1.1 Introducción al lenguaje Java

1. Historia

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos, que fue diseñado específicamente para que los desarrolladores de aplicaciones escribieran el programa una vez y lo ejecutaran en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

El lenguaje de programación Java fue originalmente desarrollado por James Gosling, de Sun Microsystems (constituida en 1982 y posteriormente adquirida el 27 de enero de 2010 por la compañía Oracle). Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos.

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar el paradigma de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos.
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

2. Bytecode, JVM, JRE, JDK

En el mundo de la programación siempre se ha hablado de lenguajes compilados y de lenguajes interpretados. El resultado del proceso de compilación (en realidad de compilación y enlazado) es un archivo ejecutable. Un **archivo ejecutable** es un programa que se puede lanzar directamente en un sistema operativo. La ventaja es que los programas ejecutables no necesitan compilarse de nuevo, son programas terminados. El problema es que el formato y la forma en que los sistemas operativos manejan estos archivos son diferentes, es decir, un archivo ejecutable en Linux no sería compatible con Windows ni con macOS.

En Java el código no se traduce a código ejecutable. En Java el proceso se conoce como precompilación y sirve para producir un archivo (de extensión **class**) que contiene código que no es directamente ejecutable (no es código Java). Es un código intermedio llamado **bytecode**. Al no ser ejecutable, el archivo class no puede ejecutarse directamente con un doble clic en el sistema. El bytecode tiene que ser interpretado (es decir, traducido línea a línea) por una aplicación conocida como la máquina virtual de Java (**JVM**).

JRE es el Java Runtime Environment o, en español, el Entorno de Ejecución de Java. Contiene a la JVM y otras herramientas que permiten la ejecución de las aplicaciones Java. La gran ventaja es que el entorno de ejecución de Java se fabrica para todas las plataformas; lo que significa que un archivo class se puede ejecutar en cualquier ordenador o máquina que incorpore el JRE. Sólo hay una pega, si programamos utilizando por ejemplo la versión 10 de Java, el ordenador en el que queramos ejecutar el programa deberá incorporar el JRE al menos de la versión 10.

A la forma de producir código final de Java se la llama JIT (**Just In Time**,justo en el momento) ya que el código ejecutable se produce sólo en el instante de ejecución del programa. Es decir, no hay en ningún momento código ejecutable.

JRE no posee compiladores ni herramientas para desarrollar las aplicaciones Java, solo posee las herramientas para ejecutarlas. **JDK** es el Java Development Kit o, en español, Herramientas de Desarrollo de Java. Sirve para construir programas usando el lenguaje de programación Java. Trae herramientas útiles como el compilador (javac), el debugger, herramientas de evaluación de rendimiento de aplicaciones, etc. Una instalación de JDK ya contiene un JRE dentro de las carpetas.

Para programar en Java, el primer paso que tiene que realizar el alumno es instalarse el JDK de la última versión de Java. La descarga la puede efectuar desde la página web de Oracle https://www.oracle.com/technetwork/java/javase/downloads/index.html en la pestaña *Downloads*. Buscar la última versión y descargar el JDK del Sistema Operativo con el que el alumno va a trabajar. Pero si al instalar el JDK ya existe una instalación del JRE en el ordenador de una versión anterior, el JDK no actualizará el JRE a la última versión. En este caso, hay que desinstalar primero el JRE y ya después instalar el JDK de la última versión.

3. Especificaciones oficiales del Java SE (Standard Edition)

La página de Oracle proporciona las especificaciones oficiales del Java SE (Standard Edition), incluyendo tanto el lenguaje de programación Java como la Máquina Virtual Java (JVM). Cada versión del Java SE tiene su propia especificación detallada en formatos HTML y PDF. Estas especificaciones incluyen las características del lenguaje, mejoras en la JVM y nuevas funcionalidades de cada versión.

Puedes consultar los detalles específicos de cada versión en https://docs.oracle.com/javase/specs/.

4. Versiones de Java

https://javaalmanac.io/ es una herramienta que permite explorar las diferentes versiones de Java y comparar los cambios que se han realizado entre ellas, es decir, recopila información sobre la historia de Java principalmente desde un punto de vista técnico. En https://javaalmanac.io/features/ podemos encontrar ejemplos de código.

Las versiones con status `EOL` (End of Life), son versiones que han llegado al final de su ciclo de vida y ya no reciben soporte oficial, actualizaciones de seguridad ni correcciones de errores. Las versiones con status `LTS` (Long-Term Support), son versiones que reciben soporte oficial, actualizaciones de seguridad y correcciones de errores importantes durante varios años.

5. Entornos de desarrollo integrado (IDE): Eclipse

El código en Java se puede escribir en cualquier editor de texto, y para compilar el código en bytecodes, sólo hace falta descargar la versión del JDK deseada. Sin embargo, la escritura y compilación de programas hecha de esta forma es un poco incómoda. Por ello numerosas empresas fabrican sus propios entornos de edición, algunos incluyen el compilador y otras utilizan el propio JDK de Java.

