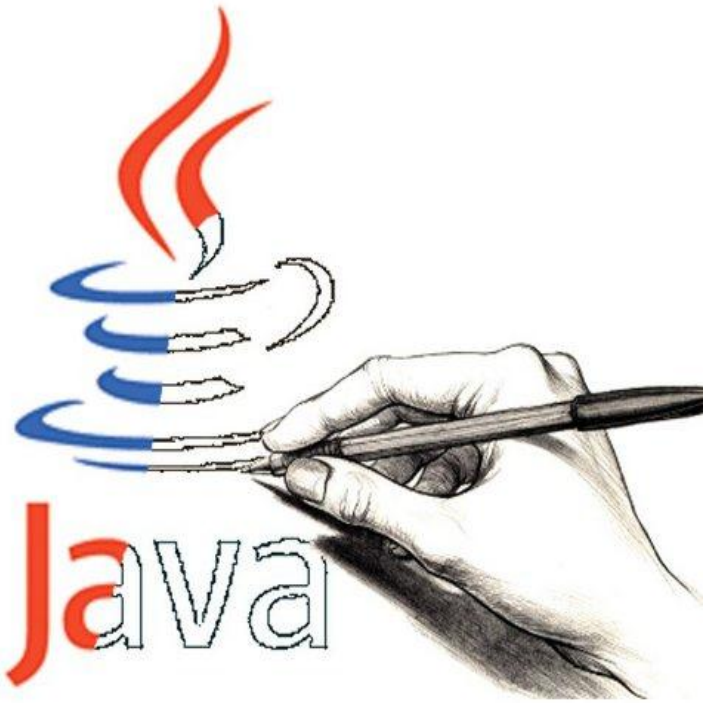
Introducción a la Programación



Autor:

Tclgo. César O. Vanegas Mendoza

2014-2015

Índice de contenido

1. Introducción: ¿por qué Java?

¿Qué es Java?

Java es un lenguaje de programación de ordenadores, diseñado como una mejora de C++, y desarrollado por ***Sun Microsystems*** (compañía actualmente absorbida por Oracle).

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; incluso muchos teléfonos móviles actuales son capaces de usar programas creados en Java). Y aún hay más: el sistema operativo Android para teléfonos móviles propone usar Java como lenguaje estándar para crear aplicaciones. Como inconveniente, la existencia de ese paso intermedio hace que los programas Java no sean tan rápidos como puede ser un programa realizado en C, C++ o Pascal y optimizado para una cierta máquina en concreto.

¿Por qué usar Java?

Puede interesarnos si queremos crear programas que se vayan a manejar a través de un interfaz web (sea en Internet o en una Intranet de una organización), programas distribuidos en general, o programas que tengan que funcionar en distintos sistemas sin ningún cambio (programas "portables"), o programas para un Smartphone Android, entre otros casos.

¿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.

¿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).
- ✓ Al igual que C++, es un lenguaje orientado a objetos, con las ventajas que eso puede suponer a la hora de diseñar y mantener programas de gran tamaño.
- ✓ Java permite crear programas multitarea.
- ✓ Permite excepciones, como alternativa más sencilla para manejar errores, como ficheros inexistentes o situaciones inesperadas.
- ✓ Es más difícil cometer errores de programación que en C y C++ (no existen los punteros).
- ✓ Se pueden crear entornos "basados en ventanas", gráficos, acceder a bases de datos, etc.

¿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 Enviroment"*** (JRE), que se puede descargar libremente desde Java.com (unos 10 Mb de descarga).

Otra forma (actualmente menos frecuente) en que podemos encontrar programas creados en lenguaje Java, es dentro de páginas Web. Estas aplicaciones Java incluidas en una página Web reciben el nombre de "Applets", y para utilizarlos también deberíamos tener instalada la máquina virtual Java (podría no ser necesario si nuestro Navegador Web reconoce automáticamente el lenguaje Java, algo que no es habitual hoy en día).

¿Qué hace falta para crear un programa en Java?

Existen diversas herramientas que nos permitirán crear programas en Java. La más habitual es la propia que suministra Sun (ahora Oracle), y que se conoce como JDK (***Java Development Kit***). Es de libre distribución y se puede conseguir en la propia página Web de Oracle.

El inconveniente del JDK es que puede 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 que de Visual C#, Visual Basic o Delphi, 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 o NetBeans. Ambos son buenos y gratuitos.

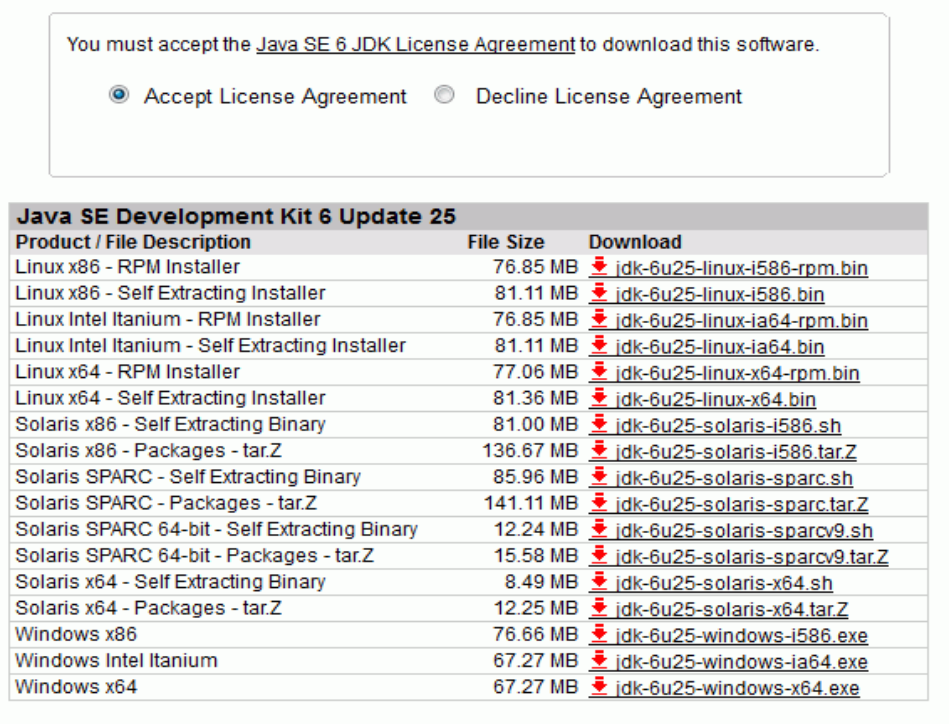
En nuestro caso, los programas que crearemos estarán probados con la versión 1.6.0 del JDK, salvo que se indique lo contrario.

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):

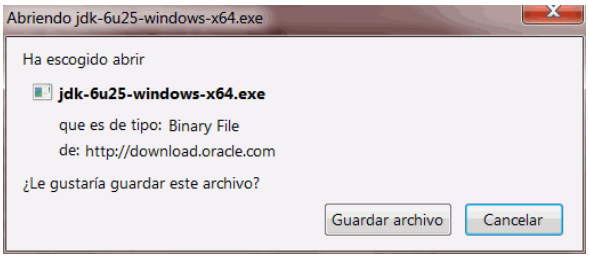
www.oracle.com/technetwork/java/javase/downloads.

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

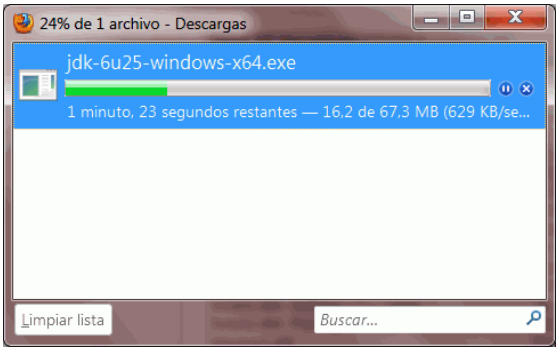
En primer lugar, deberemos escoger nuestro sistema operativo y (leer y) aceptar las condiciones de la licencia:

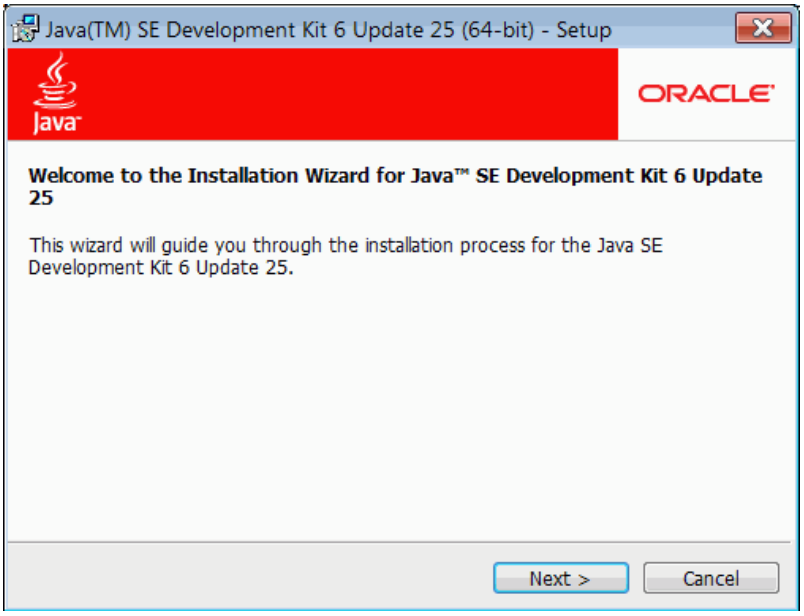


Entonces empezaremos a recibir un único fichero de gran tamaño (cerca de 70 Mb, según versiones):

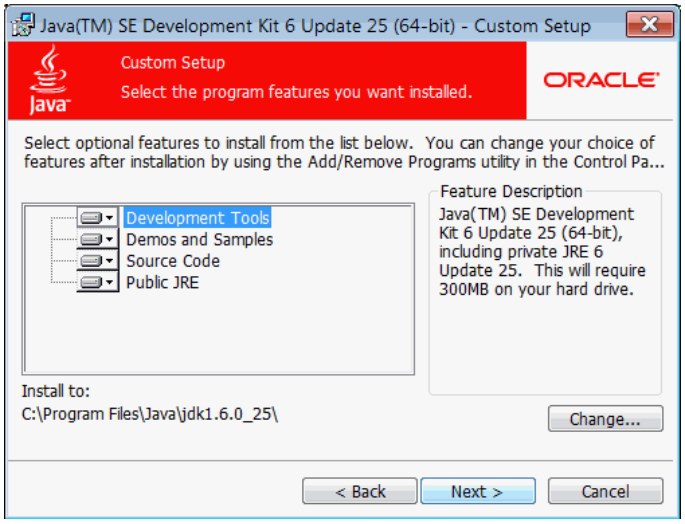


Al tener un tamaño tan grande, la descarga puede ser lenta, según la velocidad de nuestra conexión a Internet:

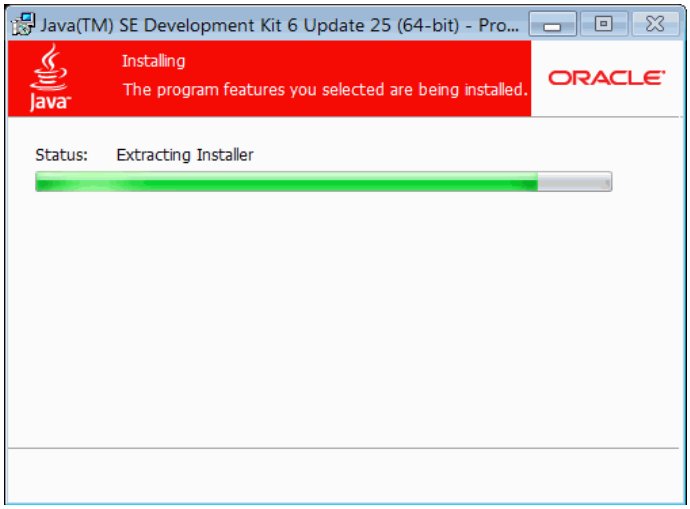




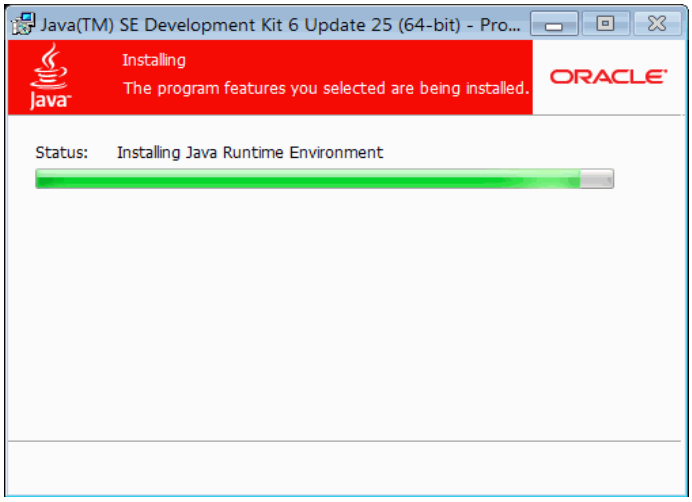
Podremos afinar detalles como la carpeta de instalación, o qué partes no queremos instalar (por ejemplo, podríamos optar por no instalar los ejemplos). Para no complicarnos, si tenemos suficiente espacio (posiblemente unos 400 Mb en total), podemos hacer una instalación típica, sin cambiar nada:



