes fabricantes de software de desarrollo han sacado al mercado herramientas que facilitan esta tarea. El desarrollo de aplicaciones con un tamaño y complejidad elevadas requiere del uso de **entornos de desarrollo integrados** (IDEs) de altas prestaciones. Estas herramientas integran distintas utilidades en un mismo paquete, por ejemplo, un editor, un compilador, un depurador, etc. Suelen ser altamente visuales, empleando un gran número de las facilidades gráficas de los entornos de ventanas. Su objetivo es facilitar las tareas del desarrollador haciendo que el diseño de software sea más rápido, eficiente y fácil de depurar.

Asimismo, suelen incluir una herramienta que permite navegar a través de la jerarquía de clases, editores de código que resaltan las palabras reservadas del lenguaje con diferentes colores, generación automática de código, compiladores just-in-time, integración con el repositorio de código fuente, ejecución de pruebas unitarias, etc.

En la actualidad existen un gran número de herramientas de este tipo para programar en Java. En la siguiente tabla se proporcionan algunas de las existentes y se adjuntan algunas direcciones de Internet donde se puede obtener más información al respecto.

IDE	Empresa	Dirección Web
BlueJ	BlueJ Org	https://www.bluej.org/
Eclipse	Eclipse Foundation	https://www.eclipse.org/
IntelliJ IDEA	JetBrains	https://www.jetbrains.com/idea/
JDeveloper	Oracle	https://www.oracle.com/database/technologies/developer-tools/jdeveloper.html
NetBeans	Oracle	https://netbeans.org/

Para la realización del curso será necesario preparar un entorno desde el que se puedan realizar los ejercicios propuestos, y que nos permitirá editar, compilar y ejecutar las aplicaciones.

El alumno podrá elegir entre dos opciones:

- Opción 1: Instalación del JDK de java y ejecución en línea de comandos
- Opción 2: Entorno de desarrollo ECLIPSE

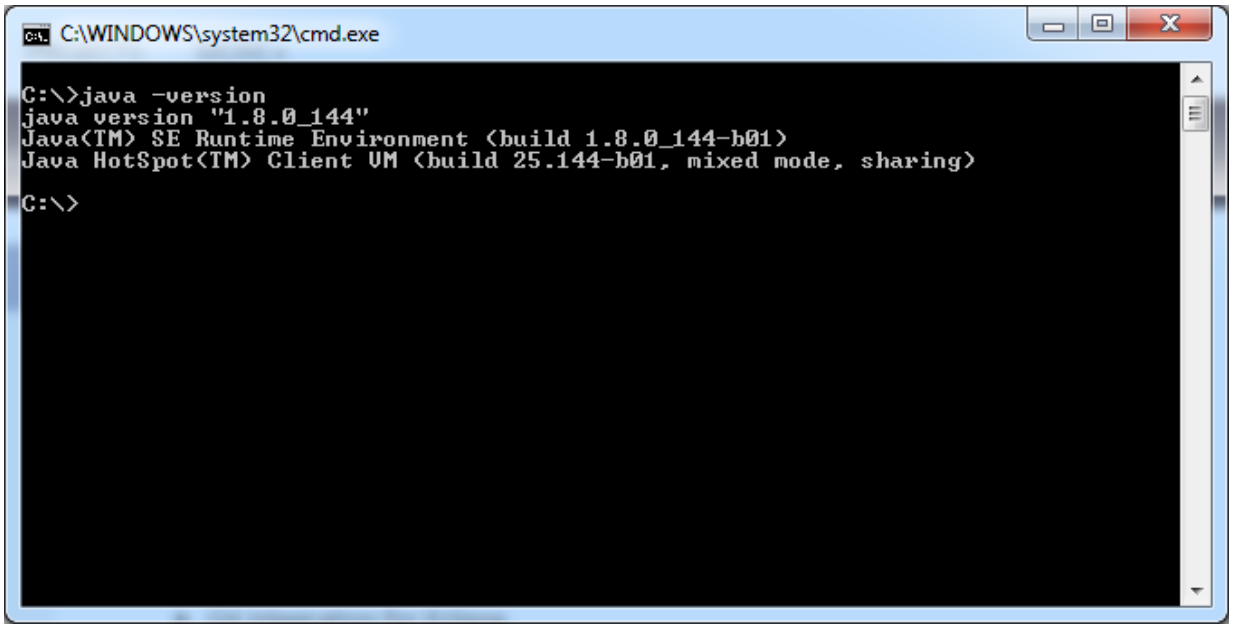
El entorno que recomendamos instalar es ECLIPSE, ya que es un entorno muy extendido entre la comunidad de desarrolladores y además es *open source* y gratuito.

A continuación se describen los pasos para instalar cada uno de los dos posibles entornos. Las explicaciones están adaptadas a un entorno Windows por ser el más extendido, pero tanto java como Eclipse pueden ejecutarse en otros sistemas operativos como Linux o Mac, por lo que no habría problema en realizar el curso en una máquina con alguno de estos otros sistemas operativos.

3.3. ENTORNO ECLIPSE

3.3.1. REQUISITOS PREVIOS

Eclipse es una aplicación basada en Java, y por tanto para su funcionamiento requiere que previamente tengamos instalado el entorno de ejecución java en el equipo (JRE). Es muy posible que en el equipo en que estemos trabajando ya esté instalado y no tengamos que hacer nada. Para averiguarlo, abriremos una ventana de línea de comandos MS-DOS (Menú Inicio → Programas → Accesorios → Símbolo del sistema) y escribiremos “java –version”. Si Java está instalado, el comando mostrará la versión de java:



En este ejemplo la versión de java es 1.8.0_144, lo cual quiere decir que tenemos instalado Java 8. Si empezara por 1.7.xx sería Java 7, y así sucesivamente.

En caso de que java no esté instalado o la versión sea inferior a la 8, hay que ir a la página oficial de java y seguir las instrucciones:

http://www.java.com/es/download/

Es posible que para instalar java se requieran permisos de administrador, por lo que si estamos en el ordenador de la oficina puede que se deba solicitar la instalación o actualización al correspondiente departamento de informática.

3.3.2.INSTALACIÓN DE ECLIPSE

Eclipse tiene varias distribuciones dependiendo del tipo de programa que se vaya a desarrollar. En nuestro caso, elegiremos la distribución *"Eclipse IDE for Java Developers"*, que se encuentra disponible aquí:

http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygenr

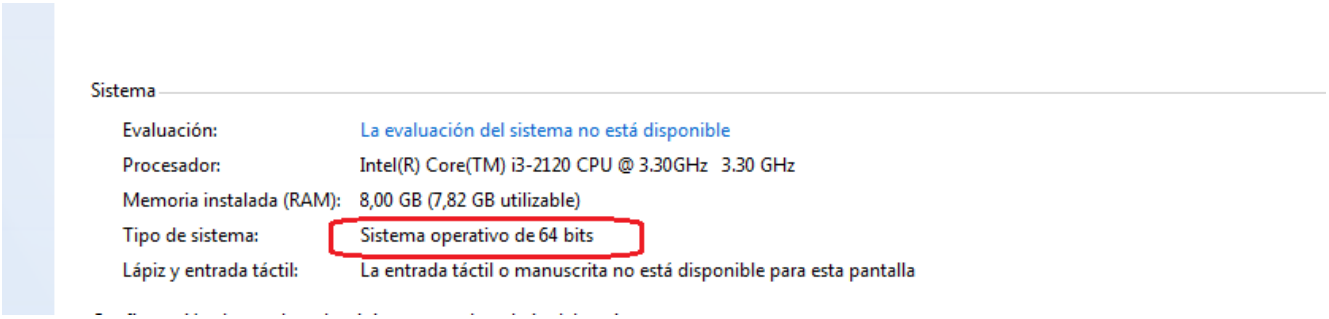
Esta versión (Oxygen) necesita Java 8, de forma que si la versión de Java en nuestro equipo fuera la 7 o anterior no sería posible ejecutarlo.

