Primeros pasos con Java

1 El fichero fuente

En Java, los ficheros que contienen el código fuente que escribe el programador llevan la extensión **.java**, y pueden ser escritos con cualquier editor de texto[1]. En el siguiente cuadro se puede ver el típico ejemplo de primer programa en Java, que guardaremos en el fichero *HolaMundo.java* ubicado en nuestro directorio ***src***.

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

Como vemos, el programa está formado por una clase **HolaMundo**, que a su vez contiene un método ***main***. Este método contiene una única instrucción, que invoca el método **println** del objeto **out**, que a su vez pertenece a la clase **System**. La clase **System** es una clase del sistema, que está siempre disponible para ser usada sin necesidad de declarar o crear nada.

En este primer ejemplo, debemos destacar 3 cosas:

1. El nombre de la clase que contiene el fichero debe ser el mismo[2] que el nombre del fichero (sin la extensión **.java**).
2. En Java, los identificadores distinguen entre mayúsculas y minúsculas (por ejemplo, escribir '*string*' en lugar de '*String*' provocará un error del compilador, al no saber a qué se refiere el identificador '*string*').
3. Para que una clase se pueda ejecutar, debe contener un método ***main*** con un parámetro de tipo *String[]* (un array de cadenas, como veremos más adelante).

2 El Java Development Kit (JDK)

Para programar en Java se usa un entorno de trabajo conocido como el Java Development Kit (JDK) que, además del compilador (programa ***javac***), incluye otros programas y una serie de librerías que componen la API de Java, y que pueden ser usadas por cualquier programador para construir sus programas.

El entorno Java (lenguaje y APIs) está vivo y en constante evolución. Año tras año se van creando nuevas APIs y modificando las anteriores, a la vez que se van añadiendo al propio lenguaje nuevas características (palabras clave, instrucciones, convenciones…). Todo ello bajo el control de la empresa Oracle, la propietaria de los derechos de Java.

Por ello, no hay un único JDK, sino que hay una serie de ellos, con números de versión incrementales, y que recogen las características del entorno en un momento determinado. Por ejemplo:

- Java 8: JDK publicado en marzo de 2014.
- Java 11: JDK publicado en septiembre de 2018.
- Java 17: JDK publicado en septiembre de 2021.

Por supuesto, también existen las versiones intermedias, como Java 9 o Java 15, pero las tres anteriores son versiones importantes: Java 8 inició un nuevo esquema de numeración, y las otras dos son versiones denominadas LTS (*Long Term Support*), para las que Oracle ofrece un ciclo de vida (mantenimiento y soporte -de pago-) más largo del habitual.

En la asignatura *Programación II* vamos a usar la versión Java 11.

---

[1] Nunca emplees un procesador de textos (como Microsoft Word), ya que pueden introducir caracteres de formato ocultos que el compilador interpreta como errores, y que a veces resultan difíciles de detectar.

[2] Esto no siempre tiene por qué ser así, pero, por ahora, respetaremos este convenio.

El JDK se instala en una ruta determinada del sistema de ficheros, que dependerá del sistema operativo, y que podemos elegir o modificar. Normalmente, la variable de entorno JAVA_HOME apunta a esa ruta, que en los ordenadores del laboratorio debería ser:

```
JAVA_HOME = /usr/lib/jvm/java-11-openjdk-amd64
```

Los comandos que usaremos habitualmente durante el desarrollo y la ejecución de programas Java (por ejemplo, el compilador ***javac***) están en el directorio *bin* del JDK, es decir, en el directorio *JAVA_HOME/bin*, que en el laboratorio es */usr/lib/jvm/java-11-openjdk-amd64/bin*. Si al ejecutar el siguiente comando:

```
~/src> javac -version
```

no informa de que es la versión 11, seguid los pasos de la entrada de MOOVI "Configurar el JDK 11".

3 La compilación de un fichero fuente

3.1 El compilador

Como hemos dicho, el compilador del lenguaje Java se llama ***javac***. Se trata de un programa que toma como entrada un fichero **.java** (el fichero fuente), lo compila, y genera un fichero **.class**, que recibe el nombre de *bytecode*. El *bytecode* es un código intermedio entre el código fuente y el código objeto. Por tanto, no es algo directamente ejecutable, como ocurre al compilar en C, sino que necesita ser leído, interpretado y ejecutado por otro programa, como veremos más adelante.

