

Introducción a Java

Por Nacho Cabanes,

Versión 0.40 de Julio de 2015

Este texto se puede distribuir libremente,
siempre y cuando no se modifique

Última versión disponible en

www.nachocabanes.com/java

Tabla de contenido

1. Introducción: ¿por qué Java?	5
1.1. ¿Qué es Java?.....	5
1.2. ¿Por qué usar Java?	5
1.3. ¿Cuándo no usar Java?.....	5
1.4. ¿Qué más aporta Java?.....	6
1.5. ¿Qué hace falta para usar un programa creado en Java?.....	6
1.6. ¿Qué hace falta para crear un programa en Java?	6
2. Instalación del entorno de desarrollo	8
2.1. Instalación del JDK y NetBeans bajo Windows	8
2.2. Instalación de Eclipse en Windows	16
2.3. Instalación del JDK y Eclipse bajo Linux	17
2.4. Instalación de Netbeans bajo Linux.....	19
2.5. La alternativa ligera: Geany.....	21
3. Nuestro primer programa	22
3.1. Un programa que escribe "Hola Mundo"	22
3.2. Entendiendo este primer programa	22
3.2.1. Escribir en pantalla	23
3.2.2. Formato libre	23
3.2.3. El cuerpo del programa	23
3.2.4. El nombre del programa	24
3.2.5. Comentarios	24
3.2.6. Varias órdenes	25
3.2.7. Escribir y avanzar de línea	25
3.3. Compilar y lanzar el programa desde línea de comandos.....	26
3.4. La alternativa básica pero cómoda: Geany	27
3.5. Manejo básico de NetBeans	29
3.6. Contacto con Eclipse	33
4. Variables y operaciones matemáticas básicas	38
4.1. Las variables.....	38
4.2. Operaciones matemáticas básicas.....	40
4.3. Operar con datos introducidos por el usuario	40
4.4. Incremento y asignaciones abreviadas	42

4.5. Otros tipos de datos numéricos.....	44
5. Comprobación de condiciones	47
5.1. if.....	47
5.2. El caso contrario: else	48
5.3. Operadores relacionales	49
5.4. Operadores lógicos para enlazar condiciones.....	50
5.5. switch.....	51
5.6. El operador condicional.....	54
6. Bucles: partes del programa que se repiten.....	56
6.1. while.....	56
6.2. do-while	57
6.3. for	59
6.4. break y continue.....	63
6.5. Etiquetas.....	64
7. Booleanos, caracteres, cadenas de texto y arrays	66
7.1 Datos booleanos	66
7.2. Caracteres	67
7.3. Las cadenas de texto	68
7.3.1. String	68
7.3.2. StringBuilder.....	73
7.4. Los arrays.	75
8. Las Matemáticas y Java	81
9. Contacto con las funciones.	85
9.1.Descomposición modular	85
9.2. Parámetros.....	88
9.3. Valor de retorno	88
10. Clases en Java.	91
10.2. Varias clases en Java	93
10.3. Herencia	98
10.4. Ocultación de detalles	100
10.5. Sin "static"	101
10.6. Constructores	103

11. Ficheros.....	105
11.1. ¿Por qué usar ficheros?	105
11.2. Escribir en un fichero de texto	105
11.3. Leer de un fichero de texto.....	106
11.4. Leer de un fichero binario.....	108
11.5. Leer y escribir bloques en un fichero binario	109
Cambios en el curso.....	111

1. Introducción: ¿por qué Java?

1.1. ¿Qué es Java?

Java es un lenguaje de programación de ordenadores, desarrollado por Sun Microsystems en 1995 (compañía que fue posteriormente absorbida por Oracle, en 2010).

Hay varias hipótesis sobre su origen, aunque la más difundida dice que se creó para ser utilizado en la programación de pequeños dispositivos, como aparatos electrodomésticos (desde microondas hasta televisores interactivos). Se pretendía crear un lenguaje con algunas de las características básicas de C++, pero que necesitara menos recursos y que fuera menos propenso a errores de programación.

De ahí evolucionó (hay quien dice que porque el proyecto inicial no acabó de funcionar) hasta convertirse en un lenguaje muy aplicable a Internet y programación de sistemas distribuidos en general.

Pero su campo de aplicación no es exclusivamente Internet: una de las grandes ventajas de Java es que se procura que sea totalmente independiente del hardware: existe una "máquina virtual Java" para varios tipos de ordenadores. Un programa en Java podrá funcionar en cualquier ordenador para el que exista dicha "máquina virtual Java" (hoy en día es el caso de los ordenadores equipados con los sistemas operativos Windows, Mac OS X, Linux, y algún otro). Pero aun hay más: el sistema operativo Android para teléfonos móviles propone usar Java como lenguaje estándar para crear aplicaciones.

1.2. ¿Por qué usar Java?

Puede interesarnos si queremos crear programas que tengan que funcionar en distintos sistemas operativos sin ningún cambio, o programas cliente/servidor, o aplicaciones para un Smartphone Android, entre otros casos. Tampoco es un mal lenguaje para aprender a programar, aunque en ocasiones resulta un tanto engorroso para un principiante.

1.3. ¿Cuándo no usar Java?

Como debe existir un paso intermedio (la "máquina virtual") para usar un programa en Java, no podremos usar Java si queremos desarrollar programas para un sistema concreto, para el que no exista esa máquina virtual. Y si necesitamos que la velocidad sea la máxima posible, quizá no sea admisible lo (poco) que ralentiza ese paso intermedio.

1.4. ¿Qué más aporta Java?

Tiene varias características que pueden sonar interesantes a quien ya es programador, y que ya irá conociendo poco a poco quien no lo sea:

- La sintaxis del lenguaje es muy parecida a la de C++ (y a la de C, C#, PHP y algún otro). Eso simplifica el aprendizaje de Java si se conoce alguno de esos lenguajes, y también permite aprovechar los conocimientos de Java para aprender después uno de estos otros lenguajes.
- Es un lenguaje orientado a objetos, lo que supondrá ventajas a la hora de diseñar y mantener programas de gran tamaño.
- Permite crear programas "multitarea" (formados por varios hilos de ejecución), lo que ayuda a sacar mejor partido de los modernos procesadores con múltiples núcleos.
- Incluye control de excepciones, como alternativa más sencilla para manejar errores inesperados, como un fichero inexistente o una conexión de red perdida.
- Es más difícil cometer errores de programación que en otros lenguajes más antiguos, como C y C++ (por ejemplo, no existen los "punteros", que son una fuente de quebraderos de cabeza en esos lenguajes).
- Se pueden crear programas en modo texto, entornos "basados en ventanas", dibujar gráficos, acceder a bases de datos, etc.

1.5. ¿Qué hace falta para usar un programa creado en Java?

Vamos a centrarnos en el caso de un "ordenador convencional", ya sea de escritorio o portátil.

Las aplicaciones que deban funcionar "por sí solas" necesitarán que en el ordenador de destino exista algún "intérprete" de Java, eso que hemos llamado la "máquina virtual". Esto es cada vez más frecuente (especialmente en sistemas como Linux), pero si no lo tuviéramos (como puede ocurrir en Windows), basta con instalar el "Java Runtime Environment" (JRE), que se puede descargar libremente desde Java.com (unos 10 Mb de descarga).

1.6. ¿Qué hace falta para crear un programa en Java?

Existen diversas herramientas que nos permitirán crear programas en Java. La propia Oracle suministra un kit de desarrollo oficial que se conoce como JDK (Java

Development Kit). Es de libre distribución y se puede conseguir en la propia página Web de Oracle.

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

El inconveniente del JDK es que no incluye un editor para crear nuestros programas, sólo las herramientas para generar el programa ejecutable y para probarlo.

Por eso, puede resultar incómodo de manejar para quien esté acostumbrado a otros entornos integrados, como los de Visual C# y Visual Basic, que incorporan potentes editores. Pero no es un gran problema, porque es fácil encontrar editores que hagan más fácil nuestro trabajo, o incluso sistemas de desarrollo completos, como Eclipse (<http://www.eclipse.org/downloads/>) o NetBeans. (<https://netbeans.org/downloads/index.html>). Ambos son buenos y gratuitos. En el próximo tema veremos cómo descargarlos e instalarlos, así como alguna alternativa para ordenadores menos potentes.

2. Instalación del entorno de desarrollo

Para poder programar en Java, necesitarás tanto el compilador (el llamado "Kit de desarrollo", JDK) como algún editor. Veremos los entornos más habituales y su instalación en Windows y Linux. En principio, el entorno más recomendable para un principiante es NetBeans, y hay alternativas aún más ligeras, como Geany, cuya instalación es inmediata en Linux pero ligeramente más incómoda en Windows.

2.1. Instalación del JDK y NetBeans bajo Windows

El JDK (Java Development Kit) es la herramienta básica para crear programas usando el lenguaje Java. Es gratuito y se puede descargar desde la página oficial de Java, en el sitio web de Oracle (el actual propietario de esta tecnología, tras haber adquirido Sun, la empresa que creó Java):

www.oracle.com/technetwork/java/javase/downloads

Allí encontraremos enlaces para descargar (download) la última versión disponible.



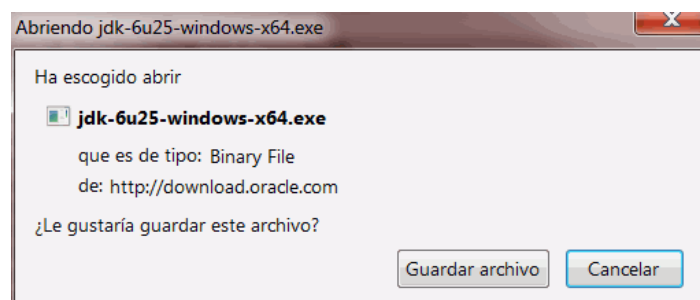
En primer lugar, deberemos escoger nuestro sistema operativo y (leer y) aceptar las condiciones de la licencia:

You must accept the [Java SE 6 JDK License Agreement](#) to download this software.

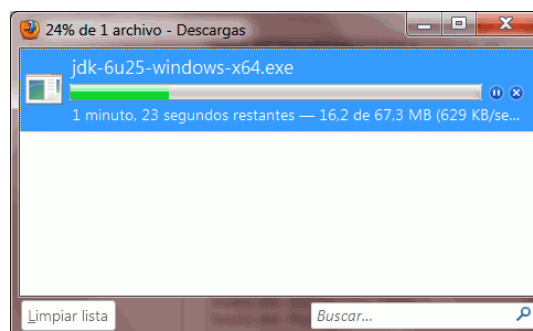
☒ Accept License Agreement
 ☐ Decline License Agreement

Java SE Development Kit 6 Update 25		
Product / File Description	File Size	Download
Linux x86 - RPM Installer	76.85 MB	jdk-6u25-linux-i586-rpm.bin
Linux x86 - Self Extracting Installer	81.11 MB	jdk-6u25-linux-i586.bin
Linux Intel Itanium - RPM Installer	76.85 MB	jdk-6u25-linux-ia64-rpm.bin
Linux Intel Itanium - Self Extracting Installer	81.11 MB	jdk-6u25-linux-ia64.bin
Linux x64 - RPM Installer	77.06 MB	jdk-6u25-linux-x64-rpm.bin
Linux x64 - Self Extracting Installer	81.36 MB	jdk-6u25-linux-x64.bin
Solaris x86 - Self Extracting Binary	81.00 MB	jdk-6u25-solaris-i586.sh
Solaris x86 - Packages - tar.Z	136.67 MB	jdk-6u25-solaris-i586.tar.Z
Solaris SPARC - Self Extracting Binary	85.96 MB	jdk-6u25-solaris-sparc.sh
Solaris SPARC - Packages - tar.Z	141.11 MB	jdk-6u25-solaris-sparc.tar.Z
Solaris SPARC 64-bit - Self Extracting Binary	12.24 MB	jdk-6u25-solaris-sparcv9.sh
Solaris SPARC 64-bit - Packages - tar.Z	15.58 MB	jdk-6u25-solaris-sparcv9.tar.Z
Solaris x64 - Self Extracting Binary	8.49 MB	jdk-6u25-solaris-x64.sh
Solaris x64 - Packages - tar.Z	12.25 MB	jdk-6u25-solaris-x64.tar.Z
Windows x86	76.66 MB	jdk-6u25-windows-i586.exe
Windows Intel Itanium	67.27 MB	jdk-6u25-windows-ia64.exe
Windows x64	67.27 MB	jdk-6u25-windows-x64.exe

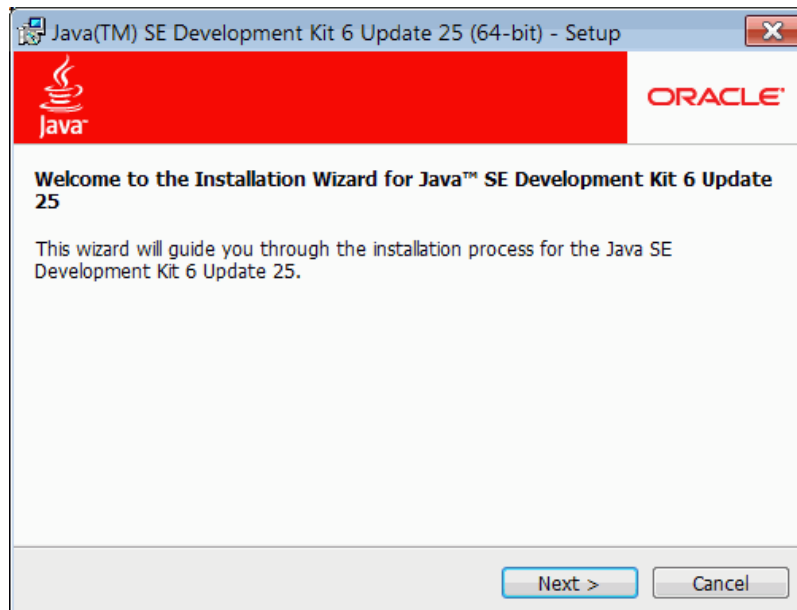
Entonces empezaremos a recibir un único fichero de gran tamaño (cerca de 70 Mb, según versiones):



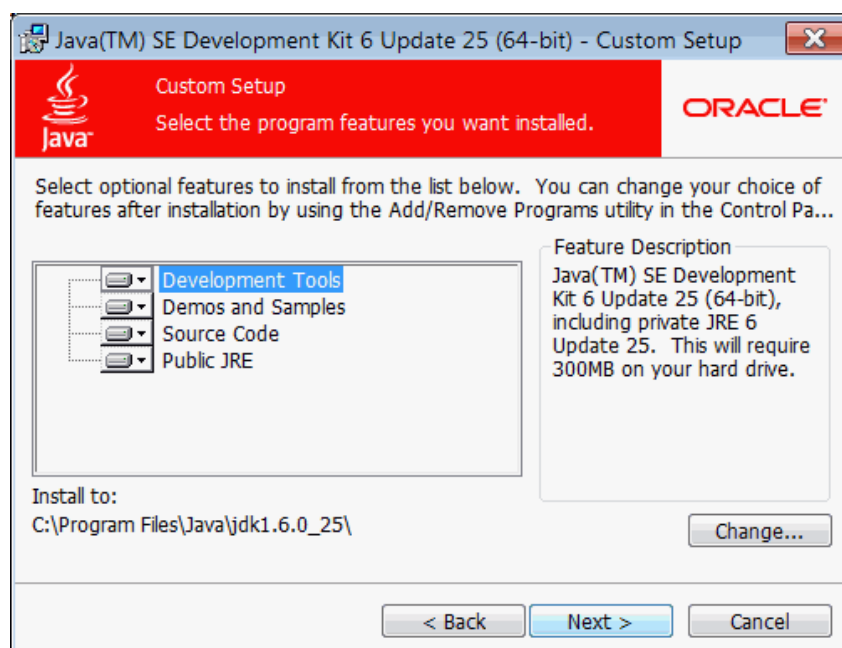
Al tratarse de un fichero de gran tamaño, la descarga puede ser lenta, dependiendo de la velocidad de nuestra conexión a Internet (y de lo saturados que estén los servidores de descarga):



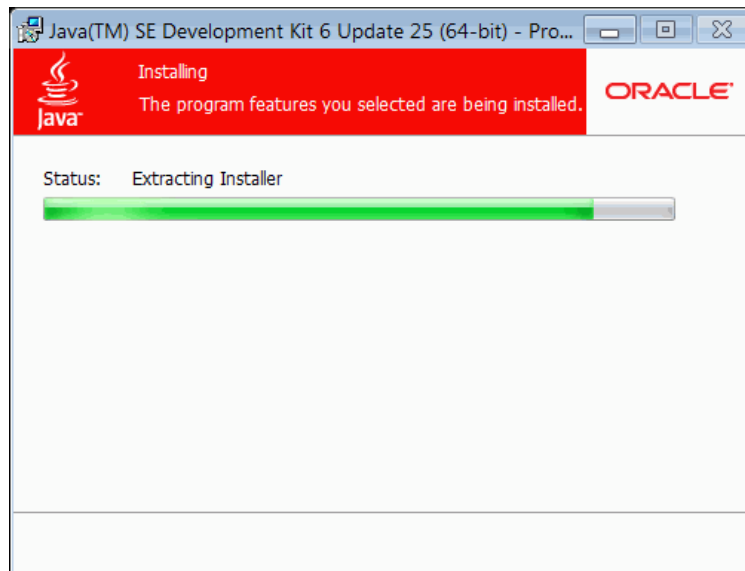
Cuando hayamos descargado, haremos doble clic en el fichero, para comenzar la instalación propiamente dicha:



Podremos afinar detalles como la carpeta de instalación, o qué partes no queremos instalar (por ejemplo, podríamos optar por no instalar los ejemplos). Si tenemos suficiente espacio (posiblemente unos 400 Mb en total), generalmente la opción más sencilla hacer una instalación típica, sin cambiar nada:

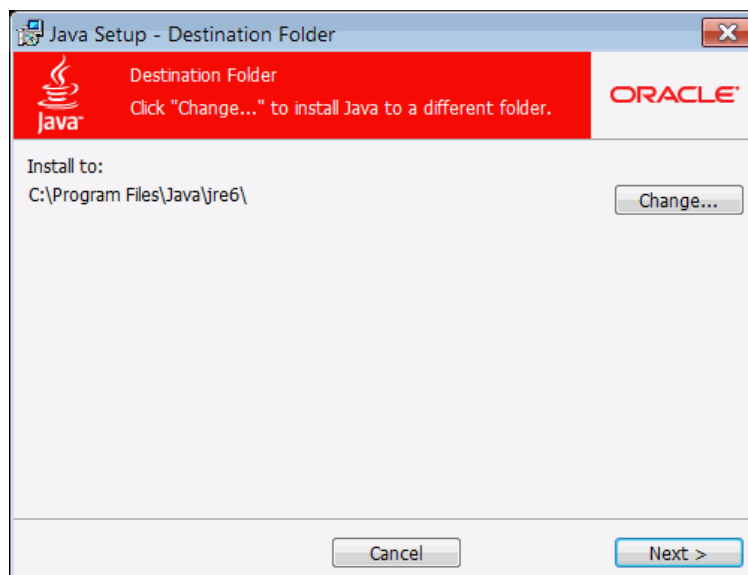


Ahora deberemos tener paciencia durante un rato, mientras se descomprime e instala todo:

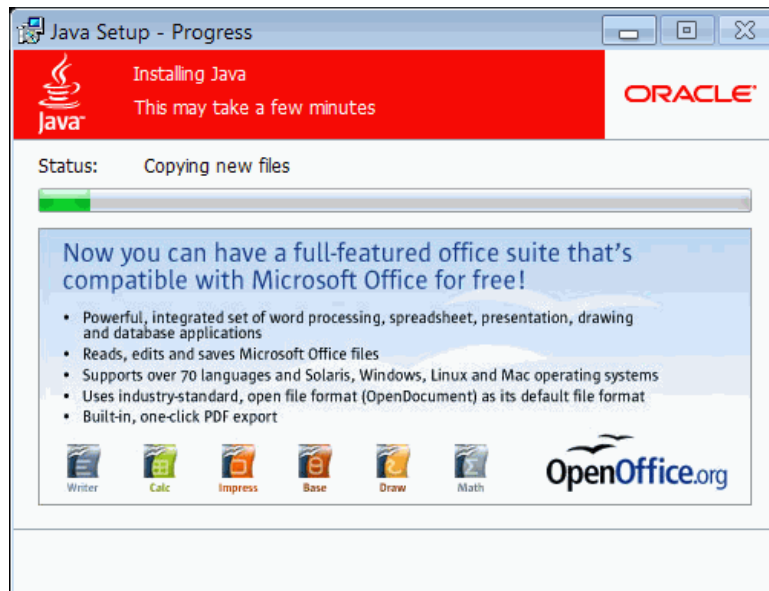


En cierto punto se nos preguntará si queremos instalar la máquina virtual Java (Java Runtime Environment, JRE). Lo razonable será responder que sí, para poder probar los programas que creemos.

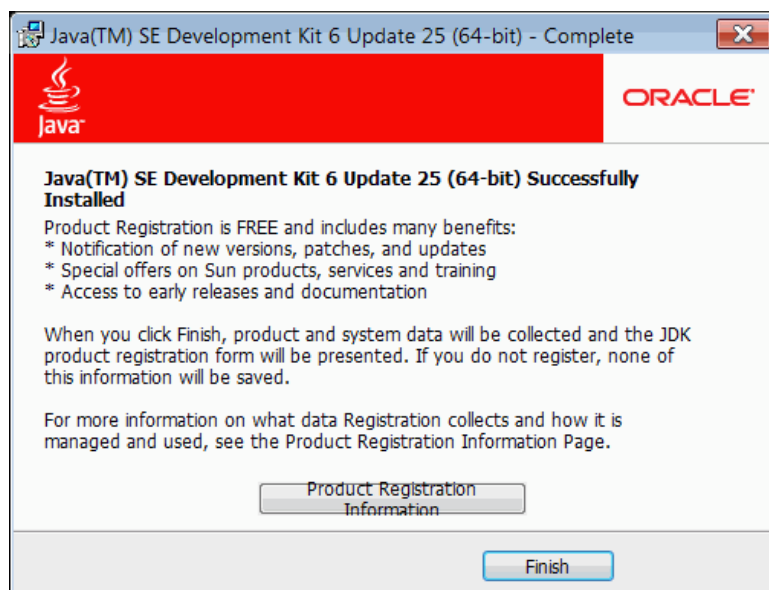
Igual que para el JDK, podríamos cambiar la carpeta de instalación del JRE:



Tendremos que esperar otro momento...



Y si todo ha ido bien, deberíamos obtener un mensaje de confirmación:



Y se nos propondrá registrar nuestra copia de Java en la página de Oracle (no es necesario):

Java Development Kit (JDK)

Registration

Please take the time to register your software.

ORACLE

You need an Oracle.com account to register your product. [Create an account now](#), or if you already have one continue by registering your product below.

Create An Account

Need an account?

[Create an Oracle.com account now.](#)

Use My Account

Please accept the terms of use below and click "Register Now" to register your product.

☐ I accept the terms of use for registering Oracle programs.

[View terms of use](#)

[Register Now](#)

Oracle Corporation respects your privacy. For more information on Oracle's Privacy Policy see <http://www.oracle.com/html/privacy.html> or contact privacy_ww@oracle.com.

ORACLE

Copyright 2010, Oracle Corporation and/or its affiliates

Con eso ya tenemos instalada la herramienta básica, el **compilador** que convertirá nuestros programas en Java a algo que pueda ser utilizado desde cualquier otro equipo que tenga una máquina virtual Java.

Pero el kit de desarrollo (JDK) **no incluye ningún editor** con el que crear nuestros programas. Podríamos instalar un "editor genérico", porque tenemos muchos gratuitos y de calidad, como Notepad++. Aun así, si nuestro equipo es razonablemente moderno, puede ser preferible instalar un entorno integrado, como **NetBeans**, que encontraremos en

<http://netbeans.org>

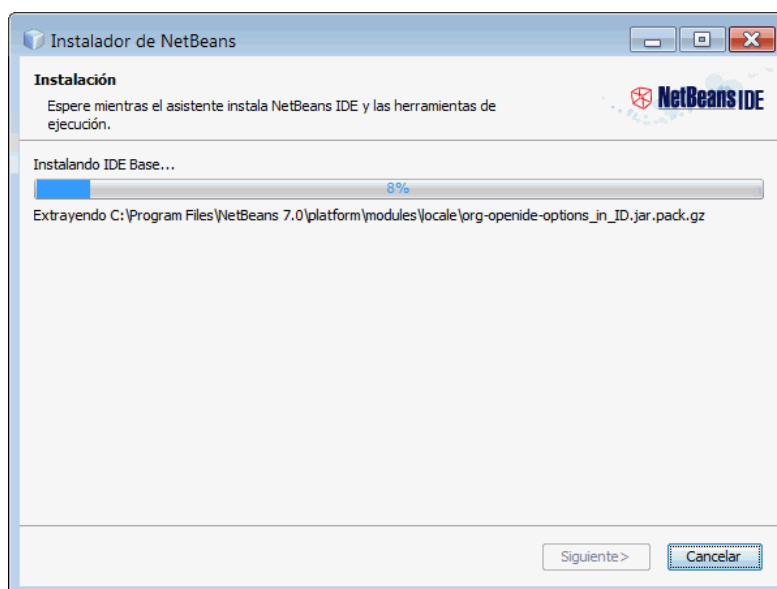
The screenshot shows the NetBeans IDE 7.0 website. At the top, there's a navigation bar with links: Home, IDE, Platform, Plugins, Docs & Support, Community, and Partners. The main content area is titled 'NetBeans IDE 7.0' and describes it as a tool for developing desktop, mobile, and web applications with Java, PHP, C/C++, and more. It mentions it runs on Windows, Linux, Mac OS X, and Solaris. There's a prominent 'Download FREE' button for NetBeans IDE 7.0 and a link to 'Learn More about NetBeans IDE'. To the right, there's a video overview of NetBeans IDE 7.0 with a 'Watch Now' button. Below the main content, there's a 'Featured News' section with three columns: 'Tutorials and Demos' (listing links like General Java Development, Java GUI Applications, etc.), 'Plugins' (listing links like PHP Twig, Show and change line endings, etc.), and 'NetBeans Community Poll' (showing a loading spinner).

Si hacemos clic en "Download", se nos llevará a la página de descargas, en la que tenemos varias versiones para elegir. Lo razonable "para un novato" es descargar la versión para **"Java SE"** (Standard Edition; las alternativas son otros lenguajes, como PHP o C++, versiones profesionales como Java EE -Enterprise Edition-, o una versión que engloba todas estas posibilidades).

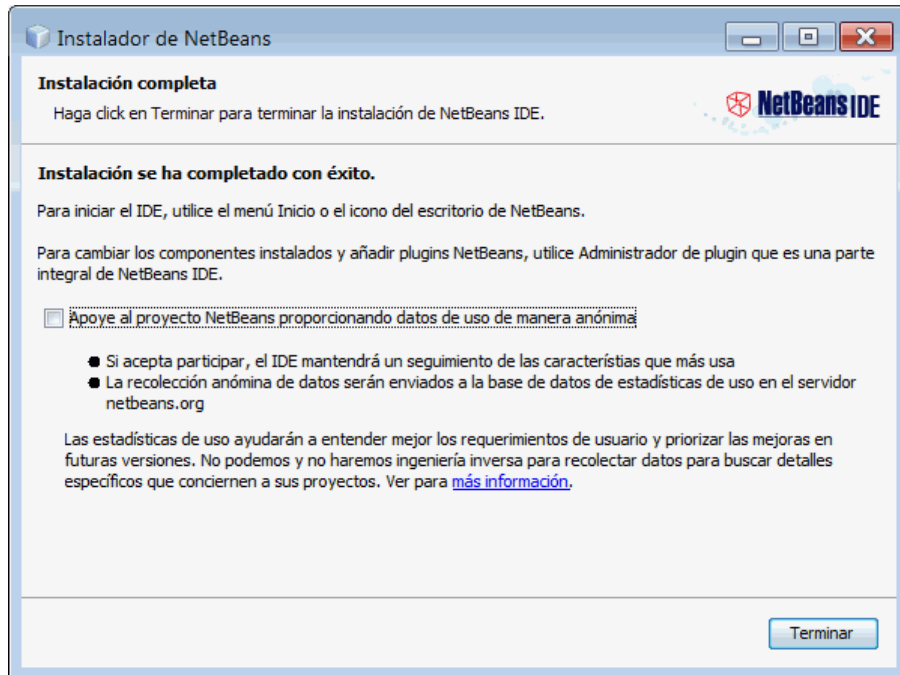
Es posible que también podamos escoger el Español como idioma, en vez del inglés (sólo en algunas versiones).