Elegimos nuestro sistema operativo en el menú de la derecha y descargamos el programa.

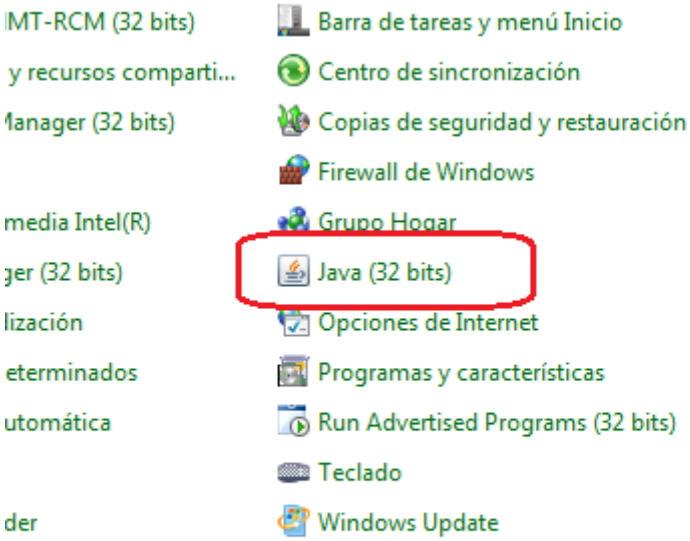
Nota importante para los usuarios de Windows con arquitectura de 64 bits: es posible que aunque el equipo sea de 64 bits la versión de java que esté instalada en el ordenador sea la de 32 bits y se ejecute en modo de compatibilidad.

En un ordenador con Windows y arquitectura de 64 bits, podemos elegir entre: - Java de 32 bits y Eclipse de 32 bits (modo compatibilidad) - Java de 64 bits y Eclipse de 64 bits Es imprescindible que ambos programas sean o bien de 32 o bien de 64, si mezclamos no funcionará.

Para saber si nuestra versión de Windows es de 32 o 64 bits se puede mirar en el Panel de Control -> Sistema:



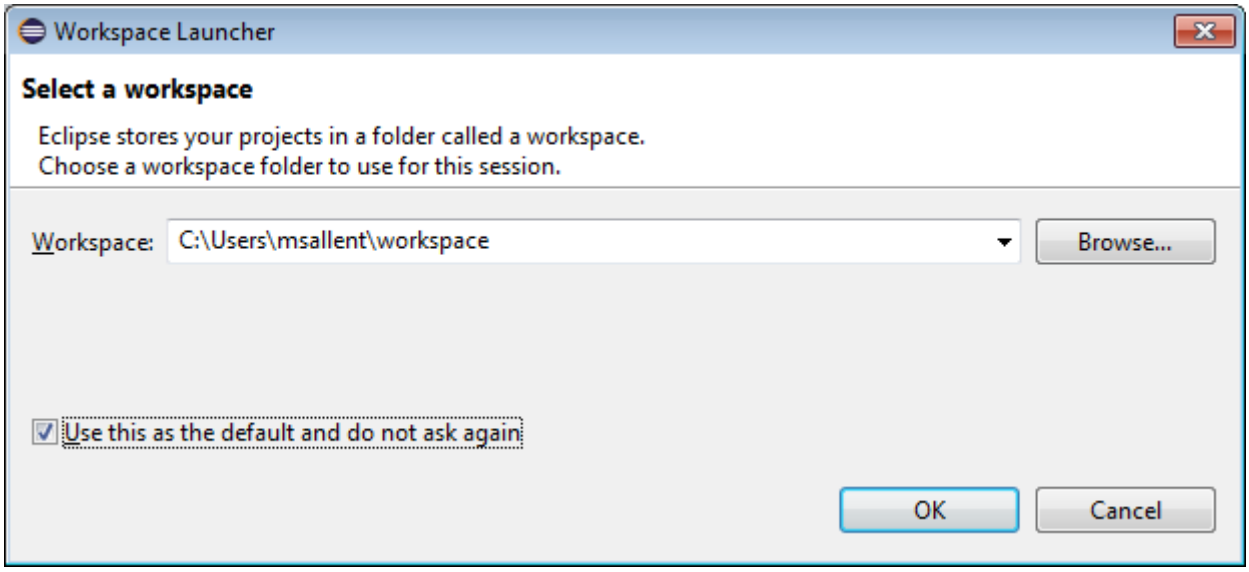
Para saber si nuestra versión de Java es de 32 o 64 bits, podemos ver si hay una carpeta con el nombre "Java" dentro de "C:\Program Files (x86)" en lugar de en "C:\Program Files". En caso afirmativo la versión de Java instalada es la de 32 bits. También podemos mirarlo en el panel de control:



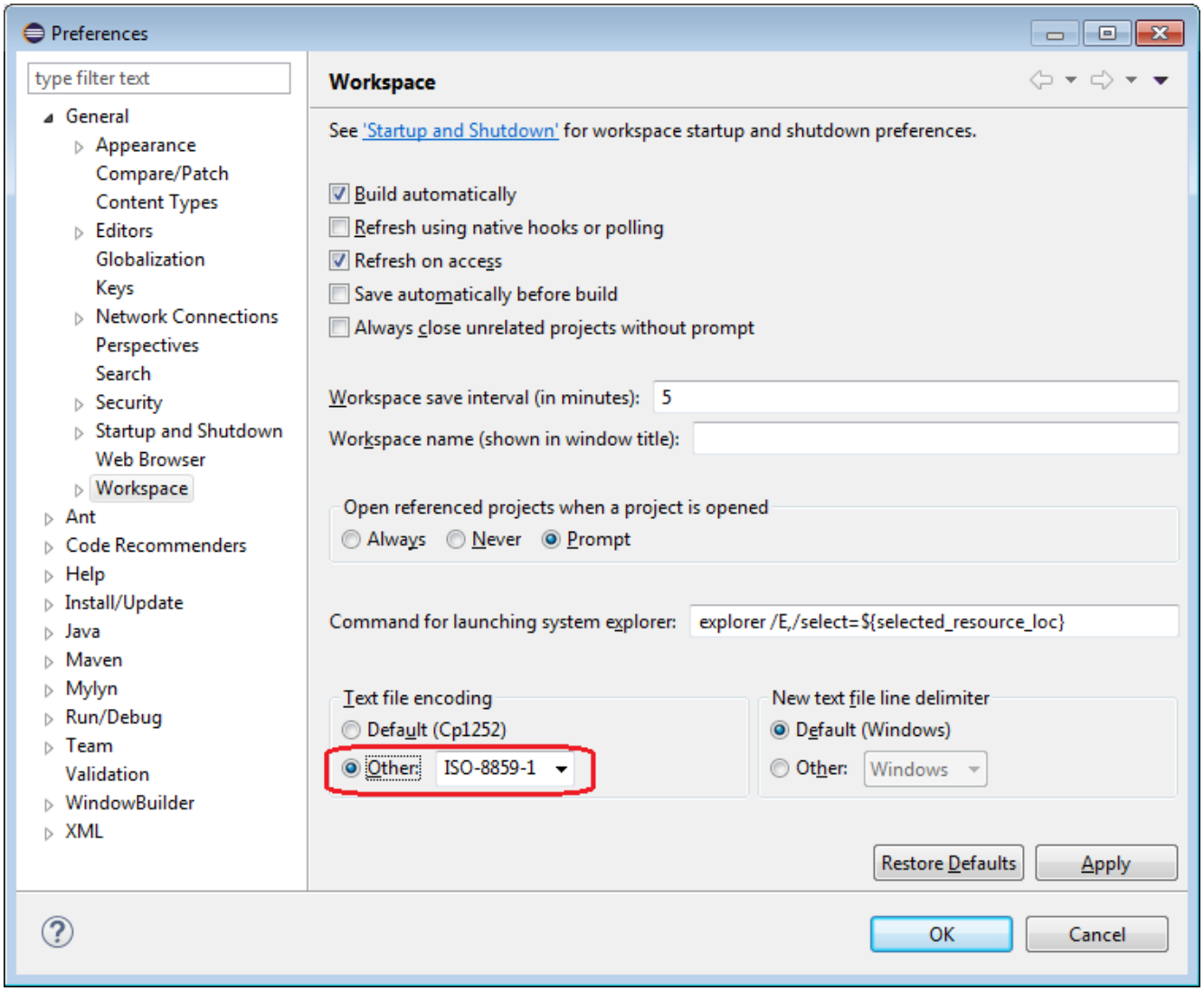
En caso de que la versión de Java sea la de 32 bits, debemos descargar la versión de Eclipse de 32 bits, no la de 64.