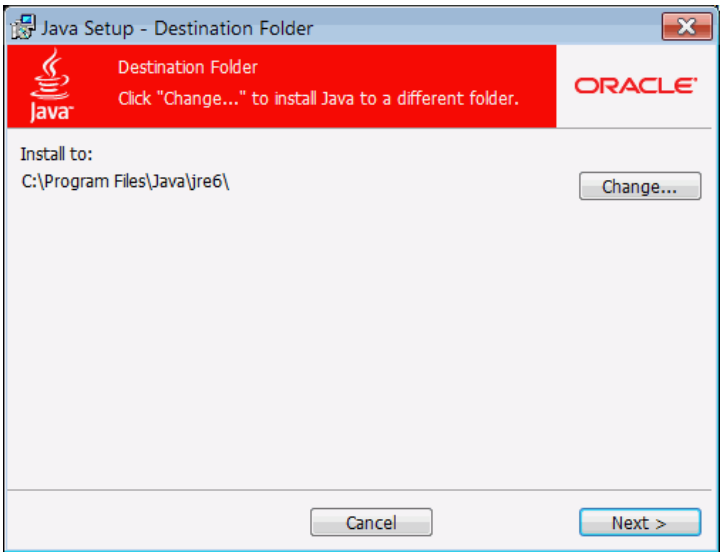
Ahora deberemos tener paciencia durante un rato, mientras se descomprime e instala todo:



En cierto punto se nos preguntará si queremos instalar la máquina virtual Java (***Java Runtime Environment, JRE***). Lo razonable será responder que sí:

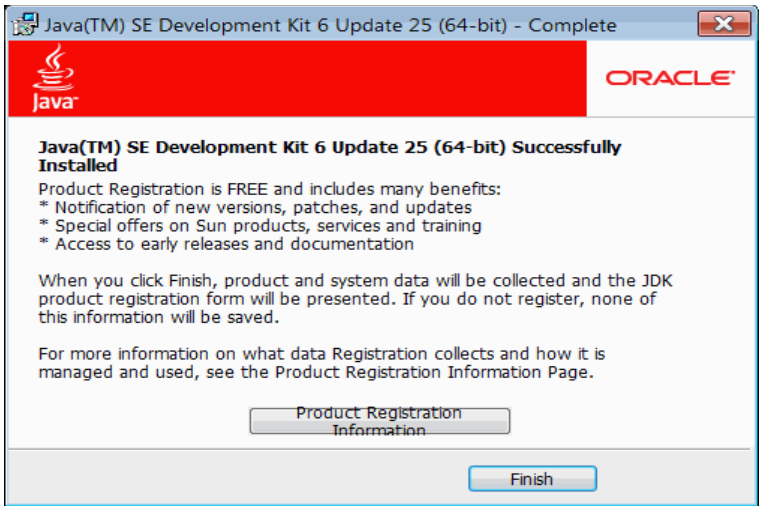


Igual que para el JDK, podríamos cambiar la carpeta de instalación:



Tendremos que esperar otro momento

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

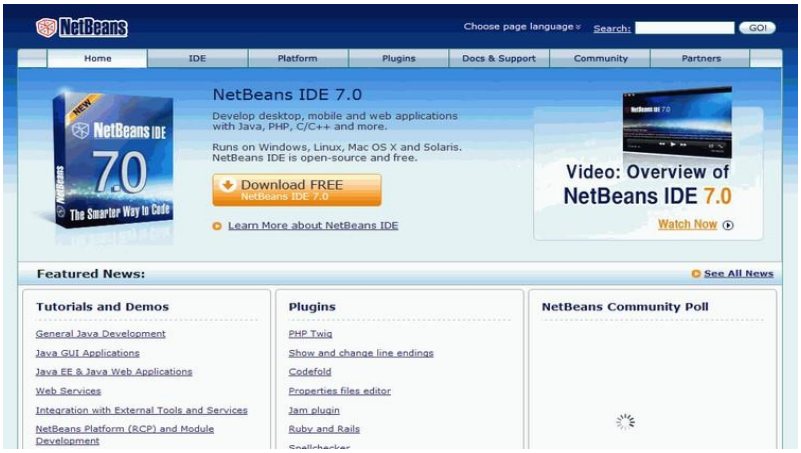


Y se nos propondrá registrar nuestra copia en la página de Oracle (no es necesario):

Con eso ya tenemos instalada la herramienta básica.

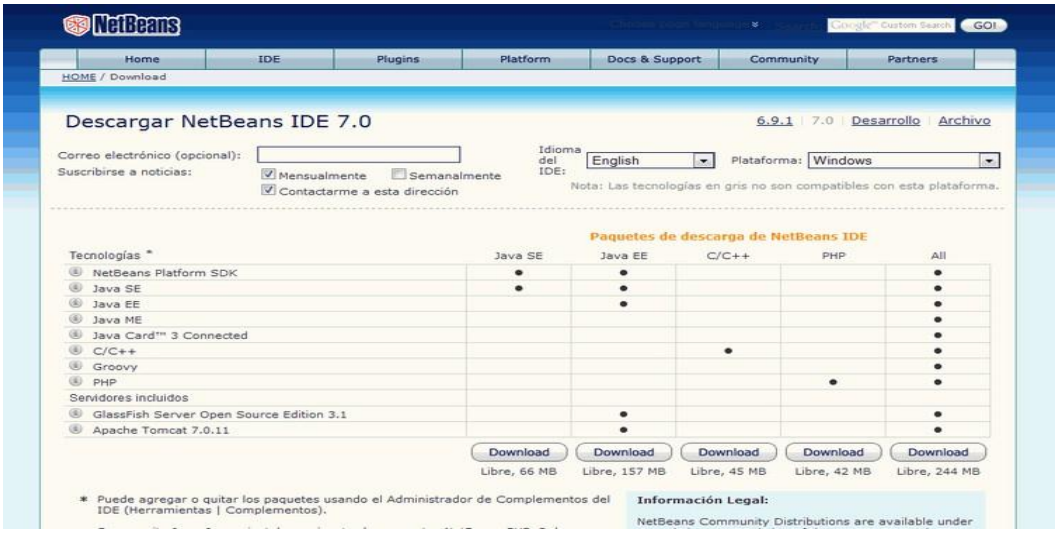
Pero el kit de desarrollo (JDK) no incluye ningún editor con el que crear nuestros programas. Podríamos instalar un "editor genérico", porque tenemos muchos gratuitos y de calidad, como Notepad++. Aun así, si nuestro equipo es razonablemente moderno, puede ser preferible instalar un entorno integrado,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

www.netbeans.org

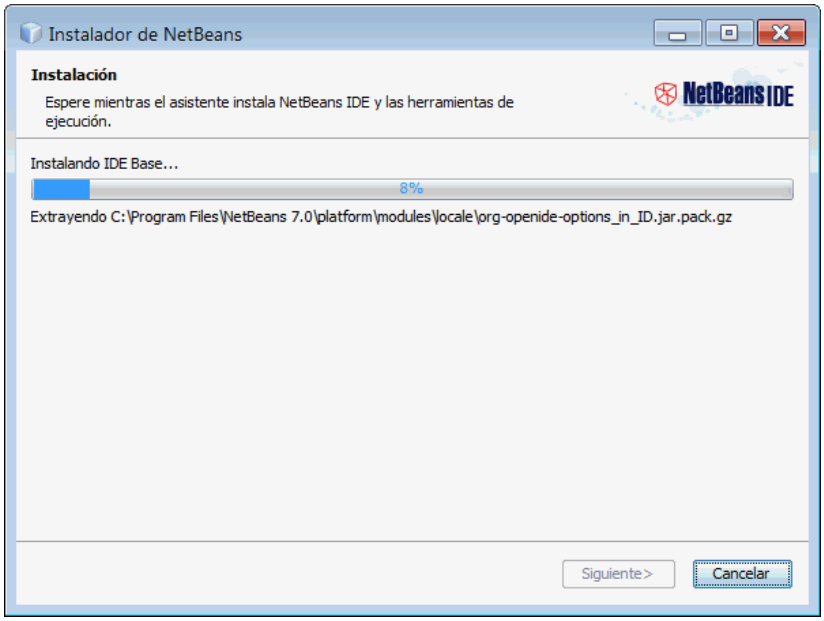


Si hacemos clic en "Download", se nos llevará a la página de descargas, en la que tenemos varias versiones para elegir. Lo razonable "para un novato" es descargar la versión para "Java SE" (las alternativas son otros lenguajes, como PHP o C++, versiones profesionales como Java EE (Enterprise Edition),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

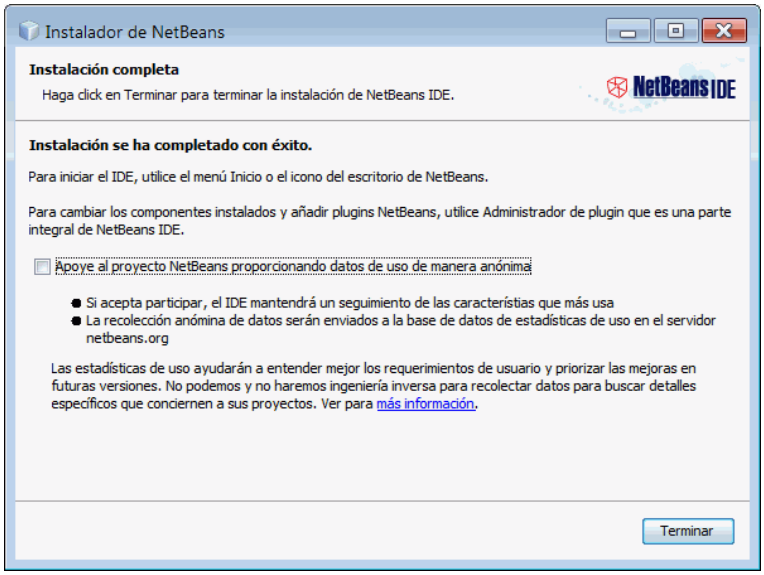
Es fácil que también podamos escoger el español como idioma, en vez del inglés.



La instalación no será posible si no hemos instalado Java antes, pero si lo hemos hecho, debería ser simple y razonablemente rápida:

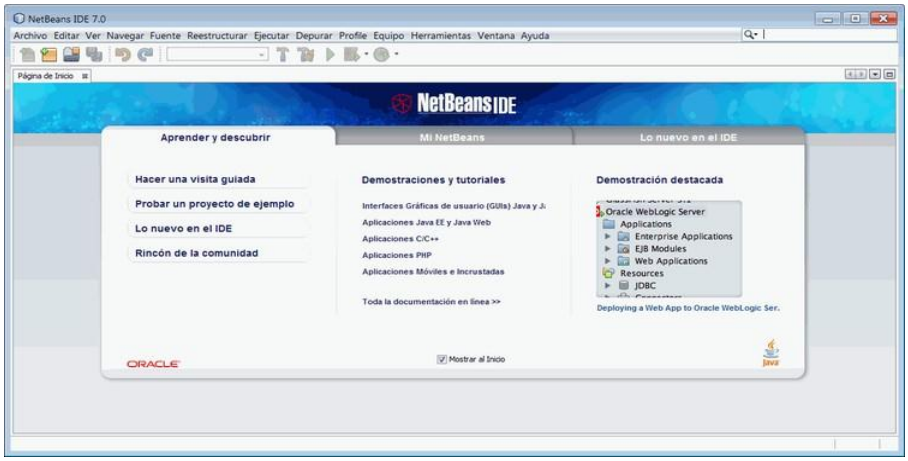


Y al final quizá se nos pregunte si queremos permitir que se recopile estadísticas sobre nuestro uso:



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. Nuestra Primera Aplicación con Netbeans

2.1. Escribiendo "Hola Mundo"

Comenzaremos por crear un pequeño programa en modo texto. Este primer programa se limitará a escribir un texto en la pantalla.

Quien ya conozca otros lenguajes de programación, verá que conseguirlo en Java parece más complicado. Por ejemplo, en BASIC bastaría con escribir PRINT "HOLA MUNDO!". Pero esta mayor complejidad inicial es debida al "cambio de mentalidad" que tendremos que hacer cuando empleamos Java, y dará lugar a otras ventajas más adelante, cuando nuestros programas sean mucho más complejos.