La instalación no se podrá completar si no hemos instalado Java antes, pero si lo hemos hecho, debería ser simple y razonablemente rápida:



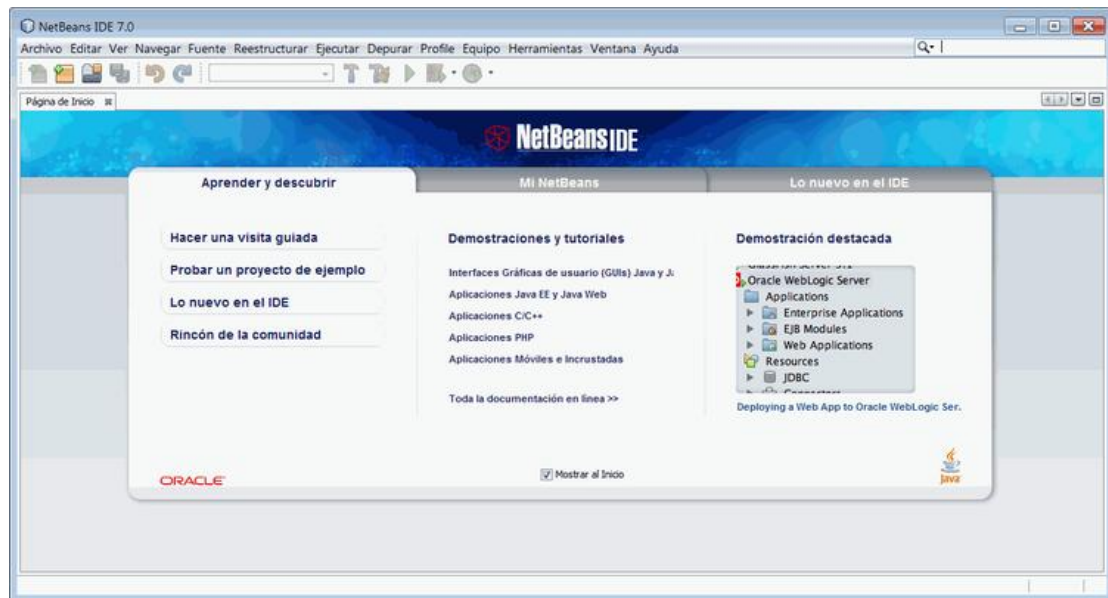
Y al final quizá se nos pregunte si queremos permitir que se recopile estadísticas sobre nuestro uso:



Todo listo. Tendremos un nuevo programa en nuestro menú de Inicio. Podemos hacer doble clic para comprobar que se ha instalado correctamente, y debería aparecer la pantalla de carga:



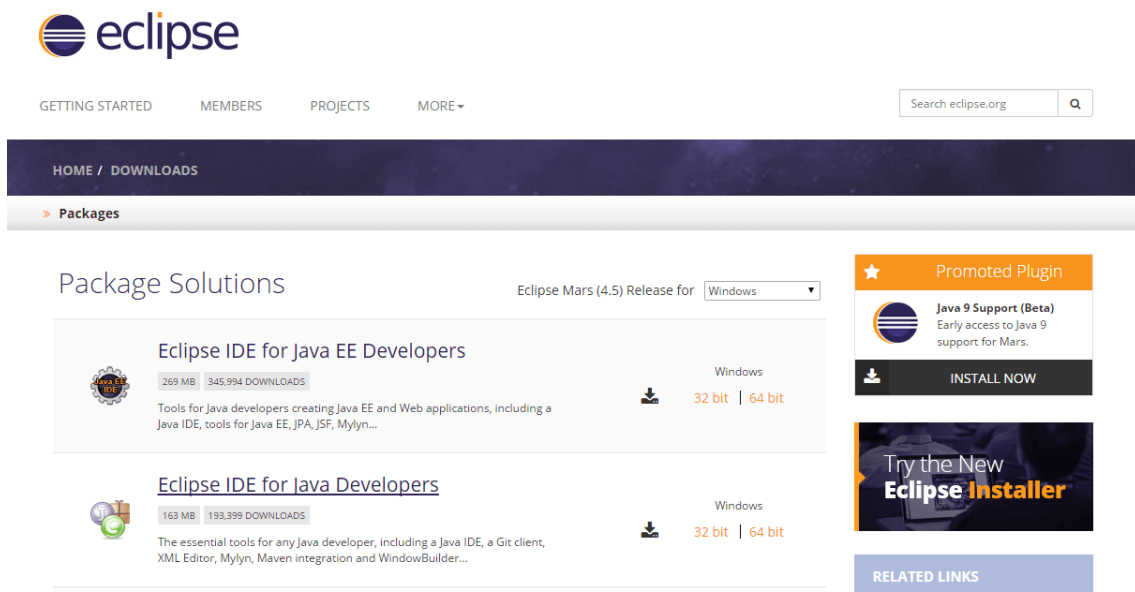
Y después de un instante, la pantalla "normal" de NetBeans:



Ya estaríamos listos para empezar a crear nuestro primer programa en Java, pero eso queda para la siguiente lección...

2.2. Instalación de Eclipse en Windows

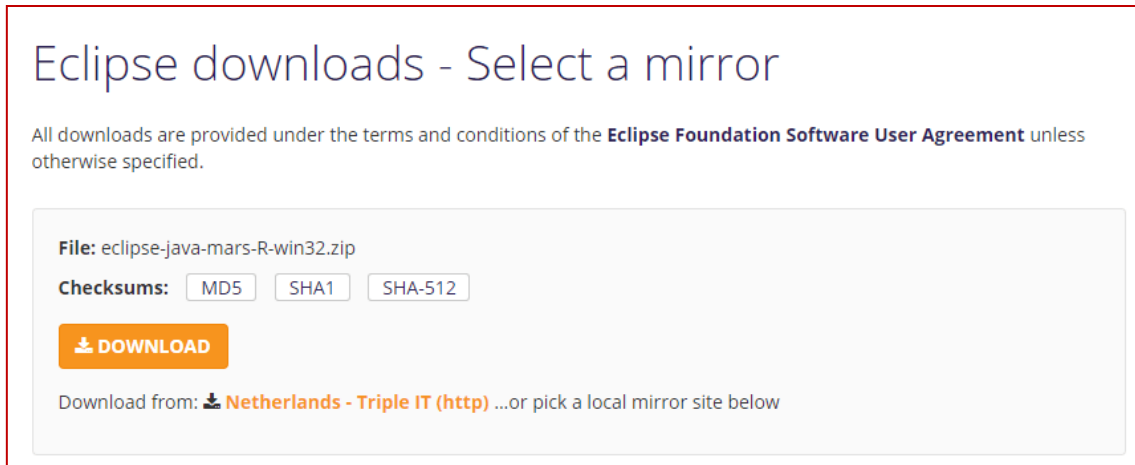
Eclipse es otro entorno de desarrollo para Java, alternativo a NetBeans, también muy extendido, aunque algo más pesado (ocupa más espacio y es más lento en funcionamiento). Se puede descargar desde eclipse.org:



Al igual que ocurre con NetBeans, existen versiones de Eclipse para crear programas en distintos lenguajes o incluso con diferentes versiones de Java. Lo razonable será usar "Eclipse for Java Developers" (no la versión de Java EE,

Enterprise Edition, que añade herramientas que un principiante no va a necesitar y que pueden complicar el entorno).

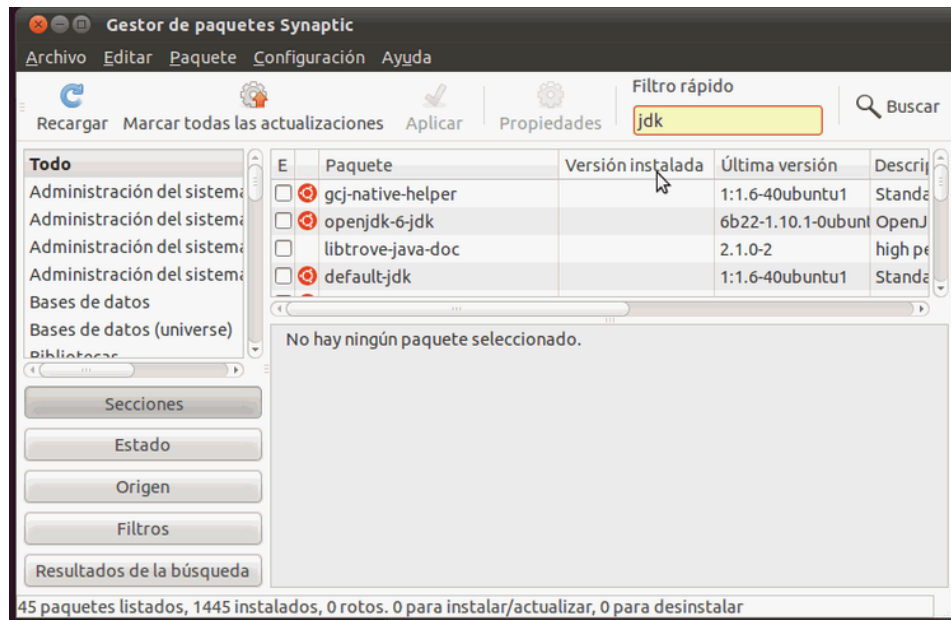
Es habitual que se nos proponga descargar un fichero ZIP, que deberemos descomprimir en la carpeta que nosotros escojamos y lanzar después con un doble clic (no se crearía una opción en el menú de Inicio).



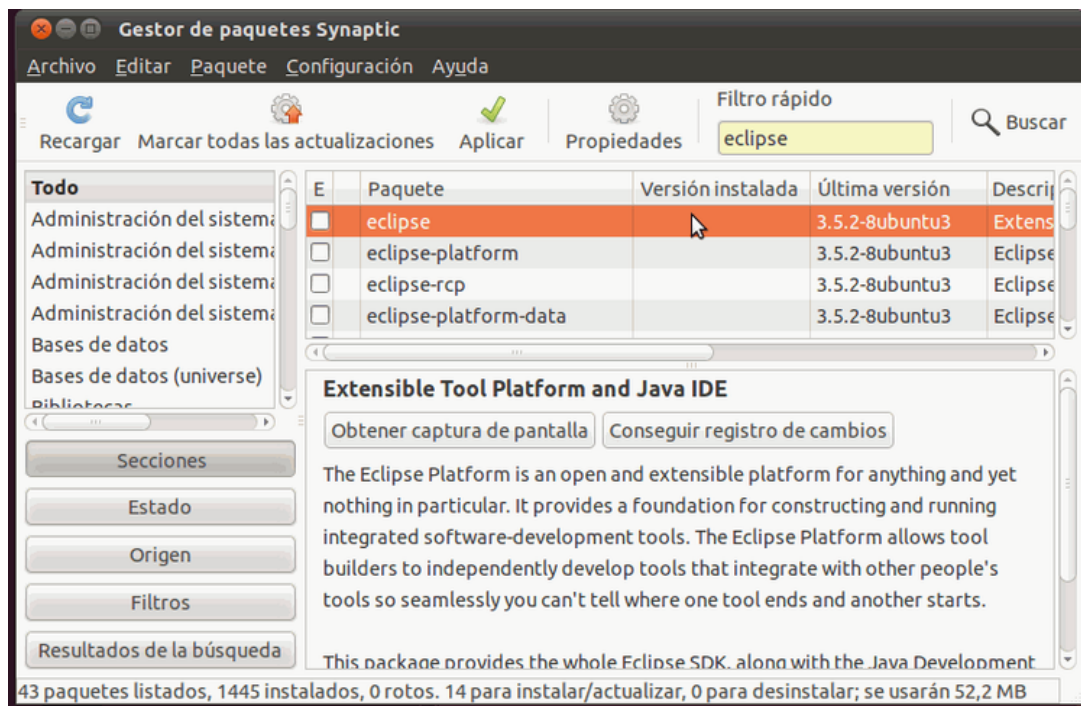
Estas mayores complicaciones en la instalación son otro punto en su contra y a favor de NetBeans. Aun así, si se ha instalado previamente el JDK y se ha reiniciado el equipo antes de intentar instalar Eclipse, es esperable que funcione todo a la primera.

2.3. Instalación del JDK y Eclipse bajo Linux

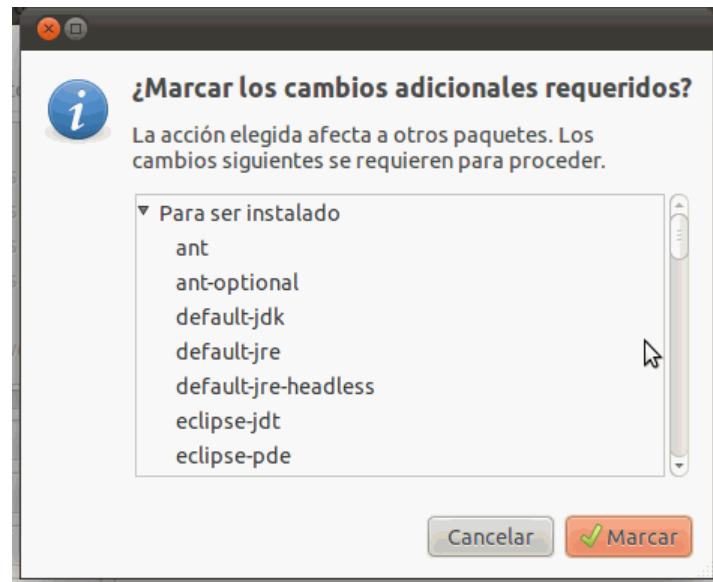
Instalar el JDK (Java Development Kit) en Linux puede ser aún más fácil que en Windows. En casi cualquier distribución Linux actual bastaría con entrar al "Gestor de paquetes Synaptic" (o la herramienta que use nuestro sistema para instalar nuevos programas), buscar "jdk" y hacer doble clic en el paquete llamado "openjdk-6-jdk" (o similar) para marcarlo:



De paso, podemos aprovechar para instalar Eclipse, que también se suele poder escoger desde el gestor de paquetes de la mayoría de distribuciones de Linux:



En ambos casos, se nos avisará de que esos paquetes dependen de otros que también habrá que instalar:



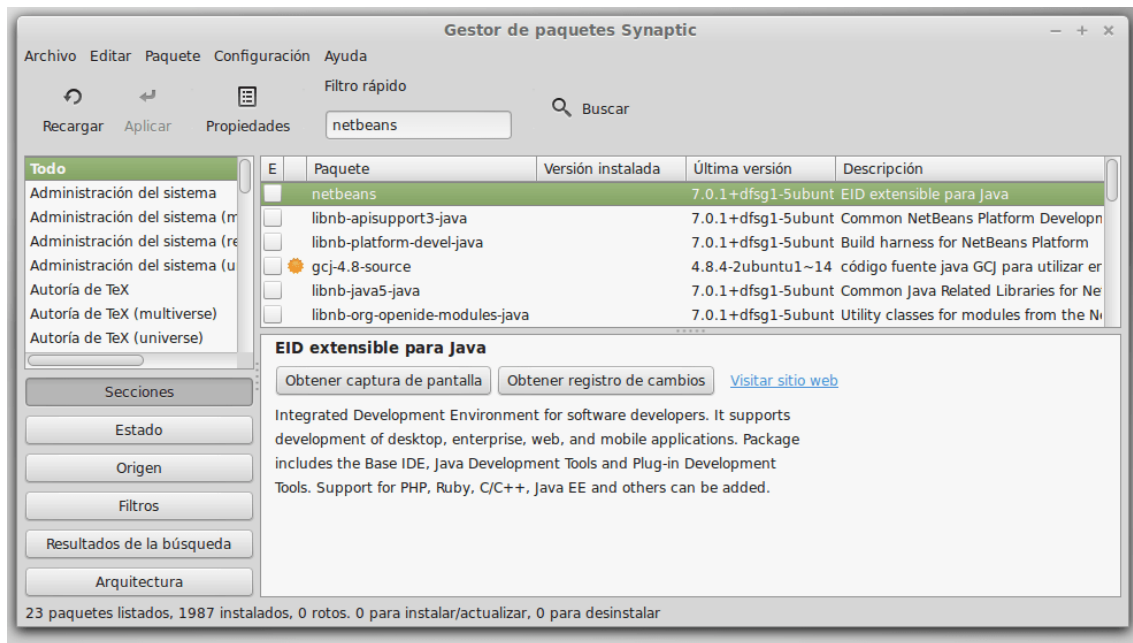
Cuando hagamos clic en el botón "Aplicar", se descargarán todos los paquetes (casi 200 Mb en total) y se instalarán (debería ocupar algo más de 300 Mb de disco). A partir de entonces, en nuestro menú, en el apartado de Programación (o Desarrollo), tendremos un acceso a Eclipse. Si entramos, deberíamos ver algo parecido a esto:



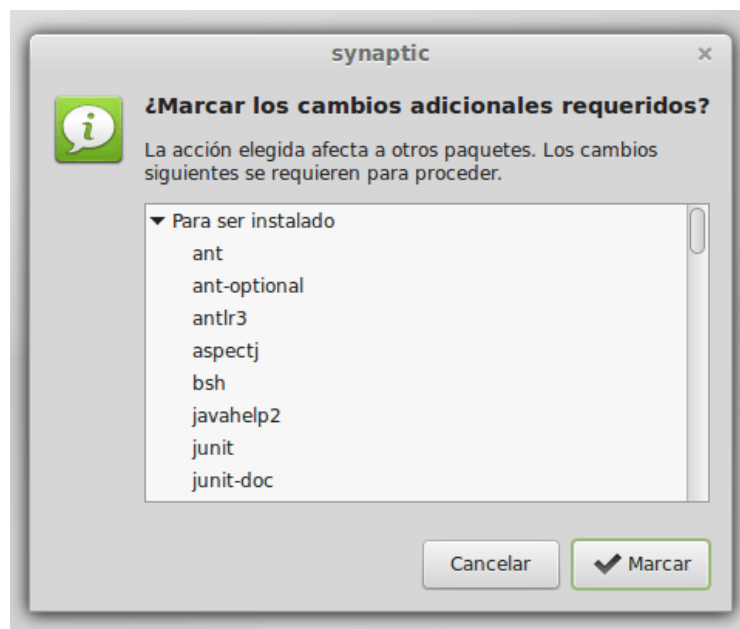
(Los detalles de cómo empezar a programar con Eclipse los veremos en el próximo tema)

2.4. Instalación de Netbeans bajo Linux

En la mayoría de distribuciones de Linux actuales, NetBeans también estará disponible desde Synaptic (o el gestor de paquetes que incluya esa distribución):



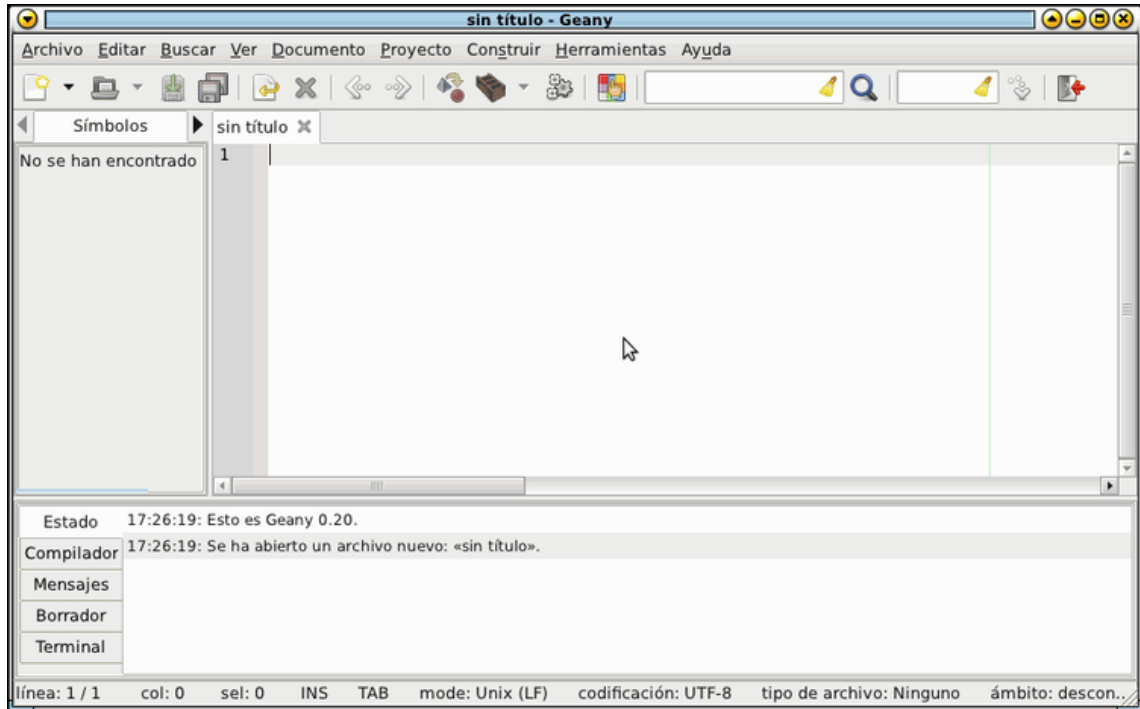
Y, al igual que hemos visto que ocurría con el JDK y con Eclipse, se nos avisará en caso de que sea necesario instalar algún paquete extra:



Si no pudiéramos instalarlo desde el gestor de paquetes, deberíamos descargar desde su página oficial, bien sea en su "versión para Linux" oficial, o bien a partir una versión comprimida en fichero ZIP capaz de funcionar en cualquier sistema operativo (para el que exista una versión de la máquina virtual Java, claro), que deberíamos descomprimir en una carpeta de nuestra elección.

2.5. La alternativa ligera: Geany

Podemos usar otras muchas herramientas, además de NetBeans y Eclipse. Si nuestro ordenador no está muy sobrado de recursos, disponemos de alternativas más simples, editores avanzados como Geany (que podríamos instalar en un par de clics usando Synaptic o el gestor de paquetes de cualquier distribución Linux). Su apariencia es ésta:



También se puede usar Geany en otros sistemas operativos, como Windows, pero en este caso, el manejo puede no ser tan sencillo: tendremos que instalar primero el JDK, luego Geany, y quizá aun así no podamos compilar y lanzar directamente los programas desde Geany, sino que obtengamos un mensaje de error que nos diga que no encuentra Java. Para solucionarlo, habría que cambiar la configuración de Geany o incluir Java en el "Path" de Windows, pero son tareas que quizá no estén al alcance de un principiante, así que las esquivaremos por ahora...

3. Nuestro primer programa

3.1. Un programa que escribe "Hola Mundo"

Comenzaremos por crear un pequeño programa en **modo texto**. Este primer programa se limitará a escribir el texto "Hola Mundo!" en la pantalla. En primer lugar, veremos cómo es este programa, luego comentaremos un poco (todavía con poco detalle) las órdenes que lo forman y finalmente veremos cómo probar ese programa con distintos entornos.

Nuestro primer programa será:

```
// HolaMundo.java
// Aplicación HolaMundo de ejemplo

class HolaMundo {
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
}
```

Dos detalles que hay que considerar antes de seguir adelante:

- Puede parecer complicado para ser un primer programa. Es cierto, lo es. Si Java es tu primer lenguaje de programación, tienes que asumirlo: Java es así, no pretende que lo simple sea simple, sino que lo complicado no sea terrible. Por eso, los grandes proyectos serán más fáciles de mantener y menos propensos a errores que con otros lenguajes más antiguos como BASIC o C, pero los programas básicos parecerán innecesariamente complejos.
- Dentro de poco veremos cómo teclearlo. Será importante respetar las mayúsculas y minúsculas exactamente como están en el ejemplo. El hecho de escribir "system" en vez de "System" hará que obtengamos un error de compilación.

Ejercicio propuesto 3.1.1: Crea un programa en Java que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

3.2. Entendiendo este primer programa

La única parte del programa que necesitamos comprender por ahora es la línea central. Por eso, vamos a analizarlo de dentro hacia fuera, dejando todavía algunos detalles en el aire.

3.2.1. Escribir en pantalla

La orden que realmente escribe en pantalla es: **System.out.print("Hola Mundo!");** La orden que se encarga de escribir es "**print**". Se trata de una orden de salida (**out**) de nuestro sistema (**System**), y deberemos escribirla siempre usando esas tres palabras, una tras otra y separadas por puntos: **System.out.print**

Lo que deseemos escribir se indicará entre paréntesis. Si se trata de un texto que deba aparecer tal cual, irá además encerrado entre **comillas**.

También es importante el **punto y coma** que aparece al final de esa línea: cada orden en Java deberá terminar con punto y coma (nuestro programa ocupa varias líneas pero sólo tiene una orden, que es "print").

Por eso, la forma de escribir un texto será:

```
System.out.print( "Hola Mundo!" );
```

3.2.2. Formato libre

Java es un lenguaje "de formato libre". Antes de cada orden, podemos dejar tantos espacios o tantas líneas en blanco como nos apetezca. Por eso, es habitual escribir un poco más a la derecha cada vez que empecemos un nuevo bloque (habitualmente cuatro espacios), para que el programa resulte más legible.

Ese es el motivo de que nuestra orden "print" no estuviera pegada al margen izquierdo, sino más a la derecha (ocho espacios, porque está dentro de dos bloques).

```
System.out.print( "Hola Mundo!" );
```

3.2.3. El cuerpo del programa

En todos los programas creados usando Java, debe existir un bloque llamado "**main**", que representa el cuerpo del programa. Por motivos que veremos más adelante, este bloque siempre comenzará con las palabras "**public static void**" y terminará con "**(String args[])**":

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

El contenido de cada bloque del programa se debe detallar entre llaves. Por eso, la línea "print" aparece después de "main" y rodeada por llaves:

```
public static void main( String args[] ) {  
    System.out.print( "Hola Mundo!" );  
}
```

Estas llaves de comienzo y de final de un bloque se podrían escribir en cualquier punto del programa antes del contenido del bloque y después de su contenido, ya que, como hemos dicho, Java es un lenguaje de formato libre. Aun así, por convenio, es habitual colocar la llave de apertura **al final** de la orden que abre el bloque, y la llave de cierre **justo debajo** de la palabra que abre el bloque, como se ve en el ejemplo anterior.

3.2.4. El nombre del programa

Cada programa Java debe "tener un nombre". Este **nombre** puede ser una única palabra o varias palabras unidas, e incluso contener cifras numéricas, pero debe empezar por una letra y no debe tener espacios intermedios en blanco. El nombre se debe indicar tras la palabra "**class**" (más adelante veremos a qué se debe el uso de esa palabra) y a continuación se abrirá el bloque que delimitará todo el programa, con llaves:

```
class HolaMundo {  
    public static void main( String args[] ) {  
        System.out.print( "Hola Mundo!" );  
    }  
}
```

El programa se debe guardar en un **fichero**, cuyo nombre coincida exactamente con lo que hemos escrito tras la palabra "class", incluso con las mismas mayúsculas y minúsculas. Este nombre de fichero terminará con ".java". Así, para nuestro primer programa de ejemplo, el fichero debería llamarse "HolaMundo.java".

3.2.5. Comentarios

Las dos primeras líneas del programa son:

```
// HolaMundo.java  
// Aplicación HolaMundo de ejemplo
```

Estas líneas, que comienzan con una doble barra inclinada (//) son **comentarios**. Nos sirven a nosotros de aclaración, pero nuestro ordenador las ignora, como si no hubiésemos escrito nada.