Un **IDE (*integrated development environment*)** es un entorno de programación que consiste básicamente en un editor de código, un compilador y un depurador.

Algunas ventajas que ofrecen son:

- Facilidades para escribir código: coloreado de las palabras clave, autocorrección al escribir, abreviaturas,...
- Facilidades de depuración, para probar el programa.
- Facilidad de configuración del sistema.
- Facilidades para organizar los archivos de código.
- Facilidad para exportar e importar proyectos.

Algunos IDEs para programar en Java son *Eclipse*, *Netbeans* e *IntelliJ IDEA*.

En esta asignatura utilizaremos como IDE el Eclipse, que el alumno puede descargar de la página web https://www.eclipse.org/downloads/eclipse-packages/. Buscar la versión más reciente del ***Eclipse IDE for Java Developers*** del Sistema Operativo con el que el alumno va a trabajar.

A continuación, hay que definir en el Eclipse, la versión del compilador de Java en *Windows -> Preferences -> Java -> Compiler* de la versión del JRE que se desea utilizar. También hay que marcar dicha versión por defecto en *Windows -> Preferences -> Java -> Installed JREs*.

6. Sentencias

Una sentencia es la unidad mínima de ejecución de un programa. Un programa se compone de conjunto de sentencias que acaban resolviendo un problema. Al final de cada una de las sentencias encontraremos un punto y coma (`;`).

Veamos algunos ejemplos de sentencias en java:

- Sentencias de declaración: `int x;`
- Invocaciones o llamadas a métodos de tipo void: `System.out.println("Bienvenidos a Programación");`
- Sentencias de control de flujo: alteran el fujo de ejecución para tomar decisiones o repetir sentencias.

7. Expresiones

Una expresión es una combinación de operadores y operandos que se evalúa generándose un único resultado de un tipo determinado.

La diferencia entre las sentencias y los expresiones es que las expresiones devuelven un valor y las sentencias no devuelven nada.

8. Bloques

Un bloque es un conjunto de sentencias las cuales están delimitadas por llaves:

```
{
    sentencias
}
```

9. CamelCase

CamelCase es un estilo de escritura que se aplica a frases o palabras compuestas. Consiste en juntar palabras poniendo la inicial de cada palabra en mayúsculas. Ejemplo: PrimerPrograma. El nombre se debe a que las mayúsculas a lo largo de una palabra en CamelCase se asemejan a las jorobas de un camello.

Existen dos tipos de CamelCase:

- UpperCamelCase, cuando la primera letra de cada una de las palabras es mayúscula. Ejemplo: *EjemploDeUpperCamelCase*.
- lowerCamelCase, igual que la anterior con la excepción de que la primera letra es minúscula. Ejemplo: *ejemploDeLowerCamelCase*.

La notación CamelCase es utilizado por muchos lenguajes de programación, entre ellos Java.

10. Import

En cualquier lenguaje de programación existen librerías que contienen código ya escrito que nos facilita la creación de programas. En el caso de Java no se llaman librerías, sino **paquetes**. Los paquetes son una especie de carpetas que contienen clases y más paquetes. Cuando se instala el kit de desarrollo de Java, además de los programas necesarios para compilar y ejecutar código Java, se incluyen miles de clases dentro de cientos de paquetes ya listos que facilitan la generación

de programas. Algunos paquetes sirven para utilizar funciones matemáticas, funciones de lectura y escritura, comunicación en red, programación de gráficos,...

`java.lang` es el paquete básico por lo que todas sus clases se importan por defecto. Dentro de este paquete están gran parte de las clases más utilizadas dentro de las aplicaciones o programas creados con tecnología Java. Por ejemplo la clase `System` está dentro del paquete java.lang y posee el método `out.println` que necesitamos para escribir fácilmente por pantalla. Para utilizar la clase System, no es necesario poner el correspondiente `import java.lang`. Si quisiéramos utilizar la clase `Arrays` que proporciona utilidades para los arrays, necesitaríamos incluir una instrucción que permita incorporar el código de Arrays cuando compilemos nuestro trabajo. Para eso sirve la instrucción **import**.

La sintaxis de esta instrucción es `import paquete.subpaquete.subsubpaquete....clase`. Esta instrucción se coloca arriba del todo en el código. Para la clase Arrays sería `import java.util.Arrays;`. Esto significa importar en el código la clase Arrays que se encuentra dentro del paquete util que, a su vez, está dentro del gran paquete llamado java.

También se puede utilizar el asterisco de esta forma: `import java.util.*;`
Esto significa que se va a incluir en el código todas las clases que están dentro del paquete util de java.

