Introducción a Java

1. Introducción

Java es un lenguaje de programación de propósito general, originalmente desarrollado por Sun Microsystems y actualmente mantenido por Oracle[1]. Fue diseñado para poder ejecutarse de forma transparente en múltiples plataformas gracias a la introducción de la JVM.

La mayoría de los lenguajes de programación pueden categorizarse como compilados o interpretados, pero mediante el uso de la JVM java optó por un esquema híbrido, donde el código se compila a un *bytecode* que posteriormente es interpretado por la JVM. Esto permite a los desarrolladores distribuir la misma versión del *bytecode* a distintos sistemas (siempre que dispongan de una JVM compatible), sin necesidad de recompilar el software para cada sistema.

Por otro lado, la JVM de Java incluye un recolector de basura o GC (garbage collector). Este recolector es una capa de lógica que controla que variables existen y cuales no van a volver a ser utilizadas, para liberar la memoria correspondinete de forma automática. En otros lenguajes como C, la reserva y liberación de memoria debe realizarse de forma manual, pudiendo producirse fugas de memoria en caso de una programación incorrecta[2].

En esta asignatura no se pretende explicar y aprender java en su totalidad, ya que no se dispone del tiempo que necesitaría dedicarle. Posteriormente en LP se dedicará la totalidad de la asignatura a la programación con Java. Los objetivos que se persiguen en este momento son los siguientes:

1) Habilitarnos una herramienta que nos será de utilidad para realizar las prácticas de la asignatura.

2) Mantener cierto contenido de programación para evitar olvidar lo aprendido en la asignatura de informática.

3) Servir de introducción a los contenidos que se ampliarán posteriormente en RO y LP.

Puede probar los ejemplos de este manual en los servidores virtualizados de la universidad[3] o en W3Schools Tryit Editor[4].

---

[1] https://docs.oracle.com/en/java/

[2] https://en.wikipedia.org/wiki/Memory_leak

[3] https://vdibroker.unavarra.es/

[4] https://www.w3schools.com/java/tryjava.asp?filename=demo_helloworld

1.1 Compilación y ejecución

Veamos un ejemplo de un programa básico. El siguiente programa imprime por salida estándar el texto *"Hello World"*.

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Posteriormente se detallará lo que significan exactamente las distintas líneas del programa, pero por ahora únicamente deberemos identificar la declaración de la clase Main, la declaración del método main y su contenido.

Todo programa de java deberá definir una clase principal cuyo nombre deberá coincidir con el nombre del fichero en el que se guarda el programa (no necesariamente Main). Por ejemplo, el programa anterior deberá guardarse en un fichero con nombre `Main.java`. Dentro de esta clase principal deberá definirse un método main, definido tal y como se encuentra en el ejemplo. Este será el punto de entrada del programa y al iniciar se ejecutarán secuencialmente las líneas que definamos en su interior.

Como se ha comentado anteriormente, los programas en java deben ser compilados antes de que puedan ejecutarse en la JVM. Para ello se disponde del programa `javac`.

```
$ javac Main.java
```

Si la compilación es satisfactoria, se generará un fichero `Main.class` cuyo contenido será el *bytecode* del programa generado por el compilador. Este será el fichero cuyo contenido lee la JVM para ejecutar el programa. Es importante recordar esto, ya que si modificamos el contenido del fichero `.java`, pero no recompilamos el programa, no se producirá ningún error al ejecutar el programa, pero se estará utilizando la versión anterior.

Una vez disponemos del fichero `Java.class`, podemos ejecutar el programa con el comando `java`, que invoca la JVM para leer el fichero `.class` e interpretar su contenido. En este caso indicaremos el nombre de la clase principal donde se encuentra nuestro método main, que deberá coincidir con el nombre del fichero sin la extensión.

```
$ java Main
Hello World
```

2. Sintaxis

Los programas escritos en java deben cumplir con las reglas sintácticas del lenguaje. A diferencia de python, donde se utilizan saltos de línea e indentaciones para distinguir entre distintos bloques o niveles del código, en java las sentencias deben ir terminadas por un punto y coma, y los distintos bloques deben ir delimitados por llaves `{}`. Por ejemplo:

```
# En python
for i in range(10):
  print(i)


// En java
for (int i = 0; i < 10; i++) {
  System.out.println(i);
}
```

Cabe destacar que al igual que python, Java es *case sensitive*, por lo que distingue entre el uso de mayúsculas y minúsculas.

Como se puede ver en los ejemplos anteriores, en Java se pueden incluir comentarios de una sola línea precediéndolos por dos barras `//`. También se pueden incluir comentarios de múltiples líneas incluyéndolos entre `/*` y `*/`. Por ejemplo:

```
// Esta linea es un comentario
System.out.println(5);  // Esta línea también lleva un comentario.


/* Esto
sería un comentario
en varias lineas diferentes
*/
```

3. Variables y tipos de datos

En python las variables se definen y el tipo de dato se ajusta de forma dinámica según el valor asignado. Java, por el contrario, es un lenguaje de programación estáticamente tipado[5], por lo que debe definirse explícitamente el tipo de dato al declarar una variable.

Java soporta los siguientes tipos de datos primitivos:

Data type	**Description**
byte	Stores whole numbers from -128 to 127
short	Stores whole numbers from -32,768 to 32,767
int	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits
boolean	Stores true or false values
char	Stores a single character/letter or ASCII values.

Al definir variables se define en primer lugar el tipo de dato y en segundo el nombre de la variable o identificador. No es necesario inicializarlas en el momento de definirlas, pero de hacerlo sería siguiendo el nombre de la variable de un igual y el valor deseado. La sentencia debe terminar en punto y coma.

Por ejemplo:

```
boolean esCierto = true; // Declaramos e inicializamos un booleano.
int x;                   // Declaramos un entero sin inicializar.
int y = 10;              // Declaramos e inicializamos un entero.
x = y + 5;        // Inicializamos un entero previamente declarado.
float n = 1.55f   // Inicializamos un decimal
```

[5] https://stackoverflow.com/a/1517670

También es posible definir y asignar múltiples variables en la misma línea:

```
int x, y, z;
x = y = z = 10;
int j = 1, k = 2;
```

Los identificadores tienen también algunas restricciones en cuanto a cuáles son los nombres que se pueden utilizar. Se puede encontrar la definición formal en la documentación[6], pero por norma general es cualquier serie de letras, números y barras bajas que no coincidan con una palabra reservada[7] y no empiecen por un número.