Si queremos que un comentario ocupe varias líneas, o sólo un trozo de una línea, en vez de llegar hasta el final de la línea, podemos preceder cada línea con una doble barra, como en el ejemplo anterior, o bien indicar dónde queremos empezar con `/*` (una barra seguida de un asterisco) y dónde queremos terminar con `*/` (un asterisco seguido de una barra), así:

```
/* Esta es la
Aplicación HolaMundo
de ejemplo */
```

3.2.6. Varias órdenes

Si queremos "hacer varias cosas", podremos escribir varias órdenes dentro de "main". Por ejemplo, podríamos escribir primero la palabra "Hola", luego un espacio y luego "Mundo!" usando tres órdenes "print" distintas:

```
// HolaMundo2.java
// Segunda aplicación HolaMundo de ejemplo

class HolaMundo2 {
    public static void main( String args[] ) {
        System.out.print( "Hola" );
        System.out.print( " " );
        System.out.print( "Mundo!" );

    }
}
```

Como puedes imaginar, este programa deberá estar guardado en un fichero llamado "HolaMundo2.java". Como también podrás intuir, el formato libre se refiere a las órdenes del lenguaje Java, no a los mensajes en pantalla: si escribes varios espacios con una orden "print", todos ellos serán visibles en pantalla.

3.2.7. Escribir y avanzar de línea

En ocasiones no queremos escribir todo en la misma línea. Para avanzar de línea tras escribir algo, basta con usar "println" en vez de "print":

```
// HolaMundo3.java
// Tercera aplicación HolaMundo de ejemplo

class HolaMundo3 {
    public static void main( String args[] ) {
        System.out.println( "Hola..." );
        System.out.println( "Mundo!" );
    }
}
```

```
}  
}
```

Ejercicio propuesto 3.2.1: Crea un programa en Java que te salude en pantalla por tu nombre. En la primera línea de pantalla aparecerá la palabra "Hola" y será en la segunda línea de pantalla donde aparezca tu nombre. Usa dos órdenes "println" distintas.

3.3. Compilar y lanzar el programa desde línea de comandos

Si quieres aprender a programar, deberías no tener miedo a la "línea de comandos", esa pantalla que típicamente tiene fondo negro y letras blancas, y en la que se teclean las órdenes en vez de hacer clic. En Linux normalmente se le llama "Terminal" y en Windows puede aparecer como "Símbolo del sistema".

Si has instalado el JDK pero ningún entorno de desarrollo, podrías abrir esa línea de comandos, lanzar desde ella un editor, después compilar el programa y finalmente, si todo ha ido bien, lanzar el programa ejecutable resultante.

En un Linux, típicamente el primer paso será lanzar un editor (como "gedit" en la versiones basadas en Gnome) y decirle el nombre que queremos que tenga el fichero que se va a crear:

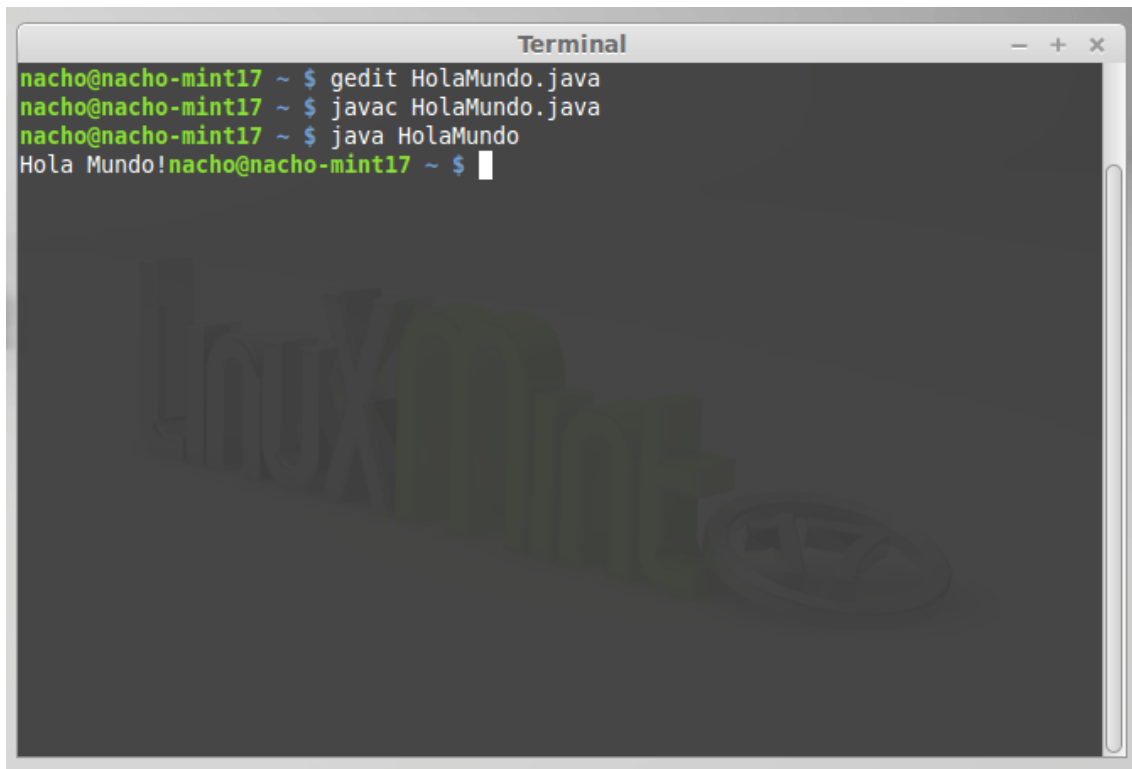
```
gedit HolaMundo.java
```

El segundo paso será lanzar el compilador, llamado "javac", indicándole el nombre del programa que queremos analizar:

```
javac HolaMundo.java
```

Si nuestro programa contiene algún error, se nos mostrará un mensaje que dirá la línea y columna en la que se ha detectado. Si no es así, obtendremos un fichero ejecutable llamado "HolaMundo.class", que podremos poner en funcionamiento tecleando:

```
java HolaMundo
```

A terminal window titled "Terminal" with standard window controls. It shows a series of commands and their output in a Linux environment. The prompt is "nacho@nacho-mint17 ~ \$". The commands are: "gedit HolaMundo.java", "javac HolaMundo.java", and "java HolaMundo". The output of the last command is "Hola Mundo!".

```
nacho@nacho-mint17 ~ $ gedit HolaMundo.java
nacho@nacho-mint17 ~ $ javac HolaMundo.java
nacho@nacho-mint17 ~ $ java HolaMundo
Hola Mundo!nacho@nacho-mint17 ~ $
```

En Windows, si todo va bien (cosa que no siempre ocurre), los pasos serían los mismos, pero el editor que viene preinstalado es el bloc de notas (notepad), luego comenzaríamos por teclear:

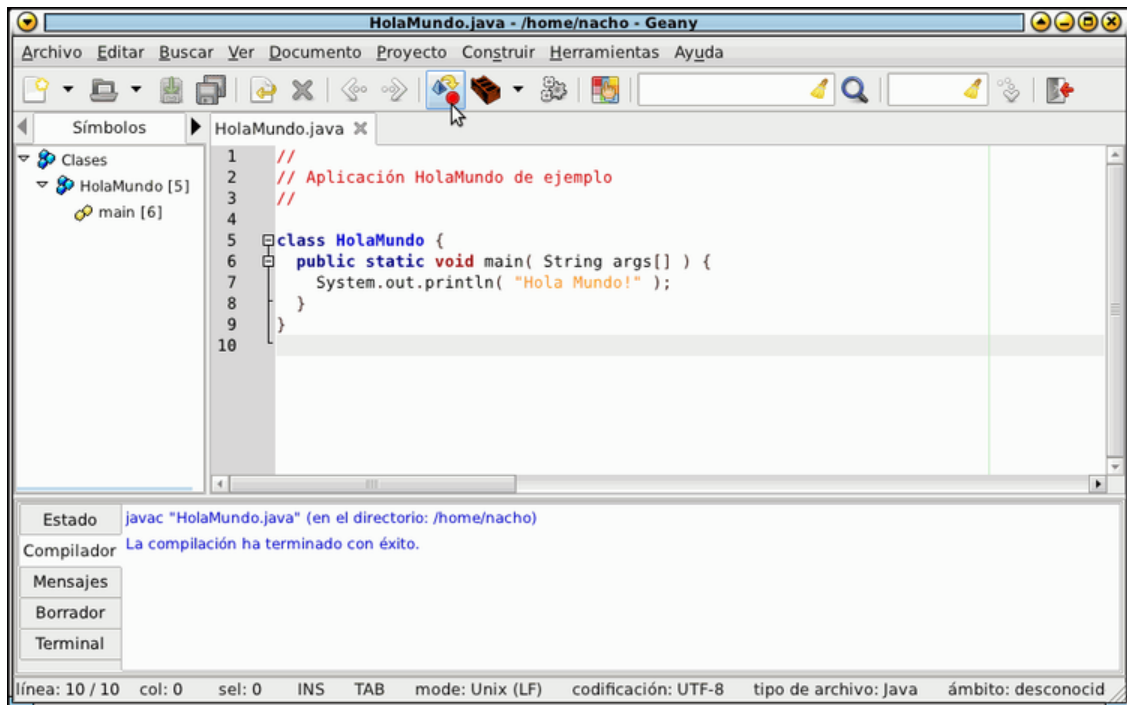
```
notepad HolaMundo.java
```

Ejercicio propuesto 3.3.1: Prueba a compilar desde "línea de comandos" el programa que te saludaba en pantalla por tu nombre (si no lo consigues, dentro de poco podrás hacerlo usando NetBeans).

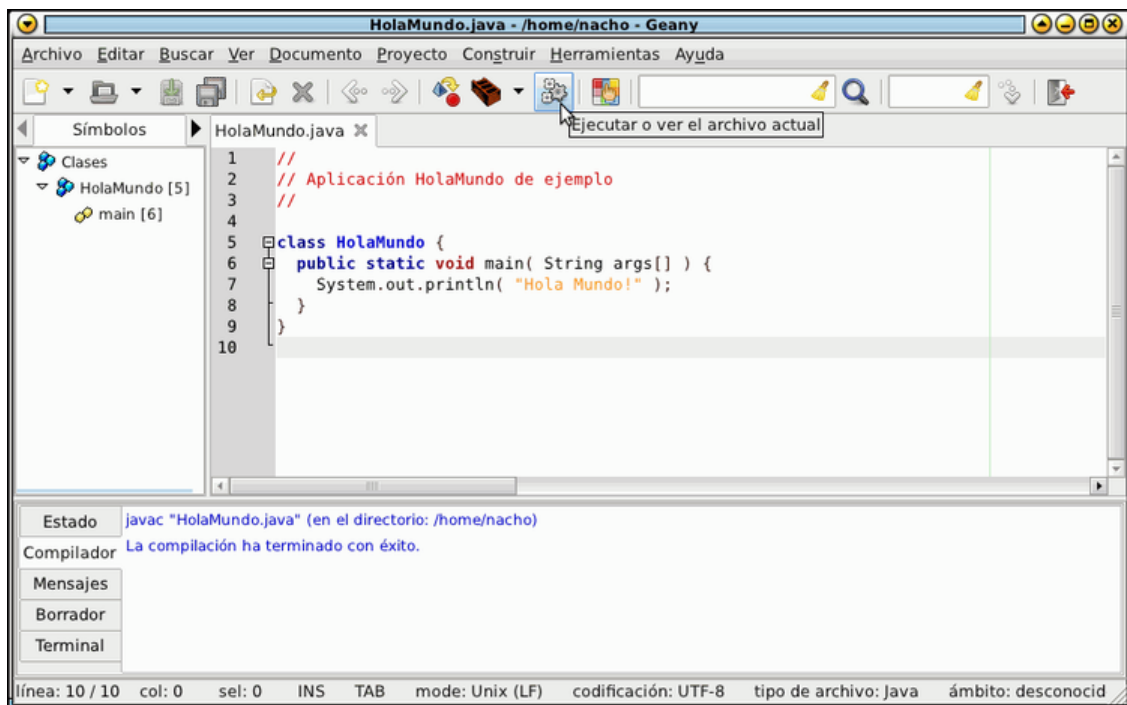
3.4. La alternativa básica pero cómoda: Geany

Geany es uno de los entornos más sencillos que permite teclear nuestros programas y también compilarlos y probarlos sin abandonar el editor.

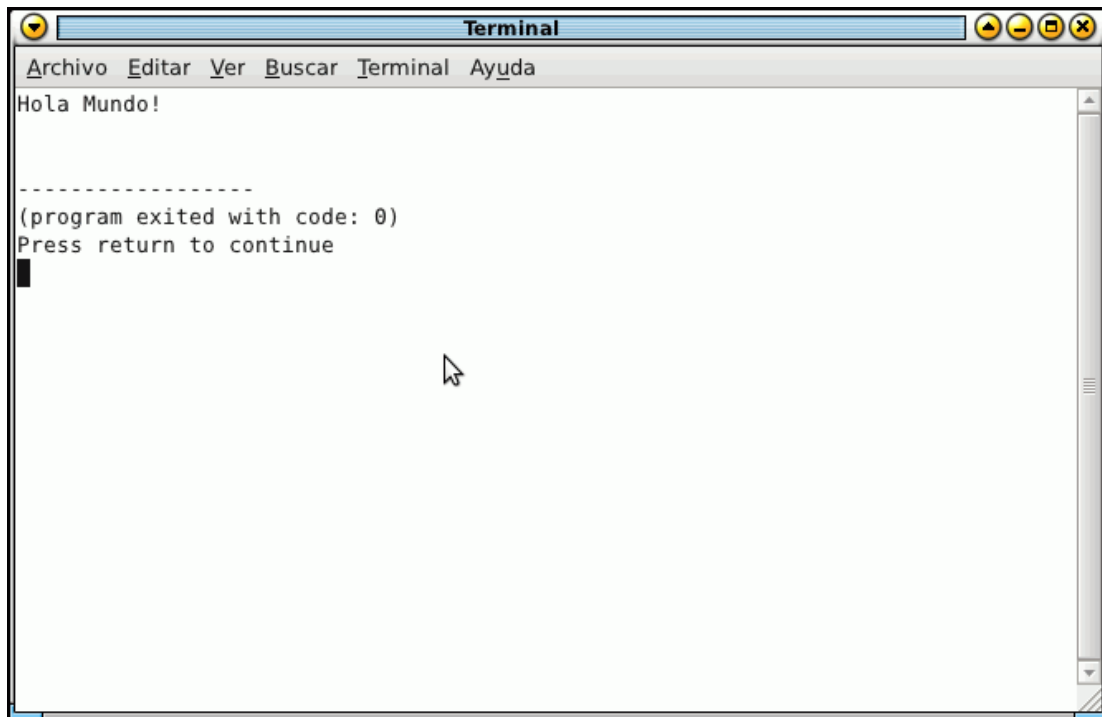
Basta con escribir nuestro fuente, y guardarlo en un fichero llamado "HolaMundo.java" (en general, el mismo nombre que la clase, respetando mayúsculas y minúsculas). Cuando pulsemos el botón de "compilar" el fuente, en la parte inferior de la ventana se nos mostrará si todo ha ido bien:



Si no ha habido errores, a continuación podremos pulsar el botón de Ejecutar para poner nuestro programa en marcha:



Y aparecerá una ventana de terminal, que mostrará el resultado:



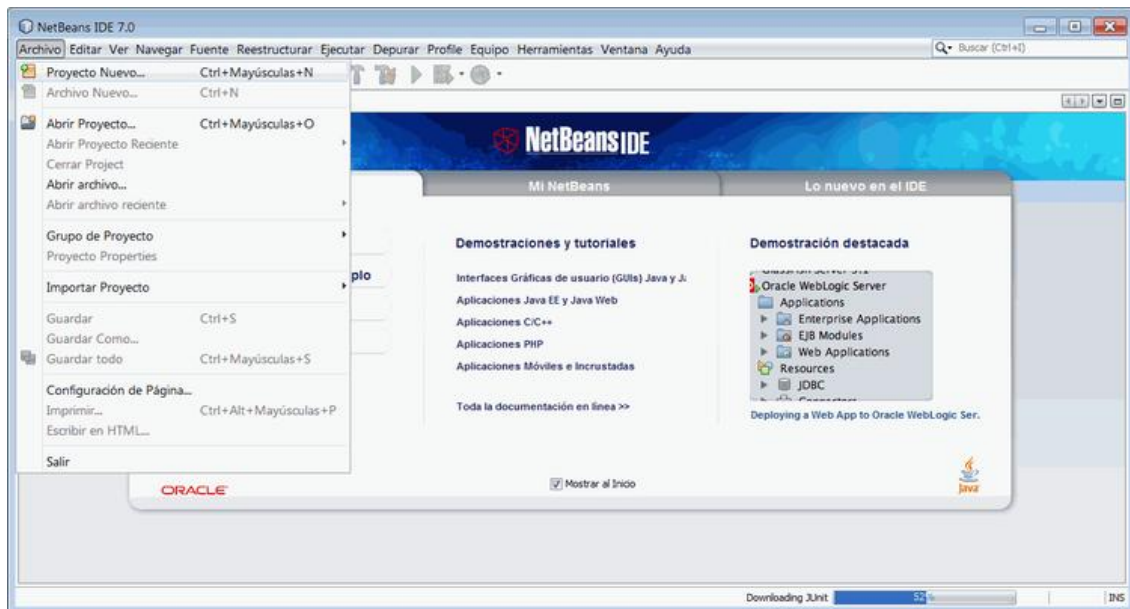
En caso de existir algún error de compilación, al volver al editor se nos mostrarían subrayadas en color rojo las líneas incorrectas.

Ejercicio propuesto 3.4.1: Crea un programa en Java, usando Geany, que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

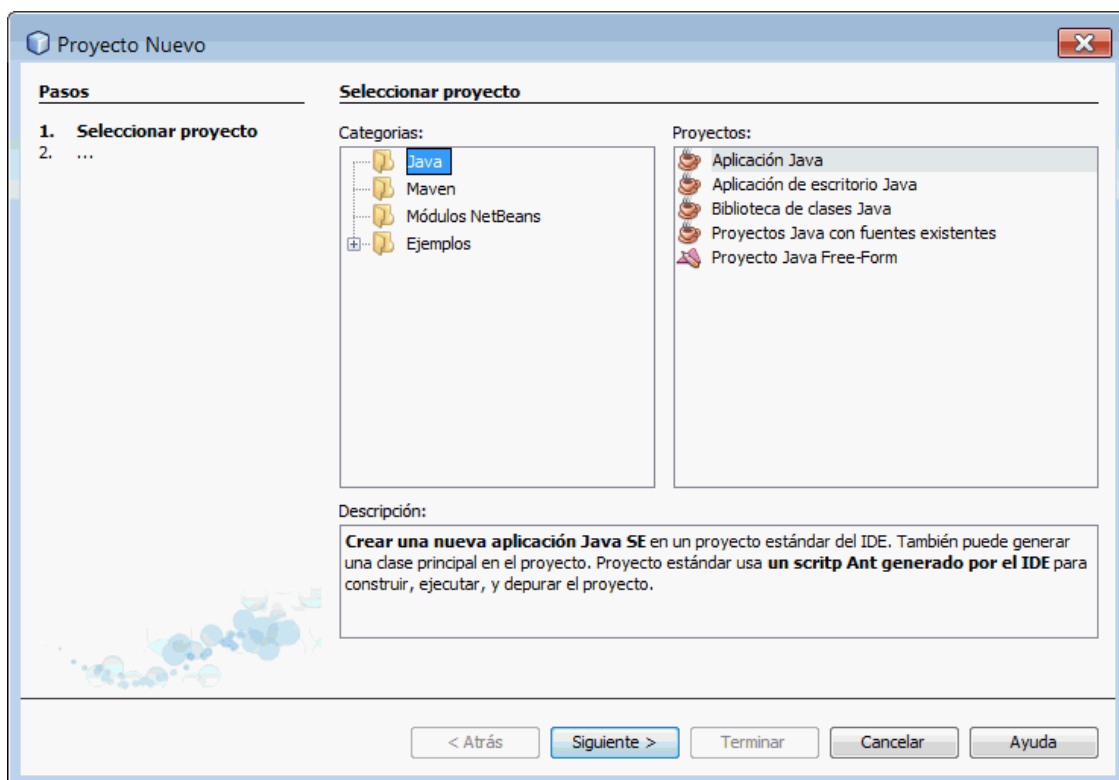
3.5. Manejo básico de NetBeans

Vamos a ver qué pasos dar en NetBeans para crear un programa como ese.

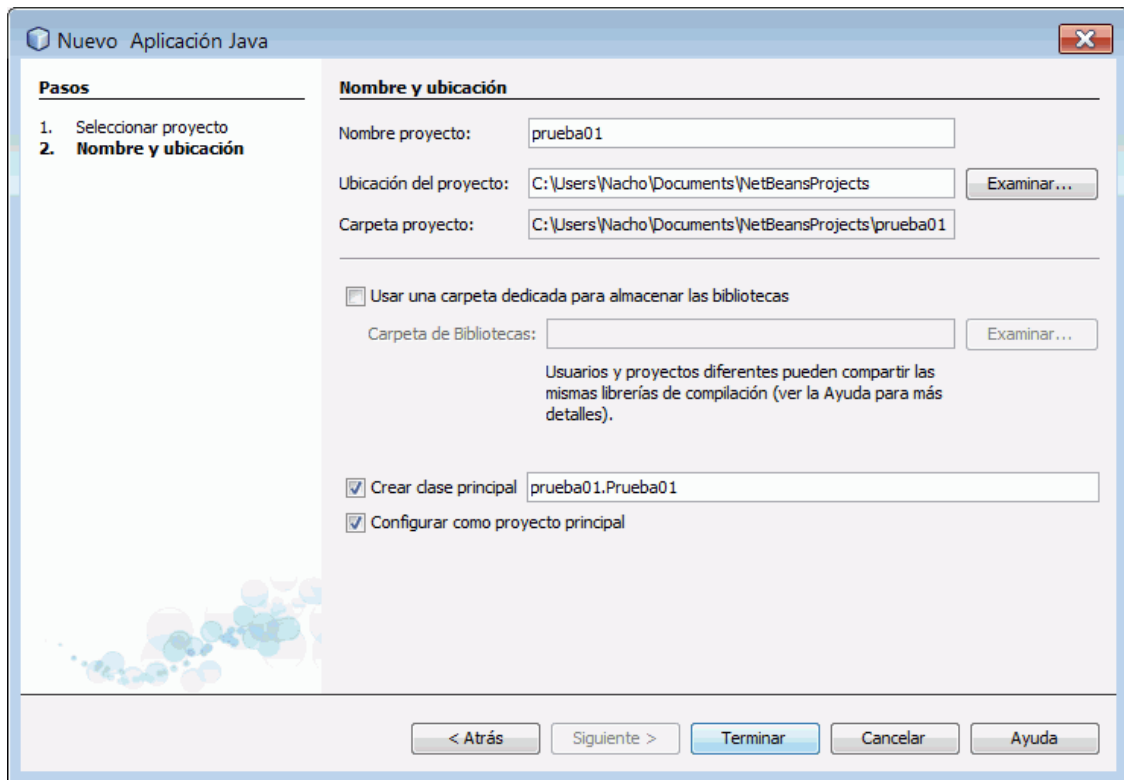
En primer lugar, deberemos entrar al menú "Archivo" y escoger la opción "Proyecto Nuevo":



Se nos preguntará el tipo de proyecto. Se tratará de una "Aplicación Java".

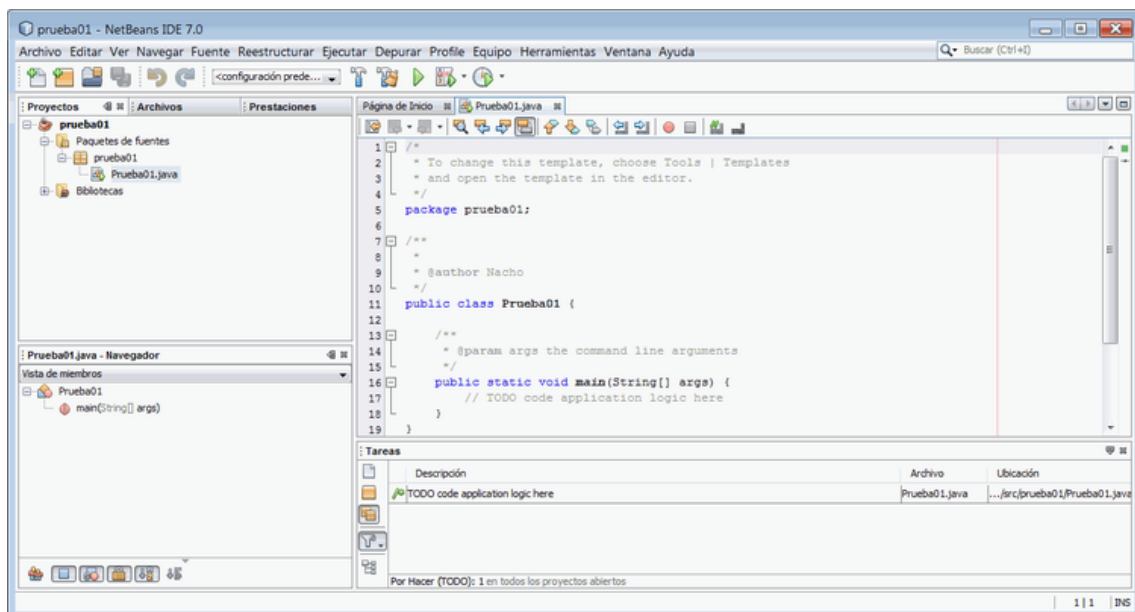


Deberemos indicar un nombre para el proyecto. Tenemos también la posibilidad de cambiar la carpeta en que se guardará.