En realidad no es obligatorio incluir la palabra import para utilizar una clase dentro de un paquete, pero sin ella deberemos escribir el nombre completo de la clase. Es decir, no podríamos utilizar el nombre `Arrays`, sino `java.util.Arrays`.

El import puede ir acompañado también por la palabra clave **static**, utilizado para importar los miembros estáticos de una clase o interfaz. Esto se llama **Static Import**. Al usar *Static Import*, es posible hacer referencia a miembros estáticos directamente por sus nombres, sin tener que calificarlos con el nombre de su clase. Esto simplifica y acorta la sintaxis requerida para usar un miembro estático.

Ejemplo: `import static java.lang.Math.*`
De esa forma, para utilizar por ejemplo la función *pow* de la clase *Math* , sería suficiente con llamarla `pow` en lugar de `Math.pow`.

11. Variables

Las variables son contenedores que sirven para almacenar los datos que utiliza un programa. Dicho más sencillamente, son nombres que asociamos a determinados datos. La realidad es que cada variable ocupa un espacio en la memoria RAM del ordenador para almacenar el dato al que se refiere. Es decir, cuando utilizamos el nombre de la variable realmente estamos haciendo referencia a un dato que está en memoria.

Las variables tienen un nombre (un **identificador**) que debe cumplir lo siguiente:

- Se escribe en minúscula.
- Si se juntan palabras, se debe utilizar la notación lowerCamelCase. Ejemplo: myFirstVariable.
- No deben comenzar con los caracteres guion bajo (_) o el signo de dólar ($), aunque ambos se admiten. Ejemplo: my_first_variable.
- Se admiten los números pero no como primer carácter.
- Deben ser cortos pero significativos. La elección de un nombre de variable debe ser mnemónico, es decir, diseñado para indicar al observador casual la intención de su uso. Por ejemplo, si queremos usar una variable para almacenar una edad, la llamaremos *edad*.

- Se deben evitar los nombres de variables de un solo carácter excepto para las variables temporales "usar y tirar". Los nombres comunes de las variables temporales son i, j, k, m, y n para enteros; c, d, y e para los caracteres.

11.1 Declaración de variables

Antes de poder utilizar una variable, esta se debe declarar de la siguiente manera: `tipo nombrevariable;`
Donde *tipo* es el tipo de datos que almacenará la variable (texto, números enteros,...) y *nombrevariable* es el identificador de la variable. Ejemplos:

```
int days; // days es un número entero, sin decimales
boolean exit; //exit sólo puede ser verdadera o falsa
```

Java es un lenguaje muy estricto al utilizar tipos de datos. Variables de datos distintos son incompatibles. Algunos autores hablan de lenguaje **fuertemente tipado** o incluso lenguaje muy tipificado. Se debe a una traducción muy directa del inglés strongly typed referida a los lenguajes que, como Java, son muy rígidos en el uso de tipos.
El caso contrario sería el lenguaje C en el que jamás se comprueban de manera estricta los tipos de datos.
Parte de le seguridad y robustez de las que hace gala Java se deben a esta característica.

Por convención de código, todas las declaraciones de variables se ponen al principio.

11.2 Inicialización de variables

En Java se utiliza el operador asignación **=** para inicializar una variable, es decir, para darle un valor inicial.

La inicialización se puede realizar:

- En la misma línea de código que la declaración:

```
int x=7;
```

- En cualquier otro momento, pero siempre después de haberla declarado:

```
int x;
...
x=7;
```

Se puede declarar más de una variable a la vez del mismo tipo en la misma línea si las separamos con comas:

```
int days, year, weeks;
```

Incluso se pueden también inicializar:

```
int days=365, year=2019, weeks;
```

También se puede utilizar una expresión para asignar un valor a una variable:

```
int x;
...
x=7;
...
x=x*2;//Utilización de una expresión en la asignación de la variable
```

Incluso se puede utilizar una expresión en la misma inicialización:

```
int a=13, b=18;
int c=a+b; //es válido, c vale 31
```

11.3 Ámbito de vida de las variables

Toda variable tiene un ámbito de vida. Esto es la parte del código en la que una variable se puede utilizar, que es en el bloque donde se ha declarado. De hecho las variables tienen un ciclo de vida:

1. En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia)
2. Se la asigna su primer valor (la variable nace)
3. Se la utiliza en diversas sentencias. Cuando finaliza el bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio que ocupa esa variable en memoria.
4. Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida. Ejemplo:

   ```
   {//Se utiliza { para indicar el comienzo del bloque de código
       int x=9;
   }//Se utiliza } para indicar el fin del bloque de código
   int y=x; //error, ya no existe x
   ```

11.4 Palabra clave var

Java 10 introdujo el uso de la palabra clave `var` que permite declarar variables locales sin especificar explícitamente el tipo, ya que el compilador lo infiere (deduce) automáticamente:

```
var count = 10;  // El compilador infiere que count es de tipo int
```

12. Constantes

Una constante es un valor que no puede ser modificado durante la ejecución de un programa, únicamente puede ser leído.