El archivo descargado no es ejecutable, sino que es un zip que hay que descomprimir en el directorio en que queramos instalar la aplicación. Se recomienda un directorio que no tenga espacios en blanco en su ruta (por ejemplo C:\eclipse).

Una vez descomprimido, ejecutamos el archivo eclipse.exe. Inicialmente nos pedirá seleccionar un *workspace* o entorno de trabajo. Dejamos la opción por defecto y marcamos "recordar":

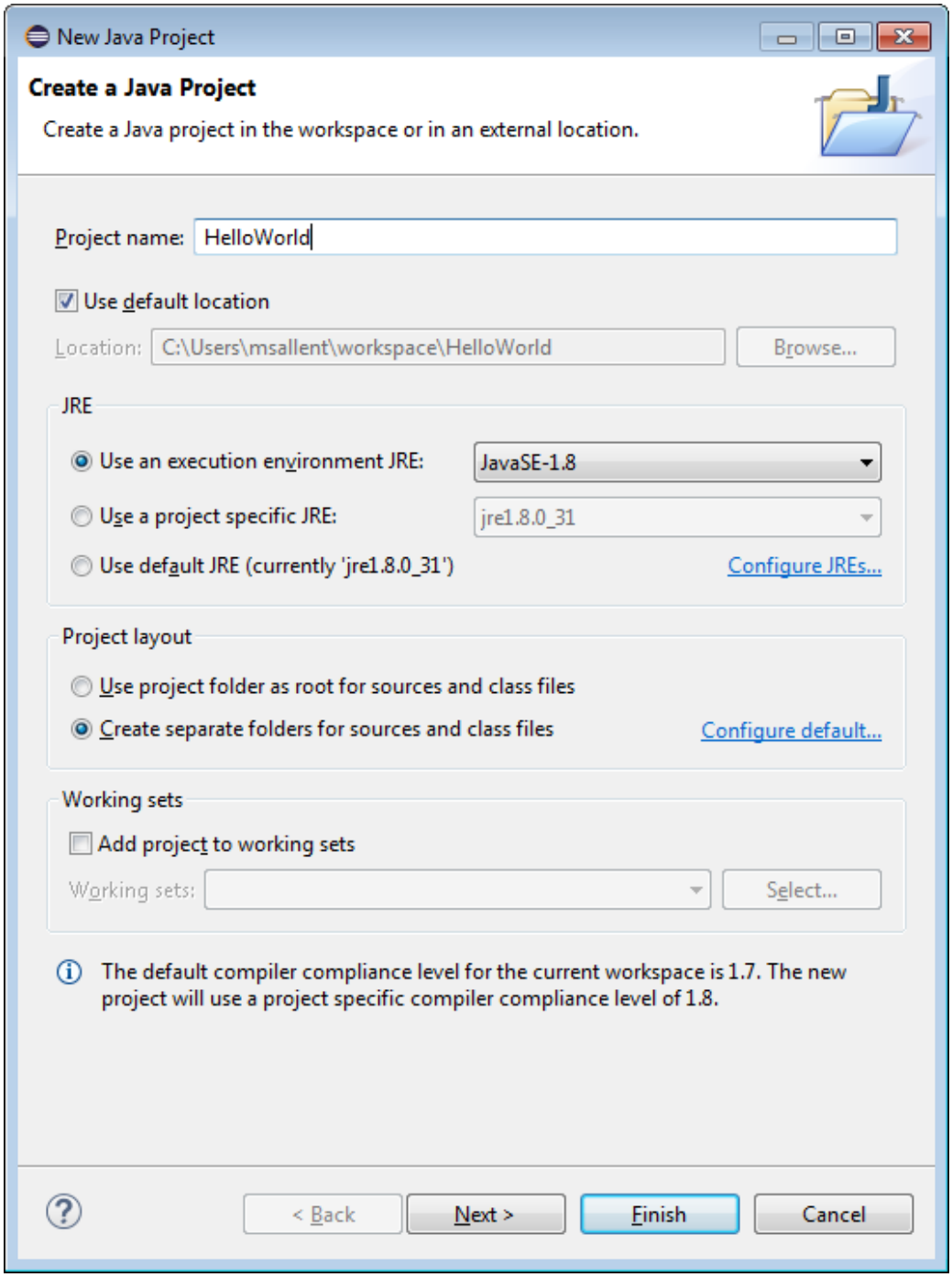


Finalmente configuraremos la aplicación para que los ficheros de código fuente trabajen con el encoding ISO-8859-1, que es el más habitual para castellano. Para ello iremos al menú Window →*Preferences* →*General* →*Workspace:*