Y entonces aparecerá un esqueleto de programa que recuerda al que nosotros queremos conseguir... salvo por un par de detalles:

- Falta la orden "System.out.print"
- Sobra una orden "package"



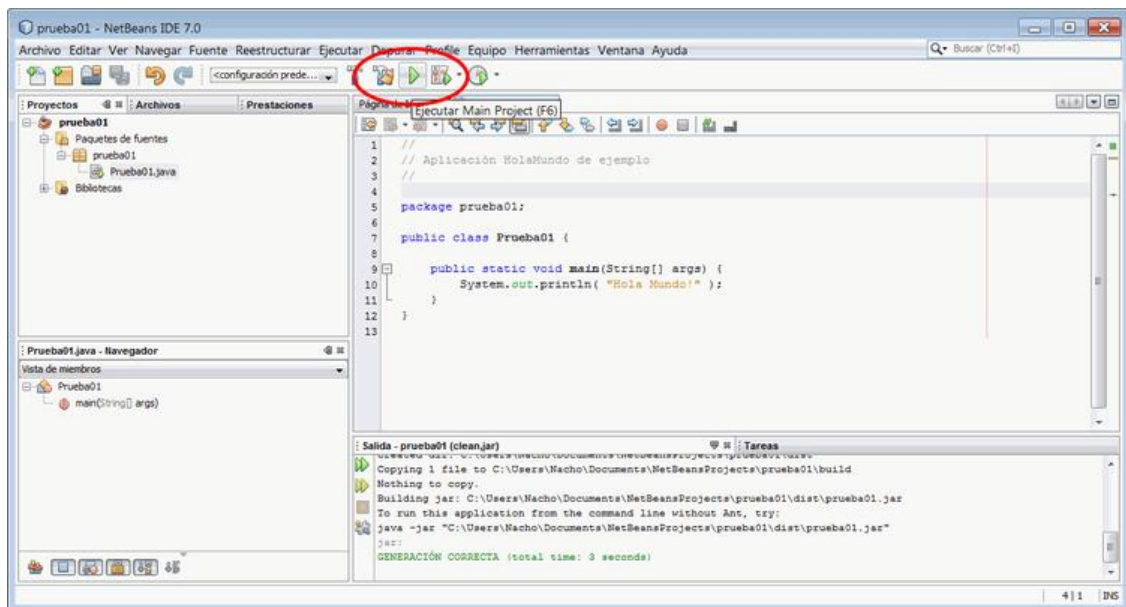
La orden "package" deberemos conservarla si usamos NetBeans, para indicar que nuestro programa es parte de un proyecto. La orden "print" deberemos añadirla, o

de lo contrario nuestro programa no escribirá nada en pantalla. Podríamos borrar los comentarios adicionales, hasta llegar a algo como esto:

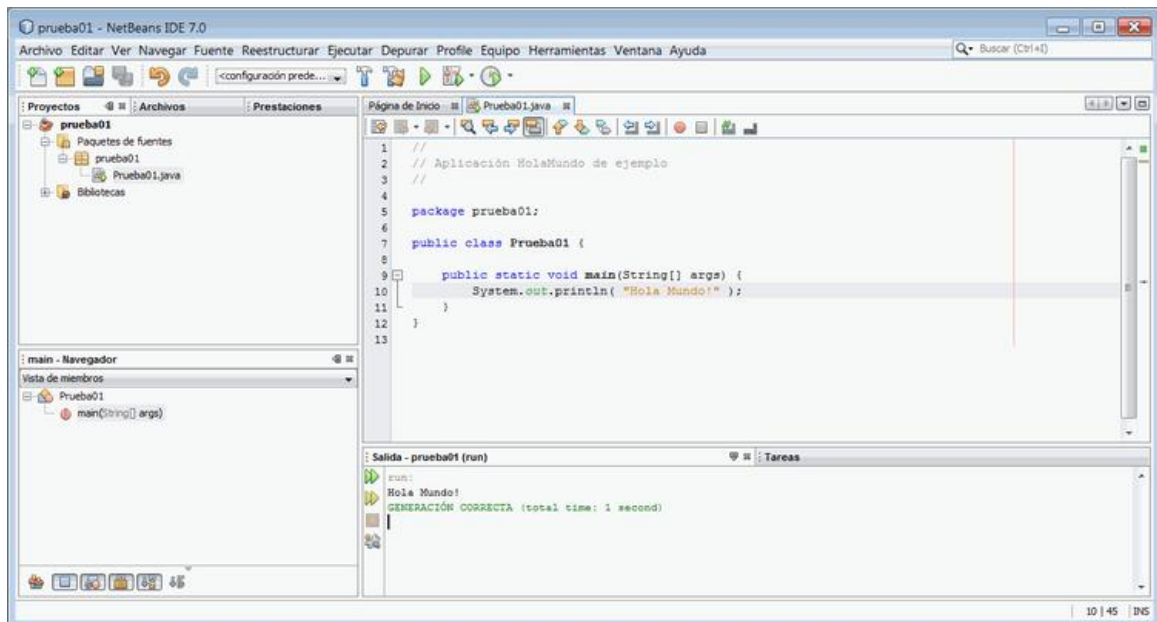
```
// HolaMundoNetBeans.java
// Aplicación HolaMundo de ejemplo, para compilar con NetBeans

package prueba01;

public class HolaMundoNetBeans {
    public static void main( String args[] ) {
        System.out.print( "Hola Mundo!" );
    }
}
```



Si hacemos clic en el botón de Ejecutar (el que muestra una punta de flecha de color verde), nuestro programa se pondrá en marcha (si no tiene ningún error), y su resultado se mostrará en la parte inferior derecha de la pantalla de trabajo de NetBeans:

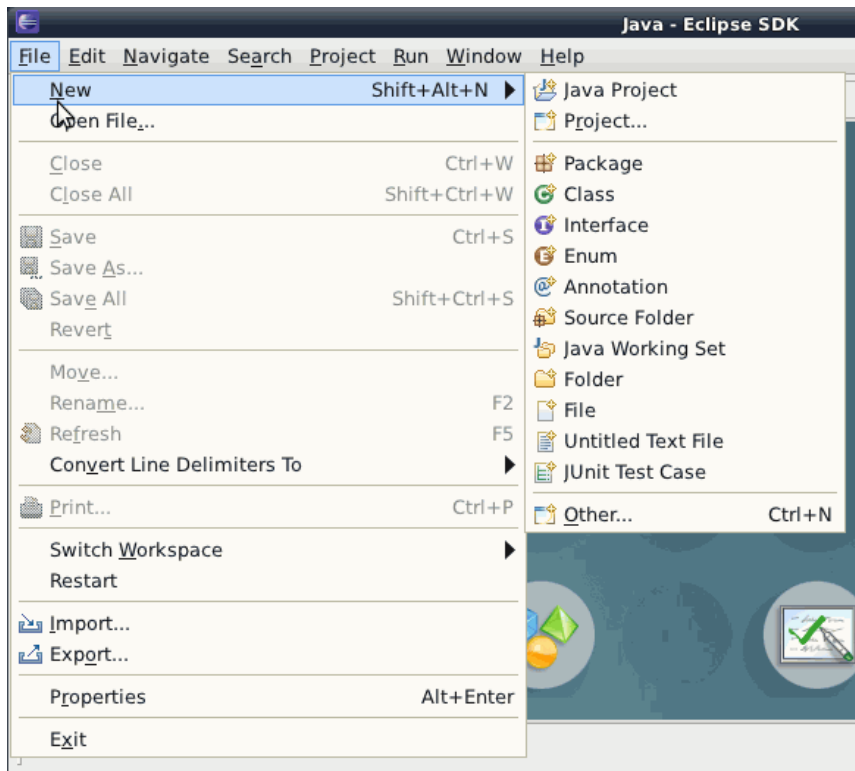


Ejercicio propuesto 3.5.1: Crea un programa en Java, usando NetBeans, que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

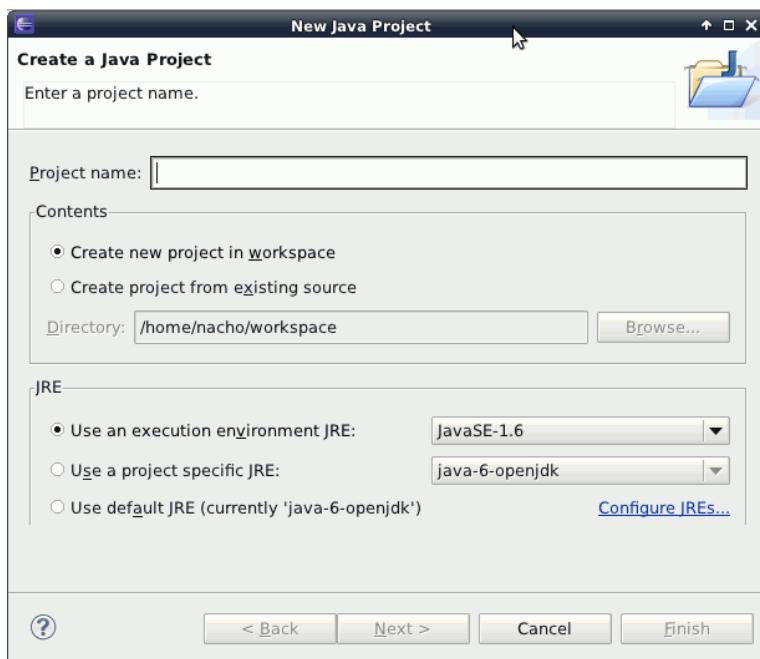
3.6. Contacto con Eclipse

Personalmente, NetBeans me parece un entorno más sencillo, más amigable y más rápido que Eclipse. Aun así, por si no has conseguido hacer funcionar tu programa con NetBeans o por si quieres conocer también Eclipse, vamos a ver los pasos básicos para probar un programa como el nuestro.

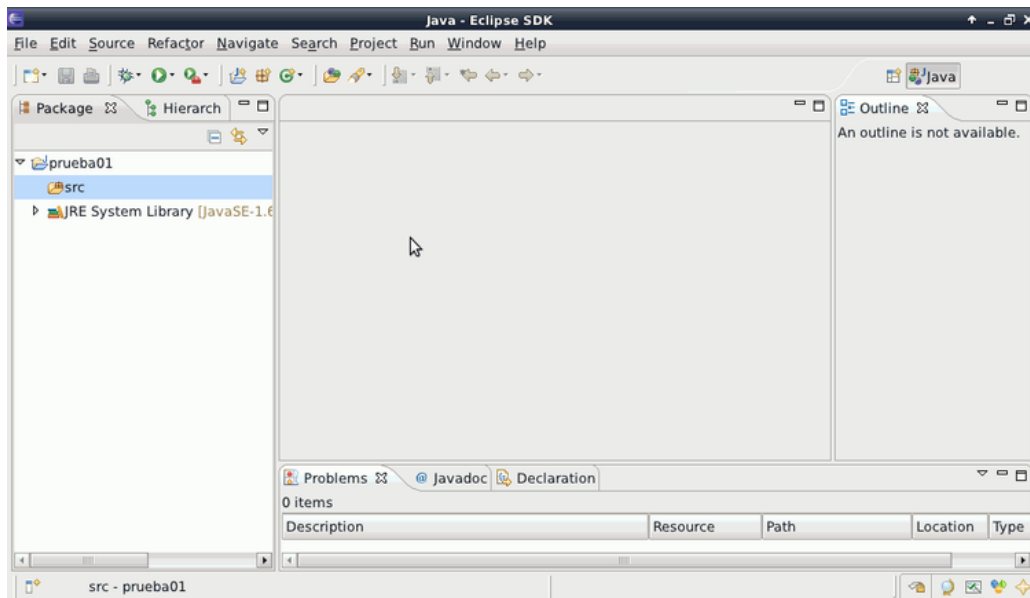
Para crear nuestro proyecto con Eclipse, deberemos entrar al menú "File" (Archivo), y en la opción "New" (Nuevo) tendremos "Java Project":



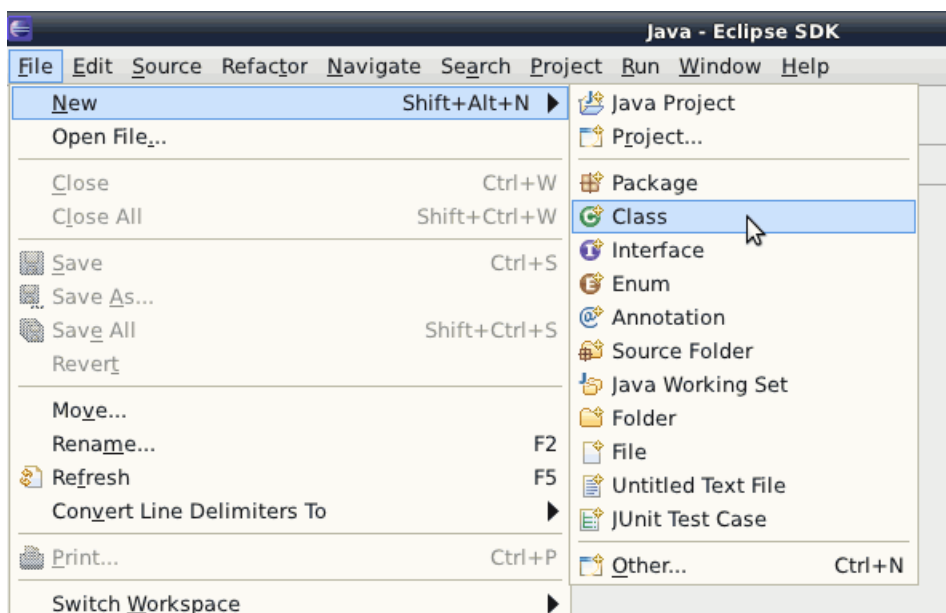
Se nos pedirá un nombre para el proyecto, y que confirmemos si se trata de un nuevo proyecto (como es nuestro caso) o si queremos partir de otros fuentes que ya existen (no por ahora):



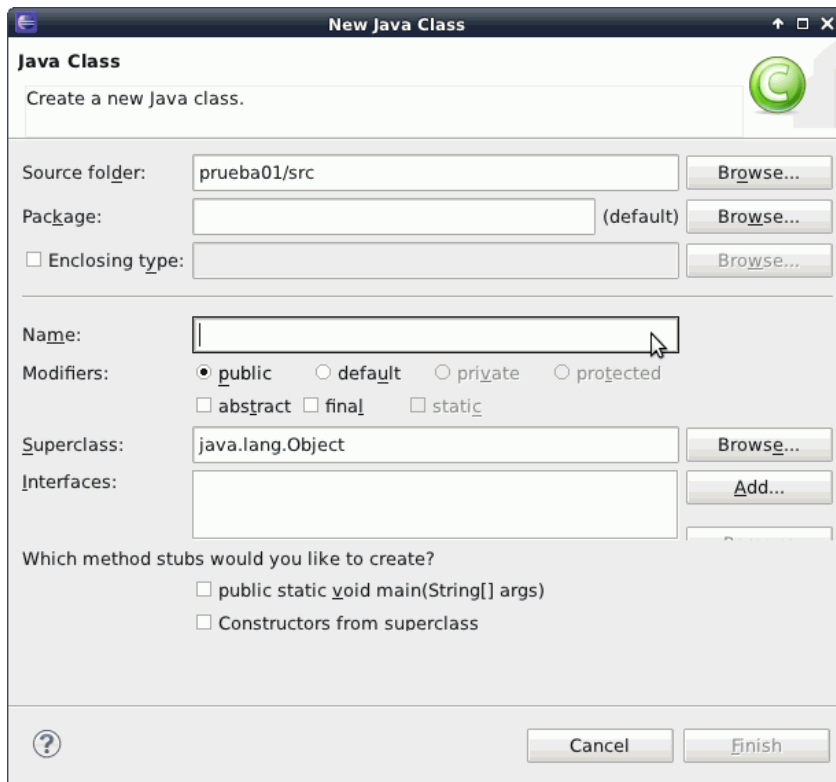
Aparecerá nuestro proyecto vacío:



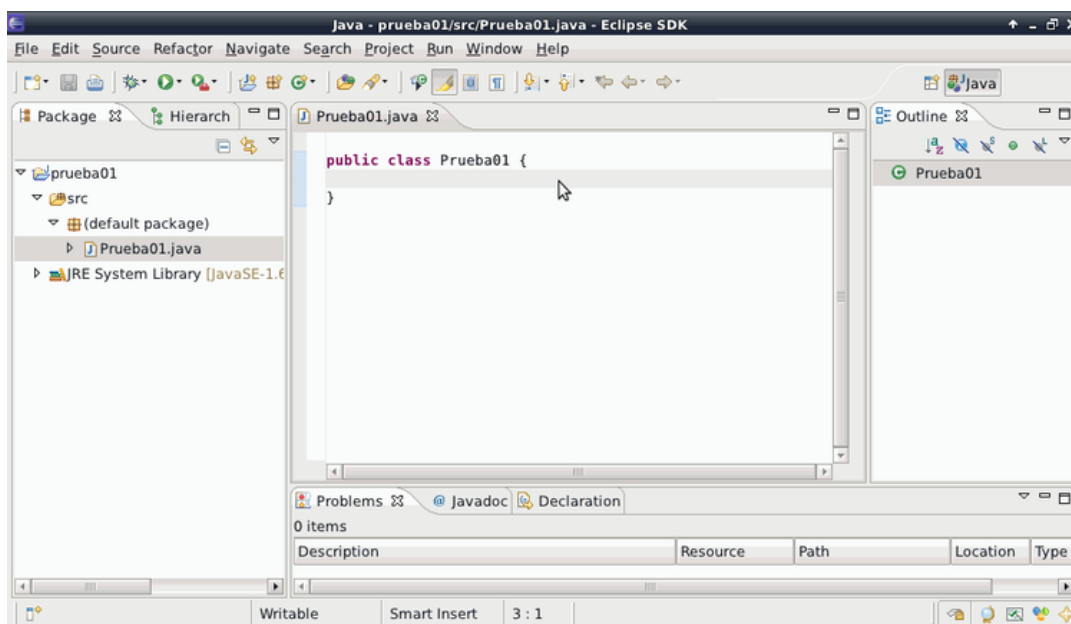
Y ahora deberemos añadir una clase a nuestro proyecto (File / New / Class):



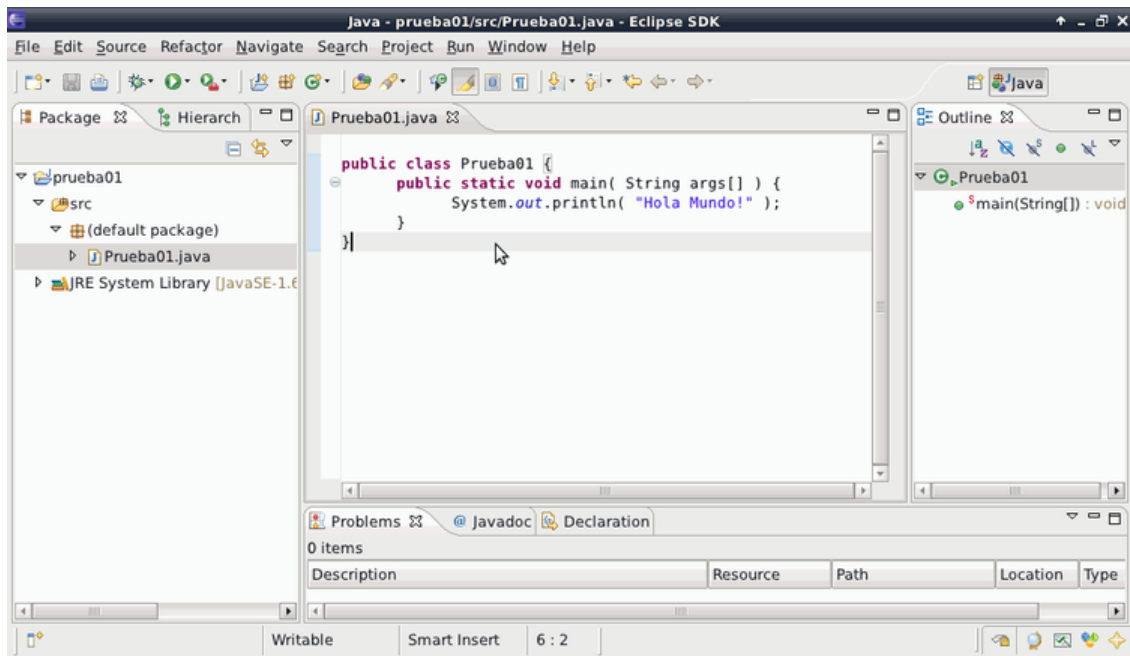
Se nos preguntará el nombre para la clase (junto con otros datos que por ahora no necesitamos cambiar):



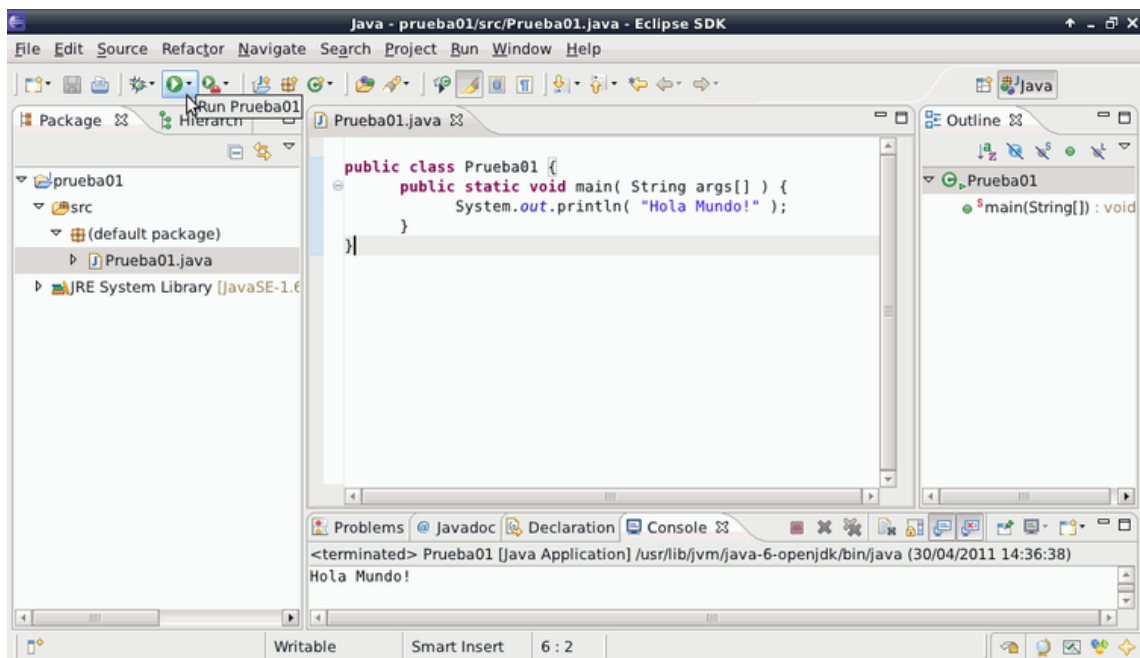
Y nos aparecerá un esqueleto de clase (algo más vacío que en el caso de NetBeans):



Sobre ese esqueleto, escribiremos (o copiaremos y pegaremos) los detalles de nuestro fuente:



Y si pulsamos el botón "Run", el programa se compilará y se pondrá en marcha, y su resultado se mostrará en la parte inferior de la pantalla, en la pestaña "Console":



Ejercicio propuesto 3.6.1: Crea un programa en Java, usando Eclipse, que te salude en pantalla por tu nombre (por ejemplo, "Hola, Nacho").

4. Variables y operaciones matemáticas básicas

4.1. Las variables.

En nuestro primer ejemplo escribíamos un texto en pantalla, pero este texto estaba prefijado dentro de nuestro programa.

De forma muy similar, podemos realizar operaciones matemáticas si las indicamos sin comillas. Así, el ordenador sabrá que no debe escribir algo "tal cual", sino que debe intentar descubrir su significado:

```
// Suma1.java
// Ejemplo de suma con datos prefijados
// Introducción a Java, Nacho Cabanes

class Suma1 {

    public static void main( String args[] ) {
        System.out.println(
            "La suma de 56 y 23 es:" ); // Mostramos un mensaje de aviso
        System.out.println(
            56+23 ); // y el resultado de la operación
    }
}
```

Pero esto no es lo habitual: normalmente los datos que maneje nuestro programa serán el resultado de alguna operación matemática o de cualquier otro tipo, a partir de datos introducidos por el usuario, leídos de un fichero, obtenidos de Internet... Por eso, necesitaremos un espacio en el que ir almacenando valores temporales y resultados de las operaciones.

En casi cualquier lenguaje de programación podremos reservar esos "espacios", y asignarles un nombre con el que acceder a ellos. Esto es lo que se conoce como **"variables"**.

Por ejemplo, si queremos que el usuario introduzca dos números y el ordenador calcule la suma de ambos números y la muestre en pantalla, necesitaríamos el espacio para almacenar al menos esos dos números iniciales. No sería imprescindible reservar espacio también para la suma, porque podemos mostrarla en pantalla en el mismo instante en que la calculamos. Los pasos a dar serían los siguientes:

Avisar al usuario de que deseamos que introduzca un número.

- Almacenar ese valor que introduzca el usuario (por ejemplo, en un espacio de memoria al que podríamos asignar el nombre "primerNumero").

- Pedir al usuario que introduzca otro número.
- Almacenar el nuevo valor (por ejemplo, en otro espacio de memoria llamado "segundoNumero").
- Mostrar en pantalla el resultado de sumar "primerNumero" y "segundoNumero".

Pues bien, en este programa estaríamos empleando dos variables llamadas "primerNumero" y "segundoNumero". Cada una de ellas se usaría para acceder a un espacio de memoria, que será capaz de almacenar un número.

Para no desperdiciar memoria de nuestro ordenador, el espacio de memoria que hace falta "reservar" se podrá optimizar según lo grande que pueda llegar a ser dicho número (la cantidad de cifras), o según la precisión que necesitemos para ese número (cantidad de decimales). Por eso, tenemos disponibles diferentes **"tipos de variables"**.

Por ejemplo, si vamos a manejar números sin decimales ("números enteros") de como máximo 9 cifras, nos interesaría el tipo de datos llamado "int" (esa palabra es la abreviatura de "integer", "entero" en inglés). Este tipo de datos consume un espacio de memoria de 4 bytes. Si no necesitamos guardar datos tan grandes (por ejemplo, si nuestros datos van a ser números inferiores a 10.000), podemos emplear el tipo de datos llamado "short" (entero "corto"), que ocupa la mitad de espacio.

Con eso, vamos a ver un programa que sume dos números enteros (de no más de 9 cifras) prefijados y muestre en pantalla el resultado:

```
// Suma2.java
// Ejemplo de suma con variables
// Introducción a Java, Nacho Cabanes

class Suma2 {

    public static void main( String args[] ) {

        int primerNumero = 56; // Dos enteros con valores prefijados
        int segundoNumero = 23;

        System.out.println(
            "La suma de 56 y 23 es:" ); // Muestro un mensaje de aviso
        System.out.println(
            primerNumero+segundoNumero ); // y el resultado de la operación
    }
}
```

Como se ve, la forma de "**declarar**" un variable es detallando primero el tipo de datos que podrá almacenar ("int", por ahora) y después el nombre que daremos la variable. Además, opcionalmente se puede indicar un valor inicial.

Ejercicio propuesto 4.1.1: Crea un programa en Java que escriba en pantalla el producto de dos números prefijados (pista: el símbolo de la multiplicación es el asterisco, "*").

4.2. Operaciones matemáticas básicas.

Hay varias operaciones matemáticas que son frecuentes. Veremos cómo expresarlas en Java, así como otra operación menos habitual:

Operación matemática	Símbolo correspondiente
Suma	+
Resta	-
Multiplicación	*
División	/
Resto de la división	%

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar. La única operación "menos habitual" es el **resto** de la división. Por ejemplo, si dividimos 14 entre 3, obtenemos 4 como cociente y 2 como resto, de modo que el resultado de $14 \% 3$ sería 2.

Ejercicio propuesto 4.2.1: Crea un programa que muestre la división de dos números prefijados.

Ejercicio propuesto 4.2.2: Crea un programa que muestre el resto de dividir 100 entre 30.

4.3. Operar con datos introducidos por el usuario

Vamos a ver cómo hacer que sea el usuario quien introduzca valores para esas variables, que es algo mucho más habitual. Nuestro primer ejemplo sumará dos números, que en esta ocasión no estarán prefijados:

```
// Suma3.java
// Ejemplo a las variables introducidas por el usuario
```



```
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Suma3 {

    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);
        System.out.print( "Introduzca el primer número: " );
        int primerNumero = teclado.nextInt();
        System.out.print( "Introduzca el segundo número: " );
        int segundoNumero = teclado.nextInt();

        System.out.print( "Su suma es: " );
        System.out.println( primerNumero+segundoNumero );

    }
}
```

En este programa hay varias cosas nuevas:

- Vamos a usar una característica que no es algo básico del lenguaje. Por eso le decimos que deseamos "importar" nuevas funcionalidades. En nuestro caso, se trata de un tal "Scanner", que nos permitirá analizar textos: **import java.util.Scanner;**
- En concreto, nuestro scanner va a tomar datos desde la entrada del sistema (System.in), por lo que lo declaramos con: **Scanner teclado = new Scanner(System.in);** El nombre "teclado" podría ser "entrada" o cualquier otro.
- A partir de entonces, cada vez que llamemos a **".nextInt()"** se leerá un número desde la entrada estándar del sistema (el teclado): `int primerNumero = teclado.nextInt();`

Son varias novedades, pero tampoco debería resultar difícil. Este programa escribirá algo como (dependiendo de los datos que introduzca el usuario):

```
Introduzca el primer número: 34
Introduzca el segundo número: 56
Su suma es: 90
```

Es habitual declarar las variables al principio del programa, antes de que comience la lógica real que resuelve el problema. Si varias variables van a guardar datos del mismo tipo, se pueden declarar todas ellas a la vez, separadas por comas, como en el siguiente ejemplo:

```
// Suma3b.java
// Dos variables declaradas a la vez
```

```
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Suma3b {

    public static void main( String args[] ) {

        Scanner teclado;
        int primerNumero, segundoNumero;

        teclado = new Scanner(System.in);
        System.out.print( "Introduzca el primer número: " );
        primerNumero = teclado.nextInt();
        System.out.print( "Introduzca el segundo número: " );
        segundoNumero = teclado.nextInt();

        System.out.print( "Su suma es: " );
        System.out.println( primerNumero+segundoNumero );

    }
}
```

Como la forma de asimilar todo esto es probándolo, aquí tienes varios ejercicios propuestos:

Ejercicio propuesto 4.3.1: Crea un programa que calcule y muestre el producto de dos números enteros que introduzca el usuario.