La forma de declarar constantes es la misma que la de las variables pero hay que anteponer la palabra **final** que es la que indica que estamos declarando una constante:

```
final double PI=3.141591;
PI=4; //Error, no podemos cambiar el valor de PI
```

Los nombres de las constantes se deben escribir en mayúsculas. Pueden contener también guiones bajos. Incluso pueden contener dígitos pero no como primer carácter.

Ejemplos:

```
final int MIN1 = 1;
final int MAX_PARTICIPANTS = 10;
```

Cuando un mismo valor se utilice en varias partes del código, entonces hay que declararlo como una constante ya que si en algún momento de la vida de la aplicación, ese valor varía, solamente hay que cambiar el valor de la constante y no estar cambiándolo en todos los sitios del código donde aparezca.

13. Tipos de datos primitivos

Se llaman **tipos primitivos** a los tipos de datos originales de un lenguaje de programación, esto es, aquellos que nos proporciona el lenguaje. Java posee los siguientes:

Tipo de variable	Bytes que ocupa	Rango de valores
boolean	1	true, false
char	2	Caracteres en Unicode
byte	1	-128 a 127
short	2	-32.768 a 32.767
int	4	-2.147.483.648 a 2.147.483.647
long	8	$-9 \cdot 10^{18}$ a $9 \cdot 10^{18}$
float	4	$-3{,}4 \cdot 10^{38}$ a $3{,}4 \cdot 10^{38}$
double	8	$-1{,}79 \cdot 10^{308}$ a $1{,}79 \cdot 10^{308}$

13.1 Enteros

Los tipos **byte**, **short**, **int** y **long** sirven para almacenar datos enteros. Los enteros son números sin decimales.

Un **literal** es un elemento de programa que representa directamente un valor:

```
int number=16; //16 es un literal
```

Los literales se pueden expresar de varias maneras:

- En decimal, es como se representan por defecto: `16`
- En binario, anteponiendo 0b: `0b10000`
- En octal, anteponiendo 0: `020`
- En hexadecimal, anteponiendo 0x: `0x10`

Por defecto, un literal entero es de tipo int. Si se le coloca detrás la letra **L**, entonces el literal será de tipo long.

```
int number=16; //16 es un literal entero
long number_long=16L; //16L es un literal de tipo long
```

No se acepta en general asignar variables de distinto tipo pero existen excepciones. Por ejemplo, sí se pueden asignar valores de variables enteras a variables enteras de un tipo superior (por ejemplo asignar un valor int a una variable long). Pero al revés no se puede:

```
int i=12;
byte b=i; //Error de compilación, posible pérdida de precisión
```

La solución es hacer un **casting**. Esta operación permite convertir valores de un tipo a otro. Se usa así:

```
int i=12;
byte b=(byte) i; //El casting evita el error
```

Hay que tener en cuenta en estos castings que si el valor asignado sobrepasa el rango del elemento, el valor convertido no tendrá ningún sentido ya que no puede almacenar todos los bits necesarios para representar ese número:

```
int i=1200;
byte b=(byte) i; //El valor de b no tiene sentido
```

Si lo que asignamos a la variable es un literal, java hace una conversión implícita siempre y cuando el literal esté dentro del rango permitido para dicho tipo. Por ejemplo, el siguiente código no da error porque 127 está dentro del rango de los byte aunque el literal sea por defecto int:

```
byte b=127;
```

Sin embargo, el siguiente código sí da error porque 128 sobrepasa el rango de los tipos *byte*:

```
byte b=128; //Error de incompatibilidad de tipos: no puede convertir de int a
byte
```

En el siguiente ejemplo, utilizamos `System.out.println` para escribir en pantalla el valor de las variables:

```
package tema1_1_IntroduccionAlLenguajeJava;

public class Integers {

    public void show() {

        int i;
        long l;
        byte b;
        short s;

        i = 16;// 16 decimal
        System.out.println(i);//16
        i = 020;// 20 octal=16 decimal
        System.out.println(i);//16
        i = 0x10;// 10 hexadecimal=16 decimal
        System.out.println(i);//16
        i = 0b10000;// 10000 binario=16 decimal
        System.out.println(i);//16

        l = 6985742369L; // Si se le quita la L da error
        System.out.println(l);//6985742369
```

```
        b = 127; // No da error porque está dentro del rango de los byte aunque
el literal sea por defecto int
        System.out.println(b);//127
        s = 32767; // No da error porque está dentro del rango de los short
aunque el literal sea por defecto int
        System.out.println(s);//32767

        i = 1200;
        System.out.println(i);//1200
        b = (byte) i;
        System.out.println(b); //-80 El valor de b no tiene sentido

        // A partir de java7, se pueden usar guiones para facilitar la lectura al
programador:
        System.out.println("Número: " + 1_000_000); //Número: 1000000


    }

    public static void main(String[] args) {

        new Integers().show();

    }

}
```

13.2 Números decimales

Los decimales se almacenan en los tipos **float** y **double**. Los decimales no son almacenados de forma exacta por eso siempre hay un posible error y se habla de precisión. Es mucho más preciso el tipo double que el tipo float.