Las variables en java se definen en ámbito local[8], por lo que cualquier variable definida dentro de un bloque (función, if, while, ...) deja de ser accesible fuera de dicho bloque.

Por ejemplo:

```
if (true) {
  int x = 10;
}
System.out.println(x);
```

Si intentásemos compilar el programa anterior se produciría un error indicando que no se encuentra la variable.

3.1. Arrays

Hay ocasiones en los que resulta necesario almacenar varios valores del mismo tipo y que 1) tener una variable para cada uno de ellos no resulte práctico, o 2) el número de valores es dinámico y no se puede conocer antes de iniciar el programa. Para este tipo de casos, Java y otros tantos lenguajes de programación, permiten el uso de arrays.

En java un array es una lista de tamaño fijo de valores de un tipo concreto. El tamaño puede ser o no conocido a priori, pero una vez definido este no se puede modificar (aunque si es posible reasignar la variable a otro array de distinto tamaño). Para definir una variable de este tipo, deberemos declarar el tipo de dato del array, seguido de `[]` y el nombre de la variable.

Los arrays pueden dejarse sin inicializar (como cualquier otra variable), inicializarse con una lista conocida de valores (lista de elementos separados por

[6] https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.8
[7] https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.9
[8] https://www.geeksforgeeks.org/dsa/difference-between-local-variable-and-global-variable/

comas, contenidos entre llaves), o inicializarse a una lista vacía cuyo contenido se asignará posteriormente (hay que indicar el tamaño de la lista).

```
int[] valores_no_init;
int[] valores_init_filled = {1, 2, 3};
int[] valores_init_empty = new int[3];
```

Se puede acceder a los distintos valores de un array mediante el índice del elemento deseado. Los índices se cuentan desde 0 (primer elemento) hasta `longitud - 1` (último elemento). Esto funciona de igual manera tanto para lectura como escritura.

```
// import java.util.Arrays;
int[] valores = {10, 20, 30};
System.out.println(valores[0]);   // 10
valores[1] = 200;
System.out.println(Arrays.toString(valores));  // [10, 200, 30]
```

Para que el código anterior funcione es necesario incluir la línea del import (sin comentar) al inicio del fichero, fuera de la definición de la clase principal. Esto nos permite utilizar la funcionalidad de `Arrays.toString()` para imprimir el array de forma cómoda. Se explicará esto en detalle más adelante.

Se puede acceder a la longitud de un array accediendo a la propiedad length.

```
int[] valores = {10, 20, 30};
System.out.println(valores.length);  // 3
```

3.2. Strings

Un Strings es un tipo de dato no primitivo que permite almacenar una cadena de carácteres de longitud indefinida. En concreto es un objeto (se verán más adelante) al cual Java da un tratamiento especial. Se puede definir igual que el resto de tipo de variables vistas hasta el momento:

```
String text = "Hello World";
```

Al ser un objeto, este puede define métodos adicionales que pueden ser utilizados a partir de la variable. Se puede encontrar la lista completa en la documentación[9]. Se muestran a continuación unos ejemplos:

```
String text = "Hello World";
System.out.println(text.length());        // 11
System.out.println(text.toUpperCase());   // HELLO WORLD
System.out.println(text.toLowerCase());   // hello world
```

[9] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#method.summary

3.3. Casting

Es posible transformar (o reinterpretar) una variable de un tipo de dato a otro mediante un proceso llamado *casting*. Esto se realiza reasignando la variable a una del tipo destino deseado e indicando el tipo de dato de la transformación de forma explícita. Por ejemplo:

```
float x = 5.5f;
int y = (int) x;  // 5
x = (float) y;    // 5.0
```

En algunos casos, en concreto cuando no hay pérdida de información (por ejemplo, de entero a decimal) el *casting* puede realizarse de forma implícita sin necesidad de indicar el tipo destino entre paréntesis.

3.4. Parsing

De forma similar al *casting*, es posible realizar la conversión de valores tipo String a valores primitivos (siempre que el valor contenido en el string sea convertible). Para ello se hace uso de los siguientes métodos:

Method	**Result type**	**Example**
`Integer.parseInt()`	`int`	`Integer.parseInt("42")`
`Long.parseLong()`	`long`	`Long.parseLong("42")`
`Double.parseDouble()`	`double`	`Double.parseDouble("3.14")`
`Float.parseFloat()`	`float`	`Float.parseFloat("3.14")`
`Short.parseShort()`	`short`	`Short.parseShort("42")`
`Byte.parseByte()`	`byte`	`Byte.parseByte("42")`
`Boolean.parseBoolean()`	`boolean`	`Boolean.parseBoolean("true")`

También es posible realizar la operación contraria y convertir tipos primitivos a String. En este caso el método a utilizar sería `String.valueOf`, de igual manera para todos los tipos primitivos (gracias al method overloading de java[10]). Por ejemplo:

```
String v1 = String.valueOf(42);    // "42"
String v2 = String.valueOf(42.5);  // "42.5"
String v3 = String.valueOf(true);  // "true"
```

[10] https://www.w3schools.com/java/java_methods_overloading.asp

4. Operadores

Java nos proporciona una serie de operadores que nos permiten realizar operaciones con los distintos valores de variables y literales. Hay que tener en cuenta que estos operadores pueden comportarse de forma ligeramente distinta según los tipos de datos utilizados en la operación.

Por ejemplo, una division entre números enteros devolverá siempre un número entero como resultado, independientemente de si la división es o no exacta.