Nuestro primer programa será:

```
//
// Aplicación HolaMundo de ejemplo
//

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

A quien no conozca ningún lenguaje de programación, todo le sonará extraño, pero dentro de muy poco (apenas veamos que nuestro programa funciona) volveremos atrás para ver paso a paso qué hace cada una de las líneas que habremos tecleado.

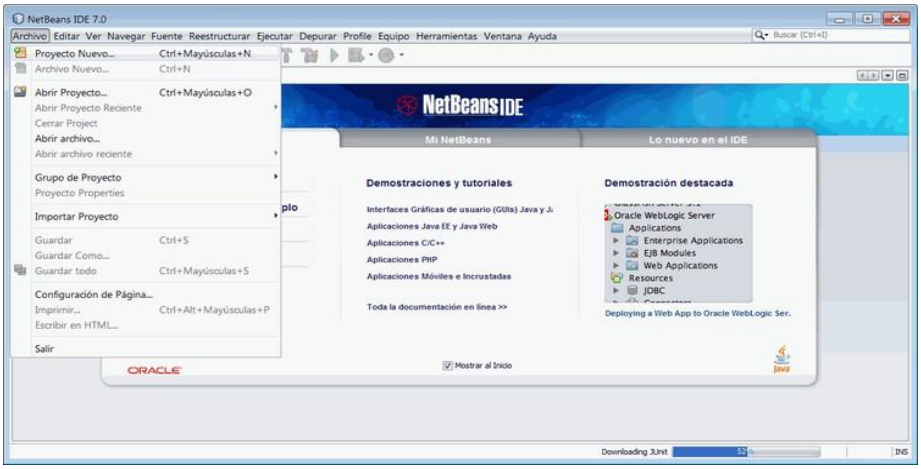
La única línea que nos interesa por ahora es la que dice: ***System.out.println("Hola Mundo!")***; Esa es la orden que se encarga de escribir Hola Mundo! en pantalla, avanzando de línea (por eso el ***"println":*** "print" quiere decir "escribir", y ***"ln"*** es la abreviatura de "line", línea, para indicar que se debe avanzar a la siguiente línea después de escribir ese texto). Se trata de una orden de salida ***(out)*** de nuestro sistema ***(System).*** Esta orden es más compleja que el PRINT de otros lenguajes más antiguos, como BASIC, pero todo tiene su motivo, que veremos más adelante.

De igual modo, a estas alturas del curso no entraremos todavía en detalles sobre qué hacen las demás líneas, que supondremos que deben existir.

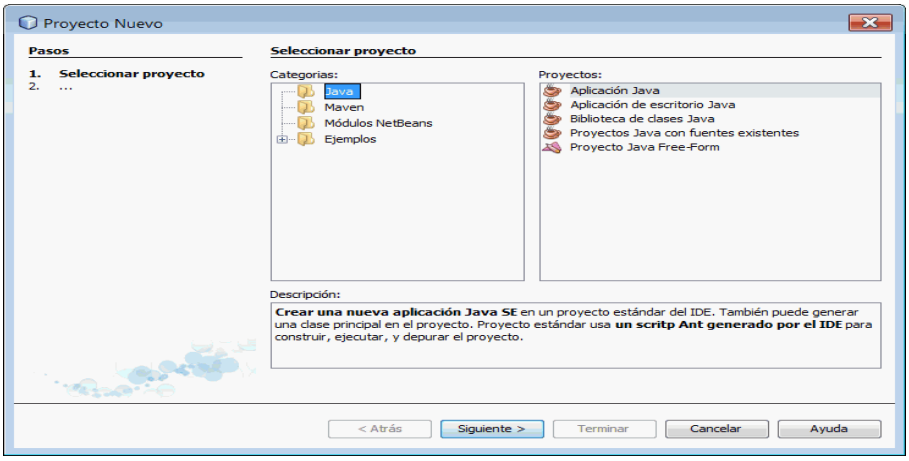
Nota: El uso de mayúsculas y minúsculas es irrelevante para Windows y para algunos lenguajes, como BASIC, pero no lo es para Java (ni para otros muchos lenguajes y sistemas). Por eso, deberemos respetar las mayúsculas y minúsculas tal y como aparezcan en los ejemplos, o éstos no funcionarán.

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

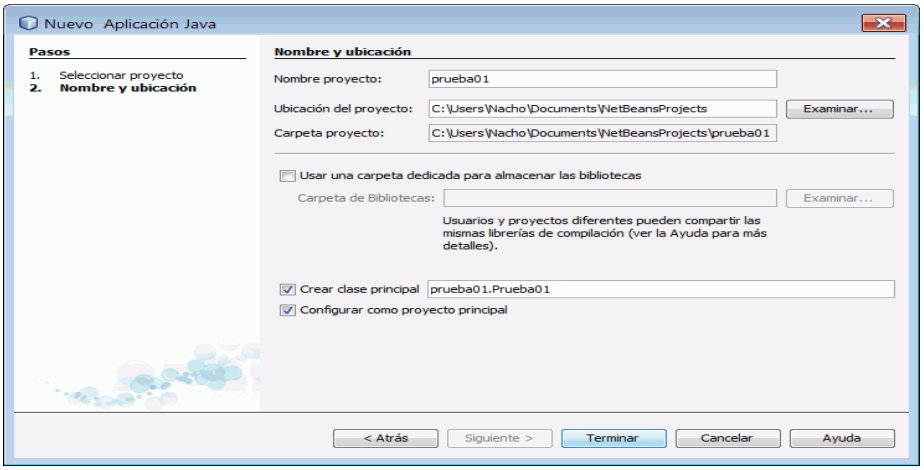
En primer lugar, deberemos entrar al menú "Archivo" y escoger la opción "Proyecto Nuevo":



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

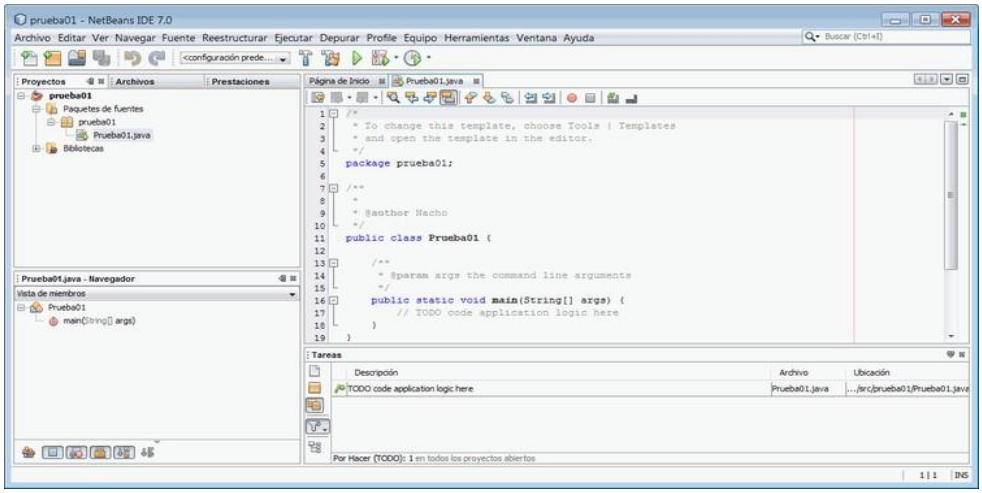


Deberemos indicar un nombre para el proyecto. Tenemos también la posibilidad de cambiar la carpeta en que se guardará.



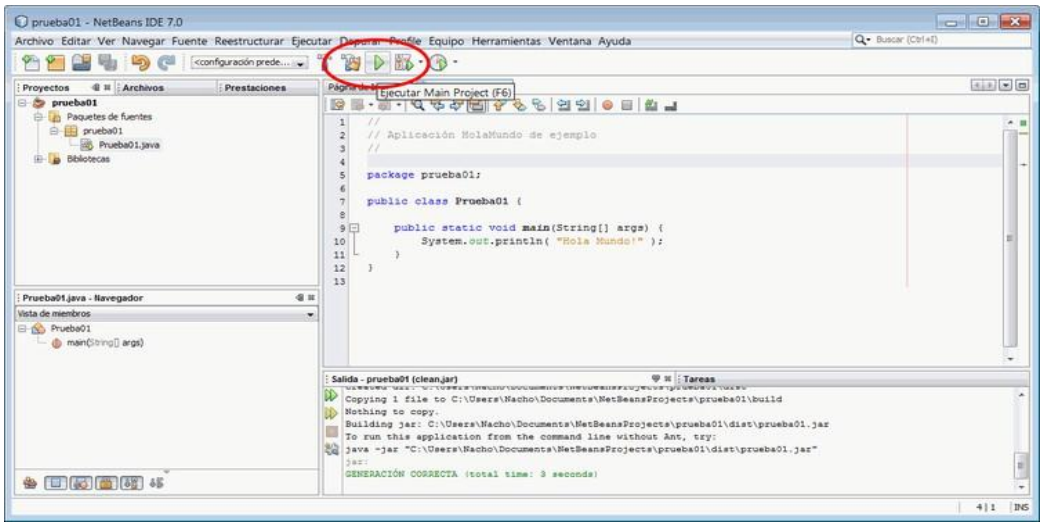
Y entonces aparecerá un esqueleto de programa que recuerda al que nosotros queremos conseguir... salvo por un par de detalles:

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

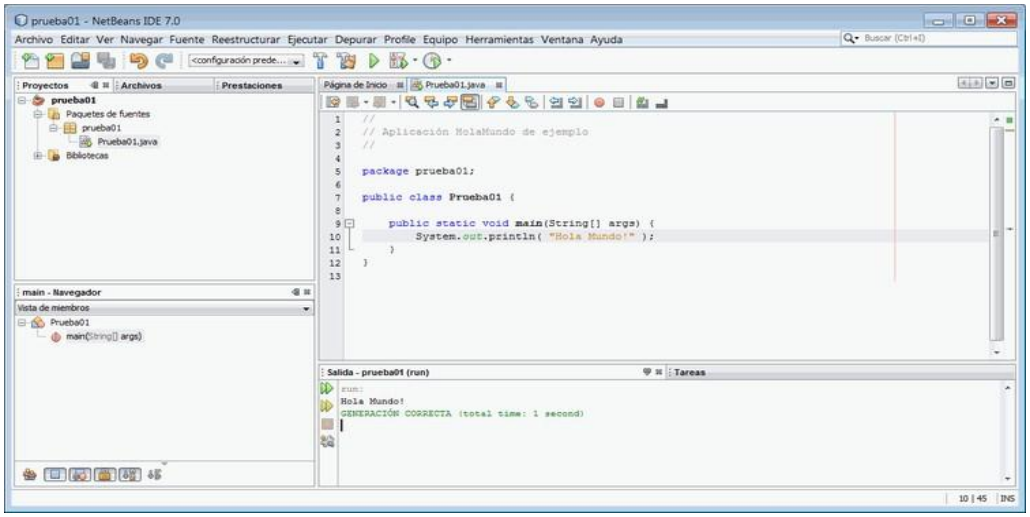


La orden "package" deberemos conservarla si usamos NetBeans, para indicar que nuestro programa es parte de un proyecto. La orden "println" deberemos añadirla, o nuestro programa no escribirá nada en pantalla. Podríamos borrar las líneas de color gris, hasta llegar a algo como esto:

```
//
// Aplicación HolaMundo de ejemplo
//
package prueba01;
public class HolaMundo {
  public static void main( String
    args[] ) { System.out.println(
    "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:



Si todo ha funcionado correctamente, llega el momento de practicar...

Ejercicio propuesto1: Crea un programa en Java, usando NetBeans, que te salude en pantalla por tu nombre (por ejemplo, "Hola, Martha Julia").

2.2. Entendiendo esa primera aplicación

Nuestro primer programa, que debería funcionar aunque todavía no sepamos bien por qué, era así:

```
//
// Aplicación HolaMundo de ejemplo
//

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

Ahora llega el momento de entender qué hace cada línea:

El comienzo del programa: un comentario.

Las tres primeras líneas son:

```
//
// 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 */
```

Continuamos definiendo una clase.

Después aparece un bloque de varias órdenes; vamos a eliminar las líneas del centro y conservar solamente las que nos interesarán en primer lugar:

```
class HolaMundo {
    [... aquí faltan más
        cosas ...]
}
```

Esto es la definición de una "clase" llamada "HolaMundo". Por ahora, podemos pensar que, a nuestro nivel de principiantes, "clase" es un sinónimo de "programa": nuestro programa se llama HolaMundo. Más adelante veremos que los programas complejos realmente se suelen descomponer en varios bloques, planteándolos como una serie de objetos que cooperan los unos con los otros. Cuando llegue ese momento, hablaremos más sobre "class" y su auténtico significado.

¿Qué hace nuestra clase?

Como ya hemos comentado, hará poco más que pedir a la pantalla que muestre un cierto mensaje. Los detalles concretos de lo que debe hacer nuestra clase los indicamos entre llaves ({ y }), así:

```
class HolaMundo {
  [... aquí faltan más cosas ...]
}
```

Estas llaves serán frecuentes en Java, porque las usaremos para delimitar cualquier conjunto de órdenes dentro de un programa (y normalmente habrá bastantes de estos "conjuntos de órdenes"...)

Java es un lenguaje de formato libre, de modo que podemos dejar más o menos espacios en blanco entre las distintas palabras y símbolos, así que la primera línea de la definición de nuestra clase se podría haber escrito más espaciada

```
class
HolaMundo    {
  [... aquí faltan más cosas ...]
}
```

o bien así (formato que prefieren algunos autores, para que las llaves de principio queden justo encima de las de final)

```
class HolaMundo
{
  [... aquí faltan más cosas ...]
}
```

o incluso cualquier otra "menos legible":

```
   class
HolaMundo
      {
  [... aquí faltan más cosas
             ...]
   }
```

Volvamos a nuestro programa...

Nuestra clase "HolaMundo" sólo contiene una cosa: un bloque llamado "main", que representa el cuerpo del programa.