Para asignar valores literales a una variable decimal, hay que tener en cuenta que el separador decimal es el punto y no la coma: `x=2.75;`

A un valor **literal** (como 1.5 por ejemplo), se le puede indicar con una **f** al final del número que es float (1.5f por ejemplo) o una **d** para indicar que es double. Si no se indica nada, un número literal siempre se entiende que es double, por lo que al usar tipos float hay que convertir los literales:

```
double d=3.49;//El literal 3.49 por defecto es double
float f=3.49f;//El literal 3.49 se tiene que convertir a float
float f = 2.75;//Error de incompatibilidad de tipos: no puede convertir de double
a float
```

Lógicamente no podemos asignar valores decimales a tipos de datos enteros:

```
int x=9.5; //Error de incompatibilidad de tipos: no puede convertir de double a
int
//Podemos mediante un casting pero perdemos los decimales, es decir, x valdrá 9:
int x=(int) 9.5;
```

El caso contrario sin embargo sí se puede hacer:

```
int x=9;
double y=x; //Correcto
```

La razón es que los tipos decimales son más grandes que los enteros, por lo que no hay problema de pérdida de valores.

13.3 Booleanos

Los valores booleanos o lógicos se almacenan en el tipo **boolean**. Sirven para indicar si algo es verdadero (**true**) o falso (**false**).

```
boolean b=true;
boolean c=false;
```

Por otro lado, a diferencia del lenguaje C, no se puede en Java asignar números a una variable booleana (en C, el valor false se asocia al número 0, y cualquier valor distinto de cero se asocia a true).
Tampoco tiene sentido intentar asignar valores de otros tipos de datos a variables booleanas mediante casting:

```
boolean b=(boolean) 9; //No tiene sentido
```

13.4 Caracteres

Los valores de tipo carácter sirven para almacenar símbolos de escritura. En Java se puede almacenar cualquier código Unicode en el tipo **char**.

Los **literales** carácter van entre comillas simples, como por ejemplo: 'a'.

En programación, secuencias de escape es el conjunto de caracteres que en el código es interpretado con algún fin. En Java, la barra invertida `\` se denomina **carácter de escape**, el cual indica que el carácter puesto a continuación será convertido en carácter especial o, si ya es especial, dejará de ser especial. Por ejemplo, el carácter `n` no es especial pero con la `\` delante se convierte en especial ya que `\n` se interpreta como un salto de línea. La `\` es un carácter especial pero con otra `\` delante deja de ser especial y simplemente es una barra invertida.

En la siguiente tabla tenemos algunas secuencias de escape en Java:

Carácter	Significado
\t	Tabulador
\n	Salto de línea
\\	Barra invertida
\udddd	Representa el carácter Unicode cuyo código es representado por dddd en hexadecimal

La descripción completa del estándar Unicode está disponible en la página web https://unicode.org/. En el enlace *Code Charts* encontraremos las tablas de caracteres en hexadecimal. Los caracteres básicos del español los encontraremos en *Latin → Basic Latin (ASCII)* y los caracteres especiales del español como por ejemplo, las vocales acentuadas y la ñ, en *Latin1 → Supplement*.

Para saber el código de los caracteres en decimal, podemos acceder al siguiente enlace: https://unicode-table.com/es/. Los caracteres imprimibles son del 32 al 126 y del 161 al 255.

Otras páginas también interesantes de caracteres Unicode son https://symbl.cc/es/unicode-table/ y https://www.compart.com/en/unicode/.

Para insertar en el código caracteres no disponibles en el teclado, se hace de manera diferente según el Sistema Operativo:

- Linux: *Ctrl+Shift* y luego se pulsa *u*(para indicar que es Unicode) y el código Unicode en hexadecimal.
- Windows: *Alt* y el código Unicode en hexadecimal.
- macOS: *Ctrl+Command+Espacio* para abrir el Visor de caracteres. Se busca el carácter y se hace clic para insertarlo.

```
package tema1_1_IntroduccionAlLenguajeJava;

public class Characters {

    public void show() {

        char character;

        character = 'C'; // Los literales carácter van entre comillas simples
        System.out.println(character);//C
        character = 67; // El código Unicode de la C es el 67. Esta línea hace lo
mismo que character='C'
        System.out.println(character);//C
        character = '\u0043'; // El código Unicode de la C en hexadecimal es el
0043. Esta línea hace lo mismo que character='C'
        System.out.println(character);//C
        character = '\n'; // Carácter especial Nueva línea
        System.out.print(character);//Se produce un salto de línea
        character = '\''; // Carácter especial Comillas simples
        System.out.println(character);//'
        character = '\"'; // Carácter especial Dobles comillas
        System.out.println(character);//"
        character = '"'; // Carácter especial Dobles comillas se puede utilizar
sin el carácter de escape en un literal carácter
        System.out.println(character);//"
        character = '\\'; // Carácter especial Barra inclinada
        System.out.println(character);//\
        character = 9752; // Código decimal del carácter trébol
        System.out.println(character);//☘
        character = '\u2618';// Código hexadecimal del carácter trébol
        System.out.println(character);//☘
        character = '☘';// Carácter trébol
        System.out.println(character);//☘

    }

    public static void main(String[] args) {

        new Characters().show();
```

```
    }

}
```