4.1 Operadores aritméticos

Encontramos los siguientes operadores aritméticos básicos:

Operator	**Description**
+	Adds to values together
-	Subtracts one value from another
*	Multiplies two values
/	Divides one value by another
%	Returns the division remainder
++	Increases the value of a variable by 1
--	Decreases the value of a variable by 1

Por ejemplo:

```
int x = 10;
int y = 3;
System.out.println(x + y); // 13
System.out.println(x - y); // 7
System.out.println(x * y); // 30
System.out.println(x / y); // 3. Division de enteros
System.out.println(x % y); // 1
```

Los operadores de incremento y decremento tienen dos formas de usarse. En el caso de los incrementos, se puede realizar un pre-incremento y un post-incremento, situando el operador antes o después del nombre de la variable. Como los nombres indican, el incremento se realiza antes o después de hacer uso del valor. Por ejemplo:

```
int z = 5;
z++;     // Al no hacer ningún uso adicional con el valor
++z;     // en estas expresiones no hay diferencia.
System.out.println(z);   // 7
System.out.println(++z); // 8. Se incrementa e imprime el valor.
```

```
System.out.println(z--); // 8. Se imprime y decrementa el valor.
System.out.println(z);   // 7
```

Por lo general los operadores solo pueden utilizarse con los tipos primitivos, pero los Strings son un caso particular que si acepta el uso del operador suma. En este caso, este operador realiza una concatenación de Strings.

```
String v1 = "Hello";
String v2 = "World";
System.out.println(v1 + " " + v2); // Hello World
```

4.2 Operadores de comparación

Encontramos los siguientes operadores de comparación:

Operator	Description
==	Returns true if both values are the same
!=	Returns true if both values are distinct
>	Returns true if the left value is greater than the right one
<	Returns true if the left value is lower than the right one
>=	Returns true if the left value is greater or equal than the right one
<=	Returns true if the left value is lower or equal than the right one

Por ejemplo:

```
int x = 10;
System.out.println(x == 10); // true
System.out.println(x != 10); // false
System.out.println(x > 5) ;  // true
System.out.println(x <= 5);  // false
```

Para el caso de objetos es posible utilizar los operadores de comparación == y !=. No obstante, es importante tener en cuenta que en java los valores primitivos se almacenan y utilizan por valor, mientras que los objetos funcionan mediante referencias. Por esto, al realizar una comparación entre dos objetos lo que se compara es si la referencia es igual (es el mismo objeto), que en caso negativo no necesariamente cumple que el valor también es distinto. En el caso concreto de los Strings, java les da un tratamiento especial que puede provocar que los definidos literalmente si almacenen la misma referencia. De cualquier forma, para la comparación de objetos se recomienta siempre utilizar el método `equals`, que compara el valor de los objetos independientemente de su referencia. Por ejemplo:

```
String v1 = "Hello";
String v2 = "World";
String v3 = v1 + " " + v2;
System.out.println(v1 == "Hello"); // true
System.out.println(v3 == "Hello World"); // false
System.out.println(v3.equals("Hello World")); // true
```

4.3 Operadores lógicos

Encontramos los siguientes operadores lógicos:

Operator	**Description**
&&	Returns true if both statements are true
||	Returns true if one of the statements is true
!	Reverses the result

Por ejemplo:

```
int x = 10;
System.out.println(x > 5 && x < 100); // true
System.out.println(x > 0 && x != 10); // false
System.out.println(!(x == 10))        // false
```

4.4 Operadores de asignación

Además del operador de asignación simple (signo igual), Java también ofrece varios operadores de asignación que son formas más concisas de expresar operaciones aritméticas y asignaciones. Por ejemplo:

Operator	Example	Equivalence
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

5. Estructuras de control

Las estructuras de control son sentencias que permiten controlar el flujo de ejecución del programa. En java disponemos de las siguientes:

5.1. If-else

Es posible definir una sentencía *if* para ejecutar ciertas líneas de código de forma condicional. Una sentencía *if* debe ir acompañada de un valor o expresión booleana contenida entre paréntesis.

```
if (condition) {
  ...
}
```

Se puede añadir una cláusula *else* para el caso contrario:

```
if (condition) {
  ...
}
else {
  ...
}
```

También es posible introducir tantas cláusulas *else if* entre medias para realizar comprobaciones adicionales. Si se necesita, puede añadirse un *else* sin condición al final.

```
if (condition1) {
  ...
}
else if (condition2) {
  ...
}
else if (condition3) {
  ...
}
```

Las condiciones pueden ser desde una variable booleana hasta cualquier expresión que resulte en un valor de este tipo. Por ejemplo:

```
int x = 5, y = 12;
if ((x * 2 < y) && !(x * 3 < y)) { ... }
```

5.2. Switch-case

En java también se dispone de la sentencia *switch.* Es apropiada cuando se quieren realizar distintas operaciones según el valor de una variable o expresión. Por ejemplo, supongamos que tenemos el siguiente bloque de *if-else.*

```
int value = 3;
if (value == 1) { ... }
else if (value == 2) { ... }
else if (value == 3) { ... }
else if (value == 4) { ... }
else { ... }
```

Es posible reescribir este fragmento de código utilizando un *switch.* Aquí se indicará un *case* para cada valor de interés, pudiendo añadir un bloque *default* que actuará como un *else*.

```
int value = 3;
switch (value) {
  case 1: ...
  case 2: ...
  case 3: ...
  case 4: ...
  default: ...
}
```

A diferencia de la mayoria de bloques, los *case* de java no llevan llaves al abrir y cerrar. Simplemente se ponen dos puntos despues del valor del case y posteriormente las líneas a ejecutar. Java entrará en el primer *case* cuyo valor coincida con el del switch, ejecutando el código que encuentre a continuación.

Es necesario incluir un *break* para evitar que ejecute el siguiente *case*. Por ejemplo, en el siguiente fragmento de código:

- Si `value=1`: No se imprimirá nada.
- Si `value=2`: Se imprimirá “`value=2`” y “`value=3`".
- Si `value=3`: Se imprimirá “`value=3`".
- Si `value=4`: Se imprimirá “`value=4`” y “`default`”.
- Para cualquier otro valor: Se imprimirá “`default`”.

Nótese que el *default* no necesita el break ya que es el último caso definido, no porque tenga un comportamiento especial en este aspecto.

```
switch (value) {
  case 1: break;
  case 2:
    System.out.println("value=2");
```

```
  case 3:
    System.out.println("value=3");
    break;
  case 4:
    System.out.println("value=4");
  default:
    System.out.println("default");
}
```

Debido a esto, también es posible reutilizar las mismas líneas de código para varios *cases* diferentes:

```
switch (value) {
  case 1:
  case 2:
  case 3:
    System.out.println("value = 1, 2 o 3");
    break;
  case 4:
    System.out.println("value=4");
    break;
  default:
    System.out.println("default");
}
```

5.3. While

Los bucles while permiten repetir un bloque de código mientras se cumpla la condición indicada. En java existen dos tipos de bucles while: `while` y `do-while`.