Por tanto, para compilar, ejecutaremos en un terminal el comando del siguiente cuadro.

```
~/src> javac HolaMundo.java
```

Tras ejecutar el comando, tendremos en el mismo directorio un nuevo fichero llamado *HolaMundo.class*.

Es posible compilar varios ficheros a la vez. Por ejemplo, el comando del siguiente cuadro genera los ficheros *HolaMundo.class*, *Auxiliar.class* y *Prueba.class* (suponiendo que existieran los correspondientes **.java**).

```
~/src> javac HolaMundo.java Auxiliar.java Prueba.java
```

3.2 Argumentos del programa *javac*

Existen una serie de argumentos que podemos emplear con el programa ***javac***, y que modificarán su comportamiento. Por ejemplo:

- **-help** (para ver todas las opciones)

  ```
  ~/src> javac -help
  ```

- ***-version*** (para ver la versión del JDK al que pertenece el compilador, como ya vimos)

- ***-d*** (para indicar que los ficheros .**class** deben generarse en otro directorio)

  El siguiente comando crea *HelloWorld.class* en el subdirectorio ***classes*** del directorio ***src***:

  ```
  ~/src> javac -d classes HelloWorld.java
  ```

  Y el siguiente crearía *HelloWorld.class en el directorio* ***/tmp/classes*** (si existe tal directorio)*:*

  ```
  ~/src> javac -d /tmp/classes HelloWorld.java
  ```

3.3 El CLASSPATH

A veces, las clases que escribimos dependen de otras clases (**NO** es el caso del único ejemplo que hemos visto, el *HolaMundo.java*). Pueden ser, por ejemplo:

- Clases nuestras, compiladas previamente.
- Clases de una librería externa (que normalmente están dentro de ficheros con la extensión **.jar**).

En ese caso, para que nuestro programa sea compilado sin errores, debemos indicarle al programa ***javac*** dónde debe buscar esas clases, en qué directorios o ficheros **.jar** debe buscarlas. Para ello, hay dos opciones:

1. Declarar esos directorios o ficheros **.jar** en la variable de entorno CLASSPATH.
   - Linux: *export CLASSPATH=/directorio1:/directorio2/lib.jar:$CLASSPATH*
   - Windows: *set CLASSPATH=C:\directorio1;C:\directorio2\lib.jar;%CLASSPATH%*

   ¡Atención! Las distintas rutas se separan con el carácter ':' en Linux, mientras que en Windows es ';'.

2. Indicar esos directorios o ficheros **.jar** al programa ***javac*** con el argumento *-classpath* (o también *-cp*). Por ejemplo, y suponiendo que queremos compilar en Linux el fichero *OtroProgramaJava.java* que **SI** depende de clases externas que están en alguna de las rutas del CLASSPATH:

```
~/src> javac -cp /directorio1:/directorio2/lib.jar OtroProgramaJava.java
```

El uso de *-classpath* (o *-cp*) es prioritario respecto a la variable CLASSPATH. Si se usa *-classpath* (o *-cp*) en el comando ***javac***, no se hará caso al contenido de la variable CLASSPATH.

3.4 Posibles problemas durante la compilación

- Por defecto, el compilador ***javac*** busca el fichero fuente a compilar en el directorio actual. Por tanto, antes de compilar, hay que asegurarse de que el fichero fuente está en el directorio en el cual ejecutamos el ***javac***. O también es posible compilar un fichero fuente que esté en otro lugar indicando su ruta. Por ejemplo, si estamos en el directorio anterior al ***src***, el siguiente comando generaría el **.class** dentro de ***src***:

```
~/> javac src/HolaMundo.java
```

- Si no se encuentra el comando ***javac***, posiblemente se debe a que la ruta donde está el programa ***javac*** (JAVA_HOME/bin) no está entre las rutas donde se buscan los ejecutables. Esas rutas se indican en la variable de entorno PATH, por lo que hay que añadir la ruta del directorio donde está el programa ***javac*** a esa variable (si hay rutas a distintos JDKs en esa variable, se ejecutará el primero de ellos, por lo que éste debe ser el del Java 11).