Si necesitamos almacenar más de un carácter, entonces debemos usar otro tipo de datos que nos permite manejar cadenas de caracteres: **String**.

En Java, las cadenas no se modelan como un dato de tipo primitivo, sino a través de la clase String. El texto es uno de los tipos de datos más importantes y por ello Java lo trata de manera especial. Para Java, las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo String.

Los **literales** cadena se escriben entre comillas dobles: `"Esto es un literal cadena"`.

Ejemplo:

```
String s="Estamos aprendiendo a programar";
```

En java existe también la **cadena vacía o nula ("")**, es decir, una cadena sin ningún carácter. Ejemplo: `String s="";` A la variable *s* se le está asignando la cadena vacía o nula.

14. Prioridad entre operadores

A veces hay expresiones con operadores que resultan confusas. Por ejemplo:

```
resultado=8+4/2;
```

Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución al de la suma. Es decir, siempre se ejecuta antes la división que la suma.
Siempre se pueden usar paréntesis para forzar el orden deseado:

```
resultado=(8+4)/2;
```

Ahora no hay duda, el resultado es seis.

¿Cómo podemos saber en qué orden se van a ejecutar los operadores en una expresión en Java? Pues se ejecutan en función de una prioridad, es decir, primero se ejecuta el que tenga más prioridad. La siguiente tabla muestra todos los operadores Java ordenados de mayor a menor prioridad. La primera línea de la tabla contiene los operadores de mayor prioridad y la última los de menor prioridad. Los operadores que aparecen en la misma línea tienen la misma prioridad. Cuando una expresión tenga dos operadores con la misma prioridad, la expresión se evalúa según su asociatividad.

Nivel	Operador	Descripcion	Asociatividad
1	[ ] . ()	acceso elementos array acceso miembros objetos paréntesis	de izquierda a derecha
2	++ - -	unario post-incremento unario post-decremento	no asociativos

Nivel	Operador	Descripcion	Asociatividad
3	++ - - + - ! ~	unario pre-incremento unario pre-decremento unario más unario menos unario lógico NOT unario NOT a nivel de bits	de derecha a izquierda
4	() new	cast creación objetos	de derecha a izquierda
5	* / %	multiplicación división módulo	de izquierda a derecha
6	+ - +	suma resta concatenación cadenas	de izquierda a derecha
7	<< >> >>>	desplazamientos a nivel de bits	de izquierda a derecha
8	< <= > >= instanceof	relacionales	no asociativos
9	= = ! =	igual distinto	de izquierda a derecha
10	&	AND a nivel de bits	de izquierda a derecha
11	^	XOR a nivel de bits	de izquierda a derecha
12	|	OR a nivel de bits	de izquierda a derecha
13	&&	AND	de izquierda a derecha
14	||	OR	de izquierda a derecha
15	?:	ternario condicional	de derecha a izquierda

Nivel	Operador	Descripcion	Asociatividad
16	= += -= *= /= %= &= ^= |= <<= >>= >>>=	asignaciones	de derecha a izquierda

Por ejemplo: `resultado = 9 / 3 * 3 ;` En este caso, la multiplicación y la división tienen la misma prioridad y su asociatividad es de izquierda a derecha por lo que se realiza primero la operación que esté más a la izquierda, que en este caso es la división. El resultado por lo tanto es nueve. Si se desea que se haga primero la multiplicación, habría que utilizar un paréntesis: `resultado = 9 / (3 * 3);` En este caso, el resultado sería 1.

Otro ejemplo: `x = y = z = 17;` Como la asociatividad de la asignación es de derecha a izquierda, primero se asigna el valor 17 a `z`, luego a `y` y por último a `x`. Esto se puede realizar porque el operador de asignación devuelve el valor asignado.

Algunos operadores son no asociativos, por ejemplo, la expresión `x <= y <= z` es inválida ya que el valor devuelto por estos operadores es de un tipo diferente (booleano) al de los operandos que necesita (numérico o carácter). En este caso, podríamos realizarla en Java de la siguiente manera: `x <= y && y <= z`.