El bucle while evalúa la condición antes de empezar la ejecución del bloque. Si esta no se cumple, se salta todo el contenido y continua el programa desde la finalización del bucle. En caso de cumplirse, se ejecuta todo el contenido del bucle. Antes de realizar otra iteración del bucle, vuelve a evaluar la condición.

```
int i = 0;
while (i < 5) {
  System.out.print(i);
  i++;
}
// Imprimirá: 01234
```

Por el contrario, el bucle `do-while` empieza realizando la ejecución del contenido del bucle una primera vez. Una vez terminada la iteración, se evalúa la condición, volviendo a iniciar otra iteración en caso de cumplirse. La diferencia principal entre

estos dos bucles es que el do-while siempre realiza una iteración, aunque la condición no se cumpla nunca. Por ejemplo:

```
int i = 0;
do {
  System.out.print(i);
  i++;
} while (i > 5);
// Imprimirá: 0
```

En ambos bucles, existe la opción de modificar intencionadamente el ciclo del mismo mediante los *keywords* `continue` y `break`. Cuando se ejecute una instrucción `continue`, el bucle terminará en ese momento su iteración actual y saltará directamente a evaluar la condición para continuar el bucle. Al ejecutarse un `break`, el bucle se interrumple completamente y continuará lo restante del programa sin evaluar la condición. Por ejemplo:

```
int i = 0;
while (i < 100) {
  i++;
  if (i >= 10) {
    break;
  }
  if (i < 5) {
    continue;
  }
  System.out.print(i);
}
// Imprimirá: 56789
```

5.4. For

El bucle for resulta adecuado cuando se conoce de antemano el número de iteraciones que se quieren realizar. El ciclo de ejecución de un bucle for se compone de 3 elementos:

- Inicialización: Se ejecutará una sola vez antes de inciiar el bucle.
- Condición: Se evalua antes de iniciar cada iteración. La iteración se realiza si se cumple la condición
- Actualización: Se ejecuta al finalizar cada interación. Es la parte que nos permite que el bucle avance.

Por ejemplo:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

En Java también se dispone del bucle `for-each`. Este bucle requiere de un objeto iterable (por ejemplo un array) y permite realizar una interación para cada elemento del mismo. En este caso es necesario indicar una variable temporal en la que se almacenará cada elemento de la iteración y el objeto iterable del que se extraerán estos elementos. Al final este tipo de bucle es una forma más cómoda se escribir un tipo concreto de bucle for habitual. Los siguientes bucles son equivalentes:

```
int[] numbers = {10, 20, 30};

for (int i = 0; i < numbers.length; i++) {
  int num = numbers[i];
  System.out.println(num);
}
for (int num: numbers) {
  System.out.println(num);
}
```

La ventaja del segundo bucle es que es más compacto y requiere escribir menos. La desventaja es que no se dispone del índice del iterable, que muchas veces no es necesario, pero de serlo habría que recurrir a la utilización de un bucle `for` convencional.

De igual manera que en los bucles `while`, es ambos tipos de bucles for también es posible hacer uso de las expresiones `break` y `continue`.

6. Funciones

Las funciones (habitualmente denominadas métodos) son bloques de código reutilizables que pueden ser invocadas desde cualquier punto de un programa. A estas funciones es posible pasarles argumentos de tal forma que pueda controlarse en cierta medida la ejecución desde fuera de la función. A su vez, las funciones pueden devolver un valor como resultado de su ejecución.

Para definir una función en java es necesario indicar el tipo de valor que devuelve (de haber alguno), el nombre de la función y los argumentos o parámetros de entrada. Por convenio, en java las funciones se nombran siguiendo la semántica de *caselCase*[11]. Por ejemplo:

```
public static float midPoint(float a, float b) {
  return (a + b) / 2;
}
```

Posteriormente se explicará en algo más de detalle, pero por el momento, todas las funciones las definiremos con los modificadores `public static`. En este ejemplo, vemos como se define el tipo de retorno (`float`), el nombre de la función (`midPoint`) y los argumentos de entrada, `a` y `b`, ambos de tipo `float`. Como la función tiene un valor de retorno, la función deberá devolver un valor de este tipo utilizando el *keyword* `return`.

En caso de una función no requiera devolver ningún tipo de dato, se utiliza el *keyword* `void` para declarar el tipo de retorno. En este caso, no necesita definir un `return`, pero puede definirlo sin valor si se quiere definir un punto de salida anticipado. Por ejemplo:

```
public static void greet(String name) {
  if (name.equals("Bob")) {
    return;
  }
  System.out.println("Hey " + name);
}
```

En este ejemplo, esta función imprimirá "Hey <name>" para todos los nombres menos para Bob, que por algún motivo no se le quiere saludar. Esta función imprime por pantalla pero no devuelve ningún valor, de ahí que se define un `return` sin valor y se declare como `void`. Si una función llega al final de su cuerpo sin ejecutar ningún `return` se comporta como si existiese un `return` vacio al final.

En este punto es posible revisitar la función main que se indicaba al principio del documento.

[11] https://www.neoguias.com/tipos-notacion-nombres/

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

Ahora se puede entender gran parte del significado de estas líneas: Se trata de una función de nombre `main`, que no devuelve ningún valor y recibe como argumentos un array de strings, llamados `args`. El propio lenguaje de Java especifica que debe definirse una función `main` con tipo de retorno `void` y un único argumento de entrada de tipo `String[]`, que si se quiere se puede cambiar el nombre de este parámetro. Java busca una función con estas características para utilizarla como punto de entrada del programa, por lo que, si se modifica el nombre de la función, el tipo de retorno o el tipo o cantidad de argumentos, java no la encontrará y se generará un error al intentar ejecutar el programa.