Ejercicio propuesto 4.3.2: Crea un programa que calcule y muestre la división de dos números enteros que introduzca el usuario.

Ejercicio propuesto 4.3.3: Crea un programa que calcule y muestre el resto de la división de dos números enteros que introduzca el usuario.

Ejercicio propuesto 4.3.4: Crea un programa que pida al usuario una longitud en millas (por ejemplo, 3) y calcule su equivalencia en metros (1 milla = 1609 m).

Ejercicio propuesto 4.3.5: Crea un programa que pida al usuario una temperatura en grados centígrados y calcule (y muestre) a cuántos grados Fahrenheit equivalen ($F = 9 \cdot C / 5 + 32$).

Ejercicio propuesto 4.3.6: Crea un programa que pregunte al usuario la base y la altura de un triángulo y muestre su superficie ($S = B \cdot A / 2$).

4.4. Incremento y asignaciones abreviadas

Hay varias operaciones muy habituales, que tienen una sintaxis abreviada en Java.

Por ejemplo, para sumar 2 a una variable "a", la forma "normal" de conseguirlo sería:

```
a = a + 2;
```

pero existe una forma abreviada en Java:

```
a += 2;
```

Al igual que tenemos el operador += para aumentar el valor de una variable, podemos emplear -= para disminuirlo, /= para dividirla entre un cierto número, *= para multiplicarla por un número, y así sucesivamente. Por ejemplo, para multiplicar por 10 el valor de la variable "b" haríamos

```
b *= 10;
```

También podemos **aumentar o disminuir en una unidad** el valor de una variable, empleando los operadores de "incremento" (++) y de "decremento" (--). Así, para sumar 1 al valor de "a", podemos emplear cualquiera de estas tres formas:

```
a = a+1;  
a += 1;  
a++;
```

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

```
a++;  
++a;
```

Pero hay una diferencia si ese valor se asigna a otra variable "al mismo tiempo" que se incrementa/decrementa:

```
int c = 5;  
int b = c++;
```

da como resultado c = 6 y b = 5, porque se asigna el valor a "b" antes de incrementar "c", mientras que

```
int c = 5;  
int b = ++c;
```

da como resultado $c = 6$ y $b = 6$ (se asigna el valor a "b" después de incrementar "c").

Por eso, para evitar efectos colaterales no esperados, es mejor no incrementar una variables a la vez que se asigna su valor a otra, sino hacerlo en dos pasos.

Ejercicio propuesto 4.4.1: Calcula "a mano" el resultado de las siguientes operaciones con números enteros, y luego crea un programa que muestre el resultado.

```
a = 5;
a++;
a*=2;
a-=3;
a%=5;
a=a+7;
```

4.5. Otros tipos de datos numéricos

No sólo se puede almacenar números enteros de hasta 9 cifras. Java nos permite usar también otros **tipos de datos numéricos**:

Nombre	¿Admite decimales?	Valor mín.	Valor máx.	Precisión	Ocupa
byte	no	-128	127	-	1 byte
short	no	-32.768	32.767	-	2 bytes
int	no	-2.147.483.648	2.147.483.647	-	4 bytes
long	no	- 9.223.372.036.854.775.808	9.223.372.036.854.775.807	-	8 bytes
float	sí	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
double	sí	4.94065645841247E-324	1.79769313486232E308	14-15 cifras	8 bytes

Los datos de tipo byte, short, int y long sirven para almacenar números **enteros**, de mayor o menor tamaño. Si se desborda ese tamaño, el programa se interrumpirá con un error, como en este ejemplo:

```
// DesbordarByte.java
// Ejemplo de desbordamiento de una variable
// Introducción a Java, Nacho Cabanes

class DesbordarByte {
    public static void main( String args[] ) {
        byte dato = 100;
        System.out.print( "El dato inicialmente es: " );
        System.out.println( dato );
        dato += 100;
        System.out.print( "Si sumamos 100, el dato ahora es: " );
        System.out.println( dato );
    }
}
```

Que mostraría:

```
El dato inicialmente es: 100
Si sumamos 100, el dato ahora es: -56
```

Por su parte, los datos float y double permiten almacenar números **reales** (con cifras decimales), que pueden almacenar los números de forma incorrecta si tienen más cifras que la permitida por ese tipo de datos, como en este ejemplo:

```
// PrecisionFloat.java
// Ejemplo de límite de precisión de un dato float
// Introducción a Java, Nacho Cabanes

class PrecisionFloat {
    public static void main( String args[] ) {
        float dato = 1.23456789f;
        System.out.print( "El dato inicialmente es: " );
        System.out.println( dato );
        dato += 1000;
        System.out.print( "Si sumamos 1000, el dato ahora es: " );
        System.out.println( dato );
    }
}
```

Que escribiría:

```
El dato inicialmente es: 1.2345679
Si sumamos 1000, el dato ahora es: 1001.23456
```

Para pedir datos de estos tipos, podemos usar un "Scanner" con ".nextByte", ".nextShort", ".nextLong", ".nextFloat" o ".nextDouble"

Por ejemplo, se podría sumar dos números reales de forma similar a como hemos visto para dos números enteros:

```
// SumaFloat.java
// Sumar dos números reales
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class SumaFloat {

    public static void main( String args[] ) {

        Scanner teclado;
        float primerNumero, segundoNumero;

        teclado = new Scanner(System.in);
        System.out.print( "Introduzca el primer número real: " );
        primerNumero = teclado.nextFloat();
        System.out.print( "Introduzca el segundo número real: " );
        segundoNumero = teclado.nextFloat();

        System.out.print( "Su suma es: " );
        System.out.println( primerNumero+segundoNumero );

    }

}
```

Un ejemplo de su posible resultado (que muestra las posibles pérdidas de precisión) podría ser:

```
Introduzca el primer número real: 2,3
Introduzca el segundo número real: 3,56
Su suma es: 5.8599997
```

Ejercicio propuesto 4.5.1: Crea un programa que calcule y muestre la suma de dos números de dos cifras (de tipo byte) que introduzca el usuario.

Ejercicio propuesto 4.5.2: Crea un programa que pida al usuario su año de nacimiento y el año actual (usando datos de tipo short) y halle la diferencia de ambos para obtener su edad.

Ejercicio propuesto 4.5.3: Crea un programa que calcule y muestre la división de dos números reales de doble precisión introducidos por el usuario.

Ejercicio propuesto 4.5.4: Crea un programa que pida al usuario una longitud en millas (por ejemplo, 3) y calcule su equivalencia en kilómetros, usando datos de tipo float (1 milla = 1.609 km).

5. Comprobación de condiciones

5.1. if

En cualquier lenguaje de programación es habitual tener que comprobar si se cumple una cierta **condición**. La forma más simple de conseguirlo es empleando una construcción que recuerda a:

```
SI condición ENTONCES órdenes
```

En Java, la forma exacta será empleando **if** (si, en inglés), seguido por la condición entre paréntesis, e indicando finalmente entre llaves los pasos que queremos dar si se cumple la condición, así :

```
if (condición) { órdenes }
```

Por ejemplo,

```
// If1.java
// Comprobación de condiciones con "if" 1: mayor que
// Introducción a Java, Nacho Cabanes

class If1 {

    public static void main( String args[] ) {

        int x = 10;

        if (x > 5) {
            System.out.println( "x es mayor que 5" );
        }
    }
}
```

Nota: Las llaves sólo serán imprescindibles cuando haya que hacer varias cosas. Si sólo queremos dar un paso en caso de que se cumpla la condición, no es necesario emplear llaves (aunque puede ser recomendable usar siempre las llaves, para no olvidarlas si más adelante ampliamos ese fragmento del programa):

```
// If2.java
// Comprobación de condiciones con "if" 2: sin llaves
// Introducción a Java, Nacho Cabanes

class If2 {

    public static void main( String args[] ) {

        int x = 10;
```

```

        if (x > 5)
            System.out.println( "x es mayor que 5" );
    }
}

```

Ejercicio propuesto 5.1.1: Crea un programa que pida un número al usuario y diga si es positivo (mayor que cero)

5.2. El caso contrario: else

Una primera **mejora** a la comprobación de condiciones con "if" es indicar qué hacer en caso de que no se cumpla la condición. Sería algo parecido a

SI condición
 ENTONCES órdenes
 EN_CASO_CONTRARIO órdenes_alternativas

que en Java escribiríamos así:

```

if (condición) { órdenes1 } else { órdenes2 }

```

Por ejemplo,

```

// If3.java
// Comprobación de condiciones con "if" 3: else
// Introducción a Java, Nacho Cabanes

class If3 {

    public static void main( String args[] ) {

        int x = 10;

        if (x > 5) {
            System.out.println( "x es mayor que 5" );
        }
        else {
            System.out.println( "x es menor o igual que 5" );
        }
    }
}

```

Ejercicio propuesto 5.2.1: Crea un programa que pida un número entero al usuario y diga si es positivo (mayor que cero) o si, por el contrario, no lo es (usando "else").

5.3. Operadores relacionales

La forma de comprobar si un dato es "mayor" que otro, que hemos usado en el apartado anterior, debería resultar totalmente intuitiva. Pero también nos interesará comprobar si un variable tiene exactamente un cierto valor, o si tiene un valor distinto, o si es menor, o si es mayor o igual que otro...

Los operadores que utilizaremos para ello son:

Operación	Símbolo
Mayor que	>
Mayor o igual que	>=
Menor que	<
Menor o igual que	<=
Igual que	== (dos símbolos de "igual")
Distinto de	!=

Así, por ejemplo, para ver si el valor de una variable "b" es distinto de 5, escribiríamos:

```
// If4.java
// Comprobación de condiciones con "if" 4: distinto de
// Introducción a Java, Nacho Cabanes

class If4 {

    public static void main( String args[] ) {

        int x = 10;

        if (x != 5) {
            System.out.println( "x no vale 5" );
        }
        else {
            System.out.println( "x vale 5" );
        }
    }
}
```

o para ver si la variable "a" vale 70, sería:

```
// If5.java
// Comprobación de condiciones con "if" 5: igual a
// Introducción a Java, Nacho Cabanes

class If5 {
```

```

public static void main( String args[] ) {

    int x = 10;

    if (x == 70) {
        System.out.println( "x vale 70" );
    }
    else {
        System.out.println( "x no vale 70" );
    }
}
}

```

Es **muy importante** recordar esta diferencia: para asignar un valor a una variable se emplea un único signo de igual, mientras que para comparar si una variable es igual a otra (o a un cierto valor) se emplean dos signos de igual.

Ejercicio propuesto 5.3.1: Crea un programa que pida al usuario que introduzca el número 12. Después debe decirle si lo ha hecho correctamente o no.

Ejercicio propuesto 5.3.2: Crea un programa que pida un número entero al usuario y diga si es positivo (mayor que cero), si es negativo (menor que cero) o si, por el contrario, es exactamente 0 (necesitarás enlazar 2 "if" uno tras otro).

Ejercicio propuesto 5.3.3: Crea un programa que pida dos números reales (con decimales) al usuario y diga cuál es el mayor de ellos.

Ejercicio propuesto 5.3.4: Crea un programa que pida al usuario y diga si ese número es múltiplo de 3 (pista: puedes utilizar la operación "módulo", el "resto de la división": %)

Ejercicio propuesto 5.3.5: Crea un programa que diga si el número introducido por el usuario es impar o no lo es.

5.4. Operadores lógicos para enlazar condiciones

Podremos enlazar varias condiciones, para indicar qué hacer cuando se cumplan ambas, o sólo una de ellas, o cuando no se cumplan. Los operadores que nos permitirán enlazar condiciones son:

Operación	Símbolo
Y	&&
O	
No	!

Por ejemplo, la forma de decir "si a vale 3 y b es mayor que 5, o bien a vale 7 y b no es menor que 4" sería:

```
// If6.java
// Comprobación de condiciones con "if" 6: varias condiciones
// Introducción a Java, Nacho Cabanes

class If6 {

    public static void main( String args[] ) {

        int a = 7;
        int b = 1;

        if ( ((a == 3) || ( b > 5))
            || ((a == 7) && ! (b < 4)) ) {
            System.out.println( "Se cumple la condición" );
        }
        else {
            System.out.println( "No se cumple la condición" );
        }
    }
}
```

Como se ve en el ejemplo anterior, la condición general del "if" deberá ir entre paréntesis, pero, por legibilidad y por evitar efectos colaterales de que alguna condición no se analice en el orden deseado, convendrá que cada condición individual vaya también entre paréntesis, y será interesante añadir paréntesis adicionales para dejar claro cómo agruparlas.

Ejercicio propuesto 5.4.1: Crea un programa que pida al usuario dos números enteros y diga si los dos son positivos (el primero Y el segundo).

Ejercicio propuesto 5.4.2: Crea un programa que pida al usuario dos números enteros y diga si al menos uno de los dos es positivo (el primero O el segundo).

Ejercicio propuesto 5.4.3: Crea un programa que pida al usuario dos números enteros y cuántos de ellos son pares.

Ejercicio propuesto 5.3.5: Crea un programa que pida tres números enteros largos al usuario y diga cuál es el mayor de los tres.

5.5. switch

Si queremos comprobar **varios posibles valores para una misma variable**, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una

forma más elegante de elegir entre distintos valores posibles que tome una cierta expresión. Su formato es éste:

```
switch (expresion) {  
    case valor1: sentencias1; break;  
    case valor2: sentencias2; break;  
    case valor3: sentencias3; break;  
    // ... Puede haber más valores  
}
```

Es decir, después de la orden **switch** indicamos entre paréntesis la expresión que queremos evaluar. Después, tenemos distintos apartados, uno para cada valor que queramos comprobar; cada uno de estos apartados se precede con la palabra **case**, indica los pasos a dar si es ese valor el que tiene la variable (esta serie de pasos no será necesario indicarla entre llaves), y termina con **break**.

Un ejemplo sería:

```
// Switch1.java  
// Comprobación de condiciones con "switch" 1: ejemplo básico  
// Introducción a Java, Nacho Cabanes  
  
class Switch1 {  
  
    public static void main( String args[] ) {  
  
        int mes = 2;  
  
        switch(mes) {  
            case 1: System.out.println( "El mes es Enero" ); break;  
            case 2: System.out.println( "El mes es Febrero" ); break;  
            case 3: System.out.println( "El mes es Marzo" ); break;  
        }  
    }  
}
```

También podemos indicar qué queremos que ocurra en el caso de que el valor de la expresión no sea ninguno de los que hemos detallado, usando la palabra **"default"**:

```
switch (expresion) {  
    case valor1: sentencias1; break;  
    case valor2: sentencias2; break;  
    case valor3: sentencias3; break;  
    // ... Puede haber más valores  
    default: sentencias; // Opcional: valor por defecto  
}
```

Por ejemplo, así:

```
// Switch2.java
// Comprobación de condiciones con "switch" 2: default
// Introducción a Java, Nacho Cabanes

class Switch2 {

    public static void main( String args[] ) {

        int mes = 4;

        switch(mes) {
            case 1: System.out.println( "El mes es Enero" ); break;
            case 2: System.out.println( "El mes es Febrero" ); break;
            case 3: System.out.println( "El mes es Marzo" ); break;
            default: System.out.println( "No es entre Enero y Marzo" ); break;
        }
    }
}
```

Podemos conseguir que se den los mismos pasos en varios casos, simplemente eliminando la orden "break" de algunos de ellos. Un ejemplo podría ser:

```
// Switch3.java
// Comprobación de condiciones con "switch" 3: casos en cascada
// Introducción a Java, Nacho Cabanes

class Switch3 {

    public static void main( String args[] ) {

        int x = 5;

        switch ( x ) {
            case 1:
            case 2:
            case 3:
                System.out.println( "El valor de x estaba entre 1 y 3" );
                break;
            case 4:
            case 5:
                System.out.println( "El valor de x era 4 o 5" );
                break;
            case 6:
                System.out.println( "El valor de x era 6" );
                int valorTemporal = 10;
                System.out.println( "Operaciones auxiliares completadas" );
                break;
            default:
                System.out.println( "El valor de x no estaba entre 1 y 6" );
                break;
        }
    }
}
```

A partir de Java 7, se puede usar la orden "switch" también para comprobar los valores de cadenas de texto:

```
// Switch4.java
// Comprobación de condiciones con "switch" 4: cadenas de texto
// (Para Java 7 o superior)
// Introducción a Java, Nacho Cabanes

class Switch4 {

    public static void main( String args[] ) {

        String nombre = "yo";

        switch(nombre) {
            case "uno": System.out.println( "Hola, uno" ); break;
            case "yo": System.out.println( "Hola, tú" ); break;
            case "otro": System.out.println( "Bienvenido, otro" ); break;
            default: System.out.println( "Nombre desconocido" ); break;
        }
    }
}
```

Ejercicio propuesto 5.5.1: Crea un programa que pida al usuario el número de un mes y escriba el nombre de ese mes. Por ejemplo, si el usuario introduce 9, deberá escribir "septiembre".

Ejercicio propuesto 5.5.2: Crea un programa que escriba como texto cualquier número del 1 al 10 que introduzca el usuario. Por ejemplo, si el usuario introduce 3, deberá escribir "tres".

5.6. El operador condicional

Existe una construcción adicional, que permite comprobar si se cumple una condición y devolver un cierto valor según si dicha condición se cumple o no. Es lo que se conoce como el "operador condicional (?) u "operador ternario":

```
condicion ? resultado_si_cierto : resultado_si_falso
```

Es decir, se indica la condición seguida por una interrogación, después el valor que hay que devolver si se cumple la condición, a continuación un símbolo de "dos puntos" y finalmente el resultado que hay que devolver si no se cumple la condición.

Es frecuente emplearlo en asignaciones (aunque algunos autores desaconsejan su uso porque puede resultar menos legible que un "if"), como en este ejemplo:

```
x = (a == 10) ? b*2 : a ;
```

En este caso, si "a" vale 10, la variable "x" tomará el valor de $b*2$, y en caso contrario tomará el valor de a. Esto también se podría haber escrito de la siguiente forma, más larga pero más legible:

```
if (a == 10)
    x = b*2;
else
    x = a;
```

Un ejemplo completo, que diera a la variable "par" el valor 1 si un número "n" es par, o el valor 0 en caso contrario, sería:

```
// Condicional1.java
// Ejemplo de "operador condicional" (o ternario)
// Introducción a Java, Nacho Cabanes

class Condicional1 {

    public static void main( String args[] ) {

        int n = 4;
        int par;

        par = n % 2 == 0 ? 1 : 0;

        System.out.print( "\"par\" vale... " );
        System.out.println( par );

    }

}
```

Ejercicio propuesto 5.6.1: Crea un programa que pida un número entero al usuario y dé a una variable par el valor 1 si ese número es par o el valor 0 si no es par. Hazlo primero con un "if" y luego con un "operador condicional".

Ejercicio propuesto 5.6.2: Crea un programa que pida dos números de tipo byte al usuario y cree a una variable "menor", que tenga el valor del menor de esos dos números. Hazlo primero con un "if" y luego con un "operador condicional".

6. Bucles: partes del programa que se repiten

Con frecuencia tendremos que hacer que una parte de nuestro programa se repita, bien sea mientras se cumpla una condición o bien un cierto número prefijado de veces. Es lo que llamaremos un **"bucle"**.

6.1. while

La orden **"while"** hace que una parte del programa se repita mientras se cumpla una cierta condición. Su formato será:

```
while (condición)
    sentencia;
```

Es decir, la sintaxis es similar a la de "if", con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (**si** se cumplía la condición), pero "while" puede repetir la sentencia más de una vez (**mientras** la condición sea cierta). Al igual que ocurría con "if", podemos realizar varias sentencias seguidas (dar "más de un paso") si las encerramos entre llaves:

```
// While1.java
// Comprobación repetitiva de condiciones con "while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class While1 {

    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);
        System.out.print("Introduce un cero: ");
        int dato = teclado.nextInt();
        while (dato != 0) {
            System.out.print("No era cero. Introduce cero: ");
            dato = teclado.nextInt();
        }
        System.out.println("Terminado!");
    }

}
```

Estas estructuras repetitivas se pueden usar también para hacer "contadores". Por ejemplo, un programa que contase del 1 al 5 (y escribiese todos esos números) podría ser:

```
// Contar1a5.java
// Contar del 1 al 5 con "while"
// Introducción a Java, Nacho Cabanes
```



```

class Contar1a5 {

    public static void main( String args[] ) {

        int x = 1;

        while (x <= 5) {
            System.out.println( x );
            x++;
        }
    }
}

```

Ejercicio propuesto 6.1.1: Crea un programa que muestre los números del 1 al 10, usando "while"

Ejercicio propuesto 6.1.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "while"

Ejercicio propuesto 6.1.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "while"

Ejercicio propuesto 6.1.4: Crea un programa que pida al usuario un número entero y muestre su cuadrado. Se repetirá mientras el usuario introduzca un número distinto de cero.

Ejercicio propuesto 6.1.5: Crea un programa que pida al usuario su login (un número entero) y su contraseña (otro número entero). Se repetirá mientras el usuario introduzca un login distinto de "1809" o una contraseña distinta de "1234".

Ejercicio propuesto 6.1.6: Crea un programa que pida al usuario su login (un número entero) y su contraseña (otro número entero). Se repetirá mientras el usuario introduzca un login distinto de "1809" o una contraseña distinta de "1234", hasta un máximo de tres veces. Tras tres fallos, se mostrará un mensaje avisando al usuario de que se le deniega el acceso.

Ejercicio propuesto 6.1.7: Crea un programa que escriba en pantalla tantos asteriscos como el usuario indique, todos ellos en la misma línea.

6.2. do-while

Existe una variante de "while", que permite comprobar la condición al final de la parte repetitiva, en vez de hacerlo antes de comenzar. Es el conjunto **do..while**, cuyo formato es:

```

do {
    sentencia;
}

```

```
} while (condición)
```

Con "while", si la condición era falsa desde un principio, los pasos que se indicaban a continuación de "while" no llegaban a darse ni una sola vez; con do-while, las "sentencias" intermedias se realizarán al menos una vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
// DoWhile1.java
// Comprobación repetitiva de condiciones con "do-while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class DoWhile1 {

    public static void main( String args[] ) {

        int password;

        Scanner teclado = new Scanner(System.in);

        do {

            System.out.print( "Introduzca su password numérica: " );
            password = teclado.nextInt();

            if (password != 1234)
                System.out.println( "No es válida." );

        } while (password != 1234);

    }

}
```

Por supuesto, también podemos crear un contador usando un do..while, siempre y cuando tengamos en cuenta que la condición se comprueba al final, lo que puede hacer que la lógica sea un poco distinta en algunos casos:

```
// Contar1a5b.java
// Contador con "do-while"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Contar1a5b {
```

```

public static void main( String args[] ) {

    int x = 1;

    do {
        System.out.println( x );
        x++;
    }
    while (x <= 5);
}
}

```

Ejercicio propuesto 6.2.1: Crea un programa que muestre los números del 1 al 10, usando "do-while"

Ejercicio propuesto 6.2.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "do-while"

Ejercicio propuesto 6.2.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "do-while"

Ejercicio propuesto 6.2.4: Crea un programa que pida al usuario dos números enteros y muestre su suma. Se repetirá hasta que los dos sean 0. Emplea do-while.

Ejercicio propuesto 6.2.5: Crea un programa que pida al usuario su login (un número entero) y su contraseña (otro número entero). Se repetirá hasta que el usuario introduzca como login "1809" y como contraseña "1234". En esta ocasión, hazlo con do-while.

Ejercicio propuesto 6.2.6: Crea un programa que pida un número de tipo byte al usuario y escriba en pantalla un cuadrado formado por asteriscos, que tendrá como alto y ancho la cantidad introducida por el usuario.

6.3. for

Una tercera forma de conseguir que parte de nuestro programa se repita es mediante la orden "**for**". La emplearemos sobre todo para conseguir un número concreto de repeticiones. En lenguajes como BASIC su formato es muy simple: "FOR x = 1 TO 10" irá recorriendo todos los valores de x, desde uno hasta 10. En Java y otros lenguajes que derivan de C, su sintaxis es más enrevesada:

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {  
    sentencias  
}
```

Es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres órdenes:

- La primera orden dará el valor inicial a una variable que sirva de control.
- La segunda orden será la condición que se debe cumplir mientras que se repitan las sentencias.
- La tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar.

Esto se verá mejor con un ejemplo. Podríamos repetir 10 veces un bloque de órdenes haciendo:

```
// For1.java  
// Repetición con "for" 1: escribir 10 veces  
// Introducción a Java, Nacho Cabanes  
  
class For1 {  
  
    public static void main( String args[] ) {  
  
        int i;  
  
        for ( i=1 ; i<=10 ; i++ ) {  
            System.out.print( "Hola " );  
        }  
    }  
}
```

(inicialmente i vale 1, hay que repetir mientras sea menor o igual que 10, y en cada paso hay que aumentar su valor una unidad),

De forma similar, podríamos hacer un contador de 1 a 5, como los que hemos creado con "while" y con "do-while":

```
// Contar1a5c.java  
// Repetición con "for" 2: contar de 1 a 5  
// Introducción a Java, Nacho Cabanes  
  
class Contar1a5c {  
  
    public static void main( String args[] ) {  
  
        int x;  
  
        for ( x=1 ; x<=5 ; x++ ) {  
            System.out.println( x );  
        }  
    }  
}
```

```
}
```

O bien podríamos contar descendiendo desde el 10 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

```
// Contar10a0.java
// Repetición con "for" 3: descontar de 10 a 0, de 2 en 2
// Introducción a Java, Nacho Cabanes

class Contar10a0 {

    public static void main( String args[] ) {

        int i;

        for ( i=10 ; i>=0 ; i-=2 ) {
            System.out.println( i );
        }

    }

}
```

Nota: se puede observar una equivalencia casi inmediata entre la orden "for" y la orden "while". Así, el ejemplo anterior se podría reescribir empleando "while", de esta manera:

```
// Contar10a0b.java
// Repetición con "while": descontar de 10 a 0, de 2 en 2
// Programa equivalente a "For2.java"
// Introducción a Java, Nacho Cabanes

class Contar10a0b {

    public static void main( String args[] ) {

        int i;

        i=10 ;
        while ( i>=0 ) {
            System.out.println( i );
            i-=2;
        }

    }

}
```

No hay obligación de declarar las variables justo al principio del programa. Es frecuente hacerlo buscando legibilidad, pero también se pueden declarar en puntos posteriores del fuente. Por eso, un sitio habitual para **declarar la variable** que actúa de contador de un "for" es dentro del propio "for", así:

```
// ForVariable.java
// Variable declarada dentro de un "for"
// Introducción a Java, Nacho Cabanes

class ForVariable {

    public static void main( String args[] ) {

        for (int i=10 ; i>=0 ; i-=2 ) {
            System.out.println( i );
        }

    }

}
```

Esto tiene la ventaja de que no podemos reutilizar por error esa variable "i" después de salir del "for", porque "se destruye" automáticamente y obtendríamos un mensaje de error si tratamos de usarla. Así evitamos efectos indeseados de que cambios en una parte del programa afecten a otra parte del programa porque estemos reusando una variable sin darnos cuenta.

Ejercicio propuesto 6.3.1: Crea un programa que muestre los números del 1 al 15, usando "for"

Ejercicio propuesto 6.3.2: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "for"

Ejercicio propuesto 6.3.3: Crea un programa que muestre la "tabla de multiplicar del 5", usando "for"

Ejercicio propuesto 6.3.4: Crea un programa que muestre los números enteros del 0 al 100 que son múltiplos de 6 (el resto al dividir por 6 sea 0).

Ejercicio propuesto 6.3.5: Crea un programa que muestre los números enteros entre 0 y el que introduzca el usuario que sean múltiplos de 3 (el resto al dividir por 3 es 0) y a la vez múltiplos de 7 (ídem).