15. Operadores

Un operador lleva a cabo operaciones sobre uno (**operador unario**), dos (**operador binario**) o tres (**operador ternario**) datos u operandos de tipo primitivo devolviendo un valor determinado también de un tipo primitivo. El tipo de valor devuelto tras la evaluación depende del operador y del tipo de los operandos. Por ejemplo, los operadores aritméticos trabajan con operandos numéricos, llevan a cabo operaciones aritméticas básicas y devuelven el valor numérico correspondiente. Los operadores se pueden clasificar en distintos grupos según se muestra en los siguientes apartados.

15.1 Operador asignación

El operador asignación **=** es un operador binario que asigna el valor del término de la derecha al operando de la izquierda. El operando de la izquierda es una variable. El término de la derecha es una expresión de un tipo de dato compatible.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
=	Operador asignación	n = 4	n vale 4

No debe confundirse el operador asignación (=) con el operador relacional de igualdad (==) que se verá más adelante. Además Java dispone de otros operadores que combinan la asignación con otras operaciones (operadores aritméticos combinados).

15.2 Operadores aritméticos

El lenguaje de programación Java tiene varios operadores aritméticos para los datos numéricos enteros y decimales.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
-	Operador unario de cambio de signo	-4	-4
+	Suma	2.5 + 7.1	9.6
-	Resta	235.6 - 103.5	132.1
*	Multiplicación	1.2 * 1.1	1.32
/	División	0.050 / 0.2 7 / 2	0.25 3
%	Módulo	20 % 7 14.5 % 2	6 0.5

El resultado exacto depende de los tipos de operandos involucrados. Es conveniente tener en cuenta las siguientes peculiaridades:

- El resultado de una expresión se convierte al tipo más general según el siguiente orden de generalidad:

  byte → short → int → long → float → double

  Teniendo esto en cuenta, tenemos que:

  - El resultado es de tipo long si, al menos, uno de los operandos es de tipo long y ninguno es decimal.
  - El resultado es de tipo int si ninguno de los operandos es de tipo long ni decimal.
  - El resultado es de tipo double si, al menos, uno de los operandos es de tipo double.
  - El resultado es de tipo float si, al menos, uno de los operandos es de tipo float y ninguno es double.
- Con los números enteros, si se divide entre cero, se genera la excepción *ArithmeticException*. Pero si se realiza la división entre cero con decimales, el resultado es infinito (`Infinity`).
- El resultado de una expresión inválida, por ejemplo, dividir infinito por infinito, no genera una excepción ni un error de ejecución: es un valor Not a Number (`NaN`).

```
package tema1_1_IntroduccionAlLenguajeJava;

public class ArithmeticOperators {

    public void show() {
```

```
        int int1 = 100, int2 = 0;
        double dec1 = 20.36, dec2 = 0;

        System.out.println(int1 / int2); //Genera ArithmeticException
        System.out.println(dec1 / dec2); //Infinity
        System.out.println(dec1 % dec2); //NaN

    }

    public static void main(String[] args) {

        new ArithmeticOperators().show();

    }

}
```

Hay que tener en cuenta que el resultado de estos operadores varía notablemente si usamos enteros o si usamos números decimales. Por ejemplo:

```
double result1, d1=14, d2=5;
int result2, i1=14, i2=5;
result1= d1 / d2; //result1=2.8
result2= i1 / i2; //result2=2
```

Es más incluso:

```
double result;
int i1=7,i2=2;
result=i1/i2; //result=3.0
result=(double)(i1/i2);//result=3.0
result=(double)i1/i2; //result=3.5
```

El operador del módulo (%) sirve para calcular el resto de una división tanto entera como decimal.

```
int remainder, i1=14, i2=5;
remainder = i1 % i2; //remainder=4
```

En los decimales, el resto se calcula asumiendo que la división produce un resultado entero.

```
double remainder, d1=7.5, d2=2;
remainder=d1 % d2;//remainder=1.5
```

15.3 Operadores aritméticos incrementales

Los operadores aritméticos incrementales son operadores unarios (un único operando). El operando puede ser numérico o de tipo char y el resultado es del mismo tipo que el operando.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
++	Incremento	4++	5

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
- -	Decremento	4 - -	3

En el caso de los caracteres, el incremento/decremento se realiza a su código Unicode. Es decir, si una variable char tiene el valor 'C', su código Unicode es 67. Si se incrementa, su código Unicode pasa a valer 68 que corresponde al valor 'D'.