Las funciones deben definirse dentro de una clase, por el momento dentro de la clase principal. Para invocarlas, simplemente hay que escribir el nombre de la función e invocarla mediante `()`. En el interior de esos paréntesis se indican los argumentos de la función si los tiene.

```
public class Main {
  public static float midPoint(float a, float b) {
    return (a + b) / 2;
  }

  public static void greet(String name) {
    if (name.equals("Bob")) {
      return;
    }
    System.out.println("Hey " + name);
  }

  public static void main(String[] args) {
    greet("ARSS");  // Hey ARSS
    float result = midPoint(10f, 15f);
    System.out.println(result);  // 12.5
  }
}
```

6.1. Parámetros y scope

Una función puede definir en su cuerpo todas las variables que necesite. Estas variables son locales y el GC las marcará para recolección una vez termine la ejecución de la función y ya no estén accesibles. Adicionalmente, además de las variables que se declaren de forma explícita en el cuerpo, los argumentos de la

propia función también son variables locales que se definen automáticamente y están disponibles en el interior de la función.

Cuando se pasan argumentos a una función es importante comprender si estos se pasan por valor o por referencia. Todos los tipos primitivos se pasan por valor, mientras que los objetos se pasan por referencia. Cuando una variable se pasa por valor, el valor se copia a la variable temporal que se define en el argumento de la función. Cualquier modificación que se realice sobre esta variable es local (ya que se modifica una variable diferente) y no tiene impacto en el exterior de la función. Por el contrario, cuando se pasa una variable por referencia, lo que se copia a la variable local es la referencia al valor original. Por lo tanto, podemos sobreescribir la referencia en nuestra variable local (reasignar la variable) sin ningún efecto secundario, pero si utilizamos dicha referencia para modificar el contenido, estas modificaciones se podrán ver desde el exterior de la función.

```
public static void timesTen(int[] numbers, int index) {
  numbers[index] *= 10;
}
public static void main(String[] args) {
  int[] numbers = {10, 20, 30};
  timesTen(numbers, 1);
  System.out.println(numbers[1]);  // 200
}
```

Sin embargo, en el siguiente ejemplo se pasa una variable por valor y esta se sobreescribe de forma local. En este caso es necesario devolver el valor si para que el resultado sea accesible desde el exterior.

```
public static float timesTen(int number) {
  number *= 10;
  return number;
}
public static void main(String[] args) {
  int[] numbers = {10, 20, 30};
  int result = timesTen(numbers[1]);
  System.out.println(numbers[1]);  // 20
  System.out.println(result);      // 200
}
```

6.2. Sobrecarga de funciones

Java permite realizar algo denominado *method overloading*. Esto consiste en definir múltiples funciones con el mismo nombre pero distintos argumentos (combinaciones de cantidad y tipos), pudiendo definir también distintos valores de retorno. Cuando se hace uso de esta función, java infiere que versión utilizar en base a los argumentos que se le indican. Por ejemplo:

```
public static int sum(int a, int b) {
  return a + b;
}
public static float sum(float a, float b) {
  return a + b;
}
public statuc float sum(int a, float b) {
  return a + b;
}

public static void main(String[] args) {
  sum(10, 5);          // Primera version
  sum(10,0f, 5.0f);    // Segunda version
  sum(10, 5.0f);       // Tercera version
  sum(10.0f, 5);       // Error al compilar.
}
```

7. Excepciones

Durante la ejecución de un programa pueden ocurrir situaciones en las cuales el programa no es capaz de realizar las operaciones solicitadas. Esto puede ocurrir por ejemplo al intentar dividir entre cero, acceder a un índice fuera de los límites de un array o parsear un texto no numérico. En Java, estos eventos se representan mediante excepciones.

Una excepción interrumpe la ejecución del programa y se propaga hacia arriba. Veamos el siguiente ejemplo:

```
public static int div(int a, int b) {
  int r = a / b;
  return r;
}
public static void main(String[] args) {
  int result = div(10, 0);
  System.out.println(result);
}
```

Cuando en el interior de la función `dev` se intenta dividir entre 0, se lanzará una excepción que interrumpirá la ejecución de la función en el punto que se produce. En este caso, no se llegará siquiera a tratar de ejecutar el `return`. Se regresa hacia arriba a la función anterior, en este caso `main`. Aquí no se llegará a ejecutar el print y la función se propagará fuera de la función `main`. Como esta es la última y no tiene a donde volver, el programa terminará de forma abrupta con un error.

Para contener este proceso de interrupciones, se puede utilizar la estructura `try-catch`. Esta permite capturar un error y gestionarlo, interrumpiendo la propagación de la excepción.

```
public static int div(int a, int b) {
  try {
    int r = a / b;
    return r;
  }
  catch (Exception e) {
    System.out.println("Error durante la division. Devolvemos 0");
    return 0;
  }
}
public static void main(String[] args) {
  int result = div(10, 0);
  System.out.println(result);
}
```

En este caso, la excepción se genera en el mismo punto e intenta propagarse hacia arriba. No se ejecutará la línea de `return r`, pero la excepción se captura en el bloque try inmediatamente superior. Cuando se captura una excepción en un `try-catch`, se ejecuta el contenido del bloque `catch`, que recibe como argumento un objeto que representa la excepción producida. Aquí se deberá definir que debe hacer en estos casos, en el ejemplo, como la función debe devolver un valor, se devuelve 0.

Otra posibilidad sería argumentar que devolver 0 no es un buen valor por defecto, ya que 0 es un posible resultado válido, pero no representa adecuadamente la indeterminación de esta división. En estos casos, es posible no capturar la excepción en este punto y delegar esta responsabilidad al punto en el que se ha invocado la función.

```
public static int div(int a, int b) {
  return a / b;
}
public static void main(String[] args) {
  try {
    int result = div(10, 0);
    System.out.println(result);
  }
  catch (Exception e) {
    System.out.println("No se puede dividir");
  }
}
```

Es posible tener un bloque de código que pueda producir múltiples excepciones y capturarlas de forma que el tratamiento de cada una sea diferente. Para ello, se pueden definir múltiples bloques `catch` donde cada uno indicará un tipo diferente de excepción. Java ejecutará uno u otro según cual sea el error generado. Si el error no está contemplado entre los distintos tipos declarados, la excepción se propagará fuera del `try-catch`. Para evitarlo, es posible añadir una cláusula catch con el tipo genérico `Exception`. Por ejemplo:

```
public static void main(String[] args) {
  try {
    int n = Integer.parseInt(args[0]);
    float inverse = 1f / n;
    System.out.println(inverse);
  }
  catch (NumberFormatException e) {
    System.out.println("No es un entero: " + args[0]);
  }
  catch (ArithmeticException e) {
    System.out.println("No se puede calcular el inverso de 0");
```

```
  }
  catch (Exception e) {
    System.out.println("Otro tipo de error");
  }
}
```