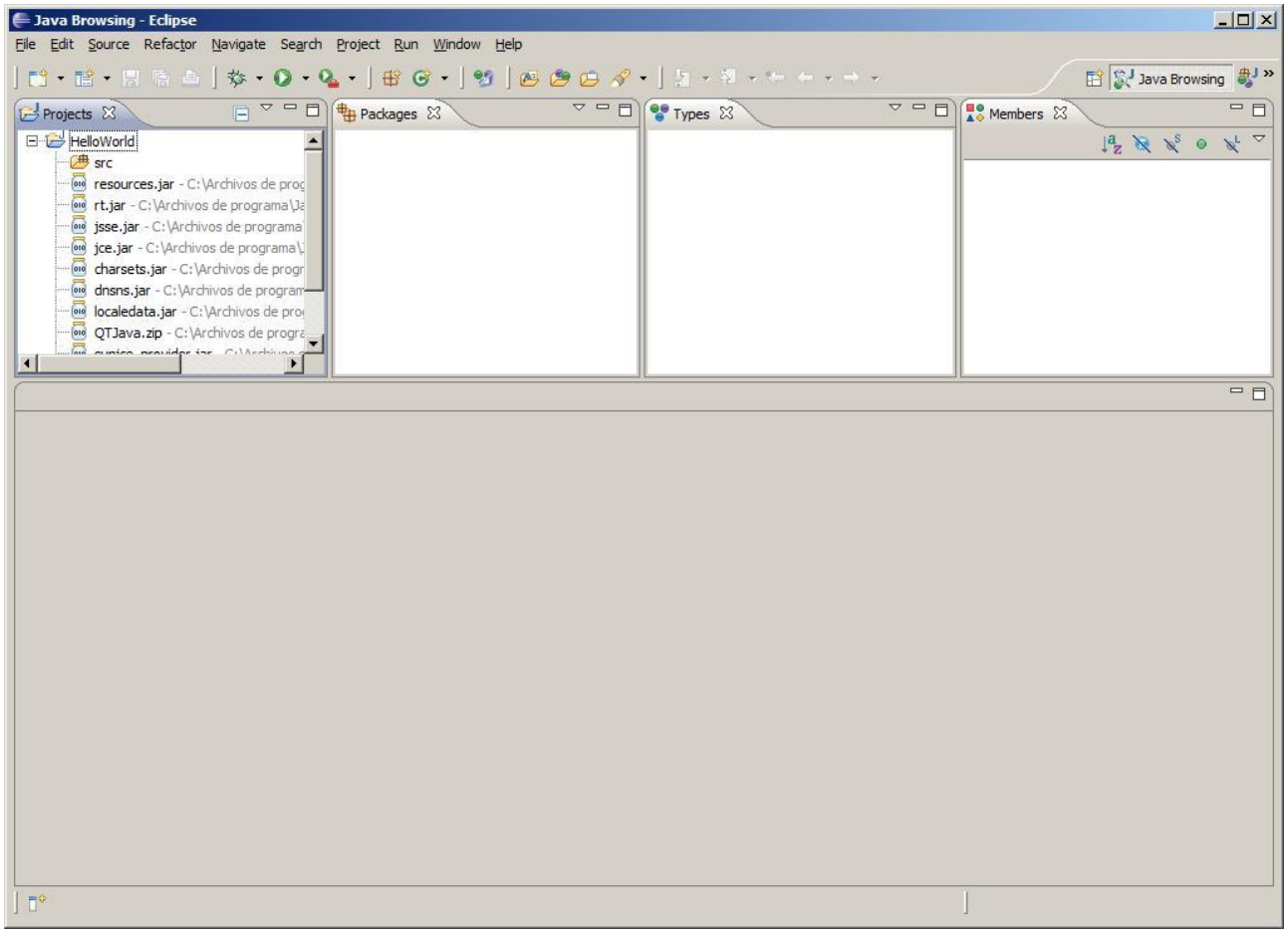


3.3.3.USO

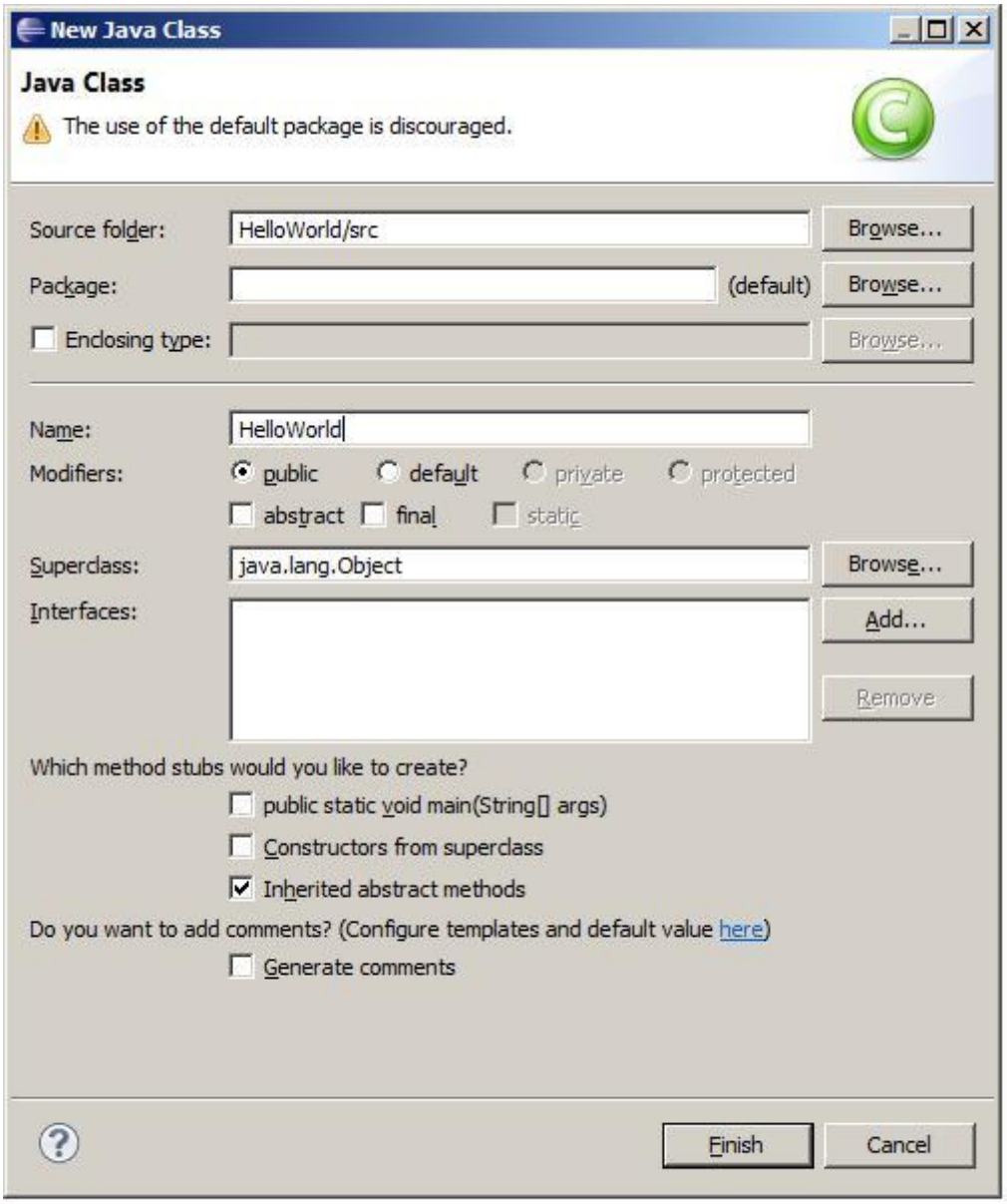
Para comenzar, crearemos un proyecto nuevo, desde el menú *File* → *New* → *Java Project*. Elegimos un nombre para el proyecto y dejamos el resto de parámetros a su valor por defecto:



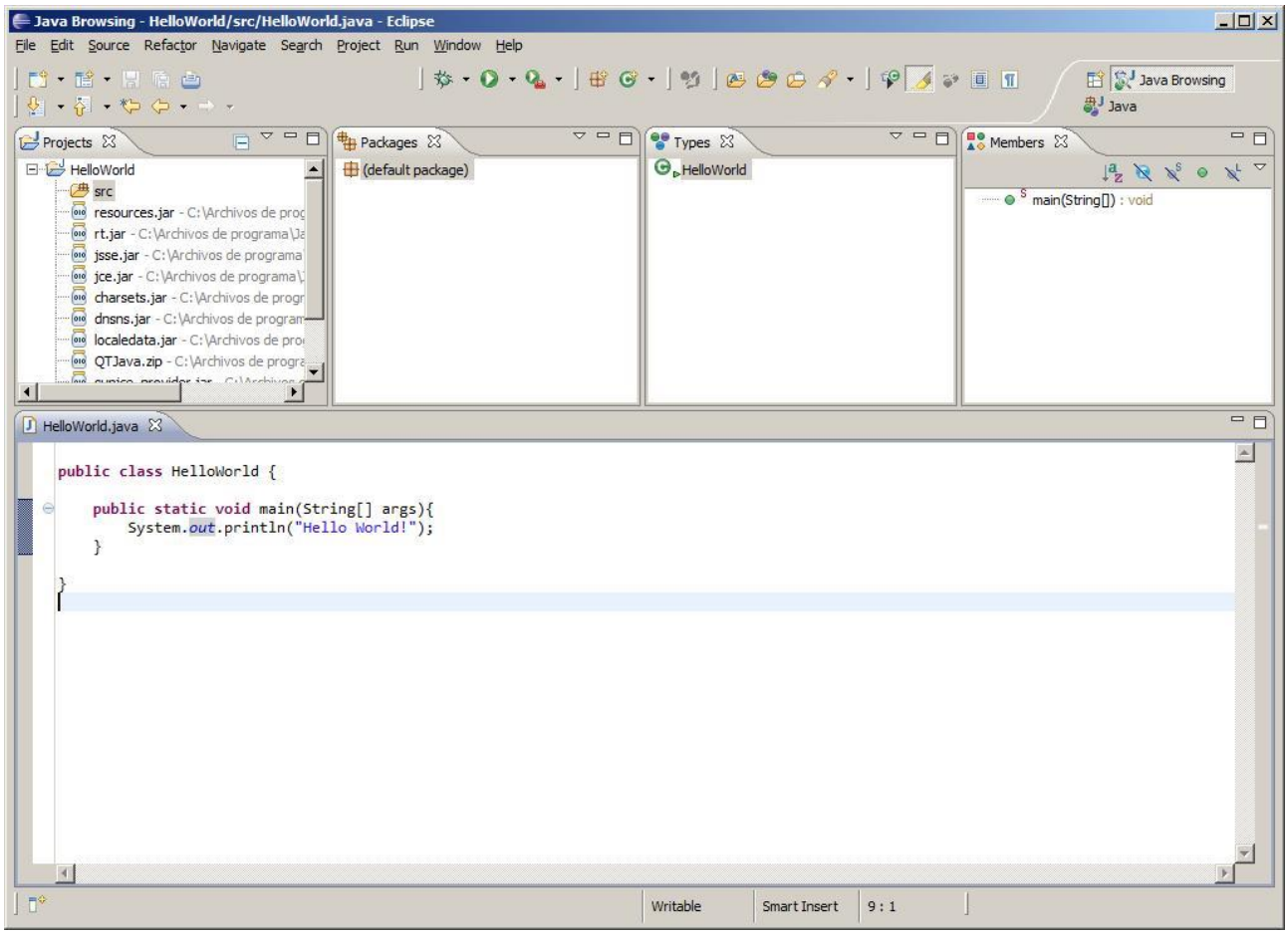
A continuación elegimos una vista para trabajar, menú *Window* → *Open Perspective* → *Java Browsing*. Vemos que aparece el proyecto recién creado:



A continuación creamos una clase: menú *File* → *New* → *Class*. Introducimos el nombre y dejamos el resto de opciones a su valor por defecto:

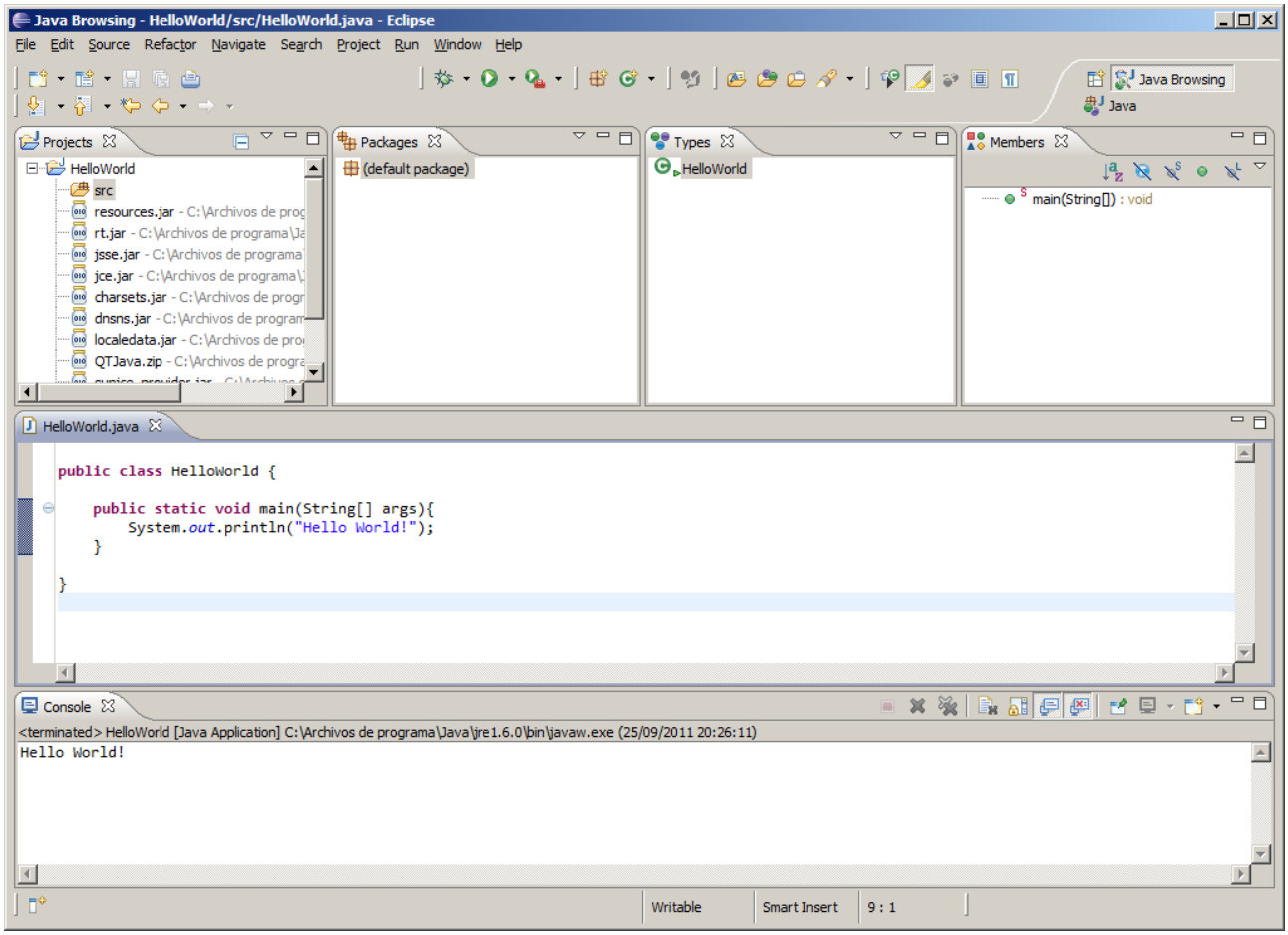


En la parte inferior tenemos el código fuente de nuestra clase, lo editamos y escribimos el código:



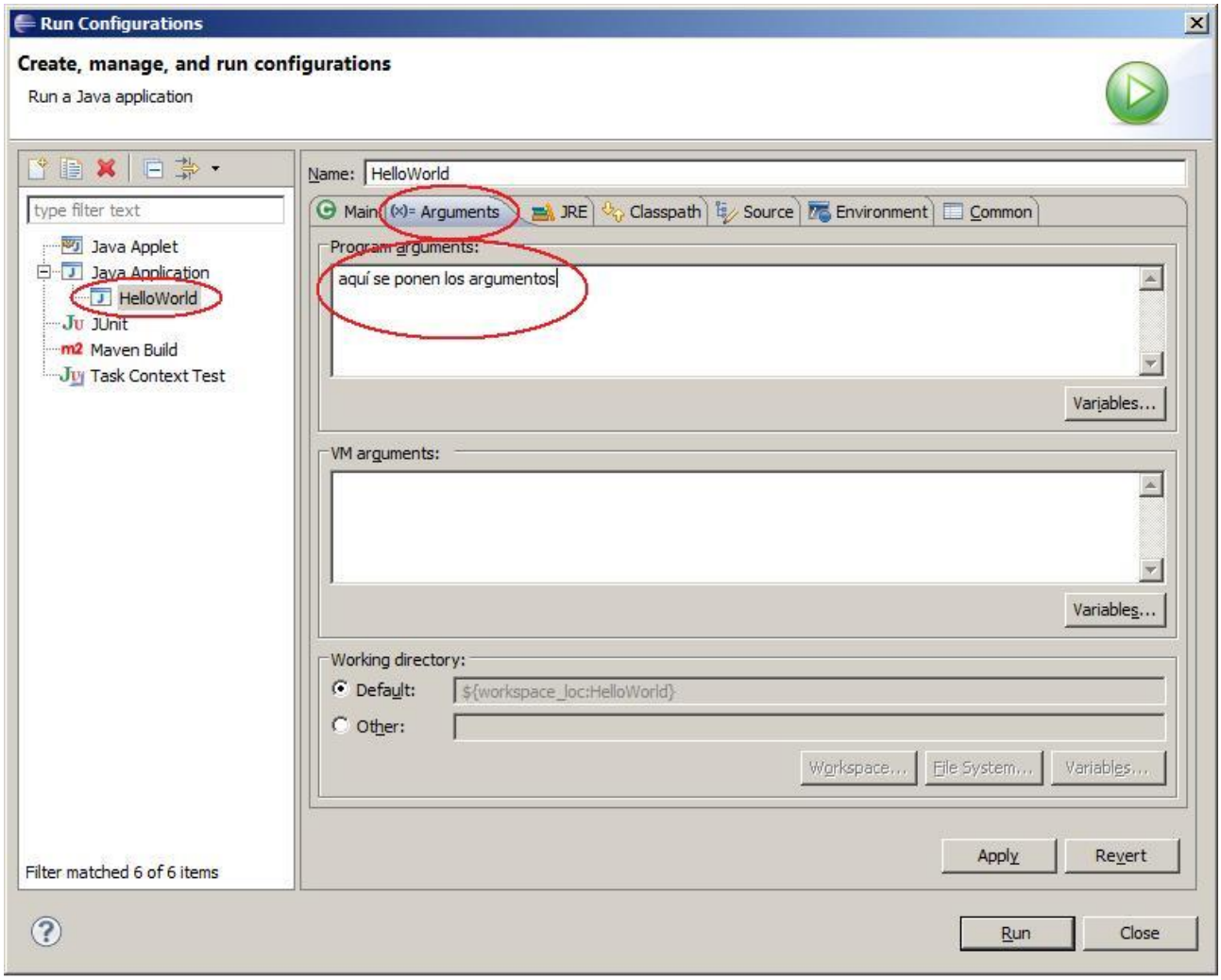
Vemos que a medida que vamos escribiendo, se van mostrando los errores de compilación, por lo que no es necesario invocar al compilador explícitamente.

Por último ejecutamos la aplicación desde el menú *Run* → *Run* (o bien mediante el botón de la barra de herramientas). Este es el resultado:



En la parte inferior vemos que aparece una nueva ventana "*Console*", que es el resultado de la ejecución de nuestro programa.

Si deseamos pasar argumentos a nuestro programa, tal y como haríamos en la línea de comandos, lo que tenemos que hacer es ir al menú *Run → Run Configurations*, seleccionar la pestaña *Arguments* y escribir los argumentos:



Podemos crear un proyecto para cada ejercicio que tengamos que realizar, ya que cada ejercicio puede tener varias clases y paquetes, y de esta forma lo tendremos organizado.

3.4. ENTORNO EN LÍNEA DE COMANDOS

Nota: la instalación del entorno de desarrollo en línea de comandos sólo es necesaria si no se ha podido instalar el entorno ECLIPSE.

3.4.1. INSTALACIÓN

En primer lugar instalaremos el entorno de desarrollo JDK. En este punto conviene recordar la diferencia entre JRE y JDK

- JRE (*Java Runtime Enviroment*) nos permite ejecutar aplicaciones, pero no nos permite compilar nuestros desarrollos. Suele estar instalado en la mayoría de los ordenadores ya que permite ejecutar aplicaciones java.
- JDK (*Java Development Kit*). Además del JRE, incluye también un compilador (*javac*) así como otras herramientas de desarrollo (depurador, etc).

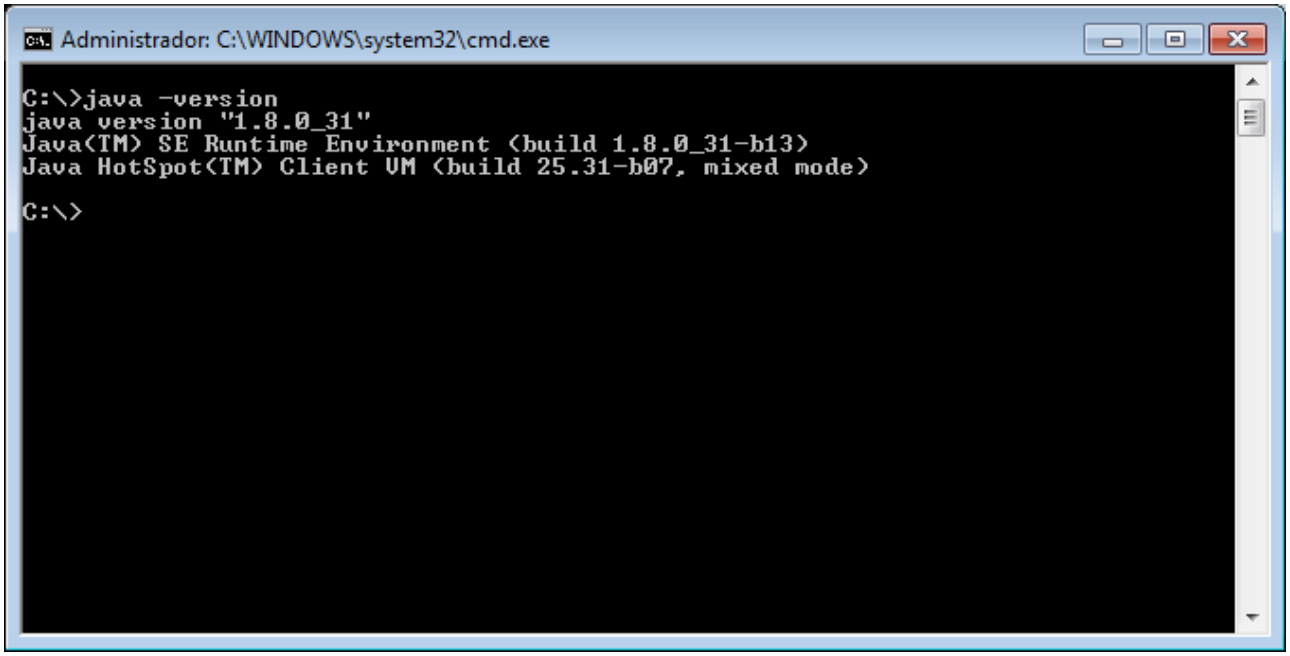
Queda claro entonces que lo que debemos instalar es el JDK para poder compilar nuestros ejercicios.

La versión con la que trabajaremos será Java SE 8, disponible aquí:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Elegimos la versión adecuada a nuestro sistema operativo, descargamos el programa de instalación y lo ejecutamos.