Ejercicio propuesto 6.3.6: Crea un programa que pida dos números de tipo byte al usuario y escriba en pantalla un rectángulo formado por asteriscos, que tendrá como alto el primer número y como ancho el segundo número.

Ejercicio propuesto 6.3.7: Crea un programa que pida un número de tipo byte al usuario y escriba en pantalla un cuadrado hueco de ese ancho, que tendrá un borde formado por asteriscos y su interior serán espacios en blanco. Por ejemplo, para un tamaño de 4 sería:

```
****
*  *
*  *
*  *
****
```

Ejercicio propuesto 6.3.8: Crea un programa que pida al usuario un número entero largo y muestre cuáles son sus divisores (aquellos entre los que se puede dividir, obteniendo como resto 0).

Ejercicio propuesto 6.3.9: Crea un programa que pida al usuario un número entero largo y diga si es primo (si sólo es divisible entre 1 y él mismo).

Ejercicio propuesto 6.3.10: Crea un programa que pida al usuario dos números enteros largos y diga cuántos números primos hay entre ellos (ambos incluidos).

Precaución con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. Si planteamos mal la condición de salida, nuestro programa se puede quedar "colgado", repitiendo sin fin los mismos pasos (dentro de un "bucle sin fin").

6.4. break y continue

Se puede modificar un poco el comportamiento de estos bucles con las órdenes "break" y "continue".

La sentencia "**break**" hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente. Como ejemplo:

```
// Break1.java
// Ejemplo de uso de "break"
// Introducción a Java, Nacho Cabanes

class Break1 {

    public static void main( String args[] ) {

        int i;
        System.out.println( "Empezamos..." );
        for ( i=1 ; i<=10 ; i++ ) {
            System.out.println( "Comenzada la vuelta" );
            System.out.println( i );
            if (i==8)
                break;
            System.out.println( "Terminada esta vuelta" );
        }
        System.out.println( "Terminado" );
    }
}
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

La sentencia "**continue**" hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente **iteración** (la siguiente "vuelta" o "pasada"). Como ejemplo:

```
// Continue1.java
// Ejemplo de uso de "continue"
// Introducción a Java, Nacho Cabanes

class Continue1 {

    public static void main( String args[] ) {

        int i;
        System.out.println( "Empezamos..." );
        for ( i=1 ; i<=10 ; i++ ) {
            System.out.println( "Comenzada la vuelta" );
            System.out.println( i );
            if (i==8)
                continue;
            System.out.println( "Terminada esta vuelta" );
        }
        System.out.println( "Terminado" );
    }
}
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, pero sí se darían la pasada de i=9 y la de i=10.

Ejercicio propuesto 6.4.1: Crea un programa que muestre los números del 1 al 20, excepto el 15, usando "for" y "continue"

Ejercicio propuesto 6.4.2: Crea un programa que muestre los números del 1 al 10, usando "for" que vaya del 1 al 20 y "break"

6.5. Etiquetas

También existe la posibilidad de usar una "**etiqueta**" para indicar dónde se quiere saltar con break o continue. Sólo se debería utilizar cuando tengamos un bucle que a su vez está dentro de otro bucle, y queramos salir de golpe de ambos. Es ese caso (muy poco frecuente), deberemos crear una etiqueta justo antes de la orden que queremos que se interrumpa, y luego usar "break" seguido del nombre de esa etiqueta, así

```
// BreakEtiqueta.java
// Ejemplo de uso de "break" con etiqueta de salto
// Introducción a Java, Nacho Cabanes

class BreakEtiqueta {
```



```

public static void main( String args[] ) {
    int i;
    bucleAInterrumpir:
    for ( i=1 ; i<=10 ; i++ ) {
        System.out.println( "Comenzada la vuelta" );
        System.out.println( i );
        if (i==8) break bucleAInterrumpir;
        System.out.println( "Terminada esta vuelta" );
    }
    System.out.println( "Terminado" );
}
}

```

En este caso, a mitad de la pasada 8 se interrumpiría el bloque que hemos etiquetado como "bucleAInterrumpir" (una etiqueta se define como se ve en el ejemplo, con su nombre terminado con el símbolo de "dos puntos").

Ejercicio propuesto 6.5.1: Crea un programa que muestre los números del 1 al 10, usando "for" que vaya del 1 al 20 y un "break" con etiqueta.

7. Booleanos, caracteres, cadenas de texto y arrays

Hemos visto cómo manejar datos numéricos, tanto enteros como reales, y con mayor o menor precisión, pero para muchos problemas reales necesitaremos datos de otros tipos: textos, datos estructurados e incluso otros datos simples que aún no hemos tratado, como letras individuales y valores lógicos "verdadero" o "falso".

7.1 Datos booleanos

Un dato "booleano" es uno que puede tener sólo dos valores posibles: verdadero (true) o falso (false), que son los dos valores existentes en la "lógica de Boole", de la que toman su nombre.

Es habitual utilizarlos para hacer que las condiciones complicadas resulten más legibles. Un ejemplo de su uso podría ser:

```
// Booleano1.java
// Primer ejemplo de variables "bool"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Booleano1 {

    public static void main( String args[] ) {

        int dato;
        boolean todoCorrecto;
        Scanner teclado = new Scanner(System.in);

        do
        {
            System.out.print("Introduce un dato del 0 al 10: ");
            dato = teclado.nextInt();
            todoCorrecto = true;

            if (dato < 0)
                todoCorrecto = false;

            if (dato > 10)
                todoCorrecto = false;

            if ( ! todoCorrecto )
                System.out.println("No es válido!");
        }
        while ( ! todoCorrecto );
        System.out.println("Terminado!");
    }
}
```

No es necesario usar un "if" para darles valores, sino que se puede hacer directamente asignándoles el valor de una condición, así:

```
// Booleano2.java
// Segundo ejemplo de variables "bool"
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Booleano2 {

    public static void main( String args[] ) {

        int dato;
        boolean todoCorrecto;
        Scanner teclado = new Scanner(System.in);

        do
        {
            System.out.print("Introduce un dato del 0 al 10: ");
            dato = teclado.nextInt();
            todoCorrecto = (dato >= 0) && (dato <= 10);
            if ( ! todoCorrecto )
                System.out.println("No es válido!");
        }
        while ( ! todoCorrecto );
        System.out.println("Terminado!");
    }
}
```

Ejercicio propuesto 7.1.1: Crea un programa que pida al usuario un número entero y, si es par, dé el valor "true" a una variable booleana llamada "esPar"; en caso contrario, le dará el valor "false".

7.2. Caracteres

El tipo de datos **char** lo usaremos para almacenar una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación, o cualquier otro carácter. Ocupa 2 bytes. Sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII). Sus valores se deben indicar entre comillas simples: char inicial = 'n';

Por ejemplo, un programa que prefije dos datos de tipos "char" y los muestre sería

```
// Char1.java
// Segundo ejemplo de variables "bool"
// Introducción a Java, Nacho Cabanes

class Char1 {

    public static void main( String args[] ) {

        char letra1, letra2;
```

```

        letra1 = 'a';
        letra2 = 'b';

        System.out.print("La primera letra es : ");
        System.out.println(letra1);

        System.out.print("La segunda letra es : ");
        System.out.println(letra2);
    }
}

```

Java no permite usar un "Scanner" para leer desde teclado letra a letra, así que la mayoría de operaciones con letras las haremos dentro de poco, cuando sepamos manejar cadenas de texto.

7.3. Las cadenas de texto

Una cadena de texto (en inglés, "string") es un conjunto de letras, que usaremos para poder almacenar palabras y frases. Realmente en Java hay **dos "variantes"** de las cadenas de texto: existe un tipo de datos llamado "String" (con la primera letra en mayúsculas, al contrario que los tipos de datos que habíamos visto hasta ahora) y otro tipo de datos llamado "StringBuilder". Un **"String"** será una cadena de caracteres constante, que no se podrá modificar (podremos leer su valor, extraer parte de él, etc.; pero para cualquier modificación, deberemos volcar los datos a una nueva cadena), mientras que un **"StringBuilder"** se podrá modificar "con más facilidad" (podremos insertar letras, dar la vuelta a su contenido, etc) a cambio de ser ligeramente menos eficiente (más lento).

7.3.1. String

Las cadenas se declaran con la palabra **"String"** (cuidado: la primera letra debe estar en mayúsculas) y su valor se asigna entre dobles comillas:

```

// String1.java
// Primer contacto con los String (cadenas de texto)
// Introducción a Java, Nacho Cabanes

class String1 {
    public static void main( String args[] ) {

        String saludo = "Hola";

        System.out.print( "El saludo es... " );
        System.out.println( saludo );

    }
}

```

Podemos "concatenar" cadenas (juntar dos cadenas para dar lugar a una nueva) con el signo +, el mismo que usamos para sumar números. Es frecuente utilizarlo para escribir varias cosas en una misma frase:

```
// String2.java
// Primer contacto con los String: concatenación
// Introducción a Java, Nacho Cabanes

class String2 {
    public static void main( String args[] ) {

        String saludo = "Hola " + "tú";
        System.out.println( "El saludo es... " + saludo );

    }
}
```

Para comprobar si una cadena de texto tiene un cierto valor o no, podemos emplear los símbolos "==" y "!=", igual que con los números:

```
// String3.java
// Tercer contacto con los String: comparación
// Introducción a Java, Nacho Cabanes

class String3 {
    public static void main( String args[] ) {

        String saludo = "Hola";

        if (saludo == "Hola")
            System.out.println( "El saludo es Hola" );
        else
            System.out.println( "El saludo no es Hola" );

    }
}
```

Por otra parte, existe una serie de "operaciones con nombre" (también llamadas "métodos") que podemos aplicar a una cadena. Vamos a ver un ejemplo de algunas de las más frecuentes y luego detallaremos un poco:

```
// String4.java
// Ejemplo más detallado de uso de String
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class String4 {
    public static void main( String args[] ) {

        // Forma "sencilla" de dar un valor
        String texto1 = "Hola";
```

```

// Declarar y dar valor usando un "constructor"
String texto2 = new String("Prueba");

// Declarar sin dar valor
String resultado;

// Manipulaciones básicas
System.out.print( "La primera cadena de texto es: " );
System.out.println( texto1 );

resultado = texto1 + texto2;
System.out.println( "Si concatenamos las dos: " + resultado);

resultado = texto1 + 5 + " " + 23.5 + '.';
System.out.println( "Podemos concatenar varios: " + resultado);
System.out.println( "La longitud de la segunda es: "
    + texto2.length() );
System.out.println( "La segunda letra de texto2 es: "
    + texto2.charAt(1) );

// En general, las operaciones no modifican la cadena
texto2.toUpperCase();
System.out.println( "texto2 no ha cambiado a mayúsculas: " + texto2
);

resultado = texto2.toUpperCase();
System.out.println( "Ahora sí: " + resultado );

// Podemos extraer fragmentos
resultado = texto2.substring(2,5);
System.out.println( "Tres letras desde la posición 2: " + resultado
);

// Y podemos comparar cadenas
System.out.println( "Comparamos texto1 y texto2: "
    + texto1.compareTo(texto2) );
if (texto1.compareTo(texto2) < 0)
    System.out.println( "Texto1 es menor que texto2" );

// Finalmente, pedimos su nombre completo al usuario
System.out.print( "¿Cómo te llamas? ");
Scanner teclado = new Scanner(System.in);
String nombre = teclado.nextLine();
System.out.println( "Hola, " + nombre);

// O podemos bien leer sólo la primera palabra
System.out.print( "Teclea varias palabras y espacios... ");
String primeraPalabra = teclado.next();
System.out.println( "La primera es " + primeraPalabra);
}
}

```

Vamos a ver los detalles más importantes:

- **String texto1 = "Hola";** es la forma normal de dar valor a una variable que vaya a guardar cadenas de texto.

- **String texto2 = new String("Prueba");** es una forma alternativa de dar el valor inicial a una variable de tipo String. Parece una manera más rebuscada, pero es un tipo de construcción muy habitual en Java. Más adelante hablaremos de "clases" y veremos el motivo de escribir ese tipo de órdenes.
- **String resultado;** es la forma de crear una variable que guardará cadenas, pero cuyo valor no se conoce aún.
- **resultado = texto1 + texto2;** Una cadena se puede crear "concatenando" otras dos cadenas, como ya hemos visto.
- **resultado = texto1 + 5 + " " + 23.5 + '.';** Si uno de los datos es numérico, se convertirá a la expresión de cadena equivalente para poderlo concatenar.
- **texto2.length()** devuelve un número, la cantidad de letras que hay en la cadena de texto que se haya guardado dentro de la variable "texto2". Será muy habitual usar esa longitud para recorrer la cadena de texto letra a letra.
- **texto2.charAt(1)** permite obtener el carácter que se encuentra en una cierta posición, en este caso la segunda (porque la primera posición es la número cero).
- **texto2.toUpperCase();** crea una copia de una cadena de texto y la convierte a mayúsculas, pero su valor se pierde si no se vuelca a otra variable o se muestra directamente con una orden "print". La forma correcta será **resultado = texto2.toUpperCase();**
- **if (texto1.compareTo(texto2) > 0)** permite ver si una cadena es "menor" que otra, es decir, si aparecería antes en un diccionario. Si *texto1* es "menor" que *texto2*, se obtendría un valor negativo; por el contrario, si *texto1* es "mayor" que *texto2*, se obtendría un valor positivo.
- **String nombre = teclado.nextLine();** permite leer toda una frase desde teclado usando un Scanner y guardarla en una cadena.
- **String primeraPalabra = teclado.next();** lee desde teclado pero sólo hasta encontrar un espacio en blanco (o el final de la línea porque se pulse Intro, lo que primero ocurra).

Una lista un poco más detallada de los "métodos" (operaciones con nombre) que se pueden aplicar sobre una cadena podría ser::

Método	Cometido
length()	Devuelve la longitud (número de caracteres) de la cadena
charAt (int pos)	Devuelve el carácter que hay en una cierta posición
toLowerCase()	Devuelve la cadena convertida a minúsculas
toUpperCase()	Devuelve la cadena convertida a mayúsculas
substring(int desde, int cuantos)	Devuelve una subcadena: varias letras a partir de una posición dada
replace(char antiguo, char nuevo)	Devuelve una cadena con un carácter reemplazado por otro
trim()	Devuelve una cadena sin espacios de blanco iniciales ni finales
startsWith(String subcadena)	Indica si la cadena empieza con una cierta subcadena
endsWith(String subcadena)	Indica si la cadena termina con una cierta subcadena
indexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el principio, a partir de una posición opcional)
lastIndexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el final, a partir de una posición opcional)
valueOf(objeto)	Devuelve un String que es la representación como texto del objeto que se le indique (número, boolean, etc.)
concat(String cadena)	Devuelve la cadena con otra añadida a su final (concatenada) También se pueden concatenar cadenas con "+"
equals(String cadena)	Mira si las dos cadenas son iguales (lo mismo que "=")
equals-IgnoreCase(String cadena)	Comprueba si dos cadenas son iguales, pero despreciando las diferencias entre mayúsculas y minúsculas
compareTo(String cadena2)	Compara una cadena con la otra (devuelve 0 si son iguales, negativo si la cadena es "menor" que cadena2 y positivo

	si es "mayor").
--	-----------------

En ningún momento estamos modificando el String de partida. Eso sí, en muchos de los casos creamos un String modificado a partir del original.

El método "compareTo" se basa en el **orden lexicográfico**: una cadena que empiece por "A" se considerará "menor" que otra que empiece por la letra "B"; si la primera letra es igual en ambas cadenas, se pasa a comparar la segunda, y así sucesivamente. Las mayúsculas y minúsculas se consideran diferentes.

Ejercicio propuesto 7.3.1.1: Crea un programa que escriba un triángulo con las letras de tu nombre, mostrando primero la primera letra, luego las dos primeras y así sucesivamente, hasta llegar al nombre completo, como en este ejemplo:

N

Na

Nac

Nach

Nacho

(Pista: tendrás que usar "substring" y un bucle "for")

Ejercicio propuesto 7.3.1.2: Crea un programa que pida su nombre al usuario y lo escriba con un espacio entre cada par de letras. Por ejemplo, a partir de "Nacho" escribiría "N a c h o " (Pista: tendrás que usar "charAt" y un bucle "for")

Ejercicio propuesto 7.3.1.3: Crea un programa que pida su nombre al usuario y lo escriba con al revés. Por ejemplo, a partir de "Nacho" escribiría "ohcaN".

Ejercicio propuesto 7.3.1.4: Crea un programa que pida su nombre al usuario y lo escriba con la primera letra en mayúsculas y el resto en minúsculas. Por ejemplo, a partir de "nAcho" escribirá "Nacho".

Ejercicio propuesto 7.3.1.5: Crea un programa que pida su nombre al usuario tantas veces como sea necesario, hasta que escriba "nacho" (o el nombre que tú prefijas en el programa). Deberá permitir que ese nombre se introduzca tanto en mayúsculas como en minúsculas. Cuando introduzca el nombre correcto, se le saludará y terminará el programa.

7.3.2. *StringBuilder*

Por su parte, los métodos básicos de un **StringBuilder**, que representa una cadena de texto modificable, son:

Método	Cometido
length()	Devuelve la longitud (número de caracteres) de la cadena
setLength()	Modifica la longitud de la cadena (la trunca si hace falta)
charAt (int pos)	Devuelve el carácter que hay en una cierta posición
setCharAt(int pos, char letra)	Cambia el carácter que hay en una cierta posición
toString()	Devuelve el StringBuilder convertido en String
reverse()	Cambia el orden de los caracteres que forman la cadena
append(objeto)	Añade otra cadena, un número, etc. al final de la cadena
insert(int pos, objeto)	Añade otra cadena, un número, etc. en una cierta posición

Un ejemplo de su uso sería:

```
// EjemploStringBuilder.java
// Aplicación de ejemplo con StringBuilder
// Introducción a Java, Nacho Cabanes

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

        StringBuilder texto3 = new StringBuilder("Otra prueba");

        texto3.append(" mas");
        System.out.println( "Texto 3 es: " + texto3 );
        texto3.insert(2, "1");
        System.out.println( "Y ahora es: " + texto3 );
        texto3.reverse();
        System.out.println( "Y ahora: " + texto3 );

        System.out.println( "En mayúsculas: " +
            texto3.toString().toUpperCase() );
    }
}
```

El resultado de este programa sería el siguiente:

```
Texto 3 es: Otra prueba mas
Y ahora es: Otlra prueba mas
Y ahora: sam abeurp arl1O En mayúsculas: SAM ABEURP AR1TO
```

Ejercicio propuesto 7.3.2.1: Crea un programa que pida su nombre al usuario y lo escriba con al revés, usando un `StringBuilder`. Por ejemplo, a partir de "Nacho" escribiría "ohcaN".

Ejercicio propuesto 7.3.2.2: Crea un programa que pida su nombre al usuario y cambie la primera letra por una "A", excepto en el caso de que ya fuera una "A", y entonces se convertiría en una "B".

Ejercicio propuesto 7.3.2.3: Crea un programa que pida su nombre al usuario y cree una nueva cadena de texto formada por "Don " seguido del nombre. Hazlo de dos formas: primero concatenando dos cadenas y luego usando "insert" en un `StringBuilder`.

Ejercicio propuesto 7.3.2.4: Crea un programa que pida su nombre al usuario y convierta las letras impares a mayúsculas y las pares a minúsculas. Por ejemplo, a partir de la cadena "nAcho" se obtendría "NaChO".

7.4. Los arrays.

Imaginemos que tenemos que realizar cálculos estadísticos con 10 números que introduzca el usuario. Parece evidente que tiene que haber una solución más cómoda que definir 10 variables distintas y escribir 10 veces las instrucciones de avisar al usuario, leer los datos que teclee, y almacenar esos datos. Si necesitamos manejar 100, 1.000 o 10.000 datos, resulta todavía más claro que no es eficiente utilizar una variable para cada uno de esos datos.

Por eso se emplean los arrays (que en ocasiones también se llaman "matrices" o "vectores", por similitud con estas estructuras matemáticas, y que algunos autores traducen como "arreglos"). Un **array** es una variable que puede contener varios datos del mismo tipo. Para acceder a cada uno de esos datos, indicaremos su posición entre corchetes. Por ejemplo, si definimos una variable llamada "m" que contenga 10 números enteros, accederemos al primero de estos números como `m[0]`, el último como `m[9]` y el quinto como `m[4]` (se empieza a numerar a desde 0 y se termina en `n-1`). Veamos un ejemplo que halla la media de cinco números (con decimales, "double"):

```
// Array1.java
// Aplicación de ejemplo con Arrays
// Introducción a Java, Nacho Cabanes

class Array1 {
    public static void main( String args[] ) {
        double[] a = { 10, 23.5, 15, 7, 8.9 };
        double total = 0;
        int i;
```

```

    for (i=0; i<5; i++)
        total += a[i];

    System.out.println( "La media es:" );
    System.out.println( total / 5 );
}

```

Para definir la variable podemos usar **dos formatos**: "double a[]" (que es la sintaxis habitual en C y C++, no recomendada en Java) o bien "double[] a" (que es la sintaxis recomendada en Java, y posiblemente es una forma más "razonable" de escribir "la variable a es un array de doubles").

Lo habitual no será conocer los valores en el momento de teclear el programa, como hemos hecho esta vez. Será mucho más frecuente que los datos los teclee el usuario o bien que los leamos de algún fichero, los calculemos, etc. En este caso, tendremos que reservar el espacio, y los valores los almacenaremos a medida que vayamos conociéndolos.

Para ello, primero **declararíamos** que vamos a utilizar un array, así:

```
double[] datos;
```

y después **reservaríamos** espacio (por ejemplo, para 1.000 datos) con

```
datos = new double [1000];
```

Si conocemos el tamaño desde el primer momento, estos dos pasos se pueden dar en uno solo, así:

```
double[] datos = new double [1000];
```

y daríamos los **valores** de una forma similar a la que hemos visto en el ejemplo anterior:

```
datos[25] = 100 ; datos[0] = i*5 ; datos[j+1] = (j+5)*2;
```

Vamos a ver un ejemplo algo más completo, con tres arrays de números enteros, llamados a, b y c. A uno de ellos (a) le daremos valores al definirlo, otro lo definiremos en dos pasos (b) y le daremos fijos, y el otro lo definiremos en un paso y le daremos valores calculados a partir de ciertas operaciones:

```

// Array2.java
// Aplicación de ejemplo con Arrays
// Introducción a Java, Nacho Cabanes

class Array2 {
    public static void main( String args[] ) {

```

```

int i; // Para repetir con bucles "for"

// ----- Primer array de ejemplo, valores prefijados
int[] a = { 10, 12345, -15, 0, 7 };
System.out.print( "Los valores de a son: " );
for (i=0; i<5; i++)
    System.out.print( a[i] + " " );
System.out.println( );

// ----- Segundo array de ejemplo, valores uno por uno
int[] b;
b = new int [3];
b[0] = 15; b[1] = 132; b[2] = -1;
System.out.print( "Los valores de b son: " );
for (i=0; i<3; i++)
    System.out.print( b[i] + " " );
System.out.println( );

// ----- Tercer array de ejemplo, valores con "for"
int j = 4;
int[] c = new int[j];
for (i=0; i < j; i++)
    c[i] = (i+1)*(i+1);
System.out.print( "Los valores de c son: " );
for (i=0; i < j; i++)
    System.out.print( c[i] + " " );
System.out.println( );
}
}

```

Y si queremos que sea el usuario el que introduzca los valores y luego mostrar esos valores, lo podríamos conseguir de esta manera:

```

// Array3.java
// Leer de teclado y mostrar datos de un array
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Array3 {
    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);

        int[] datos = new int[5];
        for (int i=0; i<5; i++)
        {
            System.out.print( "Deme el dato "+i+": " );
            datos[i] = teclado.nextInt();
        }

        for (int i=4; i>=0; i--)
        {
            System.out.println( "El dato "+ i +" vale " + datos[i] );
        }
    }
}

```

```
}
```

Un programa no debería tener "**números mágicos**" prefijados dentro del fuente, como ese 5. Pueden no ser evidentes cuando se revise el programa más adelante, y pueden dar lugares a errores si tenemos más adelante que modificarlos y no lo hacemos en todos los sitios en los que aparezca. Por eso, será preferible usar "**constantes**" para esos números que indican valores especiales como el tamaño del array y que no van a cambiar durante el funcionamiento del programa, así:

```
// Array4.java
// Leer y mostrar datos de un array, usando constantes
// Introducción a Java, Nacho Cabanes

import java.util.Scanner;

class Array4 {
    public static void main( String args[] ) {

        Scanner teclado = new Scanner(System.in);
        final int TAMANYO = 5;

        int[] datos = new int[TAMANYO];
        for (int i=0; i<TAMANYO; i++)
        {
            System.out.println( "El dato " + i + " vale " + datos[i] );
        }
    }
}
```

Como se ve en ese ejemplo, para indicar que un dato será constante usaremos la palabra "**final**" y existe el convenio de usar nombres totalmente en **mayúsculas** para las constantes.

Ejercicio propuesto 7.4.1: Crea un programa que pida al usuario 5 números reales, que guardará en un array. Luego calculará y mostrará su media (la suma de todos los datos, dividida entre la cantidad de datos) y los valores que están por encima de la media.

Ejercicio propuesto 7.4.2: Crea un programa que pida al usuario 8 números enteros, los guarde en un array, halle y muestre el valor más alto que contiene (su máximo) y el valor más bajo que contiene (su mínimo). Pista: puedes empezar suponiendo que el primer valor es el máximo provisional; a partir de ahí, recorre todos los demás con un "for"; si alguno de los datos es mayor que el máximo provisional, pasará a ser el nuevo máximo; el mínimo se calculará de forma similar.

Ejercicio propuesto 7.4.3: Crea un programa que prepare un array que contenga la cantidad de días que hay en cada mes del año. A partir de entonces el usuario introducirá un número de mes (por ejemplo, 4 para abril) y

el programa le responderá cuántos días tiene ese mes. Se repetirá hasta que el usuario introduzca un mes menor que 1 o mayor que 12.

Ejercicio propuesto 7.4.4: Crea un programa que prepare un array que contenga el nombre de cada mes del año. El usuario introducirá un número de mes (por ejemplo, 4 para abril) y el programa le dirá el nombre de ese mes.

Ejercicio propuesto 7.4.5: Si has estudiado estadística, crea un programa que pida al usuario 10 números reales de doble precisión, los guarde en un array y luego muestre su media aritmética, varianza y desviación típica.

Ejercicio propuesto 7.4.6: Crea un programa que pida al usuario 10 números enteros largos, los guarde en un array y después pregunte qué número quiere buscar y le contestará si ese número forma parte de los 10 iniciales o no. Volverá a preguntar qué número desea buscar, hasta que el usuario introduzca "-1" para indicar que no quiere seguir buscando.

Ejercicio propuesto 7.4.7: Crea un programa que prepare un array de hasta 1000 números enteros. El usuario podrá elegir añadir un nuevo dato (tendrás que llevar la cuenta la cantidad de datos que ya hay introducidos), buscar para comprobar si aparece un cierto dato, o salir del programa.

Ejercicio propuesto 7.4.8: Si has estudiado lo que es un vector, crea un dos arrays de tres elementos, cada uno de los cuales representará un vector de tres dimensiones. Pide al usuario los datos para ambos vectores, luego muéstralos en la forma "(5, 7, -2)", después calcula (y muestra) su producto escalar y su producto vectorial.

Si no necesitamos una única serie de datos sino varias, podemos usar un **array bidimensional**. Por ejemplo, podríamos crear un array con dos filas de dos elementos cada una, así:

```
// ArrayBi.java
// Ejemplo de array bidimensional
// Introducción a Java, Nacho Cabanes

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

        int[][] datos = new int[2][2];
        datos[0][0] = 5;
        datos[0][1] = 1;
        datos[1][0] = -2;
        datos[1][1] = 3;

        for (int i=0; i<2; i++)
            for (int j=0; j<2; j++)
                System.out.println( "El dato " + i + "," + j +
                                     " vale " + datos[i][j] );
    }
}
```

}

Ejercicio propuesto 7.4.9: Si has estudiado lo que es una matriz, crea un array de 3x3 elementos, pide al usuario los datos de esas 9 posiciones y luego calcula el determinante de esa matriz bidimensional.