Este programa espera un primer argumento de ejecución para el cual se intentará calcular su inverso. Veamos los distintos casos:

```
$ java Main abc
No es un entero: abc
$ java Main 0
No se puede calcular el inverso de 0
$ java Main
Otro tipo de error
$ java 2
0.5
```

Adicionalmente, es posible añadir también un bloque `finally`. Este se ejecutará siempre, y estrictamente después de los bloques `try` y `catch`, independientemente de si se produce o no una excepción.

```
public static void main(String[] args) {
  try {
    int n = Integer.parseInt(args[0]);
    float inverse = 1f / n;
    System.out.println(inverse);
  }
  catch (Exception e) {
    System.out.println("Error");
  }
  finally {
    System.out.println("Fin del programa");
  }
}
```

```
$ java Main 0
Error
Fin del programa
$ java Main 2
0.5
Fin del programa
```

Por lo general el bloque `finally` puede sustituirse simplemente por añadir más líneas de código una vez terminado el `try-catch`, pero en el caso de funciones que contengan un `return` dentro alguno de estos bloques, hay que tener en cuenta que el `finally` sí que se ejecuta, a diferencia de las líneas posteriores.

```
public static int div(int a, int b) {
  try {
    return a / b;
  }
  catch (Exception e) {
    return 0;
  }
  finally {
    System.out.println("finally");
  }
}
```

Además de recibir excepciones generadas en otros métodos o en operaciones nativas del lenguaje, también es posible lanzar excepciones de forma manual. Para ello, se utiliza el *keyword* `throw` junto con la excepción deseada y sus argumentos. Por ejemplo:

```
public static int div(int a, int b) {
  if (b == 0) {
    throw new ArithmeticException("El divisor no puede ser 0");
  }
  return a / b;
}
```

Java cuenta con una larga lista de excepciones incluidas en la librería estándar del lenguaje[12], pero también es posible definir tipos de excepciones personalizados[13]. Java distingue entre *checked* y *unchecked exceptions*. La documentación de oficial indica que todas las excepciones son *checked* a excepción de `RuntimeException` y aquellas que heredan de esta. La diferencia con estos tipos es que el compilador de Java obliga a gestionar las excepciones comprobadas, o en su defecto a declarar con `throws` que estas pueden producirse. Esto genera un efecto de bola de nieve hacia arriba hasta que una se encuentre el `try-catch` correspondiente o el propio método main declare el `throws`. Por ejemplo:

```
public static int div(int a, int b) throws Exception {
  if (b == 0) {
    throw new Exception("El divisor no puede ser 0");
  }
  return a / b;
}
```

Si se intentase compilar esta función sin incluir el `throws Excepcion` en la declaración de la función el compilador generaría un error. Si intentamos hacer

[12] https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html
[13] https://www.geeksforgeeks.org/java/user-defined-custom-exception-in-java/

uso de esta función, el compilador nos obligará a incluir un `try-catch` o a declarar nuevamente el `throws`.

8. OOP

Al hablar de Java es habitual escuchar el término *Object Oriented Programming* o OOP. Esto simplemente es una forma de diseñar el software entorno a objetos para modelar las distintas entidades lógicas de un programa. Un objeto es una entidad que combina un estado (datos almacenados) y un comportamiento (operaciones que puede realizar). Para definir un objeto es necesario definir en primer lugar su clase. Esta definición actúa como plantilla, definiendo que datos se van a almacenar en el objeto y como se va a comportar.

Una vez disponemos de una clase definida, es posible crear diferentes objetos (instancias) de ese tipo. Este proceso de creación es llamado instanciación y se realiza mediante el operador `new`. Se ha omitido anteriormente para strings, pero si que se ha dejado ver para arrays (ambos son objetos).

```
int[] numbers = new int[] {1, 2, 3, 4};
String text = new String("hello");
```

8.1. Atributos

Dentro de una clase es posible definir sus atributos, que serán las distintas variables que componen el estado del objeto. Por ejemplo, en un contexto de redes puede ser interesante modelar una traza ethernet como un objeto que almacena los distintos campos de la traza.

```
class EthernetFrame {
  String dstMAC;
  String srcMAC;
  int etherType;
  byte[] payload;
  long CRC;
}
```

A partir de la definición de la clase es posible crear tantas instancias con el operador `new`, seguido de invocar la clase como si fuese una función:

```
EthernetFrame f1 = new EthernetFrame();
EthernetFrame f2 = new EthernetFrame();
```

En este punto se han creado dos objetos independientes, donde cada uno tiene sus 5 atributos. No obtante, todos ellos están inicializados a su valor por defecto (0 para los numeros y `null` para las referencias). Se pueden reasignar los valores de los distintos atributos accediendo a ellos a partir de la instancia del objeto mediante un punto. Se puede acceder a los valores de la misma forma para utilizarlos según sea necesario.

```
EthernetFrame f1 = new EthernetFrame();
f1.dstMAC = "FF:FF:FF:FF:FF:FF";
f1.srcMAC = "01:12:23:34:45:56";
System.out.println(f1.srcMAC + " -> " + f1.dstMAC);
// 01:12:23:34:45:56 -> FF:FF:FF:FF:FF:FF
```

8.2. Métodos

Los métodos son las funciones definidas dentro del cuerpo de una clase que determinan el comportamiento del objeto. Pueden consultar o modificar los atributos del mismo.

El constructor es un método especial que se invoca automáticamente en el momento en que se crea una nueva instancia de la clase. Se utiliza para inicializar el estado del objeto dando valores a sus atributos.

El constructor debe cumplir las siguientes reglas:

1. Su nombre debe ser igual al nombre de la clase.
2. No define ningún tipo de retorno, ni siquiera `void`.
3. Puede aceptar parámetros para recibir los datos iniciales y establecerlos directamente o derivarlos.

Si no se define ningún constructor, Java automáticamente genera un constructor por defecto sin argumentos (el que se ha usado en el ejemplo anterior). En el momento que se define explícitamente un constructor, el constructor por defecto deja de generarse. De igual manera que el resto de funciones, los constructores soportan sobrecarga, por lo que es posible definir múltiples constructores con distintos argumentos.