Una vez instalado comprobamos que está funcionando correctamente, para ello abrimos una ventana de línea de comandos[1], y ejecutamos el comando "java -version". Este comando nos muestra la versión de java que está instalada.



Si observamos que aparece una versión distinta de la que hemos instalado, esto puede deberse a que en la máquina tenemos varios entornos java instalados, y está tomando como entorno por defecto uno distinto al que queremos. Para establecer el entorno por defecto, deberemos editar la variable de entorno PATH, y asegurarnos de que la ruta a la carpeta *bin* de nuestro JDK se encuentre delante de la de cualquier otro JDK o JRE instalado. Por ejemplo, en Windows esto se hace desde el Panel de Control → Sistema → Variables de Entorno.

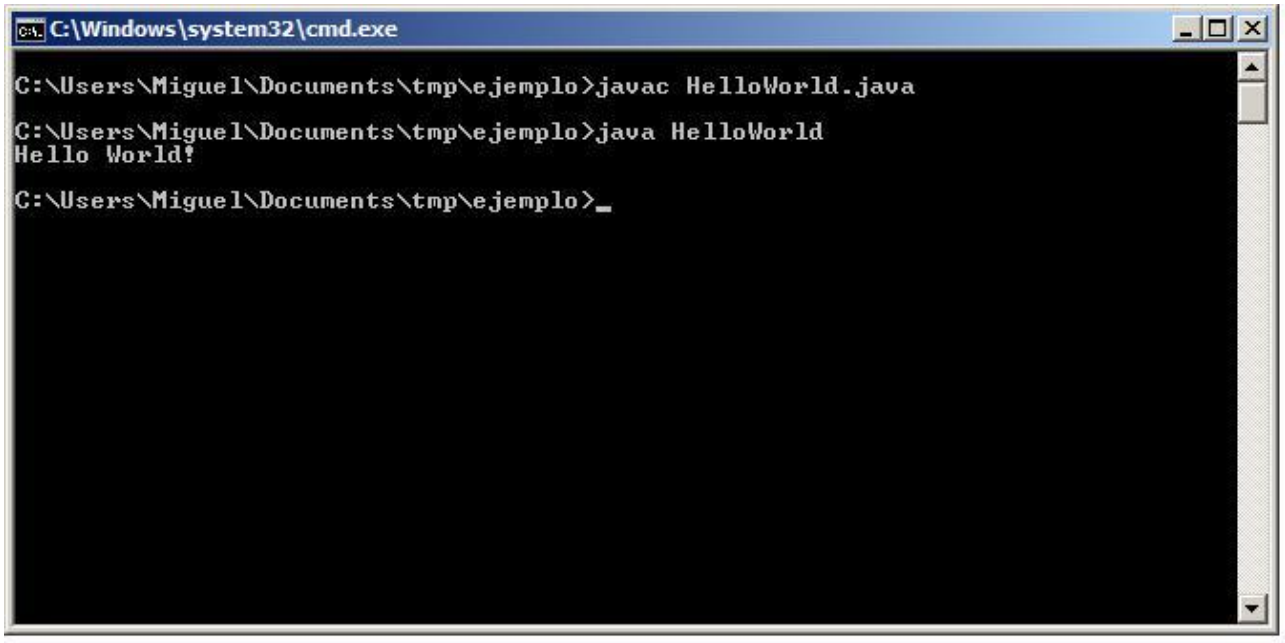
1 En Windows es la consola de MS-DOS

3.4.2.USO

Una vez instalado el JDK, crearemos nuestro fichero de código fuente java, lo compilaremos y lo ejecutaremos. Para ello, abrimos el bloc de notas (o cualquier otro editor de texto plano) y creamos un archivo que se llame "HelloWorld.java". El contenido será el siguiente:

```
public class HelloWorld {

  public static void main(String[] args){

     System.out.println("Hello World!");

  }
}
```

El siguiente paso es compilarlo. Para ello abrimos una ventana de comandos MS-DOS, nos situamos en el directorio en el que hayamos guardado nuestro código fuente, y ejecutamos "javac HelloWorld.java". Una vez hecho esto, si no hay errores de compilación se habrá creado un fichero "HelloWorld.class", que es el fichero ejecutable (binario bytecode). Para ejecutarlo, escribimos "java HelloWorld".



Obsérvese que al compilador javac se le pasa como argumento el nombre del fichero fuente (terminado en .java), y al intérprete java se le pasa sólo el nombre de la clase (sin incluir el .class).

El entorno de desarrollo en Java es muy sensible al contexto, es decir, diferencia entre mayúsculas y minúsculas no sólo dentro del código fuente, sino también en el nombrado de los ficheros. Es necesario que los nombres de los ficheros coincidan exactamente con los de las clases a los que se asocian. Además, los ficheros fuente en lenguaje Java deben tener la extensión *.java,* y los ficheros bytecode extensión *.class*.

Ejemplo: si desarrollamos una clase denominada *Aplicación*, el nombre del fichero que la contenga debe ser *Aplicación.java*, no siendo válido, por ejemplo, *aplicación.java*. Una vez compilado el fichero fuente, se obtendrá el correspondiente fichero bytecode, que para este ejemplo sería *Aplicación.clas*s.

4. CONCEPTOS BÁSICOS DE ORIENTACIÓN A OBJETOS

Antes de comenzar con la sintaxis básica de Java, es necesario presentar los conceptos básicos de orientación a objetos.

4.1. OBJETO

Un objeto es una unidad dentro de un programa que consta de un **estado** y un **comportamiento**. Los objetos a menudo se usan para modelar objetos del mundo real: por ejemplo una bicicleta, un perro, etc.

Los objetos tienen estado (por ejemplo el perro tendrá *nombre*, *raza*, *color*, *edad*) y comportamiento (*ladrar*, *correr*, *comer*, *beber*, *etc*). En programación orientada a objetos, el estado se almacena en los **atributos** del objeto, que son como variables pero pertenecen al objeto. Y el comportamiento se expone a través de los **métodos.** Por ejemplo podríamos tener un objeto lámpara con un atributo *encendida*, que sólo puede ser verdadero o falso, y dos métodos: *encender* y *apagar.* Los métodos pueden por tanto modificar el estado del objeto.

Un paso más es definir la visibilidad de cada atributo o método: en general podemos elegir si queremos que un determinado atributo o método sea visible desde fuera del objeto (público) o sólo desde dentro (privado). Se trata de ocultar al exterior los detalles de implementación del objeto y mostrar al exterior sólo lo necesario para que pueda ser utilizado. A esto se le llama **encapsulamiento.**

Por ejemplo, supongamos que tenemos un objeto que representa una bicicleta, que tendrá un atributo *marcha* que representa la marcha seleccionada. Este atributo será de tipo entero. Pero la bicicleta sólo tiene seis marchas, por lo que no se debería poder establecer el valor de este atributo fuera del rango entre 1 y 6. Para conseguir esto ocultamos el atributo *marcha* para que no pueda ser

modificado directamente desde el exterior del objeto, y sólo podrá ser modificado llamando a los métodos *subirMarcha* y *bajarMarcha,* que no sólo modificarán el atributo interno sino que además comprobarán que nunca esté fuera del rango entre 1 y 6. Por tanto el encapsulamiento nos permite restringir el acceso a los detalles internos del objeto de forma que garantizamos siempre un estado consistente y un uso adecuado de la funcionalidad.

Además del encapsulamiento el uso de objetos tiene las siguientes ventajas:

- Modularidad: el código de un objeto puede ser escrito y mantenido de forma independiente del código de otros objetos.
- Reusabilidad: si un objeto ya existe, podremos reusar ese objeto en nuestro programa. Esto permite que los especialistas diseñen objetos que están perfectamente escritos, probados, depurados, etc, de forma que podemos confiar en ellos para usarlos en nuestro código.