  Para añadir rutas al valor de esa variable, podemos usar los siguientes comandos en el terminal:

  - Linux: *export PATH=/usr/lib/jvm/java-11-openjdk-amd64/bin:$PATH*
  - Windows: *set PATH=C:\jdkX.Y\bin;%PATH%;*

  En ambos casos se cogerá el valor actual de PATH (*$PATH* o *%PATH%*) y se le añadirá al principio la ruta indicada al directorio ***bin*** del JDK. Recuerda que el separador en Linux es el carácter ':', mientras que en Windows es ';'.

  También es posible realizar esa modificación de forma permanente a través de los ficheros de configuración de vuestra cuenta. Por ejemplo, en el caso de Linux, suponiendo que se use el shell *bash*:

  - Crea el fichero *~/.bashrc*, que contenga la línea:
    export PATH=/usr/lib/jvm/java-11-openjdk-amd64/bin:$PATH
  - Crea el fichero *.bash_profile*, que contenga la línea:
    source ~/.bashrc

- Si el compilador indica que no se encuentra alguna clase auxiliar, comprueba que la ruta a esa clase (o su fichero **.jar**) está en la variable CLASSPATH o en el parámetro *-classpath* (o *-cp*), si se usa.

4 Ejecutando el programa

4.1 El programa *java*

La lectura y ejecución del *bytecode* de una clase (siempre que tenga un método ***main***, recuerda que es algo imprescindible) la realiza un programa que se denomina Máquina Virtual Java (*Java Virtual Machine* o JVM), y que se invoca mediante el comando ***java***, que se encuentra también en el directorio *JAVA_HOME/bin*.

```
~/src> java HolaMundo
```

¡IMPORTANTE! En este caso, lo que se le pasa al programa ***java*** no es el nombre del fichero **.class**, sino el nombre de la clase a ejecutar (la clase que contiene el método ***main***).

Por defecto, el programa ***java*** busca la clase a ejecutar en el directorio actual (debemos asegurarnos de que el fichero *HolaMundo.class* está en el directorio actual).

4.2 Ejecutar una clase que no está en el directorio actual

Si queremos ejecutar una clase que no está en el directorio actual, debemos indicarle al programa ***java*** dónde debe buscar esa clase, en qué directorios debe buscarla. Para ello, hay las dos opciones que ya vimos:

1. Declarar esos directorios en la variable de entorno CLASSPATH.
   - Linux: *export CLASSPATH=./classes:$CLASSPATH*
   - Windows: *set CLASSPATH=./classes;%CLASSPATH%;*

   Suponiendo que *HolaMundo.class* está en ***src/classes***, podríamos ejecutar desde ***src***:

   ```
   ~/src> java HolaMundo
   ```

2. Proporcionar esos directorios al programa ***java*** con el argumento *-classpath* (o *-cp*). Por ejemplo, volviendo a suponer que *HolaMundo.class* está en ***src/classes***:

```
~/src> java -cp classes HolaMundo
```

```
~/src> java -classpath /ruta-absoluta-a-classes HolaMundo
```

Recuerda que el uso de *-classpath* o *-cp* es prioritario respecto a la variable CLASSPATH. Si se usa *-classpath* (o *-cp*) en el comando, no se hará caso al contenido de la variable CLASSPATH.

4.3 Los argumentos del programa

Como vemos, el método ***main*** contiene un parámetro llamado `args` (de tipo `String[]`, es decir, un array de cadenas), en el que el programa recibe cualquier argumento con el que se le invoque en la línea de comandos.

El número de argumentos recibido se puede conocer consultando la propiedad *args.length*, mientras que el acceso a los mismos es el habitual en un array: `args[0]`, `args[1]`… (a diferencia de C, `args[0]` no es el nombre del programa).