Dentro de cualquier método de una clase se puede acceder al propio objeto mediante el *keyword* `this`. Este hace referencia al objeto concreto sobre el cual se está actuando. En el caso de los constructores es útil para distinguir entre el atributo de la clase y la variable local del argumento del constructor (si se llaman igual es necesario para evitar ambigüedad).

```
class EthernetFrame {
  String dstMAC;
  String srcMAC;
  int etherType;
  byte[] payload;
  long CRC;

  EthernetFrame(String dstMAC, String srcMAC, int etherType, byte[]
payload) {
    this.dstMAC = dstMAC;
    this.srcMAC = srcMAC;
```

```
    this.etherType = etherType;
    this.payload = payload;
    //this.CRC = se podría calcular el CRC en base al payload;
  }
}

EthernetFrame f1 = new EthernetFrame(
  "FF:FF:FF:FF:FF:FF",
  "01:12:23:34:45:56",
  0x0800,
  new byte[1500]
);
System.out.println(f1.srcMAC + " -> " + f1.dstMAC);
// 01:12:23:34:45:56 -> FF:FF:FF:FF:FF:FF
```

Además del constructor es posible definir cualquier otro método que sea necesario de forma similar a las funciones que se han visto anteriormente. En este caso omitiremos los modificadores `public static`.

```
class EthernetFrame {
  ...

  boolean isBroadcast() {
    return this.dstMAC.equals("FF:FF:FF:FF:FF:FF");
  }
  boolean isMulticast() {
    int first = Integer.parseInt(this.dstMAC.substring(0, 2), 16);
    return (first & 1) == 1;
  }
}
```

Para utilizarlos simplemente hay que acceder a los métodos a partir de la instancia e invocarlos como cualquier otra función:

```
System.out.println(f1.isBroadcast());
System.out.println(f1.isMulticast());
```

8.3. Modificadores

Java permite el uso de distintos modificadores tanto en variables como métodos. Estos modificadores permiten establecer distintas propiedades de los nombres sobre los que actúan.

En primer lugar tenemos los modificadores de acceso. El *keyword* `public` que se ha venido utilizando hasta el momento es un modificador de acceso que expone la variable o el método para que sea accesible desde cualquier lugar. Hay también

otros modificadores[14] como `private`, `protected` y el por defecto (omisión de modificador), pero para esta asignatura utilizaremos exclusivamente `public` o lo dejaremos por defecto.

Hay también otros modificadores, que a falta de nombre propio se denominan *non-access modifiers*[15]. Uno que se ha visto ya, y el único que nos va a interesar por el momento, es el *keyword* `static`. Este se puede utilizar tanto en atributos como en métodos, y en ambas lo que indica es que el nombre en cuestión pertenece a la clase en vez de a la instancia.

En el caso de un método esto implica que el método ya no actúa sobre una clase concreta (se pierde acceso al *keyword* `this`). A su vez, el método pasa a ser accesible e invocable desde la referencia de la propia clase (aunque sigue siendo accesible desde una instancia).

```
class EthernetFrame {
  static boolean isBroadcast(String dstMAC) {
    return dstMAC.equals("FF:FF:FF:FF:FF:FF");
  }
}

EthernetFrame f1 = new EthernetFrame();
EthernetFrame.isBroadcast(f1.dstMAC)
EthernetFrame.isBroadcast("01:12:23:34:56"); // false
```

En el caso de los atributos, la variable pasa a ser de la clase y por tanto independiente de los objetos. Al igual que los métodos, se puede acceder a ellas desde la clase, pero siguen siendo accesibles desde las instancias. No obstante, esta variable es la misma en todo momento y es compartida por todas las instancias.

```
class EthernetFrame {
  static final int MAX_PAYLOAD_SIZE = 1500;
  static final int MIN_PAYLOAD_SIZE = 46;
}

EthernetFrame f1 = new EthernetFrame();
System.out.println(f1.MAX_PAYLOAD_SIZE);
System.out.println(EthernetFrame.MAX_PAYLOAD_SIZE);
```

En este ejemplo se ha utilizado el modificador adicional `final`. Quien tenga interés puede investigar su significado.

[14] https://www.w3schools.com/java/java_modifiers.asp
[15] https://www.w3schools.com/java/java_non_modifiers.asp

9. Librería estándar

Java incluye una amplia colección de clases ya definidas e incluidas en el propio lenguaje. Estas facilitan el desarrollo ya que permiten reutilizar funcionalidades habituales sin tener que "reinventar la rueda" cada vez. Estas clases vienen agrupadas en paquetes según su funcionalidad.

Para poder hacer uso de estos paquetes es necesario importarlos. Para ello se utiliza el *keyword* `import`, seguido del paquete que se quiere utilizar. Estas declaraciones son interpretadas por el compilador, que busca el paquete indicado y expone esos nombres de forma que sean accesibles desde el programa. Deben ir indicadas fuera de la clase principal al inicio del fichero. Se ha mencionado brevemente en la sección de arrays:

```
import java.util.Arrays;


public class Main {
  public static void main(String[] args) {
    int[] = {1, 2, 3};
    System.out.println(Arrays.toString(valores));
  }
}
```

Hay una gran cantidad de paquetes definidos en la librería estándar de Java[16], y no es factible ni se espera que nadie la conozca entera de memoria. Lo importante es conocer de su existencia para buscar funcionalidades cuando sean necesarias antes de reimplementarlas. A medida que se hace uso de distintos paquetes uno va aprendiéndose los que usa de forma habitual.

Para el desarrollo de esta asignatura se van a comentar brevemente algunos paquetes habituales y que se espera que puedan ser de utilidad en las prácticas. No obstante, entiéndase como una breve mención para dar a conocer de su existencia, ya que es posible que se requieran funcionalidades no mencionadas. En tal caso se espera que el alumno sea capaz de buscar la información necesaria en la documentación oficial u otros recursos.

9.1. Salidas del sistema

Java proporciona dos flujos de salida predefinidos en el paquete `java.lang` para enviar información desde el programa hacia el exterior. Cabe destacar que el paquete java.lang no es necesario importarlo de forma explícita, ya que se incluyen siempre automáticamente. A lo largo de este documento se ha venido utilizando uno de ellos sin dar demasiado contexto al respecto.