```
public static void main( String
  args[] ) { [... aquí faltan más
  cosas ...]
}
```

> **Nota** para C y C++: La función "main()" es, al igual que en C y C++, la que indicará el cuerpo de un programa, pero en Java el propio programa debe ser también un objeto, de modo que "main()" debe estar dentro de la declaración de una clase.

Pero vemos que hay muchas "cosas" rodeando a "main" (public, static, void). Por ahora, simplemente asumiremos que estas "cosas" deberán estar siempre, y más adelante volveremos a ellas según las vayamos necesitando. Lo mismo ocurrirá con eso de "String args[]": ya veremos para qué sirve y en qué circunstancias nos puede resultar útil.

Al igual que decíamos de la clase, todo el conjunto de "cosas" que va a hacer esta función "main" se deberá englobar entre llaves:

```
public static void main( String
  args[] ) { [... aquí faltan más
  cosas ...]
}
```

En concreto, nuestra clase de objetos "HolaMundo" interacciona únicamente con la salida en pantalla de nuestro sistema (System.out), para enviarle el mensaje de que escriba un cierto texto en pantalla (println):

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

Como se ve en el ejemplo, el texto que queremos escribir en pantalla se debe indicar 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 "println").

Ahora ya podemos volver a leer todo nuestro programa, con la esperanza de entenderlo un poco más...

```
//
// Aplicación HolaMundo de ejemplo
//

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

Recordemos las "cosas" importantes que no se deben olvidar de este programa:

- Nuestra clase (por ahora es lo mismo que decir "nuestro programa") se llama HolaMundo.
- Sólo tiene un bloque, el único que (casi) siempre aparecerá, que se llama "main", y que corresponde al cuerpo del programa.
- Escribe (println) en la pantalla estándar (System.out) la frase "Hola Mundo!". Ejercicio propuesto 3.2.1: Crea un programa en Java que escriba en pantalla "Java no me da miedo".

Ahora ya podemos seguir avanzando...

4. Las variables

4.1. Qué son y cómo se declaran las variables

Nota: el manejo de variables básicas en Java es muy similar al de C y al de C++, y lo mismo ocurre con las operaciones matemáticas fundamentales, así que quien haya manejado ya alguno de esos dos lenguajes, podrá avanzar con mucha rapidez en este tema

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

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 nada más calcularla, sin almacenarla previamente en ningún sitio). Los pasos a dar serían los siguientes:

- Pedir al usuario 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 la 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" será distinto 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 llamado "int" (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 va a ser números inferiores a 1.000), podemos emplear el tipo de datos llamado "short" (entero "corto"), que ocupa la mitad de espacio.

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

```
/* ------------------------------*
 *  Introducción a Java - Ejemplo *
 *  Por Nacho Cabanes             *
```

```
 * -------------------------------*
 *  Fichero: Suma.java            *
 *  (Suma dos números enteros)    *
 * -------------------------------*/

class Suma {

  public static void main( String args[] ) {

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

    System.out.println(
      "La suma 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, se puede indicar un valor inicial.

Hay una importante diferencia entre las dos órdenes "println": la primera contiene comillas, para indicar que ese texto debe aparecer "tal cual", mientras que la segunda no contiene comillas, por lo que no se escribe el texto " primerNumero+segundoNumero", sino que se intenta calcular el valor de esa expresión (en este caso, la suma de los valores que en ese momento almacenan las variables primerNumero y segundoNumero, 56+23 = 89).

Podríamos pensar en mejorar el programa anterior para que los números a sumar no estén prefijados, sino que se pidan al usuario... pero eso no es trivial. El lenguaje Java prevé que quizá se esté utilizando desde un equipo que no sea un ordenador convencional, y que quizá no tenga un teclado conectado, así que deberemos hacer ciertas comprobaciones de errores que todavía están fuera de nuestro alcance. Por eso, vamos a aplazar un poco eso de pedir datos al usuario.

Ejercicio propuesto 2: Crea un programa en Java que escriba en pantalla el producto de dos números prefijados (pista: el símbolo de la multiplicación es el asterisco, "*"

Los tipos de ***Datos Numéricos*** disponibles en Java son los siguientes:

Tipos de Datos	Bytes que ocupa	Rango de valores
byte	1	-128 a 127
short	2	-32.768 a 32.767
int	4	-2.147.483.648 a 2.147.483.649
long	8	-9 · 1018 a 9 · 1018
double	8	-1,79 · 10308 a 1,79 · 10308
float	4	-3,4 · 1038 a 3,4 · 1038

Ejercicio propuesto 3: Crea un programa en Java que escriba en pantalla el producto de dos

números reales prefijados. Estos dos números deben tener 3 cifras significativas (como 1.23 y 20.8).

La forma de "declarar" variables de todos estos tipos es, como ya habíamos comentado, detallando primero el tipo de datos y después el nombre que daremos la variable, así:

```
int numeroEntero; // La variable numeroEntero será un número de tipo "int"
short distancia;  // La variable distancia guardará números "short"
long gastos;      // La variable gastos es de tipo "long"
byte edad;        // Un entero de valores "pequeños"
float porcentaje; // Con decimales, unas 6 cifras de precisión
double numPrecision; // Con decimales y precisión de unas 14 cifras
```

Se pueden declarar varias variables del mismo tipo "a la vez":

```
int primerNumero, segundoNumero;     // Dos enteros
```

Eso sí, algunos autores recomiendan que no se haga: nuestro programa será más legible si escribimos en cada línea una variable distinta, y todas ellas precedidas por su tipo.

También se puede dar un valor a las variables a la vez que se declaran:

```
int a = 5; // "a" es un entero, e inicialmente vale 5
short b=-1, c, d=4; // "b" vale -1, "c" vale 0, "d" vale 4
```

Los nombres de variables pueden contener letras y números (pero no pueden comenzar con un número) y algún otro símbolo, como el de subrayado, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles (+,-,*,/), ni llaves o paréntesis, ni vocales acentuadas (á,é,í,ó...), ni eñes...

Ejercicio propuesto 4: Crea un programa en Java, que muestre la diferencia (resta) de dos números reales (float) prefijados.

Tenemos otros dos tipos básicos de variables, que no son para datos numéricos, y que usaremos más adelante:

Tipos de Datos	Bytes que ocupa	Rango de valores
boolen	1	true o false
char	2	Carácter Unicode

- **char.** Será una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación. Ocupa 2 bytes. Sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII).

- **boolean**. se usa para evaluar condiciones, y puede tener el valor "verdadero" (true) o "false" (false). Ocupa 1 byte.

Estos ocho tipos de datos son lo que se conoce como "tipos de datos primitivos" (porque forman parte del lenguaje estándar, y a partir de ellos podremos crear otros más complejos).

Nota para C y C++: estos tipos de datos son muy similares a los que se emplean en C y en C++, apenas con alguna "pequeña" diferencia, como es el hecho de que el tamaño de cada tipo de datos (y los valores que puede almacenar) son independientes en Java del sistema que empleemos, cosa que no ocurre en C y C++, lenguajes en los que un valor de 40.000 puede ser válido para un "int" en unos sistemas, pero "desbordar" el máximo valor permitido en otros. Otra diferencia es que en Java no existen enteros sin signo ("unsigned"). Y otra más es que el tipo "boolean" ya está incorporado al lenguaje, en vez de equivaler a un número entero, como ocurre en C.

3.2. Operaciones matemáticas básicas.

Hay varias operaciones matemáticas que son frecuentes. Veremos cómo expresarlas en Java, y también veremos otras operaciones "menos frecuentes"

Las que usaremos con más frecuencia son:

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (resto de la división)

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar.

Ejercicio propuesto 5: Crea un programa que muestre la suma de dos números prefijados de tipo "byte".

Ejercicio propuesto 6: Crea un programa que muestre la división de dos números reales de doble precisión (double) prefijados.

Nota: El resto de la división (%) también es una operación conocida. 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 7: Crea un programa que calcule y muestre el resto de la división de 11 entre 5.

3.3. 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, tenemos -= 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 variable a la vez que se asigna su valor a otra, sino hacerlo en dos pasos.

Ejercicio propuesto 8: 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;
```

3.4. Operadores relacionales.

También podremos comprobar si entre dos números (o entre dos variables) existe una cierta relación del tipo "¿es a mayor que b?" o "¿tiene a el mismo valor que b?". Los operadores que utilizaremos para ello son:

operador	significado
<	Menor que
>	Mayor que
>=	Mayor o igual que
<=	Menor o igual que
== **(dos iguales)**	Igual que
!=	Distinto de

Así, por ejemplo, para ver si el valor de una variable "b" es distinto de 5, escribiríamos algo parecido (veremos la sintaxis correcta un poco más adelante) a

SI b != 5 ENTONCES ...

o para ver si la variable "a" vale 70, sería algo como (nuevamente, veremos la sintaxis correcta un poco más adelante)

SI a == 70 ENTONCES ...

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.

3.5. Operadores lógicos.

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

operador	significado
!	No lógico (**NOT**)
&&	"Y" lógico (**AND**)
||	"O" lógico (**OR**)

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" en una sintaxis parecida a la de Java (aunque todavía no es la correcta) sería:

SI (a==3 && b>5) || (a==7 && ! (b<4))

Pasemos ahora a ver cuál sería la forma correcta de comprobar condiciones...

4. Comprobación de condiciones

4.1. if

Nota: al igual que ocurría con el apartado anterior, las sentencias condicionales en Java son casi idénticas al caso de C y al de C++, así que quien haya manejado ya alguno de esos dos lenguajes, podrá avanzar con mucha rapidez en este tema

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

SI condición_a_comprobar ENTONCES pasos_a_dar

En el caso de 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)
{
  sentencias
}
```

Por ejemplo,

```
if (x == 3) {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
```

Nota: 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). Las llaves serán imprescindibles sólo cuando haya que hacer varias cosas:

```
if (x == 3)
  System.out.println( "El valor es correcto" );
```

Ejercicio propuesto 9: Crea un programa que diga si el número 371 es múltiplo de 3 (pista: puedes utilizar la operación "módulo", el "resto de la división" (%))

4.2. else

Una primera mejora es indicar qué hacer en caso de que no se cumpla la condición. Sería algo parecido a

SI condición_a_comprobar ENTONCES
pasos_a_dar
EN_CASO_CONTRARIO
pasos_alternativos

que en Java escribiríamos así:

```
if (condición)
  {
     sentencias1
  }
else
  {
     sentencias2
   }
```

Por ejemplo,

```
if (x == 3) {
  System.out.println( "El valor es correcto" );
  resultado = 5;
}
else {
  System.out.println( "El valor es incorrecto" );
  resultado = 27;
}
```

Ejercicio propuesto 10: Crea un programa que diga si el número 371 es impar o no lo es.

4.3. switch

Si queremos comprobar varias condiciones, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una forma más elegante en Java 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:

```
switch ( x * 10) {
  case 30: System.out.println( "El valor de x era 3" ); break;
  case 50: System.out.println( "El valor de x era 5" ); break;
  case 60: System.out.println( "El valor de x era 6" ); 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:

```
switch ( x * 10) {
  case 30: System.out.println( "El valor de x era 3" );
  break; case 50: System.out.println( "El valor de x era
  5" ); break; case 60: System.out.println( "El valor de
  x era 6" ); break;
  default: System.out.println( "El valor de x no  era 3, 5 ni 6" );
  break;
}
```

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

```
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" );
           valorTemporal = 10;
           System.out.println( "Operaciones auxiliares completadas" );
           break;
   default:
     System.out.println( "El valor de x no estaba entre 1 y 6" );
     break;
}
```

Ejercicio propuesto 11: Crea un programa que escriba como texto cualquier número del uno al 10. Por ejemplo, si la variable "x" vale 3, deberá escribir "tres".