En el siguiente cuadro se puede ver una versión modificada del programa inicial. Esta versión puede recibir un argumento, el nombre de una persona, que se imprimirá en la pantalla tras el saludo.

```
public class HolaMundo {

        public static void main (String[] args) {
                if (args.length == 0) System.out.println("¡Hola!");
                else System.out.println("¡Hola "+args[0]+"!");
        }
}
```

En este caso, para ejecutar el programa emplearemos el siguiente comando (cambiando *nombre* por tu nombre):

```
~/src> java HolaMundo nombre
```

Hay que tener en cuenta que cada argumento es una cadena de caracteres. Si el argumento recibido representa, por ejemplo, un entero, en realidad se lee la cadena con los dígitos de ese entero. Si queremos realizar operaciones matemáticas con él, antes debemos convertirlo de `String` a `int`, por ejemplo, mediante la llamada `Integer.parseInt(s)` que veremos más adelante.

5 Convenciones sobre el uso de identificadores

Por identificadores, entendemos los nombres de las clases, interfaces, métodos y variables. Como dijimos al comienzo, en Java se distingue entre mayúsculas y minúsculas en el uso de esos identificadores.

Para escribir un código más fácil de leer, entender y reutilizar, en Java se emplean una serie de convenciones en la selección de los identificadores. Estas convenciones, si bien no todas son obligatorias para el compilador, sí lo son desde el punto de vista de un correcto estilo de programación, por lo que obedecerlas será algo imprescindible en esta asignatura.

Como norma general, los identificadores en Java incluyen letras y números. Nunca deberían empezar por guion bajo ('_') ni por el símbolo de dólar ('$') para evitar confusiones con los identificadores de sistema, y del código generado de forma automática. Asimismo, para facilitar la compatibilidad entre plataformas y versiones de Java nos limitaremos a utilizar caracteres ASCII (no emplees, por tanto, acentos, 'ñ', o similares).

5.1 Clases e interfaces

Los nombres de las clases serán sustantivos y su primera letra se escribirá en mayúscula. Si está formada por más de una palabra, la primera letra de la cada palabra irá también en mayúscula (un estilo llamado *CamelCase*, por recordar a las jorobas de un camello). No deberían utilizarse acrónimos ni abreviaturas a la hora de nombrar una clase.

Por ejemplo, identificadores apropiados serían:

```
interface ConMotor
class Formula1 implements ConMotor
class MountainBike extends Bicicleta
```

5.2 Métodos

Los nombres de los métodos suelen ser verbos donde la primera letra se escribe en minúscula. En caso de contener varias palabras, las restantes empezarán por mayúscula. Por ejemplo:

```
void acelerar(int incremento);
void cambiarMarcha(int marcha);
```

5.3 Variables

Los nombres de las variables deben ser cortos pero significativos, siguiendo las mismas pautas descritas para los métodos. Deben ser intuitivos y permitir deducir fácilmente su uso dentro del programa.

Se deberán evitar variables de una sola letra, excepto casos puntuales como índices de bucles. Los nombres comúnmente adoptados para este tipo de variables temporales incluyen `i`, `j`, `k`, `m` y `n` para enteros, y `c`, `d`, `e` para caracteres. Por ejemplo:

```
int velocidad = 0;
int maxAltura, maxAncho;
for (int i = 0; i < limite; i++)
```

5.4 Constantes

Las constantes en Java se identifican con el modificador `final`. Se escriben con mayúsculas, separando sus posibles palabras internas mediante el símbolo de guion bajo ('_'). Al igual que las variables, deben tener nombres cortos y autoexplicativos. Por ejemplo:

```
final int CAPACIDAD = 30;
final float MAX_VALOR = 29.3;
final String FICHERO_POR_DEFECTO = “fichero.txt”;
```

5.5 Paquetes

Los nombres de los paquetes se escriben con minúsculas. En el caso de nombres de paquetes únicos, deben comenzar con un nombre de dominio de nivel superior (e.g., com, edu, gov, mil, net, org...) y los componentes posteriores del nombre del paquete dependerá de los criterios de denominación de cada organización. Todos los paquetes oficiales de Java comienzan con la palabra `java` o utilizan la palabra `java` como prefijo (e.g. `javax`). Por ejemplo:

```
java.lang
javax.swing
com.sun.eng
pkg1
```

6 Algunas diferencias entre C y Java

Java es un lenguaje de la familia C, por lo que su repertorio de instrucciones y la sintaxis de éstas es muy similar. A continuación, apuntamos algunas diferencias entre Java y C que nos ayudarán en los primeros pasos en Java.