[16] https://docs.oracle.com/javase/8/docs/api/overview-summary.html

- `System.out` (Salida estándar o *stdout*): Se utiliza habitualmente para la comunicación con el usuario. Es donde suelen imprimirse los resultados del programa, menús, etc.
- `System.err` (Salida de error o *stderr*)**:** Se utiliza habitualmente para reportar errores, trazas de excepciones o logs informativos del programa.

Ambos permiten imprimir texto en la consola por defecto, pero son dos canales diferentes que pueden ser gestionados de forma independiente. Se ha mencionado el uso habitual que se les suele dar a cada uno de ellos, pero esto queda a libre uso del programador.

Ambos comparten los mismos métodos para imprimir datos en la salida:

- `println`: Imprime el mensaje y añade un salto de línea al final.
- `print`: Imprime el mensaje sin salto de línea. El proximo texto aparecerá inmediatamente seguido al anterior.
- `printf`: Permite imprimir texto formateado en base a modificadores y una plantilla[17].

9.2. Clases envolventes

Java proporciona en el paquete `java.lang` clases envolventes de los tipos primitivos. En estas clases hay distintas definidas distintas utilidades como las vistas anteriormente en la sección de parsing. Hay distintas clases y métodos de la librería estándar de java que requieren del uso de estas clases envolventes en lugar de los tipos primitivos. No obstante, Java puede hacer la conversión entre estos tipos de forma transparente.

```
Integer n1 = 10;
int n2 = n1;
Integer n3 = n2;
```

9.3. Colecciones dinámicas

Anteriormente se han visto los *arrays* tradicionales. Si bien son el bloque fundamental para las colecciones de datos, su tamaño es fijo y no se puede modificar una vez creados. Para situaciones donde el número de elementos varía con la ejecución del programa, Java proporciona una serie de colecciones de tamaño dinámico que internamente utilizan arrays pero exponen al usuario una interfaz de tamaño dinámico.

Es importante mencionar que estas colecciones están diseñadas para almacenar referencias, por lo que no es posible almacenar directamente tipos primitivos. Para ello es necesario hacer uso de las clases envolventes vistas anteriormente.

[17] https://www.w3schools.com/java/ref_output_printf.asp

ArrayList

Un `ArrayList`[18] es una lista ordenada de elementos de tamaño dinámico. Esencialmente, es equivalente a un array convencional donde el tamaño del array se ajusta automáticamente según la cantidad de elementos contenidos en su interior.

Para instanciar un `ArrayList` es necesario en primer lugar importar el paquete correspondiente. Al definir una variable de tipo `ArrayList`, deberemos también indicar cual es el tipo de dato de los elementos de la lista, indicando el tipo entre `<>`, e instanciar el objeto con `new`. Los ArrayLists no permiten el acceso a los distintos elementos con los corchetes, pero exponen distintos métodos para realizar operaciones.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> usuarios = new ArrayList<>();

        // Añadir elementos con add
        usuarios.add("Alicia");
        usuarios.add("Bob");
        usuarios.add("Alicia");

        // Acceder a elementos con get
        System.out.println(usuarios.get(0)); // Alicia

        // Modificar elementos con set
        usuarios.set(1, "John");

        // Obtener tamaño con size
        System.out.println(usuarios.size()); // 3

        // Eliminar elementos con remove
        usuarios.remove(0); // Elimina a "Alicia"

        // Recorrer con bucle for-each
        for (String user: usuarios) {
            System.out.println(user);
        }
    }
}
```

[18] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

HashSet

Un `HashSet`[19] es una colección no ordenada de elementos no duplicados. Es similar a un `ArrayList` en el sentido de que pueden añadirse y eliminarse elementos. No obstante, no es posible acceder a ni eliminar elementos por índice.

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<String> usuarios = new HashSet<>();

        // Añadir elementos con add
        usuarios.add("Bob");
        usuarios.add("Alice");
        usuarios.add("Alice"); // Ignorado por estar repetido

        // Obtener tamaño con size
        System.out.println(usuarios.size()); // 2

        // Comprobar existencia con contains
        System.out.println(usuarios.contains("Bob"));
        System.out.println(usuarios.contains("John"));

        // Eliminar elementos con remove
        usuarios.remove("Bob");

        // Recorrer con bucle for-each
        for (String user: usuarios) {
            System.out.println(user);
        }
    }
}
```

HashMap

Un HashMap[20] es una colección no ordenada de elementos donde cada valor va asociado a una clave. Las claves de un `HashMap` deben ser únicas, y la escritura sobre una clave reemplaza el valor anteriormente contenido bajo esa clave. No hay restricción respecto a la duplicidad de valores.

```
import java.util.HashMap;
```

[19] https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
[20] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

```
public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> edades = new HashMap<>();

        // Añadir o reemplazar valores con put
        edades.put("Bob", 22);
        edades.put("Alice", 23);
        edades.put("Alice", 30);  // Sobreescribe el valor anterior

        // Acceder a valor con get
        System.out.println(edades.get("Alice")); // 30

        // Obtener tamaño con size
        System.out.println(edades.size()); // 2

        // Comprobar existencia con containsKey
        System.out.println(edades.containsKey("Bob"));
        System.out.println(edades.containsKey("John"));

        // Eliminar elementos con remove
        edades.remove("Bob");

        // Recorrer con bucle for-each
        for (String name: edades.keySet()) {
            System.out.println(name+ ": " + edades.get(name));
        }
    }
}
```

9.4. Lectura de datos

Hay multitud de clases en la librería estándar de java para lectura de datos desde distintos orígenes. La utilización de una u otra depende principalmente del origen desde el que se lee, de las funcionalidades que se necesiten y de preferencia personal.

Para solicitar input al usuario y para lectura desde *stdin* se recomienda usar la clase `Scanner`[21]. Se muestra un ejemplo de lectura desde entrada entandar:

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
```

[21] https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

```
    while (sc.hasNextLine()) {
      String line = sc.nextLine();
      System.out.println(line);
    }
  }
}
```

Para leer líneas de texto de a partir de un fichero se recomienda utilizar las clases `FileReader`[22] y `BufferedReader`[23].

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {
  public static void main(String[] args) throws Exception {
    FileReader fr = new FileReader("file.txt");
    BufferedReader br = new BufferedReader(fr);
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  }
}
```

[22] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
[23] https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html