Un poco más adelante propondremos más ejercicios que permitan afianzar todos estos

conceptos, cuando sepamos cómo pedir datos al usuario.

4.4. 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 (?)":

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;
```

Ejercicio propuesto 12: Crea un programa que diga si el número 371 es impar o no lo es, usando el operador condicional.

5. Partes del programa que se repiten

Nota: al igual que ocurría con los dos apartados anteriores, los bucles en Java son muy parecidos a los de C y C++: quien conozca alguno de esos dos lenguajes, podrá avanzar rápido en este tema

Con frecuencia tendremos que hacer que una parte de nuestro programa se repita (algo a lo que con frecuencia llamaremos "bucle"). Este trozo de programa se puede repetir mientras se cumpla una condición o bien un cierto número prefijado de veces.

5.1. while

Java incorpora varias formas de conseguirlo. La primera que veremos es la orden "while", que 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:

```
x = 20;
while ( x > 10) {
  System.out.println("Aun no se ha alcanzado el valor límite");
  x --;
}
```

Ejercicio propuesto 13: Crea un programa que muestre los números del 1 al 10, usando "while"
Ejercicio propuesto 14: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "while"
Ejercicio propuesto 1.5: Crea un programa que muestre la "tabla de multiplicar del 5", usando "while"

5.2. do-while

Existe una variante de este tipo de bucle. Es el conjunto do..while, cuyo formato es:

```
do {
    sentencia;
} while (condición)
```

En este caso, la condición se comprueba al final, lo que quiere decir que las "sentencias" intermedias se realizarán al menos una vez, cosa que no ocurría en la construcción anterior (un único "while" antes de las sentencias), porque 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.

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:

```
do {
    System.out.println("Introduzca su clave de acceso");
    claveIntentada = LeerDatosUsuario();
} while (claveIntentada != claveCorrecta)
```

En este ejemplo hemos supuesto que existe algo llamado "LeerDatosUsuario", para que resulte legible. Pero realmente la situación no es tan sencilla: no existe ese "LeerDatosUsuario", ni sabemos todavía cómo leer información del teclado cuando trabajamos en modo texto (porque supondría hablar de excepciones y de otros conceptos que

todavía son demasiado avanzados para nosotros), ni sabemos crear programas "de ventanas" en los que podamos utilizar una casilla de introducción de textos. Así que de momento nos creeremos que algo parecido a lo que hemos escrito podrá llegar a funcionar... aunque todavía no lo hace.

Ejercicio propuesto 16: Crea un programa que muestre los números del 1 al 10, usando "do-while"
Ejercicio propuesto 17: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "do-while"
Ejercicio propuesto 18: Crea un programa que muestre la "tabla de multiplicar del 5", usando"do-while"

5.3. for

Una tercera forma de conseguir que parte de nuestro programa se repita es la orden "for". La emplearemos sobre todo para conseguir un número concreto de repeticiones. Su formato es

```
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:

```
for ( i=1 ; i<=10 ; i++ ) {
  ---
}
```

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),

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

```
for ( j = 10 ; j > 0 ; j -= 2 )
```

```
System.out.println( j );
```

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:

```
j = 20;
while ( j > 0) {
  System.out.println
  ( j ); j -= 2;
}
```

Ejercicio propuesto 19: Crea un programa que muestre los números del 1 al 10, usando "for"
Ejercicio propuesto 20: Crea un programa que muestre los números pares del 20 al 2, decreciendo, usando "for"
Ejercicio propuesto 21: Crea un programa que muestre la "tabla de multiplicar del 5", usando "for"
Ejercicio propuesto 22: Crea un programa que muestre los números menores de 100 que son múltiplos de 3 (el resto al dividir por 3 es 0) y a la vez múltiplos de 7 (ídem).

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

5.4. 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:

```
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:

```
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 23: Crea un programa que muestre los números del 1 al 20, excepto el 15, usando "for" y "continue"
Ejercicio propuesto 24: Crea un programa que muestre los números del 1 al 10, usando "for" que vaya del 1 al 20 y "break"

5.5. 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 un caso poco frecuente, así que no profundizaremos más, pero sí veremos un ejemplo:

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

En este caso, a mitad de la pasada 8 se saltaría hasta la posición que hemos etiquetado como "salida" (se define como se ve en el ejemplo, terminada con el símbolo de "dos puntos"), de modo que no se escribiría en pantalla el texto "Terminado" (lo hemos "saltado").

Ejercicio propuesto 24: Crea un programa que muestre los números del 1 al 10, usando "for" que vaya del 1 al 20 y un "break" con etiqueta.

6. Arrays y cadenas de texto.

Hemos visto cómo manejar tipos de datos básicos: varios tipos de datos numéricos, letras (char) y valores verdadero/falso (boolean). Pero para poder empezar a aplicar nuestros conocimientos en ejemplos medianamente complicados, nos interesa ver al menos un par de

tipos de datos más:

- Es muy habitual tener que manejar "bloques" de letras, que darán lugar a palabras o frases. Esto serán las "cadenas de texto" o "strings".

- También es frecuente tener que manejar "bloques" de números. Es algo que sonará familiar a quien haya estudiado estadística o álgebra matricial, por ejemplo. Para eso utilizaremos los "arrays" (palabra que algunos autores traducen por "arreglos", y que, en ciertos contextos, equivalen a "matrices" o a "vectores").

Veamos una introducción a ambos tipos de datos.

6.1. Los arrays.

Imaginemos que tenemos que hallar el promedio de 10 números que introduzca el usuario (o realizar cualquier otra operación con ellos). 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 (o arreglos). Un array es una variable que puede contener varios datos del mismo tipo. Para acceder a cada uno de esos datos emplearemos 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
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++) 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").

Nota: Quien venga de C y/o C++ tiene que tener en cuenta que en Java no son válidas definiciones como float a[5]; habrá que asignar los valores como acabamos de hacer, o reservar espacio de la forma que veremos a continuación.

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];
```

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

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

```
            System.out.println(
                  b[i] );

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

Ejercicio propuesto 25: Crea un programa que, a partir de un array de números reales, calcule y muestre su media y los valores que están por encima de la media.
Ejercicio propuesto 26: Crea un programa que, a partir de un array de números enteros, halle y muestre el valor más alto que contiene (su máximo) y el valor más bajo que contiene (su mínimo).

6.2. Las cadenas de texto.

Una cadena de texto (en inglés, "string") es un bloque de letras, que usaremos para poder almacenar palabras y frases. En algunos lenguajes, podríamos utilizar un "array" de "chars" para este fin, pero en Java no es necesario, porque tenemos un tipo "cadena" específico ya incorporado en el lenguaje.

Realmente en Java hay dos "variantes" de las cadenas de texto: existe una clase llamada "String" y otra clase llamada "StringBuffer". Un "String" será una cadena de caracteres constante, que no se podrá modificar (podremos leer su valor, extraer parte de él, etc.; para cualquier modificación, realmente Java creará una nueva cadena), mientras que un "StringBuffer" 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).

Vamos a ver las principales posibilidades de cada uno de estos dos tipos de cadena de texto y luego lo aplicaremos en un ejemplo.

Podemos "concatenar" cadenas (juntar dos cadenas para dar lugar a una nueva) con el signo +, igual que sumamos números. Por otra parte, los métodos de la clase String (las "operaciones con nombre" que podemos aplicar a una cadena) son:

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 conun 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.

Hay alguna otra posibilidad, de uso menos sencillo, que no veremos (al menos por ahora), como la de volcar parte del String en un array de chars o de bytes.

Los métodos de la clase **StringBuffer** 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 StringBuffer 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

Al igual que ocurre con los **strings**, existe alguna otra posibilidad más avanzada, que no he comentado, como la de volcar parte del **String** en un array de chars, o de comprobar la capacidad (tamaño máximo) que tiene un **StringBuffer** o fijar dicha capacidad.

Un comentario extra sobre los Strings: Java convertirá a String todo aquello que indiquemos entre comillas dobles. Así, son válidas expresiones como "Prueba".length() y también podemos concatenar varias expresiones dentro de una orden System.out.println:

```
// Strings1.java
// Aplicación de ejemplo con Strings
// Introducción a Java,

class Strings1 {
   public static void main( String args[] ) {

      String texto1 = "Hola"; // Forma "sencilla"
      String texto2 = new String("Prueba"); // Usando un "constructor"

      System.out.println( "La primera cadena de texto es
      :" );
      System.out.println( texto1 );
      System.out.println( "Concatenamos las dos: " + texto1 + texto2 );
      System.out.println( "Concatenamos varios: " + texto1 + 5 + " " +
      23.5 );
      System.out.println( "La longitud de la segunda es: " +
      texto2.length() );
      System.out.println( "La segunda letra de texto2 es: "
        + texto2.charAt(1) );
      System.out.println( "La cadena texto2 en mayúsculas: "
        + texto2.toUpperCase() );
      System.out.println( "Tres letras desde la posición 1: "
        + texto2.substring(1,3) );
```

```
        System.out.println( "Comparamos texto1 y texto2: "
          + texto1.compareTo(texto2) );

        if (texto1.compareTo(texto2) < 0)
          System.out.println( "Texto1 es menor que texto2" );

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

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

    }
}
```

El resultado de este programa sería el siguiente:

```
La primera cadena de texto es : Hola
Concatenamos las dos: HolaPrueba
Concatenamos varios: Hola5 23.5
La longitud de la segunda es: 6
La segunda letra de texto2 es: r
La cadena texto2 en mayúsculas: PRUEBA
Tres letras desde la posición 1: ru
Comparamos texto1 y texto2: -8
Texto1 es menor que texto2
Texto 3 ahora es: Otra prueba mas
Y ahora es: Ot1ra prueba mas
Y ahora: sam abeurp ar1tO
```

Ejercicio propuesto 28: 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:

```
V
VA
VAN
VANE
VANEG
VANEGA
VANEGAS
```

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

8. Datos introducidos por el usuario

8.1. Cómo leer datos desde consola.

Como ya hemos comentado en más de una ocasión, leer desde teclado usando Java no es trivial:

- Habrá que interceptar los posibles errores (por ejemplo, que ni siquiera exista un teclado). Por eso, toda la construcción de lectura de teclado deberá ir encerrada en un bloque "**try...catch**" ("intentar hacer... interceptar posibles errores"; veremos esta construcción más adelante con mayor detalle).

- Deberemos usar la entrada del sistema ("**System.in**"), pero enmascarándola dentro de un flujo de datos de entrada ("**InputStreamReader**"), al que deberemos acceder mediante un buffer intermedio ("**BufferedReader**").

- Por esta estructura, deberemos indicar que queremos usar ciertas funciones de entrada y salida ("**import java.io.***;").

Con todas esas consideraciones, un programa básico que pidiera su nombre al usuario y le saludara mediante ese nombre podría ser así:

```
//
// Ejemplo 1 de como pedir datos al usuario
//

import java.io.*;

class Hola {
  public static void main( String
    args[] ) { String nombre = "";

    System.out.print( "Introduzca su nombre: " );
    try {
      BufferedReader entrada =
                    new BufferedReader(new
                InputStreamReader(System.in));
      nombre = entrada.readLine();
    }
    catch (IOException e) {} System.out.println( "Hola, " + nombre );
  }
}
```

Es complejo como para hacerlo "de memoria". Más adelante profundizaremos más, pero por ahora nos limitaremos a "copiar y pegar" esa estructura para poder hacer programas básicos "en modo texto" que pidan datos al usuario.

Ejercicio propuesto 29: Crea un programa que te pida tu nombre y escriba un triángulo con las letras que lo forman, mostrando primero la primera letra, luego las dos primeras y así

sucesivamente, hasta llegar al nombre completo, como en este ejemplo:
E
ED
EDU
EDUA
EDUAR
EDUARD
EDUARDO

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

Ejercicio propuesto 30: Crea un programa que te pida tu nombre y escriba la primera letra en mayúsculas y el resto en minúsculas.

Ejercicio propuesto 31: Crea un programa que te pida tu nombre y escriba las letras separadas por espacios. Por ejemplo, a partir de "CESAR" escribirá "C E S A R".