4.2. CLASE

En el mundo real encontraremos muchos objetos individuales del mismo tipo. Puede haber cientos de bicicletas, todas de la misma marca y modelo. Cada bicicleta tiene los mismos componentes y se comporta de igual manera. En orientación a objetos decimos que una bicicleta determinada es una **instancia** de la **clase** de objetos *Bicicleta*. Por tanto una clase define cómo serán los atributos y los métodos, y luego podemos crear y utilizar varios objetos de esa clase, cada uno con un estado distinto.

El siguiente ejemplo es una posible implementación de la clase *Bicicleta*.

```
class Bicicleta {

    int cadencia = 0;
    int velocidad = 0;
    int marcha = 1;

    void cambiaCadencia(int nuevoValor) {
         cadencia = nuevoValor;
    }

    void cambiaMarcha(int nuevoValor) {
         marcha = nuevoValor;
    }

    void aumentaVelocidad(int incremento) {
         velocidad = velocidad + incremento;
    }

    void frena(int decremento) {
         velocidad = velocidad - decremento;
    }

    void imprimeEstado() {
```

```
            System.out.println("cadencia:" +
                cadencia + " velocidad:" +
                velocidad + " marcha:" + marcha);
    }
}
```

Los atributos *cadencia, velocidad* y *marcha* representan el estado de la bicicleta, y los métodos (*cambiaCadencia, cambiaMarcha, etc.*) definen su interacción con el mundo exterior.

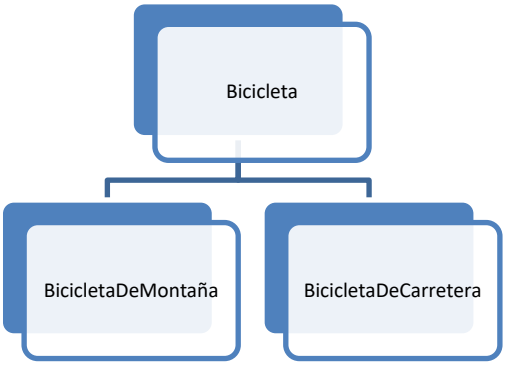
Ahora veremos cómo crear objetos de esta clase. Para ello crearemos una segunda clase que hace uso de ella:

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

        // Creamos dos objetos Bicicleta diferentes
        Bicicleta bici1 = new Bicicleta();
        Bicicleta bici2 = new Bicicleta();

        // Usamos los métodos de estos objetos
        bici1.cambiaCadencia(50);
        bici1.aumentaVelocidad(10);
        bici1.cambiaMarcha(2);
        bici1.imprimeEstado ();

        bici2.cambiaCadencia(50);
        bici2.aumentaVelocidad(10);
        bici2.cambiaMarcha (2);
        bici2.cambiaCadencia(40);
        bici2.aumentaVelocidad(10);
        bici2.cambiaMarcha(3);
        bici2.imprimeEstado();
    }
}
```

4.3. HERENCIA

Hay ciertos tipos de objetos que comparten cosas en común con otros tipos. Por ejemplo podemos tener bicicletas de montaña y bicicletas de carretera. Ambos tipos de bicicletas tienen atributos en común (velocidad, cadencia, marcha), pero además tienen otra serie de características que las diferencian: por ejemplo una bicicleta de carretera podría tener marchas adicionales.

La programación orientada a objetos permite a las clases **heredar** el estado y el comportamiento de otras clases. En nuestro ejemplo tendríamos una clase padre o superclase Bicicleta, y dos clases hijas o subclases, BicicletaDeMontaña y BicletaDeCarretera.



La sintaxis es la siguiente:

```
class BicicletaDeMontaña extends Bicicleta {

    // los nuevos campos y métodos
    // irían aquí

}
```

La clase hija tendrá todos los atributos y métodos de la clase padre, y además los atributos y métodos que defina ella.

En Java no hay herencia múltiple, de forma que una clase sólo puede tener una única clase padre.

4.4. INTERFAZ

Los objetos definen su interacción con el mundo exterior a través de los métodos que exponen. Los métodos forman el *interfaz* con el mundo exterior.

En Java podemos definir un **interfaz** como un conjunto de métodos vacíos, es decir, el interfaz sólo define cómo se comporta un objeto pero no lo implementa:

```
interface VehiculoConMarchas{

    void cambiaMarcha(int nuevoValor);

}
```

Toda clase que implemente el interfaz *VehiculoConMarchas* deberá declarar e implementar un método *cambiaMarcha*.

El interfaz es como un contrato entre una clase y el mundo exterior: toda clase que implemente ese interfaz se compromete a comportarse de una manera determinada.

Una clase que implementara este interfaz lo haría de la siguiente manera:

```
class Bicicleta implements VehiculoConMarchas {

    int cadencia = 0;
    int velocidad = 0;
    int marcha = 1;

    void cambiaCadencia(int nuevoValor) {
         cadencia = nuevoValor;
    }

    public void cambiaMarcha(int nuevoValor) {
         marcha = nuevoValor;
    }

    void aumentaVelocidad(int incremento) {
         velocidad = velocidad + incremento;
    }

    void frena(int decremento) {
         velocidad = velocidad - decremento;
    }

    void imprimeEstado() {
         System.out.println("cadencia:" +
             cadencia + " velocidad:" +
             velocidad + " marcha:" + marcha);
    }
}
```

Vemos que la clase puede declarar otros métodos además de los del interfaz.

A diferencia de la herencia, una clase sí puede implementar varios interfaces.

4.5. PAQUETE

Un paquete es un espacio de nombres que agrupa a un conjunto de clases relacionadas. Conceptualmente es como una carpeta en el ordenador en la que agrupamos los ficheros de código fuente de las clases (cada clase típicamente está en un fichero fuente independiente).

Los nombres de los paquetes por convención se escriben en minúsculas. Podemos tener paquetes dentro de paquetes, igual que en el ordenador tenemos carpetas dentro de carpetas, en este caso cada nivel de anidación se separa por un punto. Por ejemplo tenemos el paquete *java.util*, y dentro de él el paquete *java.util.concurrent, java.util.regex, java.util.zip*, etc.

Para indicar que una clase pertenece a un paquete, al comienzo del fichero escribimos por ejemplo lo siguiente:

```
package cursojava.tema1;

class Bicicleta {
   …
}
```

Cuando queramos usar esta clase desde otra clase de un paquete distinto, tenemos varias opciones, o bien escribir el nombre de la clase completo:

```
cursojava.tema1.Bicicleta bici1 = new cursojava.tema1.Bicicleta();
```

o bien importar esa clase al comienzo:

```
import cursojava.tema1.Bicicleta;
…
Bicicleta bici1 = new Bicicleta();
```

o bien importar el paquete completo al comienzo:

```
import cursojava.tema1.*;
…
Bicicleta bici1 = new Bicicleta();
```

Hay que destacar que la importación de esta manera no es recursiva, es decir, si importamos por ejemplo java.util.* no estamos importando java.util.zip.*.

Realizar la importación de una u otra forma no afecta al rendimiento del programa ya que las importaciones son sólo instrucciones para el compilador.

El uso de paquetes no sólo permite organizar mejor el código, sino que también permite resolver conflictos de nombres, ya que podemos tener clases con el mismo nombre pero en distintos paquetes. Esto cobra especial relevancia cuando usamos código desarrollado por terceros, incluso ajenos a nuestra organización. Si cada proyecto desarrolla su código en su propio espacio de nombres, no habrá conflictos.