Ejercicio propuesto 7.4.10: Crea un programa que cree un array de 5x5 caracteres, lo rellene con puntos, excepto la segunda fila, que estará rellena de letras "A" y luego muestra el contenido del array en pantalla. Deberá aparecer algo como:

```
.....  
AAAAA  
.....  
.....  
.....
```

Ejercicio propuesto 7.4.11: Crea un programa que cree un array de 5x5 caracteres, lo rellene con puntos en todas las posiciones excepto aquellas en las que el número de fila es igual al número de columna, en las que pondrá un X. Luego muestra su contenido en pantalla, que deberá ser algo como:

```
X....  
.X...  
..X..  
...X.  
....X
```


8. Las Matemáticas y Java

Ya habíamos visto las operaciones matemáticas básicas: suma, resta, división, multiplicación. También alguna menos habitual, como el resto de una división.

Pero existen otras operaciones matemáticas que son muy habituales: raíces cuadradas, potencias, logaritmos, funciones trigonométricas (seno, coseno, tangente), generación de números al azar... Todas estas posibilidades están accesibles a través de la clase `java.lang.Math`. Vamos a comentar alfabéticamente las más importantes y luego veremos un ejemplo de su uso:

Función	Significado
<code>abs()</code>	Valor absoluto
<code>acos()</code>	Arcocoseno
<code>asin()</code>	Arcoseno
<code>atan()</code>	Arcotangente entre $-\pi/2$ y $\pi/2$
<code>atan2(,)</code>	Arcotangente entre $-\pi$ y π
<code>ceil()</code>	Entero mayor más cercano
<code>cos(double)</code>	Coseno
<code>exp()</code>	Exponencial
<code>floor()</code>	Entero menor más cercano
<code>log()</code>	Logaritmo natural (base e)
<code>max(,)</code>	Máximo de dos valores
<code>min(,)</code>	Mínimo de dos valores
<code>pow(,)</code>	Primer número elevado al segundo
<code>random()</code>	Número aleatorio (al azar) entre 0.0 y 1.0

rint(double)	Entero más próximo
round()	Entero más cercano (redondeo de la forma habitual)
sin(double)	sin(double)
sqrt()	Raíz cuadrada
tan(double)	Tangente
toDegrees(double)	Pasa de radianes a grados (a partir de Java 2)
toRadians()	Pasa de grados a radianes (a partir de Java 2)

También hay disponibles dos constantes: PI (relación entre el diámetro de una circunferencia y su longitud) y E (base de los logaritmos naturales).

Las funciones trigonométricas (seno, coseno, tangente, etc) miden en radianes, no en grados, de modo que más de una vez deberemos usar "toRadians" y "toDegrees" si nos resulta más cómodo pensar en grados.

Y un ejemplo, agrupando estas funciones por categorías, sería:

```
// Matem.java
// Ejemplo de matemáticas desde Java
// Introducción a Java, Nacho Cabanes

class Matem {

    public static void main( String args[] ) {

        System.out.print( "2+3 es " );
        System.out.println( 2+3 );
        System.out.print( "2*3 es " );
        System.out.println( 2*3 );
        System.out.print( "2-3 es " );
        System.out.println( 2-3 );
        System.out.print( "3/2 es " );
        System.out.println( 3/2 );
        System.out.print( "3,0/2 es " );
        System.out.println( 3.0/2 );
        System.out.print( "El resto de dividir 13 entre 4 es " );
        System.out.println( 13%4 );

        System.out.print( "Un número al azar entre 0 y 1: " );
        System.out.println( Math.random() );
    }
}
```

```

System.out.print( "Un número al azar entre 50 y 150: ");
System.out.println( (int)(Math.random()*100+50) );
System.out.print( "Una letra minúscula al azar: ");
System.out.println( (char)(Math.random()*26+'a') );

System.out.print( "Coseno de PI radianes: ");
System.out.println( Math.cos(Math.PI) );
System.out.print( "Seno de 45 grados: ");
System.out.println( Math.sin(Math.toRadians(45)) );
System.out.print( "Arco cuya tangente es 1: ");
System.out.println( Math.toDegrees(Math.atan(1)) );

System.out.print( "Raíz cuadrada de 36: ");
System.out.println( Math.sqrt(36) );
System.out.print( "Cinco elevado al cubo: ");
System.out.println( Math.pow(5.0,3.0) );
System.out.print( "Exponencial de 2: ");
System.out.println( Math.exp(2) );
System.out.print( "Logaritmo de 2,71828: ");
System.out.println( Math.log(2.71828) );

System.out.print( "Mayor valor entre 2 y 3: ");
System.out.println( Math.max(2,3) );
System.out.print( "Valor absoluto de -4,5: ");
System.out.println( Math.abs(-4.5) );
System.out.print( "Menor entero más cercano a -4,5: ");
System.out.println( Math.floor(-4.5) );
System.out.print( "Mayor entero más cercano a -4,5: ");
System.out.println( Math.ceil(-4.5) );
System.out.print( "Redondeando -4,5 con ROUND: ");
System.out.println( Math.round(-4.5) );
System.out.print( "Redondeando 4,5 con ROUND: ");
System.out.println( Math.round(4.5) );
System.out.print( "Redondeando -4,6 con RINT: ");
System.out.println( Math rint(-4.6) );
System.out.print( "Redondeando -4,5 con RINT: ");
System.out.println( Math rint(4.5) );

}
}

```

Su resultado es:

```

2+3 es 5
2*3 es 6
2-3 es -1
3/2 es 1
3,0/2 es 1.5
El resto de dividir 13 entre 4 es 1
Un número al azar entre 0 y 1: 0.9775498588615054
Un número al azar entre 50 y 150: 71
Una letra minúscula al azar: u
Coseno de PI radianes: -1.0
Seno de 45 grados: 0.7071067811865475
Arco cuya tangente es 1: 45.0
Raíz cuadrada de 36: 6.0
Cinco elevado al cubo: 125.0
Exponencial de 2: 7.38905609893065

```

Logaritmo de 2,71828: 0.999999327347282
Mayor valor entre 2 y 3: 3
Valor absoluto de -4,5: 4.5
Menor entero más cercano a -4,5: -5.0
Mayor entero más cercano a -4,5: -4.0
Redondeando -4,5 con ROUND: -4
Redondeando 4,5 con ROUND: 5
Redondeando -4,6 con RINT: -5.0
Redondeando -4,5 con RINT: 4.0

Ejercicio propuesto 8.1: Crea un programa que muestre en pantalla el resultado de lanzar dos dados (dos números del 1 al 6).

Ejercicio propuesto 8.2: Crea un programa que dé al usuario la oportunidad de adivinar un número del 1 al 100 en un máximo de 6 intentos. En cada pasada deberá avisar de si se ha pasado o se ha quedado corto. El número a adivinar se debe generar al azar.

Ejercicio propuesto 8.3: Crea un programa que cree un array de 5x5 caracteres, lo rellene con puntos en todas las posiciones y luego escriba una O en los puntos que cumplan $y = x^2$. (Nota: puedes calcular el cuadrado usando "pow" o simplemente con "x*x") Deberá verse algo como:

```
..0..  
.....  
.....  
.0...  
0....
```

Ejercicio propuesto 8.4: Crea un programa que calcule y muestre el coseno de los ángulos que van de 0 a 90 grados (ambos inclusive), de 5 en 5 grados.

Ejercicio propuesto 8.5: Crea un programa que pida al usuario un número real "a" y un número entero "b". Deberá calcular la raíz de orden "b" del número "a". Por ejemplo, si los números son 3 y 4, tendrás que calcular (y mostrar) la raíz cuarta de 3. Deberá repetirse hasta que el número "a" sea 0. Si el número "a" es negativo y "b" es par se deberá mostrar un aviso "No se puede calcular esa raíz". (Pista: recuerda que para calcular la raíz "b" del número "a", basta con elevar "a" a "1/b".).

9. Contacto con las funciones.

9.1.Descomposición modular

En ocasiones, nuestros programas contendrán operaciones repetitivas. Crear funciones nos ayudará a que dichos programas sean más fáciles de crear y más robustos (con menos errores). Vamos a verlo con un ejemplo...

Imaginemos que queremos calcular la longitud de una circunferencia a partir de su radio, y escribirla en pantalla con dos cifras decimales. No es difícil: por una parte, la longitud de una circunferencia se calcula con $2 * \pi * \text{radio}$; por otra parte, para conservar sólo dos cifras decimales, lo podemos hacer de varias formas, una de las cuales consiste en multiplicar por 100, quedarnos con la parte entera del número y volver a dividir entre 100:

```
// FuncionesPrevio1.java
// Ejemplo previo 1 de la conveniencia de usar funciones
// para evitar código repetitivo

import java.io.*;

class FuncionesPrevio1 {
    public static void main( String args[] ) {

        int radio = 4;

        double longCircunf = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longCircunf * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longConDosDecimales);
    }
}
```

Su resultado sería

La longitud de la circunferencia de radio 4 es 25.13

Si ahora queremos hacerlo para 5 circunferencias, podemos repetir esa misma estructura varias veces:

```
// FuncionesPrevio2.java
// Ejemplo previo 2 de la conveniencia de usar funciones
// para evitar código repetitivo

import java.io.*;

class FuncionesPrevio2 {
    public static void main( String args[] ) {

        int radio1 = 4;
```

```

double longCircunf1 = 2 * 3.1415926535 * radio1;
double longConDosDecimales1 =
    Math.round(longCircunf1 * 100) / 100.0;
System.out.println( "La longitud de la circunferencia " +
    "de radio " + radio1 + " es " + longConDosDecimales1);

int radio2 = 6;

double longCircunf2 = 2 * 3.1415926535 * radio2;
double longConDosDecimales2 =
    Math.round(longCircunf2 * 100) / 100.0;
System.out.println( "La longitud de la circunferencia " +
    "de radio " + radio2 + " es " + longConDosDecimales2);

int radio3 = 8;

double longCircunf3 = 2 * 3.1415926535 * radio3;
double longConDosDecimales3 =
    Math.round(longCircunf3 * 100) / 100.0;
System.out.println( "La longitud de la circunferencia " +
    "de radio " + radio3 + " es " + longConDosDecimales3);

int radio4 = 10;

double longCircunf4 = 2 * 3.1415926535 * radio4;
double longConDosDecimales4 =
    Math.round(longCircunf4 * 100) / 100.0;
System.out.println( "La longitud de la circunferencia " +
    "de radio " + radio4 + " es " + longConDosDecimales4);

int radio5 = 111;

double longCircunf5 = 2 * 3.1415926535 * radio5;
double longConDosDecimales5 =
    Math.round(longCircunf5 * 100) / 100.0;
System.out.println( "La longitud de la circunferencia " +
    "de radio " + radio5 + " es " + longConDosDecimales5);
    }
}

```

Pero un programa tan repetitivo es muy propenso a errores: tanto si escribimos todo varias veces como si copiamos y pegamos, es fácil que nos equivoquemos en alguna de las operaciones, usando el nombre de una variable que no es la que debería ser. Por ejemplo, podría ocurrir que escribiéramos "double longCircunf5 = 2 * 3.1415926535 * radio4;" (en el último fragmento no aparece "radio5" sino "radio4"). El programa se comportaría de forma incorrecta, y este error podría ser muy difícil de descubrir.

La alternativa es crear una "función": un bloque de programa que tiene un nombre, que recibe ciertos datos, y que puede incluso devolvernos un resultado. Por ejemplo, para el programa anterior, podríamos crear una función llamada "escribirLongCircunf" (escribir la longitud de la circunferencia), que recibiría un dato (el radio, que será un número entero) y dará todos los pasos que daba nuestro primer "main":

```

public static void escribirLongCircunf( int radio ) {
    double longCircunf = 2 * 3.1415926535 * radio;
    double longConDosDecimales =
        Math.round(longCircunf * 100) / 100.0;
    System.out.println( "La longitud de la circunferencia " +
        "de radio " + radio + " es " + longConDosDecimales);
}

```

Y desde "main" (el cuerpo del programa) usaríamos esa función tantas veces como quisiéramos:

```

public static void main( String args[] ) {
    escribirLongCircunf(4);
    escribirLongCircunf(6);
    ...
}

```

De modo que el programa completo quedaría:

```

// Funciones01.java
// Primer ejemplo de funciones

import java.io.*;

class Funciones01 {

    public static void escribirLongCircunf( int radio ) {
        double longCircunf = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longCircunf * 100) / 100.0;
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longConDosDecimales);
    }

    public static void main( String args[] ) {
        escribirLongCircunf(4);
        escribirLongCircunf(6);
        escribirLongCircunf(8);
        escribirLongCircunf(10);
        escribirLongCircunf(111);
    }
}

```

Por ahora, hasta que sepamos un poco más, daremos por sentado que todas las funciones tendrán que ser "**public**" y "**static**".

Ejercicio propuesto 9.1.1: Crea una función llamada "borrarPantalla", que borre la pantalla dibujando 25 líneas en blanco. No debe devolver ningún valor. Crea también un "main" que permita probarla.

Ejercicio propuesto 9.1.2: Crea una función llamada "dibujarCuadrado3x3", que dibuje un cuadrado formato por 3 filas con 3 asteriscos cada una. Crea también un "main" para comprobar que funciona correctamente.

9.2. Parámetros

Nuestra función "escribirLongCircunf" recibía un dato entre paréntesis, el radio de la circunferencia. Estos datos adicionales se llaman "**parámetros**", y pueden ser varios, cada uno indicado con su tipo de datos y su nombre.

Ejercicio propuesto 9.2.1: Crea una función que dibuje en pantalla un cuadrado del ancho (y alto) que se indique como parámetro. Completa el programa con un "main" que permita probarla.

Ejercicio propuesto 9.2.2: Crea una función que dibuje en pantalla un rectángulo del ancho y alto que se indiquen como parámetros. Completa el programa con un "main" que permita probarla.

9.3. Valor de retorno

Las funciones "**void**", como nuestra "escribirLongCircunf" y como el propio cuerpo del programa ("main") son funciones que dan una serie de pasos y no devuelven ningún resultado. Este tipo de funciones se suelen llamar "**procedimientos**" o "**subrutinas**".

```
public static void saludar( ) {  
    System.out.println( "Bienvenido");  
    System.out.println( "Comenzamos...");  
}
```

Por el contrario, las funciones matemáticas suelen dar una serie de pasos y devolver un resultado, por lo que no serán "void", sino "**int**", "double" o del tipo que corresponda al dato que devuelven. Por ejemplo, podríamos calcular la superficie de un círculo así:

```
public static double superfCirculo( int radio ) {  
    double superf = 3.1415926535 * radio * radio;  
    double superfConDosDecimales =  
        Math.round(superf * 100) / 100.0;  
    return superfConDosDecimales;  
}
```

O descomponer la parte matemática del ejemplo anterior (el cálculo de la longitud de la circunferencia) en una función independiente, así:

```
public static double longCircunf( int radio ) {  
    double longitud = 2 * 3.1415926535 * radio;  
}
```



```

double longConDosDecimales =
    Math.round(longitud * 100) / 100.0;
return longConDosDecimales;
}

```

Y un programa completo que usara todas esas funciones quedaría:

```

// Funciones02.java
// Segundo ejemplo de funciones

import java.io.*;

class Funciones02 {

    public static double longCircunf( int radio ) {
        double longitud = 2 * 3.1415926535 * radio;
        double longConDosDecimales =
            Math.round(longitud * 100) / 100.0;
        return longConDosDecimales;
    }

    public static double superfCirculo( int radio ) {
        double superf = 3.1415926535 * radio * radio;
        double superfConDosDecimales =
            Math.round(superf * 100) / 100.0;
        return superfConDosDecimales;
    }

    public static void saludar( ) {
        System.out.println( "Bienvenido");
        System.out.println( "Comenzamos...");
    }

    public static void escribirLongCircunf( int radio ) {
        System.out.println( "La longitud de la circunferencia " +
            "de radio " + radio + " es " + longCircunf( radio ));
    }

    public static void main( String args[] ) {
        saludar();
        escribirLongCircunf(4);
        escribirLongCircunf(6);
        escribirLongCircunf(8);
        escribirLongCircunf(10);
        escribirLongCircunf(111);
        System.out.println( "La superficie del círculo " +
            "de radio 5 es " + superfCirculo(5));
    }
}

```

Volveremos más adelante a las funciones, para formalizar un poco lo que hemos aprendido, y también para ampliarlo, pero ya tenemos suficiente para empezar a practicar.

Ejercicio propuesto 9.3.1: Crea una función que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Pruébala para calcular el cubo de 3.2 y el de 5.

Ejercicio propuesto 9.3.2: Crea una función que calcule el menor de dos números enteros que recibirá como parámetros. El resultado será otro número entero.

Ejercicio propuesto 9.3.3: Crea una función que devuelva la primera letra de una cadena de texto. Prueba esta función para calcular la primera letra de la frase "Hola".

Ejercicio propuesto 9.3.4: Crea una función que devuelva la última letra de una cadena de texto. Pruébala para calcular la última letra de la frase "Hola".

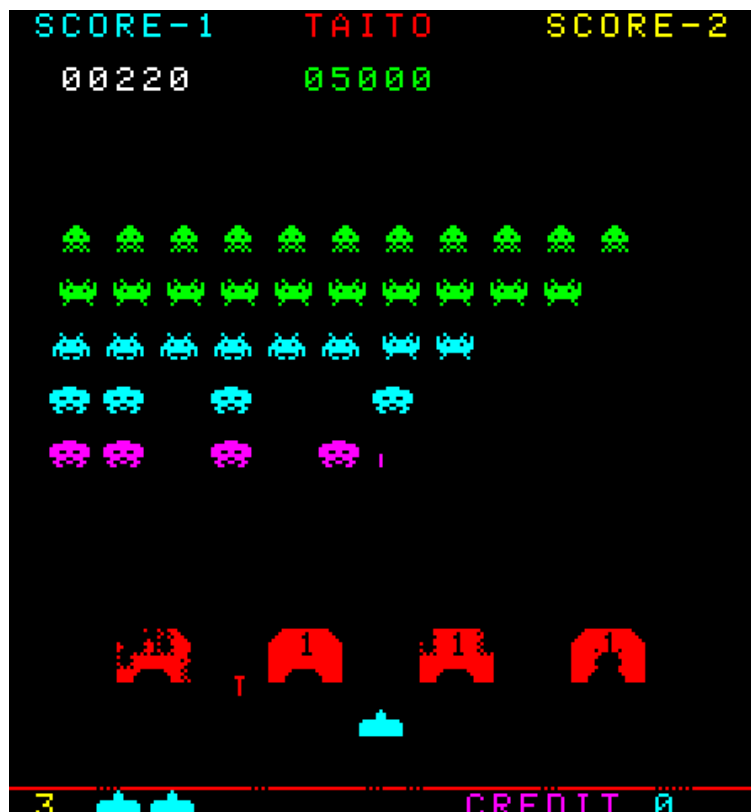
Ejercicio propuesto 9.3.5: Crea una función "esPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.

10. Clases en Java.

Cuando tenemos que realizar un proyecto grande, será necesario descomponerlo en varios subprogramas, de forma que podamos repartir el trabajo entre varias personas (pero la descomposición no debe ser arbitraria: por ejemplo, será deseable que cada bloque tenga unas responsabilidades claras).

La forma más recomendable de descomponer un proyecto será tratar de verlo como una serie de "objetos" que colaboran entre ellos, cada uno de los cuales tiene unas ciertas responsabilidades.

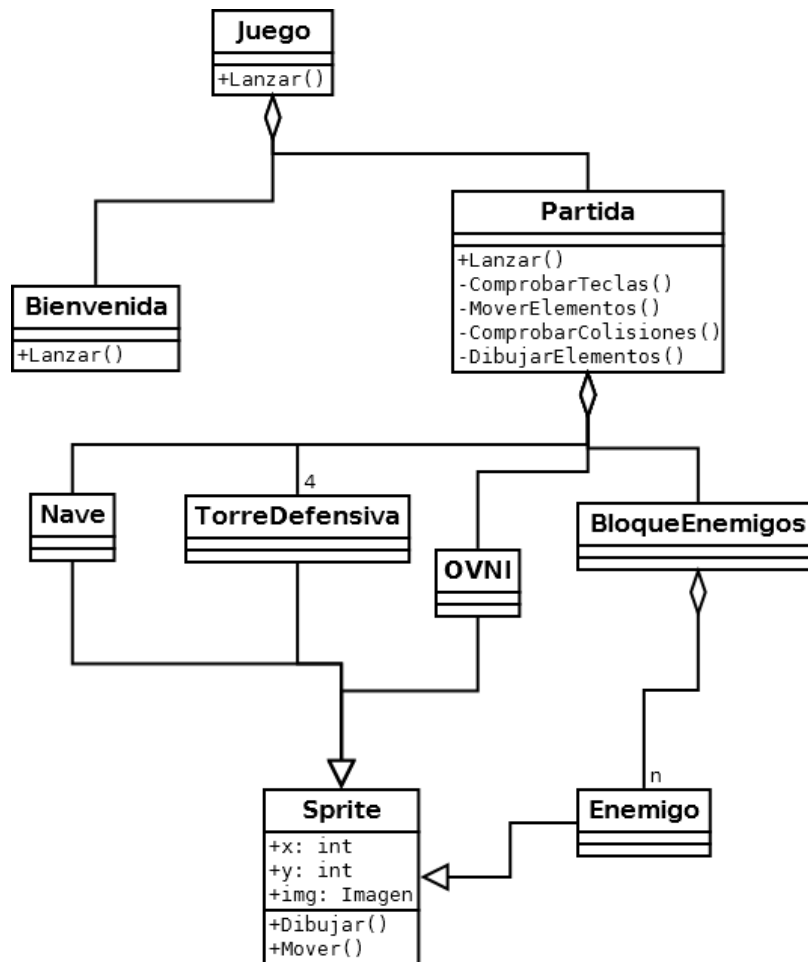
Como ejemplo, vamos a dedicar un momento a pensar qué elementos ("objetos") hay en un juego como el clásico Space Invaders:



De la pantalla anterior, se puede observar que nosotros manejamos una "nave", que se esconde detrás de "torres defensivas", y que nos atacan (nos disparan) "enemigos". Además, estos enemigos no se mueven de forma independiente, sino como un "bloque". En concreto, hay cuatro "tipos" de enemigos, que no se diferencian en su comportamiento, pero sí en su imagen. También, aunque no se ve en la pantalla anterior, en ocasiones aparece un "OVNI" en la parte superior de la pantalla, que nos permite obtener puntuación extra. También hay un "marcador", que muestra la puntuación y el record. Y antes y después de cada

"partida", regresamos a una pantalla de "bienvenida", que muestra una animación que nos informa de cuántos puntos obtenemos al destruir cada tipo de enemigo.

Para diseñar cómo descomponer el programa, se suele usar la ayuda de "diagramas de clases", que muestran de una manera visual qué objetos son los que interaccionan para, entre todos ellos, formar nuestro proyecto. En el caso de nuestro "Space Invaders", un diagrama de clases simplificado podría ser algo como:



Algunos de los detalles que se pueden leer de ese diagrama son:

- La clase principal de nuestro proyecto se llama "Juego" (el diagrama típicamente se leerá de arriba a abajo).
- El juego contiene una "Bienvenida" y una "Partida" (ese relación de que un objeto "contiene" a otros se indica mediante un rombo en el extremo de la línea que une ambas clases, junto a la clase "contenedora").
- En una partida participan una "Nave", cuatro "Torres" defensivas, un "BloqueDeEnemigos" formado por varios "Enemigos" (que, a su vez,

podrían ser de tres tipos distintos, pero no afinaremos tanto por ahora) y un "Ovni".

- Tanto la "Nave" como las "Torres", los "Enemigos" y el "Ovni" son tipos concretos de "Sprite" (esa relación entre un objeto más genérico y uno más específico se indica con las puntas de flecha, que señalan al objeto más genérico).
- Un "Sprite" es una figura gráfica de las que aparecen en el juego. Cada sprite tendrá detalles (**atributos**) como una "imagen" y una posición, dada por sus coordenadas "x" e "y". Será capaz de hacer operaciones (**métodos**) como "dibujarse" o "moverse" a una nueva posición. Cuando se programa toda esta estructura de clases, los atributos serán variables, mientras que los "métodos" serán funciones. Los subtipos de sprite **heredarán** las características de esta clase. Por ejemplo, como un Sprite tiene una coordenada X y una Y, también lo tendrá el OVNI, que es una subclase de Sprite.
- El propio juego también tendrá métodos como "comprobarTeclas" (para ver qué teclas ha pulsado el usuario), "moverElementos" (para actualizar el movimiento de los elementos que deban moverse por ellos mismos), "comprobarColisiones" (para ver si dos elementos chocan, como un disparo y un enemigo, y actualizar el estado del juego según corresponda), o "dibujarElementos" (para mostrar en pantalla todos los elementos actualizados).

En este punto, podríamos empezar a repartir trabajo: una persona se podría encargar de crear la pantalla de bienvenida, otra de la lógica del juego, otra del movimiento de los enemigos, otra de las peculiaridades de cada tipo de enemigo, otra del OVNI...

Nosotros no vamos a hacer proyectos tan grandes (al menos, no todavía), pero sí empezaremos a crear proyectos sencillos en los que colaboren varias clases, que permitan sentar las bases para proyectos más complejos, y también entender algunas peculiaridades de los temas que veremos a continuación, como el manejo de ficheros en Java.

10.2. Varias clases en Java

En Java podemos definir **varias clases** dentro de un mismo fichero, con la única condición de que sólo una de esas clases sea declarada como "pública". En un caso

general, lo más correcto será definir **cada clase en un fichero**. Aun así, vamos a ver primero un ejemplo que contenga dos clases en un solo fichero

```
// DosClases.java
// Primer ejemplo de una clase nuestra
// que accede a otra también nuestra,
// ambas definidas en el mismo fichero
// Introducción a Java, Nacho Cabanes

class Principal {

    public static void main( String args[] ) {
        Secundaria s = new Secundaria();

        s.saluda(); // Saludo de "Secundaria"
        saluda(); // Saludo de "Principal"
    }

    public static void saluda() {
        System.out.println( "Saludando desde " );
    }
}

// -----

class Secundaria {

    public void saluda() {
        System.out.println( "Saludando desde " );
    }
}
```

Como siempre, hay cosas que comentar:

- En este fuente hay **dos clases**, una llamada "Principal" y otra llamada "Secundaria".
- La clase "Secundaria" sólo tiene **un método**, llamado "saluda", mientras que la clase "Principal" tiene **dos métodos**: "main" (el cuerpo de la aplicación) y otro llamado "saluda", al igual que el de "Secundaria".
- Ambos métodos "**saluda**" se limitan a mostrar un mensaje en pantalla, que es distinto en cada caso.
- En el método "**main**", hacemos 3 cosas:
 - Primero definimos y creamos un objeto de la clase "Secundaria". Esto lo podríamos conseguir en dos pasos, definiendo primero el objeto con "Secundaria s" y creando después el objeto con "s = new

Secundaria()", o bien podemos hacer ambas cosas en un solo paso, como ya habíamos hecho con las variables sencillas.

- Después llamamos al método "saluda" de dicho objeto, con la expresión "s.saluda()"
- Finalmente, llamamos al método "saluda" de la propia clase "Principal", escribiendo solamente "saluda()".

Para **compilar** este programa desde línea de comandos, teclearíamos, como siempre:

```
javac DosClases.java
```

y entonces se crearían **dos ficheros** llamados

```
Principal.class  
Secundaria.class
```

Podríamos **probar** el resultado tecleando

```
java Principal
```

Y en pantalla se mostraría:

```
Saludando desde <Principal>  
Saludando desde <Secundaria>
```

Si usamos Geany como editor, nos interesará que el propio programa se llame Principal.java, igual que la que va a ser la clase que habrá que ejecutar posteriormente; de lo contrario, cuando pidamos lanzar el programa, se buscaría un fichero DosClases.class (porque nuestro fuente era DosClases.java), pero ese fichero no existe...