7.2. Pedir datos numéricos.

También es habitual manejar datos numéricos. Como el resultado de "**readLine**" es una cadena de texto, tendremos que calcular su valor numérico. Para un número entero, esto se hace con "**Integer.parseInt**":

```
//
// Ejemplo 2 de como pedir datos al usuario
//

import java.io.*;

class Doble {
  public static void main( String args[] ) {
    int numero = 0;

    System.out.print( "Introduzca un numero: " );
    try {
      BufferedReader entrada =
        new BufferedReader(new InputStreamReader(System.in));
      numero = Integer.parseInt( entrada.readLine() );
    }
    catch (IOException e) {}
    System.out.println( "Su doble es: " + numero*2 );
  }
}
```

Los valores que demos dentro de un bloque **try...catch** no se pueden usar desde fuera de ese bloque. Por eso hemos estado dando valores iniciales a las variables, o el programa no compilaría. Otra alternativa poco elegante, pero sencilla y aceptable en un programa tan simple creado por un principiante sería que todo el fragmento de programa que manipule esas variables esté dentro del try...catch. Así, podríamos sumar dos números haciendo:

```
//
```

```
// Ejemplo 3 de como pedir datos al usuario
//

import java.io.*;
class Sumar {
    public static void main( String args[] ) {
     try {
        BufferedReader entrada =new BufferedReader(new InputStreamReader(System.in));
        System.out.print( "Introduzca un numero: " );
        int numero1 = Integer.parseInt( entrada.readLine() );
        System.out.print( "Introduzca otro numero: " );
        int numero2 = Integer.parseInt( entrada.readLine() );
        System.out.println( "Su suma es: " + numero1+numero2 );
     }
catch (IOException e) {}

}
}
```

Ejercicio propuesto 32: Crea un programa que te pida dos números (enteros) y muestre su división (entera) y el resto de dicha división. Por ejemplo, si los números son 16 y 3, la división sería 5 y el resto sería 1.

Ejercicio propuesto 33: Crea un programa que te pida dos números enteros y diga si el primero es múltiplo del segundo.

7.3. Otra forma aparentemente más sencilla

Si nos molesta tener que usar el bloque try...catch, hay una forma más compacta de escribir casi lo mismo, añadiendo "**throws**" a la declaración de la función, para indicar que puede lanzar una excepción:

```
//
// Otra forma de pedir datos al usuario
//

import java.io.*;

class Hola2 {
  public static void main( String args[] ) throws IOException {
    String nombre;
    System.out.print( "Introduzca su nombre: " );
    BufferedReader entrada =
      new BufferedReader(new InputStreamReader(System.in));
    nombre = entrada.readLine();
    System.out.println( "Hola, " + nombre );
  }
}
```

Pero cuidado: **NO** es exactamente lo mismo. Esta construcción avisa al programa "superior" de que esa función puede lanzar una excepción, de modo que estamos aplazando la comprobación de errores con try...catch, para que no la haga nuestra función, sino el programa que la llama. En nuestro caso, como nuestra función es el cuerpo del programa ("**main**"),

estamos pasando la pelota a... ¡nadie!... así que estamos despreciando por completo la comprobación de errores, que puede no ser una solución aceptable en problemas "reales".

Ejercicio propuesto 35: Crea un programa que te pida tu nombre y escriba las letras separadas por espacios, usando "**throws**" en vez de "try-catch". Por ejemplo, a partir de "PETRONILA" escribirá "P E T R O N I L A".

7.4. Usando la clase "Scanner"

A partir de la versión 5 de Java, tenemos otra posibilidad que simplifica ligeramente el acceso a teclado: la clase Scanner. En su uso más sencillo, le indicamos qué flujo de datos debe analizar (ya sea un fichero o la entrada estándar del sistema, "Sytem.in"), y vamos obteniendo el siguiente dato con ".**next**" (el siguiente hasta llegar a un espacio en blanco):

```
// Pedir datos al usuario de forma mas simple,
// palabra por palabra (Java 5 o superior)


import
java.io.*;
import
java.util.Scanner;

class
Scanner1 {
  public static void main( String args[] ) throws IOException {
    String nombre;
    System.out.print( "Introduzca su nombre (una palabra): " );
    Scanner entrada=new Scanner(System.in);
    nombre = entrada.next();
    System.out.println( "Hola, " + nombre );
  }
}
```

No sólo podemos leer cadenas de texto. Si lo siguiente que queremos leer es un número, podemos usar ".**nextInt**", ".**nextFloat**", ".**nextDouble**"... Y si queremos obtener más de un dato, podemos repetir con ".**hasNext**" ("tiene siguiente"), que nos devolverá verdadero o falso. Típicamente se usaría como parte de un bucle "**while**".

```
while (entrada.hasNext()) {
```

Finalmente, si queremos usar la clase "**Scanner**" para que lea línea a línea, en vez de interrumpirse cada vez que encuentre un espacio en blanco, podemos cambiar el "delimitador" con **".useDelimiter".** Por ejemplo, para que se interrumpa cuando llegue a un final de línea, indicaríamos como delimitador el que esté definido en la propiedad "line.separator" (separador de línea) del sistema, así:

```
//
// Pedir datos al usuario de forma más simple,
// permitiendo espacios (Java 5 o superior)
```

```
//

import java.io.*;
import java.util.Scanner;

class Scanner2 {
  public static void main( String args[] ) throws
    IOException { String nombre;

    System.out.print( "Introduzca su nombre (una o varias

    palabras): " ); Scanner entrada=new Scanner(System.in);
    entrada.useDelimiter(System.getProperty("line.separator"));
    nombre = entrada.next();

    System.out.println( "Hola, " + nombre );
  }
}
```

Ejercicio propuesto 36: Crea un programa que te pida tu nombre y escriba las letras separadas por espacios, usando "Scanner".

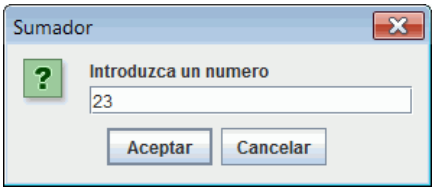
7.5. Cómo leer datos mediante una ventana.

Como lenguaje moderno que es, Java no sólo permite crear programas en "modo texto". También podemos usar "ventanas" como forma de comunicación con el usuario. Como toma de contacto, veremos apenas un par de posibilidades: los diálogos de entrada (**InputDialog**) para pedir datos, y los diálogos de mensaje (**MessageDialog**) para mostrar resultados:

A un "**InputDialog**" hay que indicarle varios detalles adicionales:

- La ventana de la que depende nuestro InputDialog; como todavía no usaremos más ventanas, para nosotros será "null"
- El texto que queremos que aparezca como pregunta al usuario, por ejemplo "Introduzca un número".
- El texto que queremos que aparezca en el título de la ventana, por ejemplo "Sumador".
- Las opciones, que indicarán el tipo de ventana del que se trata, para mostrar iconos adicionales que hagan la apariencia más vistosa. Por ejemplo, si es un "QUESTION_MESSAGE" aparecerá un icono con una interrogación, y si es un "INFORMATION_MESSAGE" aparecerá una letra "i".

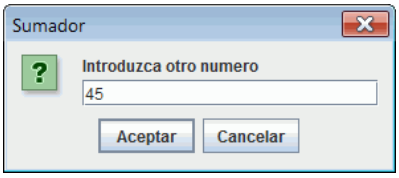
Así podremos obtener una ventana como ésta:



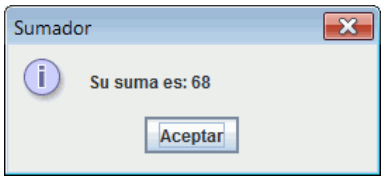
Al igual que pasaba con la orden "**readLine**", obtendremos una cadena de texto, que podríamos tratar como un número si usamos "**Integer.parseInt**". La fuente completa podría ser así:

```
//
// Ejemplo 4 de como pedir datos al usuario (interfaz grafico)
//

import javax.swing.JOptionPane;

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

      int numero1 = Integer.parseInt(JOptionPane.showInputDialog(
        null,"Introduzca un numero", "Sumador",
        JOptionPane.QUESTION_MESSAGE) );

      int numero2 = Integer.parseInt( JOptionPane.showInputDialog(
        null,"Introduzca otro numero", "Sumador",
        JOptionPane.QUESTION_MESSAGE) );

      int resultado = numero1+numero2;

      JOptionPane.showMessageDialog(null,"Su suma es: " + resultado,
        "Sumador", JOptionPane.INFORMATION_MESSAGE);
  }
}
```

En este programa se muestra una segunda ventana de introducción de datos:



Y el resultado se mostraría en una ventana de mensaje, como ésta:



Ejercicio propuesto 37: Crea un programa que te pida tu nombre (usando una ventana de introducción de datos) y lo muestre "al revés" (de la última letra a la primera), en una nueva ventana.

Ejercicio propuesto 38: Crea un programa que pida dos números enteros (usando ventanas de introducción de datos) y muestre su suma, en una nueva ventana.

7.6. Ejercicios propuestos de repaso.

Ahora que sabremos pedir datos al usuario, podemos practicar mucho más. Para los siguientes ejercicios, puedes usar entrada de datos mediante consola o mediante ventanas:

Ejercicio de repaso 39: Crear un programa que pida al usuario dos números enteros y diga cuál es el mayor de ellos.

Ejercicio de repaso 40: Crear un programa que pida una letra al usuario y diga si se trata de una vocal, usando "switch".

Ejercicio de repaso 41: Crear un programa que pida al usuario dos números enteros y diga "Uno de los números es positivo", "Los dos números son positivos" o bien "Ninguno de los números es positivo", según corresponda.

Ejercicio de repaso 42: Crear un programa que pida al usuario tres números reales y muestre cuál es el mayor de los tres (los números reales -con cifras decimales-, pueden ser de tipo "double" y entonces se convertirían con "Double.parseDouble".

Ejercicio de repaso 43: Crear un programa que pida al usuario su contraseña. Deberá terminar cuando introduzca como contraseña la palabra "clave", pero volvérsela a pedir tantas veces como sea necesario.

Ejercicio de repaso 44: Crear un programa calcule cuantas cifras tiene un número entero positivo que introduzca el usuario (pista: se puede hacer dividiendo varias veces entre 10).

Ejercicio de repaso 45: Crear un programa que pida al usuario una letra, y luego muestre las letras de la Z (mayúscula) a la A (mayúscula, descendiendo) excepto la que ha introducido el usuario.

Ejercicio de repaso 46: Crear un programa que pida al usuario un número "n" y que escriba en pantalla los números del 1 a "n" que sean múltiplos de 3 (pista: habrá que recorrer todos esos números y ver si el resto de la división entre 3 resulta 0).

Ejercicio de repaso 47: Crear un programa que pida un número entero al usuario y diga si ese número es primo.

Ejercicio de repaso 48: Crear un programa que descomponga un número (que teclee el usuario) como producto de sus factores primos. Por ejemplo, 60 = 2 · 2 · 3 · 5 (pista: puede ser más fácil escribirlo terminando siempre en " · 1", así: "60 = 2 · 2 · 3 · 5 · 1")

Ejercicio de repaso 49: Crear un programa que dé al usuario la oportunidad de adivinar un número del 1 al 100 (prefijado en el programa) en un máximo de 6 intentos. En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.

Ejercicio de repaso 50: Crear un programa que pida al usuario cinco números reales y muestre su media, sin memorizar todos los datos.

Ejercicio de repaso 51: Crear un programa que pida al usuario cinco números reales y muestre su media, almacenando los datos en un array.

Ejercicio de repaso 52: Crear un programa que pida al usuario 5 números enteros y luego los muestre en el orden contrario al que se introdujeron.

Ejercicio de repaso 53: Un programa que pida 10 nombres y los memorice. Después deberá pedir que se teclee un nombre y dirá si se encuentra o no entre los 10 que se han tecleado antes. Volverá a pedir otro nombre y a decir si se encuentra entre ellos, y así sucesivamente hasta que se teclee "fin".

Ejercicio de repaso 54: Crear un programa que pida al usuario dos números enteros y muestre su máximo común divisor.

Ejercicio de repaso 55: Crear un programa que pida al usuario una frase y diga si es palíndroma (se lee igual al derecho que al revés, como "Ama").

Ejercicio de repaso 56: Crear un programa que pida al usuario dos números enteros y muestre los números primos que hay entre ellos (incluidos). Por ejemplo, si los números son 5 y 8, la respuesta sería "5 7".

Ejercicio de repaso 57: Crear un programa que pida al usuario el radio de una circunferencia (un número real de simple precisión) y muestre la longitud de la circunferencia y la superficie del círculo correspondiente (tendrás que localizar las correspondientes fórmulas, si no las conoces).

Ejercicio de repaso 58: Crear un programa que pida al usuario una cantidad indeterminada de nombres (hasta que pulse "Intro" sin introducir ningún dato) y los guarde en un array. Luego deberá pedir al usuario una letra y mostrar los nombres que comienzan por esa letra, y repetir esa petición hasta que el usuario introduzca "fin" en vez de una letra.

8. Contacto con las funciones

8.1. 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 * 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:

```
//
// 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:

```
//
// 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:

```
//
// 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 59: 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 60: Crea una función llamada "**dibujarCuadrado3x3**", que dibuje un cuadrado formato por 3 filas con 3 asteriscos cada una. Crea también un "**main**" que permita probarla.

8.2. 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 61: 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 62: 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.

8.3. 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 quedaría que usara todas esas funciones quedaría:

```
//
// Primer 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 63: Crear una función que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5.

Ejercicio propuesto 64: Crear una función que calcule el menor de dos números enteros que recibirá como parámetros. El resultado será otro número entero.