6.1 Comentarios

Además de la forma tradicional de C (/* comentario */), en Java se puede comentar el resto de una línea mediante la cadena “//”.

int cantidad; // variable para almacenar una cantidad

6.2 Tipos de datos nativos

Hay 8 tipos de datos nativos en Java:

char, boolean, byte (8 bits), short (16), int (32), long (64), float (32), double (64)

El JDK proporciona una serie de clases envoltorio (*wrapper*) que sirven para envolver un tipo nativo. Estas clases son: `Integer`, `Long`, `Double`… En determinados contextos, se puede emplear un envoltorio allí donde se espera un tipo nativo y viceversa, ya que el compilador realiza las transformaciones apropiadas (*unboxing* y *autoboxing*).

Estas clases tienen una serie de métodos estáticos (*static*) para procesar tipos nativos, por ejemplo:

int num = Integer.parseInt(“3”); // convertir de cadena a número, algo muy habitual

String binaryRepresentation = Integer.toBinaryString(453);

6.3 Imprimir en la consola

Como ya hemos visto en los anteriores ejemplos, se puede imprimir en pantalla con los métodos ***print/println*** del objeto **out** de la clase **System**.

System.out.print(“mensaje en pantalla”);

System.out.print(“Hola ”+”mundo”); // se concatenan las dos cadenas y se imprime “Hola mundo”.

Se puede imprimir cualquier tipo nativo del lenguaje:

System.out.print(6);

E incluso mezclar tipos uniéndolos con el operador ‘+’ (sólo para generar una cadena o imprimir):

System.out.print(“El valor de la variable entera x es ”+x);

6.4 Sentencia switch

Además de enteros y booleanos, como en C, un `switch` en Java también permite emplear cadenas en los ***case***:

```
switch (var) {
        case "uno": break;
        case "dos": break;
        default:    break;
}
```

6.5 Funciones matemáticas

Para realizar operaciones matemáticas sobre distintos tipos de datos (o acceder a las constantes habituales), en Java está disponible, sin necesidad de declarar nada, la clase **Math**, que contiene varios métodos estáticos (*static*) con las operaciones matemáticas típicas (*max(), min(), sqrt(), pow(), log(), sin(), cos()…*).

```
double raiz = Math.sqrt(100);
```

6.6 Excepciones

Para proteger a un programa frente a errores inesperados (que suelen causar una terminación abrupta del mismo), Java introduce el concepto de excepción, gestionado con el constructor `try-catch`.

Básicamente, se trata de encerrar el código que queremos proteger mediante el constructor `try`, y de declarar uno o varios `catch` para tratar las correspondientes excepciones que se pueden producir, o una general que capture cualquier excepción, que entraría en acción si no entra otra antes. Es decir, sólo se ejecuta el código de la primera excepción que se capture (por ello, deben declararse en orden, de las más específicas a las más generales).

```
try  {
        // sentencias que conforman el comportamiento normal del programa
        int dividendo=3, divisor=0;
        int resultado = dividendo / divisor;   // levantaría una excepción ArithmeticException
}
catch (ArithmeticException e) {
        // sentencias para tratar la excepción ArithmeticException
        System.out.println("No se pudo dividir: "+e.toString());
}
catch (Exception e) {
        // sentencias para tratar la excepción general, que sólo entraría si no entra otra antes
        System.out.println("No se pudo dividir: motivo desconocido");
}
```

Aunque podemos poner una sección vacía dentro de un `catch`

```
catch (Exception e) { }
```

esto no es recomendable, especialmente durante la creación y depuración de un programa, ya que las posibles anomalías podrían pasarnos desapercibidas. Como mínimo, deberíamos imprimir un mensaje que notifique lo acontecido.