Ahora vamos a ver un ejemplo en el que las dos clases están en **dos ficheros distintos**. Tendremos una clase "sumador" que sea capaz de sumar dos números (no es gran cosa, sabemos hacerlo sin necesidad de crear "clases a propósito", pero nos servirá como ejemplo) y tendremos también un programa principal que la utilice.

La clase "Sumador", con un único método "calcularSuma", que acepte dos números enteros y devuelva otro número entero, sería:

```
// Sumador.java
// Segundo ejemplo de una clase nuestra
// que accede a otra también nuestra.
// Esta es la clase auxiliar, llamada
// desde "UsaSumador.java"
// Introducción a Java, Nacho Cabanes

class Sumador {

    public int calcularSuma( int a, int b ) {
        return a+b;
    }

}
```

Por otra parte, la clase "UsaSumador" emplearía un objeto de la clase "Sumador", llamado "suma", desde su método "main", así:

```
// UsaSumador.java
// Segundo ejemplo de una clase nuestra
// que accede a otra también nuestra.
// Esta es la clase principal, que
// accede a "Sumador.java"
// Introducción a Java, Nacho Cabanes

class UsaSumador {

    public static void main( String args[] ) {

        Sumador suma = new Sumador();

        System.out.println( "La suma de 30 y 55 es" );
        System.out.println( suma.calcularSuma (30,55) );
    }

}
```

Para **compilar** estos dos fuentes desde línea de comandos, si tecleamos directamente

```
javac usaSumador.java
```


recibiríamos como respuesta un mensaje de error que nos diría que no existe la clase Sumador:

```
UsaSumador.java:13: Class Sumador not found.  
Sumador suma = new Sumador();  
^  
UsaSumador.java:13: Class Sumador not found.  
Sumador suma = new Sumador();  
^  
2 errors
```

La **forma correcta** sería compilar primero "Sumador" y después "UsaSumador", para después ya poder probar el resultado:

```
javac Sumador.java  
javac UsaSumador.java  
java UsaSumador
```

La respuesta, como es de esperar, sería:

```
La suma de 30 y 55 es  
85
```

Si usamos como entorno **NetBeans**, también podemos crear programas formados por varios fuentes. Los pasos serían los siguientes:

- Crear un proyecto nuevo (menú "Archivo", opción "Proyecto nuevo")
- Indicar que ese proyecto es una "Aplicación Java".
- Elegir un nombre para esa aplicación (por ejemplo, "EjemploSumador"), y, si queremos, una carpeta (se nos propondrá la carpeta de proyectos de NetBeans).
- Añadir una segunda clase a nuestra aplicación (menú "Archivo", opción "Archivo Nuevo"). Escogeremos que ese archivo esté dentro de nuestro proyecto actual ("EjemploSumador") y que sea una "Clase Java". Después se nos preguntará el nombre que deseamos para la clase (por ejemplo "Sumador").
- Entonces completaremos el código que corresponde a ambas clases, preferiblemente empezando por las clases que son necesitadas por otras (en nuestro caso, haríamos "Sumador" antes que "EjemploSumador").

- Finalmente, haremos clic en el botón de "Ejecutar" nuestro proyecto, para comprobar el resultado.

Ejercicio propuesto 10.2.1: Crea una clase "LectorTeclado", para simplificar la lectura de datos desde teclado. Esta clase tendrá un método "pedir", que recibirá como parámetro el texto de aviso que se debe mostrar al usuario, y que devolverá la cadena de texto introducida por el usuario. Crea también una clase "PruebaTeclado", que use la anterior.

10.3. Herencia

Hemos comentado que unas clases podían "heredar" atributos y métodos de otras clases. Vamos a ver cómo se refleja eso en un programa. Crearemos un "escritor de textos" y después lo mejoraremos creando un segundo escritor que sea capaz además de "adornar" los textos poniendo asteriscos antes y después de ellos. Finalmente, crearemos un tipo de escritor que sólo escriba en mayúsculas...

```
// Herencia.java
// Primer ejemplo de herencia entre clases,
// todas definidas en el mismo fichero
// Introducción a Java, Nacho Cabanes

class Escritor {
    public static void escribe(String texto) {
        System.out.println( texto );
    }
}

class EscritorAmpliado extends Escritor {
    public static void escribeConAsteriscos(String texto) {
        escribe( "***" + texto + "***" );
    }
}

class EscritorMayusculas extends Escritor {
    public static void escribe(String texto) {
        Escritor.escribe( texto.toUpperCase() );
    }
}

// -----

class Herencia {

    public static void main( String args[] ) {

        Escritor e = new Escritor();
        EscritorAmpliado eAmp = new EscritorAmpliado();
    }
}
```

```

    EscritorMayusculas eMays = new EscritorMayusculas();

    e.escribe("El primer escritor sabe escribir");
    eAmp.escribe("El segundo escritor también");
    eAmp.escribeConAsteriscos("y rodear con asteriscos");
    eMays.escribe("El tercero sólo escribe en mayúsculas");
}
}

```

Veamos qué hemos hecho:

- Creamos una primera **clase** de objetos. La llamamos "Escritor" y sólo sabe hacer una cosa: escribir. Mostrará en pantalla el texto que le indiquemos, usando su método "escribe".
- Después creamos una clase que **amplía** las posibilidades de ésta. Se llama "EscritorAmpliado", y se basa (*extends*) en Escritor. Como "hereda" las características de un Escritor, también "sabrás escribir" usando el método "escribe", sin necesidad de que se lo volvamos a decir.
- De hecho, también le hemos añadido una nueva posibilidad (la de "escribir con asteriscos"), y al definirla podemos usar "escribe" sin ningún problema, aprovechando que un EscritorAmpliado es un tipo de Escritor.
- Después creamos una tercera clase, que en vez de ampliar las posibilidades de "Escritor" lo que hace es basarse (*extends*) en ella pero **cambiando** el comportamiento (sólo escribirá en mayúsculas). En este caso, no añadimos nada, sino que reescribimos el método "escribe", para indicarle que debe hacer cosas distintas (esto es lo que se conoce como **polimorfismo**: el mismo nombre para dos elementos distintos -en este caso dos funciones-, cuyos comportamientos no son iguales). Para rizar el rizo, en el nuevo método "escribe" no usamos el típico "System.out.println" (lo podíamos haber hecho perfectamente), sino que nos apoyamos en el método "escribe" que acabábamos de definir para la clase "Escritor".
- Finalmente, en "main", creamos un **objeto** de cada clase (usando la palabra "new" y los probamos.

El resultado de este programa es el siguiente:

```

El primer escritor sabe escribir
El segundo escritor también
**y rodear con asteriscos**
EL TERCERO SÓLO ESCRIBE EN MAYÚSCULAS

```

Ejercicio propuesto 10.3.1: Crea una nueva clase "EscritorMayusculasEspaciado" que se apoye en "EscritorMayusculas", pero reemplace cada espacio en blanco por tres espacios antes de escribir en pantalla.

10.4. Ocultación de detalles

En general, será deseable que los detalles internos (los "atributos", las variables) de una clase no sean accesibles desde el exterior. En vez de hacerlos públicos, usaremos "métodos" (funciones) para acceder a su valor y para cambiarlo.

Esta forma de trabajar tiene como ventaja que podremos cambiar los detalles internos de nuestra clase (para hacerla más rápida o que ocupe menos memoria, por ejemplo) sin que afecte a los usuarios de nuestra clase, que la seguirán manejando "como siempre", porque su parte visible sigue siendo la misma.

De hecho, podremos distinguir tres niveles de visibilidad:

- **Público** (public), para métodos o atributos que deberán ser visibles. En general, las funciones que se deban poder utilizar desde otras clases serán visible, mientras que procuraremos que los estén ocultos.
- **Privado** (private), para lo que no deba ser accesible desde otras clases, como los atributos o algunas funciones auxiliares.
- **Protegido** (protected), que es un caso intermedio: si declaramos un atributo como privado, no será accesible desde otras clases, ni siquiera las que heredan de la clase actual. Pero generalmente será preferible que las clases "hijas" de la actual sí puedan acceder a los atributos que están heredando de ella. Por eso, es habitual declarar los atributos como "protected", que equivale a decir "será privado para todas las demás clases, excepto para las que hereden de mí".

Un ejemplo sería

```
// Getters.java
// Segundo ejemplo de herencia entre clases,
//  todas definidas en el mismo fichero
// Incluye getters y setters
// Introducción a Java, Nacho Cabanes

class Escritor {
    public static void escribe(String texto) {
```

```

        System.out.println( texto );
    }
}

class EscritorMargen extends Escritor {
    static byte margen = 0;

    public static void escribe(String texto) {
        for (int i=0; i < margen; i++)
            System.out.print(" ");
        System.out.println( texto );
    }

    public static int getMargen() {
        return margen;
    }

    public static void setMargen(int nuevoMargen) {
        margen = (byte) nuevoMargen;
    }
}

// -----

class Getters {

    public static void main( String args[] ) {

        Escritor e = new Escritor();
        EscritorMargen e2 = new EscritorMargen();

        e.escribe("El primer escritor sabe escribir");
        e2.setMargen( 5 );
        e2.escribe("El segundo escritor también, con margen");
    }
}

```

Ejercicio propuesto 10.4.1: Crea una nueva clase "EscritorDosMargenes" que se base en "EscritorMargen", añadiéndole un margen derecho. Si el texto supera el margen derecho (suponiendo 80 columnas de anchura de pantalla), deberá continuar en la línea siguiente.

10.5. Sin "static"

Hasta ahora, siempre hemos incluido la palabra **"static"** antes de cada función, e incluso de los atributos. Realmente, esto no es necesario. Ahora que ya sabemos lo que son las clases y cómo se definen los objetos que pertenecen a una cierta clase, podemos afinar un poco más:

La palabra "static" se usa para indicar que un método o un atributo es igual para todos los objetos de una clase. Pero esto es algo que casi no ocurre en "el mundo

real". Por ejemplo, podríamos suponer que el atributo "cantidadDeRuedas" de una clase "coche" podría ser "static", y tener el valor 4 para todos los coches... pero en el mundo real existe algún coche de 3 ruedas, así como limusinas con más de 4 ruedas. Por eso, habíamos usado "static" cuando todavía no sabíamos nada sobre clases, pero prácticamente ya no lo volveremos a usar a partir de ahora, que crearemos objetos usando la palabra "new".

Podemos reescribir el ejemplo anterior sin usar "static", así

```
// Getters2.java
// Segundo ejemplo de herencia entre clases,
//  todas definidas en el mismo fichero
// Incluye getters y setters
// Versión sin "static"
// Introducción a Java, Nacho Cabanes

class Escritor {
    public void escribe(String texto) {
        System.out.println( texto );
    }
}

class EscritorMargen extends Escritor {
    byte margen = 0;

    public void escribe(String texto) {
        for (int i=0; i < margen; i++)
            System.out.print(" ");
        System.out.println( texto );
    }

    public int getMargen() {
        return margen;
    }

    public void setMargen(int nuevoMargen) {
        margen = (byte) nuevoMargen;
    }
}

// -----

class Getters2 {

    public static void main( String args[] ) {

        Escritor e = new Escritor();
        EscritorMargen e2 = new EscritorMargen();

        e.escribe("El primer escritor sabe escribir");
        e2.setMargen( 5 );
        e2.escribe("El segundo escritor también, con margen");
    }
}
```

```
}
```

Ejercicio propuesto 10.5.1: Crea una versión del ejercicio 10.4.1, que no utilice "static".

10.6. Constructores

Nos puede interesar dar valores iniciales a los atributos de una clase. Una forma de hacerlo es crear un método (una función) llamado "Inicializar", que sea llamado cada vez que creamos un objeto de esa clase. Pero esto es algo tan habitual que ya está previsto en la mayoría de lenguajes de programación actuales: podremos crear "**constructores**", que se lanzarán automáticamente al crear el objeto. La forma de definirlos es con una función que se llamará igual que la clase, que no tendrá ningún tipo devuelto (ni siquiera "void") y que puede recibir parámetros. De hecho, podemos incluso crear varios constructores alternativos, con distinto número o tipo de parámetros:

```
// Constructores.java
// Ejemplo de clases con constructores
// Introducción a Java, Nacho Cabanes

class Escritor {
    protected String texto;

    public Escritor(String nuevoTexto) {
        texto = nuevoTexto;
    }

    public Escritor() {
        texto = "";
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto(String nuevoTexto) {
        texto = nuevoTexto;
    }

    public void escribe() {
        System.out.println( texto );
    }
}

class EscritorMargen extends Escritor {
    byte margen = 0;
}
```

```

public EscritorMargen(String nuevoTexto) {
    texto = nuevoTexto;
}

public void escribe() {
    for (int i=0; i < margen; i++)
        System.out.print(" ");
    System.out.println( texto );
}

public int getMargen() {
    return margen;
}

public void setMargen(int nuevoMargen) {
    margen = (byte) nuevoMargen;
}
}

// -----

class Constructores {

    public static void main( String args[] ) {

        Escritor e = new Escritor("Primer escritor");
        EscritorMargen e2 =
            new EscritorMargen("Segundo escritor");

        e.escribe();
        e2.setMargen( 5 );
        e2.escribe();
    }

}

```

También existen los "**destructores**", que se llamarían cuando un objeto deja de ser utilizado, y se podrían aprovechar para cerrar ficheros, liberar memoria que hubiéramos reservado nosotros de forma manual, etc., pero su uso es poco habitual para un principiante, y menos aún en un lenguaje moderno como Java, que incluye un "recolector de basura" automático para liberar las zonas de memoria que ya no se usan, así que no los veremos por ahora.

Ejercicio propuesto 10.6.1: Crea una nueva versión de la clase "EscritorDosMargenes" que use un constructor para indicarle el texto, el margen izquierdo y el margen derecho.

11. Ficheros

11.1. ¿Por qué usar ficheros?

Con frecuencia tendremos que guardar los datos de nuestro programa para poderlos recuperar más adelante. Hay varias formas de hacerlo. Una de ellas son los ficheros, que son relativamente sencillos. Otra forma más eficiente cuando es un volumen de datos muy elevado es usar una base de datos, que veremos más adelante.

11.2. Escribir en un fichero de texto

Un primer tipo de ficheros, que resulta sencillo de manejar, son los **ficheros de texto**. Son ficheros que podremos crear desde un programa en Java y leer con cualquier editor de textos, o bien crear con un editor de textos y leer desde un programa en Java, o bien usar un programa tanto para leer como para escribir.

Para manipular ficheros, siempre tendremos que dar tres pasos:

- Abrir el fichero
- Guardar datos o leer datos
- Cerrar el fichero

Hay que recordar siempre esos tres pasos: si no guardamos o leemos datos, no hemos hecho nada útil; si no abrimos fichero, obtendremos un mensaje de error al intentar acceder a su contenido; si no cerramos el fichero (un error frecuente), puede que realmente no se llegue a guardar ningún dato, porque no se vacíe el "buffer" (la memoria intermedia en que se quedan los datos preparados hasta el momento de volcarlos a disco).

En el caso de un fichero de texto, no escribiremos con "println", como hacíamos en pantalla, sino con "**write**". Cuando queramos avanzar a la línea siguiente, deberemos usar "newline()":

```
ficheroSalida.write("Hola");  
ficheroSalida.newLine();
```

Por otra parte, para cerrar el fichero (lo más fácil, pero lo que más se suele olvidar), usaremos "close", mientras que para abrir usaremos un BufferedWriter, que se apoya en un FileWriter, que a su vez usa un "File" al que se le indica el nombre del fichero. Es menos complicado de lo que parece: ###

```
// FicheroTextoEscribir.java
```

```
// Ejemplo de escritura en un fichero de texto
// Introducción a Java, Nacho Cabanes

import java.io.*;

class FicheroTextoEscribir
{
    public static void main( String[] args )
    {
        System.out.println("Volcando a fichero de texto...");

        try
        {
            BufferedWriter ficheroSalida = new BufferedWriter(
                new FileWriter(new File("fichero.txt")));

            ficheroSalida.write("Hola");
            ficheroSalida.newLine();
            ficheroSalida.write("Este es");
            ficheroSalida.write(" un fichero de texto");
            ficheroSalida.newLine();

            ficheroSalida.close();
        }
        catch (IOException errorDeFichero)
        {
            System.out.println(
                "Ha habido problemas: " +
                errorDeFichero.getMessage() );
        }
    }
}
```

Como se ve en este ejemplo, todo el bloque que accede al fichero deberá estar encerrado también en un bloque try-catch, para interceptar errores.

Ejercicio propuesto 11.2.1: Crea un programa que pida al usuario que introduzca frases, y guarde todas ellas en un fichero de texto. Deberá terminar cuando el usuario introduzca "fin".

11.3. Leer de un fichero de texto

Para leer de un fichero de texto usaremos "readLine()", que nos devuelve una cadena de texto (un "string"). Si ese string es *null*, quiere decir que se ha acabado el fichero y no se ha podido leer nada. Por eso, lo habitual es usar un "while" para leer todo el contenido de un fichero.

Existe otra diferencia con la escritura, claro: no usaremos un BufferedWriter, sino un BufferedReader, que se apoyará en un FileReader:

```

// FicheroTextoLeer.java
// Ejemplo de lectura desde un fichero de texto
// Introducción a Java, Nacho Cabanes

import java.io.*;

class FicheroTextoLeer
{
    public static void main( String[] args )
    {
        // Volcar a un fichero las líneas de otro (de texto)
        // que empiecen por "A"
        // Errores: sólo se comprueba si no existe el de origen

        if ( ! (new File("fichero.txt")).exists() )
        {
            System.out.println("No he encontrado fichero.txt");
            return;
        }

        System.out.println("Leyendo fichero de texto...");

        try
        {
            BufferedReader ficheroEntrada = new BufferedReader(
                new FileReader(new File("fichero.txt")));

            String linea=null;
            while ((linea=ficheroEntrada.readLine()) != null) {
                System.out.println(linea);
            }

            ficheroEntrada.close();
        }
        catch (IOException errorDeFichero)
        {
            System.out.println(
                "Ha habido problemas: " +
                errorDeFichero.getMessage() );
        }
    }
}

```

Ejercicio propuesto 11.3.1: Crea un programa que muestre el contenido de un fichero de texto, cuyo nombre deberá introducir el usuario. Debe avisar si el fichero no existe.

Ejercicio propuesto 11.3.2: Crea un programa que lea el contenido de un fichero de texto y lo vuelque a otro fichero de texto, pero convirtiendo cada línea a mayúsculas.

Ejercicio propuesto 11.3.3: Crea un programa que pida al usuario el nombre de un fichero y una palabra a buscar en él. Debe mostrar en pantalla todas las líneas del fichero que contengan esa palabra.

11.4. Leer de un fichero binario

Un fichero "binario" es un fichero que contiene "cualquier cosa", no sólo texto. Podemos leer byte a byte con "read()". Si el dato lo leemos como "int", un valor de "-1" indicará que se ha acabado el fichero.

En este caso, el tipo de fichero que usaremos será un "FileInputStream":

```
// FicheroBinarioLeer.java
// Ejemplo de lectura desde un fichero de texto
// Introducción a Java, Nacho Cabanes

import java.io.*;

class FicheroBinarioLeer
{
    public static void main( String[] args )
    {
        // Cantidad de "a" en un fichero de cualquier tipo
        // Mirando errores solo con try-catch

        System.out.println("Contando \"a\"...");
        int contador = 0;

        try
        {
            FileInputStream ficheroEntrada2 =
                new FileInputStream(new File("fichero.bin"));

            int dato;

            while ((dato = ficheroEntrada2.read()) != -1) {
                if (dato == 97) // Código ASCII de "a"
                    contador++;
            }
            ficheroEntrada2.close();
        }
        catch (Exception errorDeFichero)
        {
            System.out.println(
                "Ha habido problemas: " +
                errorDeFichero.getMessage() );
        }

        System.out.println("Cantidad de \"a\": " + contador);
    }
}
```

Este ejemplo cuenta la cantidad de letras "a" que contiene un fichero de cualquier tipo, ya sea de texto, ejecutable, documento creado con un procesador de textos o cualquier otro fichero.

Ejercicio propuesto 11.4.1: Crea un programa que lea el contenido de un fichero binario, mostrando en pantalla todo lo que sean caracteres imprimibles (basta con que sean desde la A hasta la Z, junto con el espacio en blanco).

Ejercicio propuesto 11.4.2: Crea un programa que extraiga el contenido de un fichero binario, volcando a un fichero de texto todo lo que sean caracteres imprimibles (basta con que sean desde la A hasta la Z, junto con el espacio en blanco).

11.5. Leer y escribir bloques en un fichero binario

Si tenemos que leer muchos datos de un fichero de cualquier tipo, acceder byte a byte puede resultar muy lento. Una alternativa mucho más eficiente es usar un array de bytes. Podemos usar "read", pero indicándole en qué array queremos guardar los datos, desde qué posición del array (que casi siempre será la cero) y qué cantidad de datos:

Si no conseguimos leer tantos datos como hemos intentado, será porque hemos llegado al final del fichero. Por eso, un programa que duplicara ficheros, leyendo cada vez un bloque de 512 Kb podría ser:

```
// FicheroBinarioEscribir.java
// Ejemplo de escritura en un fichero binario
// Introducción a Java, Nacho Cabanes

import java.io.*;

class FicheroBinarioEscribir
{
    public static void main( String[] args )
    {
        // -----
        // Copiar todo un fichero, con bloques de 512 Kb
        // Sin ninguna comprobación de errores

        System.out.println("Copiando fichero binario...");
        final int BUFFER_SIZE = 512*1024;

        try
        {
            InputStream ficheroEntrada3 = new FileInputStream(
                new File("fichero.in"));
            OutputStream ficheroSalida3 = new FileOutputStream(
                new File("fichero2.out"));
```

```

        byte[] buf = new byte[BUFFER_SIZE];
        int cantidadLeida;
        while ((cantidadLeida = ficheroEntrada3.read(buf, 0,
            BUFFER_SIZE)) > 0)
        {
            ficheroSalida3.write(buf, 0, cantidadLeida);
        }
        ficheroEntrada3.close();
        ficheroSalida3.close();
    }
    catch (Exception errorDeFichero)
    {
        System.out.println(
            "Ha habido problemas: " +
            errorDeFichero.getMessage() );
    }

    System.out.println("Terminado!");
}
}

```

Ejercicio propuesto 11.5.1: Crea un programa que lea los primeros 54 bytes de un fichero BMP (su cabecera) y compruebe si los dos primeros bytes de esos 54 corresponden a las letras B y M. Si no es así, mostrará el mensaje "No es un BMP válido"; si lo son, escribirá el mensaje "Parece un BMP válido"

Ejercicio propuesto 11.5.2: Crea una versión del ejercicio 11.4.2, leyendo a un array: un programa que extraiga el contenido de un fichero binario, volcando a un fichero de texto todo lo que sean caracteres imprimibles (basta con que sean desde la A hasta la Z, junto con el espacio en blanco).

Cambios en el curso

- 0.01, de 21 de julio de 2006: Tema 1, de introducción.
- 0.02, de 26 de julio de 2006: Tema 2, instalación del JDK
- 0.03, de 29 de julio de 2006: Tema 3a, creando y compilado una primera aplicación
- 0.04, de 30 de julio de 2006: Tema 3.2
- 0.05, de 2 de agosto de 2006: Tema 4
- 0.06, de 4 de agosto: Tema 5
- 0.07, de 5 agosto: Tema 6
- 0.08, de 7 de agosto: Tema 7 (arrays y cadenas de texto)
- 0.09, de 10 de agosto de 2006: Tema 8 (funciones)
- 0.10, de 16 de agosto de 2006: Tema 9 (programación modular)
- 0.11, de 25 de agosto de 2006: Tema 10 (más sobre clases)
- 0.12, de 9 de octubre de 2006: Tema 11, sobre matemáticas.
- 0.13, de 21 de octubre de 2006: Tema 12, toma de contacto con los Applets.
- 0.14, de 28 de abril de 2011: Revisión de los temas 1 a 4. Ampliación del tema 2 y el tema 3, para hablar de NetBeans como entorno de desarrollo.
- 0.15, de 3 de mayo de 2011: Revisión del tema 5. Actualización del tema 2 y el tema 3, para hablar de Eclipse y de Geany.
- 0.16, de 4 de mayo de 2011: Añadido el tema 7b: introducción de datos mediante consola. Revisión del tema 8.
- 0.17, de 5 de mayo de 2011: Añadido el tema 7c: introducción de datos mediante ventanas. Añadido el tema 7d: ejercicios propuestos de repaso (16). Revisión del tema 11.
- 0.18, de 16 de mayo de 2011: Creado el tema 13. Ampliado el tema 12.
- 0.19, de 26 de junio de 2011: Revisión del tema 9, para incluir los detalles de cómo crear un proyecto de varios fuentes con NetBeans y para añadir un ejercicio propuesto. Ligera revisión del tema 10, que también incluye un ejercicio propuesto.

- 0.20, de 29 de junio de 2013: Revisados los temas 1, 2, 3
- 0.21, de 30 de junio de 2013: Revisados los temas 4, 5, 6
- 0.22, de 01 de julio de 2013: Revisados los temas 7,8
- 0.23, de 06 de julio de 2013: Revisado el tema 9
- 0.24, de 10 de julio de 2013: Revisado y parcialmente reescrito el tema 10
- 0.25, de 21 de julio de 2013: Revisado el tema 12
- 0.26, de 15 de agosto de 2013: Revisado el tema 14
- 0.26p, de 08 de noviembre de 2013: Creada una versión PDF con todo el contenido hasta la fecha.
- 0.27, de 20 de enero de 2015: Corregido el tema 6: el ejemplo del formato de "break" con etiqueta era incorrecto.
- 0.28, de 21 de enero de 2015: Ampliado el tema 4: Incluido un ejemplo de cómo sumar datos prefijados sin necesidad de variables. El apartado 4.3 mostrará como leer datos de teclado e incluye 10 ejercicios propuestos. El antiguo apartado 4.3, más avanzado, queda retrasado para otro tema posterior. Los apartados 4.5 y 4.6 se retrasan para fundirse con el tema 5.
- 0.29, de 22 de enero de 2015: Ampliado el tema 5: Incluidos en él los que eran los apartados 4.5 (operadores condicionales) y 4.6 (enlazar condiciones). Los ejemplos son más detallados. Añadidos 9 ejercicios propuestos más.
- 0.30, de 23 de enero de 2015: Ampliado el tema 6: Los ejemplos son más detallados. Dos nuevos fuentes de ejemplo. Diez nuevos ejercicios propuestos.
- 0.31, de 26 de enero de 2015: Ampliado el tema 7: Ampliadas las explicaciones con más detalles y hablando también de "constantes". Los ejemplos son más detallados. Tres nuevos fuentes de ejemplo. Catorce nuevos ejercicios propuestos.
- 0.32, de 04 de febrero de 2015: El que antes era el tema 11 (funciones matemáticas) pasa a ser el nuevo tema 8. Añadidos seis nuevos ejercicios propuestos (dos en el apartado 7.1, cuatro en el nuevo apartado 8).
- 0.33, de 04 de julio de 2015: Revisado y parcialmente reescrito el tema 8.

- 0.34, de 07 de julio de 2015: Revisado y ampliado el tema 5. Añadidos cuatro ejemplos completos de la orden "switch" y uno del operador condicional.
- 0.35, de 09 de julio de 2015: Revisado y ampliado el tema 2.
- 0.36, de 11 de julio de 2015: Reescrito y ampliado el tema 3.
- 0.37, de 14 de julio de 2015: Reescrito y ampliado el tema 4.
- 0.38, de 15 de julio de 2015: Revisado y ampliado el tema 6.
- 0.39, de 23 de julio de 2015: Revisado y ampliado el tema 7.
- 0.40, de 24 de julio de 2015: Movido el tema 14 (ficheros) al tema 11. Eliminados (al menos por ahora) los temas anticuados (applets, entrada usando BufferedReader) e incompletos (Java2D). Ligeramente revisados los temas 8, 9, 10 y 11.