Ejercicio propuesto 65: Crear una función que devuelva la primera letra de una cadena de texto. Probar esta función para calcular la primera letra de la frase "Hola".

Ejercicio propuesto 66: Crear una función que devuelva la última letra de una cadena de texto. Probar esta función para calcular la última letra de la frase "Hola".

Ejercicio propuesto 67: Crear una función "esPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.

9. Clases en Java.

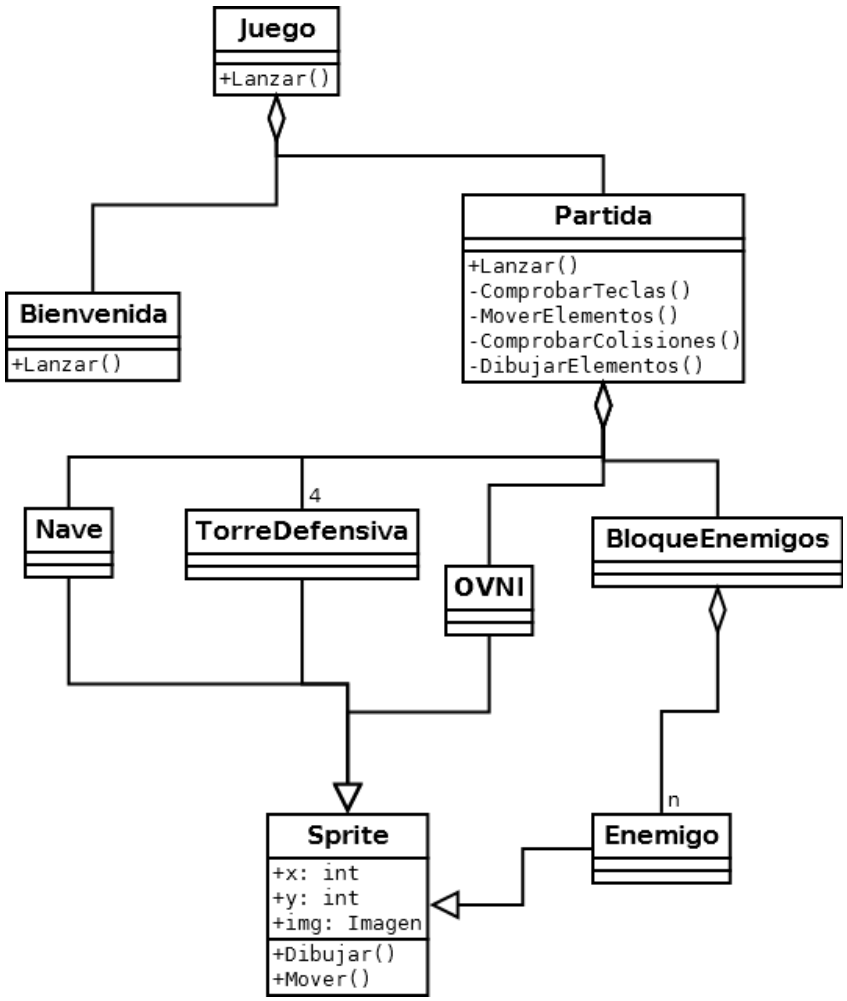
Cuando tenemos que realizar un proyecto grande, será necesario descomponerlo en varios subprogramas, de forma que podamos repartir el trabajo entre varias personas (pero la descomposición no debe ser arbitraria: por ejemplo, será deseable que cada bloque tenga unas responsabilidades claras).

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**" (esa 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.

9.1. 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,

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 <Principal>");
  }
}

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

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:

    1. 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.
    2. Después llamamos al método "**saluda**" de dicho objeto, con la expresión "**s.saluda()**"
    3. 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

que podríamos probar tecleando

java Principal

El resultado se mostraría en pantalla es:

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"

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"

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 varias 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 68: 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 (o una cadena vacía en caso de error). Crea también una clase "PruebaTeclado", que use la anterior.

9.2. 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
```

```
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 "sabrá 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 69: Crea una nueva clase "EscritorMayusculasEspaciado" que se apoye en "EscritorMayusculas", pero reemplace cada espacio en blanco por tres espacios antes de escribir en pantalla.

9.3. 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 visibles, mientras que procuraremos que los estén ocultos.

- **Privado (privado)**, 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

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 70: 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.

9.4. 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"

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 71: Crea una versión del ejercicio 70, que no utilice "static".

9.5. 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

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, así que no los veremos por ahora.

Ejercicio propuesto 72: Crea una nueva versión de la clase "EscritorDosMargenes" que use un constructor para indicarle el texto, el margen izquierdo y el margen derecho.

10. 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, e incluso otras operaciones que son casi exclusivas del mundo informático, las llamadas "operaciones a nivel de bits" (negación, producto lógico, suma lógica, desplazamientos).

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
abs()	Valor absoluto
acos()	Arcocoseno
asin()	Arcoseno
atan()	Arcotangente entre -PI/2 y PI/2
atan2(,)	Arcotangente entre -PI y PI
ceil()	Entero mayor más cercano
cos(double)	Coseno
exp()	Exponencial
floor()	Entero menor más cercano
log()	Logaritmo natural (base e)
max(,)	Máximo de dos valores
min(,)	Mínimo de dos valores
pow(,)	Primer número elevado al segundo
random()	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
```

```
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 73: Crear 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.

11. Programas para la web: los applets y los servlets

11.1. ¿Qué es un applet?

Un applet es una pequeña aplicación escrita en Java, pero que no está diseñada para funcionar "por sí sola", como los ejemplos que habíamos visto en modo texto, sino para formar parte de una página Web, de modo que necesitaremos también un navegador web (como Internet Explorer, Mozilla Firefox o Google Chrome) para poder hacer funcionar dichos applets. Hoy en día se da por sentado que cualquier ordenador tendrá una navegador web incorporado, pero las primeras versiones del entorno de desarrollo de Java incluye una utilidad adicional llamada "appletviewer", que nos permitirá probar nuestros applets sin necesidad de tener uno instalado. Hoy en día los "applet" de Java están en desuso. Es más frecuente encontrar páginas web creadas con Flash, otra herramienta que permite crear con rapidez vistosos programas para la Web. Aun así, veremos al menos una introducción a la creación de "applets".

11.2. ¿Cómo se crea un applet?

Tendremos que dar tres pasos:

1. El primer paso, similar a lo que ya hemos hecho, consiste en teclear el programa empleando un editor de texto o un entorno integrado.
2. El segundo paso es nuevo: tendremos que crear la página Web desde la que se accederá al applet (o modificarla, si se trata de una página que ya existía pero que queremos mejorar).
3. El último paso será, al igual que antes, probar el programa, pero esta vez no emplearemos directamente la utilidad llamada "java", sino cualquier navegador Web.

Vamos a verlo con un ejemplo. Crearemos un applet que escriba el texto "Hola Mundo!" en pantalla, al igual que hicimos en nuestra primera aplicación de ejemplo. Recordemos que lo que habíamos escrito en aquel momento fue:

```
//
// Aplicación HolaMundo de ejemplo
//

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

Esta vez tendremos que escribir más. Primero vamos a ver cómo debería ser el applet, y después comentaremos los cambios (bastantes) con relación a este programa:

```
//
// Primer Applet de ejemplo
//

import java.awt.Graphics;

public class AppletHola extends java.applet.Applet {

    public void paint(Graphics g) {
        g.drawString( "Hola Mundo!", 100, 50);
    }
}
```

Los cambios son los siguientes:

- Al principio aparece una orden "**import**", porque vamos a emplear una serie de posibilidades que no son parte del lenguaje Java "estándar", sino del paquete "Graphics" (funciones gráficas) que a su vez forma parte del grupo llamado AWT ("Abstract Windowing Toolkit", herramientas "abstractas" para el manejo de ventanas).
- Después no creamos una clase nueva "desde cero", sino heredando una que ya existe, la clase "Applet". Por eso aparece la palabra "extends" en la definición de la clase, como ya vimos en el apartado sobre las clases y Java.
- Sólo utilizamos un método de los que nos permite la clase Applet: el método "paint", que es el que dibuja (o escribe) en pantalla la información que nos interese. Tiene un

parámetro de tipo "Graphics", que es la pantalla en la que dibujaremos.
- En concreto, lo que hacemos en esta pantalla es dibujar una cadena de texto (drawString), en las coordenadas 100 (horizontal), 50 (vertical).

(Un poco más adelante se verá el resultado de este Applet).

Y vamos a ver los tres pasos que tenemos que dar, con mayor detalle.

11.3. Primer paso: teclear el fuente.

Repetimos básicamente lo que ya vimos:

- Tecleamos el fuente con Geany, el bloc de notas de Windows, o cualquier otro editor de textos.
- Lo guardamos, con el mismo nombre que tiene la clase. En nuestro caso sería "AppletHola.java" (si empleamos otro nombre, el compilador "javac" nos regañará).
- Lo compilamos empleando "javac" (recordemos que desde algunos editores, como Geany, permiten compilar directamente desde dentro de ellos).
- Si hay algún error, se nos informará de cual es; si no aparece ningún error, obtendremos un fichero llamado "AppletHola.class", listo para utilizar.

11.4. Segundo paso: crear la página Web.

No pretendo "formar maestros" en el arte de crear páginas Web. Apenas veremos lo necesario para poder probar nuestros Applets.

Las páginas Web internamente son ficheros de texto, escritos en un cierto lenguaje. El lenguaje más sencillo, y totalmente válido para nuestros propósitos, es el lenguaje HTML. Los ficheros HTML son ficheros de texto, que podremos crear con cualquier editor de texto (esta es la forma más trabajosa, pero con la que tenemos mayor control) o con editores específicos (como ***Dreamweaver***, ***Frontpage***, ***SeaMonkey Composer***, , que permiten "crear" más rápido, aunque a veces eso supone perder en funcionalidades y/o entender menos cada paso que se da).

En nuestro caso, recurriremos nuevamente a Geany (o al bloc de notas de Windows, o cualquier otro editor de textos) y escribiremos lo siguiente:

```
<html>
  <head>
    <title>Prueba de applet</title>
        </head>
  <body>
    <h1>Prueba de applet</h1>
    <hr>
    <applet code=AppletHola.class width=300 height=120>
          alt="El navegador no puede mostrar el APPLET"
    </applet>
  </body>
```

```
</html>
```

Grabaremos este texto con el nombre "prueba.html" (serviría cualquier otro nombre, con la única condición de que termine en ".htm" o en ".html") para utilizarlo un poco más adelante.

Vamos a ver qué hemos escrito:

- Es un fichero de texto, en el que aparecen algunas "etiquetas" encerradas entre < y >. El propio fichero en sí comienza por la etiqueta <html> y termina con </html> (esto será habitual: la mayoría de las etiquetas van "por pares", la primera crea un "bloque" y la segunda -la que contiene una barra al principio- indica el final del bloque).
- Después aparece otro bloque, opcional: la cabecera del fichero, que empieza con <head> y termina con </head>.
- Dentro de la cabecera hemos incluido una información que también es opcional: el "título" de la página, delimitado entre <title> y </title>. Este dato no es necesario, pero puede resultar cómodo, porque aparecerá en la barra de título del menú (ver las imágenes de ejemplo, un poco más abajo).
- Después va el cuerpo del fichero html, que se delimita entre <body> y </body>. (El cuerpo sí es obligatorio).
- Dentro del cuerpo, lo primero que aparece es un título (en inglés, "heading"), también opcional, delimitado entre <h1> y </h1>. Tenemos distintos niveles de títulos. Por ejemplo, podríamos poner un título de menor importancia delimitándolo entre <h2> y </h2>, o más pequeño aún entre <h3> y </h3>, y así sucesivamente.
- La etiqueta <hr> muestra una línea horizontal (en inglés, "horizontal ruler"; claramente, también es opcional), y no necesita ninguna etiqueta de cierre.
- Finalmente, la parte en la que se indica el applet comienza con <applet> y termina con </applet>. También es opcional en un "caso general", pero imprescindible en nuestro caso. En la etiqueta de comienzo se pueden indicar una serie de opciones, que veremos después con más detalle. Nosotros hemos utilizado sólo las tres imprescindibles:
- code sirve para indicar cuál es el nombre del applet a utilizar
- width y height indican la anchura y altura de un rectángulo, medida en puntos de pantalla (pixels). Este rectángulo será la "ventana de trabajo" de nuestro applet.

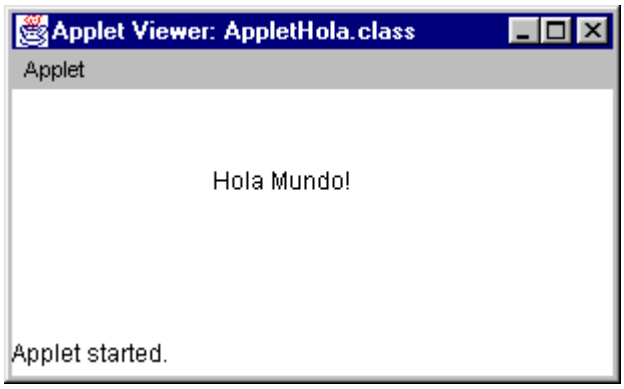
El resultado de todo esto se puede ver en el siguiente apartado...

11.5. Tercer paso: probar el resultado.

Si empleásemos la utilidad "AppletViewer" que incorporaban las primeras versiones del JDK, teclearíamos:

```
appletviewer  prueba.html
```

Y al cabo de un instante, tendríamos en pantalla algo parecido a:



También podemos usar cualquier otro navegador que soporte Java (cualquiera moderno debería permitirlo si está instalada la máquina virtual Java). Por ejemplo, el Netscape Navigator 4.5 del año 1998 mostraría lo siguiente:

Y con el vetusto Internet Explorer 4 de Microsoft, del año 1997, veríamos

Las últimas versiones de Internet Explorer, Netscape Navigator y Mozilla ya no incluyen Java en su instalación normal, pero si hemos descargado el JDK desde la página de Sun (java.sun.com)

Se debería haber añadido automáticamente a nuestro navegador.

Y si no tenemos el JDK, porque no somos programadores, pero queremos usar Java (es lo que le ocurriría normalmente a los usuarios de nuestros programas), deberíamos descargar sólo la máquina virtual Java (el "Java Runtime Environment", JRE), que se instalaría automáticamente en nuestro(s) navegador(es).

11.6. La "vida" de un applet.

En los applets hay un detalle importante que hemos dejado pasar hasta ahora: no hemos utilizado una función "main" que represente el cuerpo de la aplicación. En su lugar, hemos empleado el método "paint" para indicar qué se debe mostrar en pantalla. Esto se debe a que un applet tiene un cierto "ciclo de vida":

- Cuando el applet se carga en el navegador, se llama a su método "**init**" (que nosotros no hemos "reutilizado"). Hace las funciones de un constructor.
- Cuando el applet se hace visible, se llama a su método "**start**" (que tampoco hemos empleado). Se volverá a llamar cada vez que se vuelva a hacer visible (por ejemplo, si ocultamos nuestro navegador y luego lo volvemos a mostrar).
- Cuando el applet se va a dibujar en pantalla, se llama a su método "**paint**" (como hemos hecho nosotros).
- El método "**repaint**" se utiliza para redibujar el applet en pantalla. Es normalmente el

que nosotros utilizaremos como programadores, llamándolo cuando nos interese, en vez de "**paint**", que será el que se llame automáticamente. A su vez, "**repaint**" llama internamente a "**update**", que en ciertos casos nos puede interesar redefinir (por ejemplo, si no queremos que se borre toda la pantalla sino sólo una parte). Ambos los veremos con más detalle un poco más adelante.

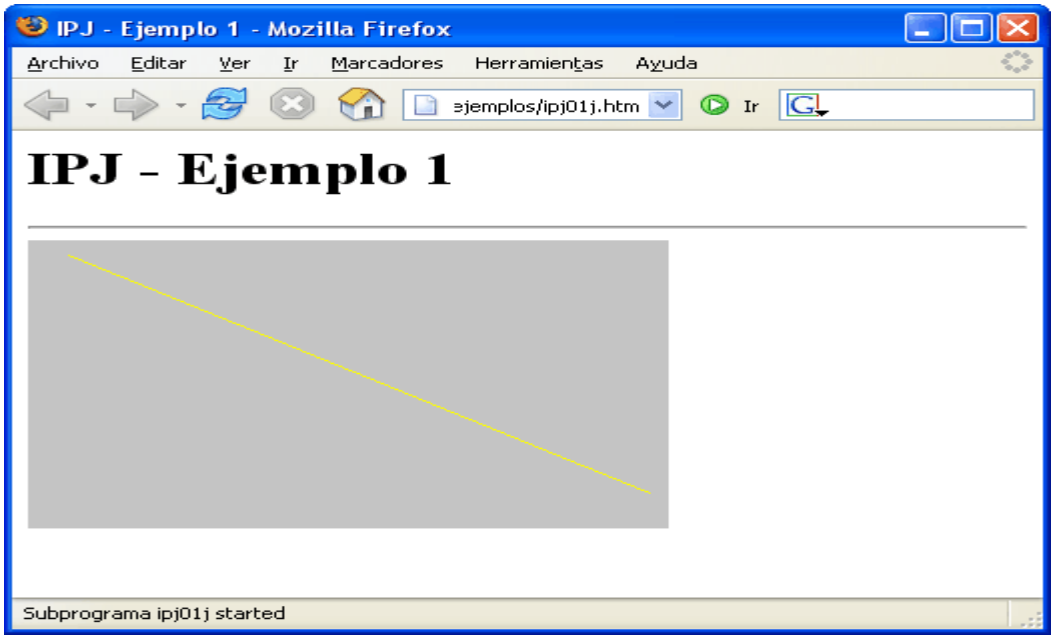
- Cuando el applet se oculta (por ejemplo, si minimizamos la ventana de nuestro navegador), se llama a su método "**stop**". Sería conveniente que desde él parásemos las animaciones o cualquier otra tarea que consumiese recursos "sin necesidad".
- Cuando se termina de utilizar el applet, se llama a su método "**destroy**".

11.7. Dibujar desde un applet

Un ejemplo básico de cómo cambiar a modo gráfico y dibujar una diagonal en un Applet podría: ser así:

```
/*-----------------------------*/
/*  Applet de ejemplo: entra a */
/*  modo grafico y dibuja una  */
/*  linea diagonal en la pan-  */
/*  talla.                     */
/*-----------------------------*/
import java.awt.*;
public class ipj01j extends java.applet.Applet {
    public void paint(Graphics g) {
    g.setColor( Color.yellow );
    g.drawLine( 20, 10,      310, 175 );
    }
}
```

Como es habitual para los Applets, deberemos compilar el fichero y crear la página web que nos permita lanzarlo. Si nuestro navegador no reconoce el lenguaje Java, veríamos el aviso "El navegador no puede mostrar el APPLET", pero si todo ha ido bien, debería aparecer algo como



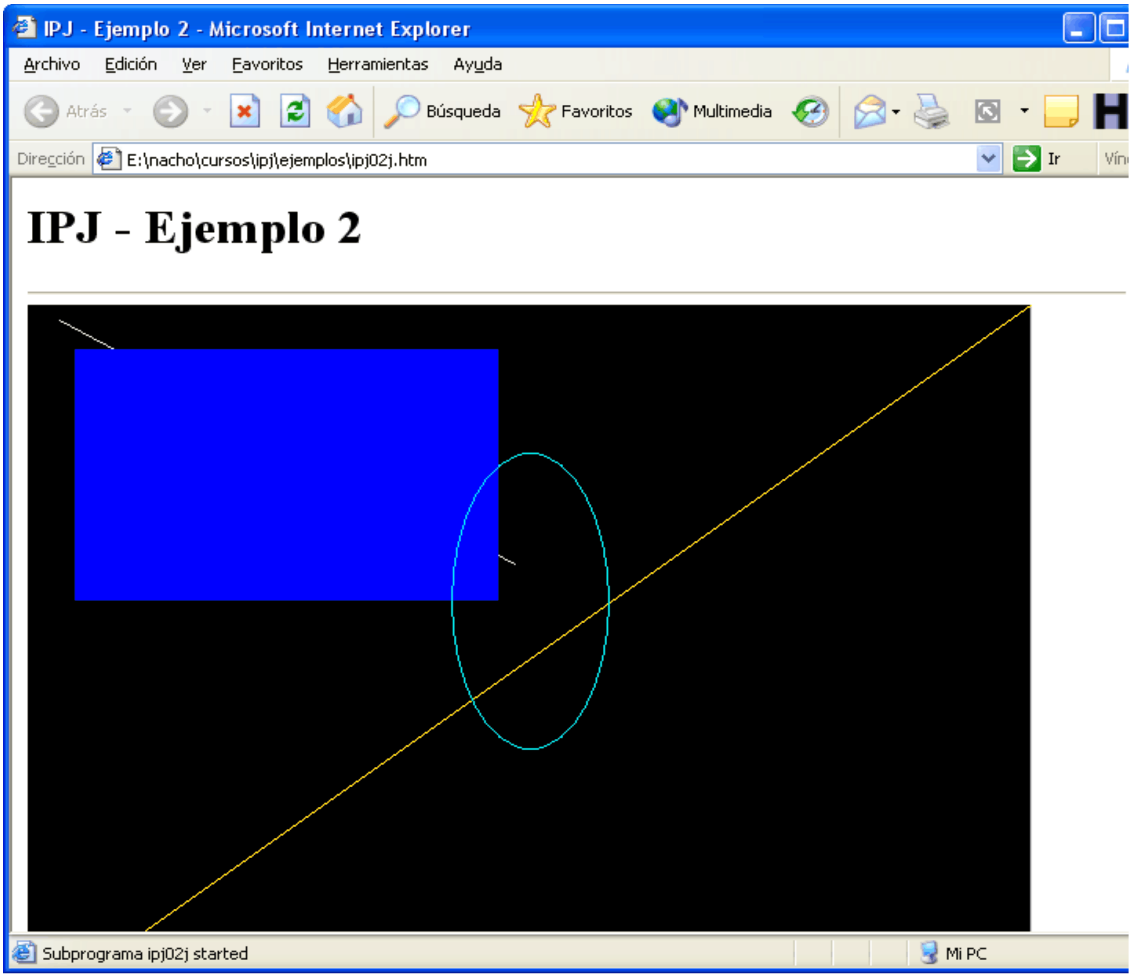
Las posibilidades más básicas son:

```
// Escribir texto: mensaje, coordenada x (horizontal) e y (vertical)
g.drawString("Hola Mundo!",100,50 );

// Color: existen nombres predefinidos
g.setColor( Color.black );

// Linea: Coordenadas x e y de los extremos
g.drawLine( x1, y1, x2, y2 );

// Rectangulo: Origen, anchura y altura
g.drawRect( x1, y1, ancho, alto );


// Rectangulo relleno: identico
g.fillRect( x1, y1, ancho, alto );

// Rectangulo redondeado: similar + dos redondeos
g.drawRoundRect( x1, y1, x2, y2, rnd1, rnd2 );

// Rectangulo redondeado relleno: igual
g.fillRoundRect( x1, y1, x2, y2, rnd1, rnd2 );

// Ovalo (elipse): Como rectangulo que lo rodea
g.drawOval( x1, y1, ancho, alto );

// Ovalo relleno: identico
g.fillOval( x1, y1, ancho, alto );
```

Un programa de prueba que utilice algunas de estas posibilidades podría quedar así:

```
/*----------------------------*/
/*  Applet de ejemplo: dibuja */
/*  figuras basicas           */
/*----------------------------*/

import java.awt.*;

public class ipj02j extends java.applet.Applet {

    public void paint(Graphics g)
                {
        // Primero borro el fondo en
        negro g.setColor( Color.black );
        g.fillRect( 0, 0, 639, 479 );
        // Y ahora dibujo las figuras del ejemplo
        g.setColor( Color.white );
        g.drawLine( 20, 10,  310, 175 );
        g.setColor( Color.yellow );
        g.drawLine( 639, 0,    0, 479);
        g.setColor( Color.blue );
        g.fillRect( 30, 30, 270, 170 );
        g.setColor( Color.cyan );
        g.drawOval( 270, 100, 100, 200
        );
    }
}
```

Y el resultado sería este:



Como los applets casi no se usan hoy en día, no veremos más detalles. En vez de eso, seguiremos profundizando con las posibilidades básicas de Java como lenguaje y con otras bibliotecas auxiliares que nos permitirán (por ejemplo) dibujar en una aplicación convencional, sea para el sistema operativo que sea.

11.8. ¿Y los servlets?

Un applet está diseñado para ejecutarse desde un navegador web, es decir, en el ordenador de un "cliente", si pensamos en una arquitectura cliente-servidor.

De forma alternativa, existe la posibilidad de que, en el servidor al que conecta nuestro navegador web, sea un programa Java el que responda a nuestras peticiones. Este tipo de aplicaciones Java que se implementan en un servidor web para aumentar las funcionalidades se suelen conocer como "servlets", pero no veremos más detalles sobre ellos... al menos por ahora...

12. Dibujar desde Java

Hemos visto las ideas básicas sobre cómo dibujar desde un Applet, pero también hemos comentado que se trata de una tecnología en desuso.

12.1. Java2D

(Pronto disponible)

13. Ficheros

13.1. ¿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.

13.2. 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:

```
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 74: 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".

13.3. 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 esnull, 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:

```
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 75: 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 76: 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 77: 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.

13.4. 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":

```
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 78: 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 79: 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).

13.5. 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:

```
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() );
         }
    }
}
```

Ejercicio propuesto 80: 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 81: Crea una versión del ejercicio 79, 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).