Estos operadores pueden emplearse de dos formas dependiendo de su posición con respecto al operando:

- si el operador está detrás del operando, primero se utiliza la variable y luego se incrementa/decrementa su valor:
  - Post-incremento: a++
  - Post-decremento: a- -
- si el operador está delante del operando, primero se incrementa/decrementa el valor de la variable y luego se utiliza.
  - Pre-incremento: ++a
  - Pre-decremento: - -a

```
package tema1_1_IntroduccionAlLenguajeJava;

public class IncrementalArithmeticOperators {

    public void show() {

        int integer1, integer2;
        char character1, character2;

        character1 = 'C';//Unicode 67
        character1++;
        System.out.println(character1);//Al incrementarse vale 'D', Unicode 68

        /*
         * También se pueden utilizar los caracteres con los operadores
         * aritméticos, pero entonces hace falta usar casting:
         */
        character2 = (char) (character1 + 6);
        System.out.println(character2);//character2 vale 'J', Unicode 74
        integer1 = character2 + 2;
        System.out.println(integer1);//integer1 vale 76
        character2++;
        System.out.println(character2);//character2 vale 'K', Unicode 75
        integer1 = character2;
        System.out.println(integer1);//integer1 vale 75

        integer1 = 5;
        integer2 = integer1++;
        System.out.println(integer1); //integer1 vale 6
        System.out.println(integer2); //integer2 vale 5
        integer1 = 5;
        integer2 = ++integer1;
        System.out.println(integer1); //integer1 vale 6
```

```
        System.out.println(integer2); //integer2 vale 6

    }

    public static void main(String[] args) {

        new IncrementalArithmeticOperators().show();

    }

}
```

15.4 Operadores aritméticos combinados

Combinan un operador aritmético con el operador asignación. Como en el caso de los operadores aritméticos, pueden tener operandos numéricos enteros o decimales y el tipo específico del resultado numérico dependerá del tipo de éstos.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
+=	Suma combinada	a+=b	a=a+b
- =	Resta combinada	a - =b	a=a - b
*=	Multiplicación combinada	a*=b	a=a*b
/=	División combinada	a/=b	a=a/b
%=	Resto combinado	a%=b	a=a%b

También se pueden utilizar con caracteres:

```
char character='a';
character+=2; //character vale 'c'
```

15.5 Operadores relacionales

Realizan comparaciones entre datos compatibles de tipos primitivos (numéricos y carácter) obteniendo siempre un resultado booleano. También se pueden emplear con operandos booleanos pero solamente pueden utilizar los operadores de igualdad y desigualdad.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
==	Igual que	7 == 38	false
!=	Distinto que	'a' != 'k'	true
<	Menor que	'G' < 'B'	false
>	Mayor que	'b' > 'a'	true
<=	Menor o igual que	7.5 <= 7.38	false

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
>=	Mayor o igual que	38 >= 7	true

15.6 Operadores lógicos o booleanos

Las puertas lógicas son circuitos electrónicos capaces de realizar operaciones lógicas básicas:

- **Puerta NOT**: la salida es la inversa de la entrada. Se corresponde con la siguiente tabla de verdad:

A (entrada)	S (salida)
0	1
1	0

- **Puerta AND**: la señal de salida se activa solo cuando se activan todas las señales de entrada. Equivale al producto lógico S = A · B y se corresponde con la siguiente tabla de verdad:

A (entrada1)	B (entrada 2)	S (salida)
0	0	0
0	1	0
1	0	0
1	1	1

- **Puerta OR**: la salida se activa cuando cualquiera de las entradas está activada. Equivale a la suma lógica S = A + B y se corresponde con la siguiente tabla de verdad:

A (entrada1)	B (entrada2)	S (salida)
0	0	0
0	1	1
1	0	1
1	1	1

Los operadores lógicos o booleanos en Java se basan en estas puertas lógicas realizando operaciones sobre datos booleanos y obteniendo como resultado un valor booleano:

Operador	Descripción Operador - Puerta lógica correspondiente	Ejemplo de expresión	Resultado del ejemplo
!	Negación - Puerta NOT	! false ! (5 == 5)	true false
||	Suma lógica - Puerta OR	true || false (5 != 5) || (5 < 4)	true false
&&	Producto lógico - Puerta AND	false && true (5 == 5) && (4 < 5)	false true

Ejemplos de uso de operadores lógicos o booleanos:

```
boolean adult, younger;
int age = 21;
adult = age >= 18; //adult será true
younger = !adult; //younger será false
```

```
boolean drivingLicense=true;
int age=20;
boolean canDrive= (age>=18) && drivingLicense;
/*Si la edad es de al menos 18 años y tiene carnet de conducir,
  entonces puede conducir*/
```

```
boolean snow =true, rain=false, hail=false;
boolean badWeather= snow || rain || hail;
//Si nieva o llueve o graniza, hace mal tiempo
```

Cuando el operador de negación se encuentra delante de un paréntesis, hay que tener en cuenta lo siguiente:

- `! (A && B) = !A || !B`
- `! (A || B) = !A && !B`

15.7 Operador ternario condicional

El operador ternario condicional permite devolver valores en función de una expresión lógica. Su sintaxis es la siguiente:

`expresionLogica ? expresion_1 : expresion2`

Si el resultado de evaluar la expresión lógica es verdadero, devuelve el valor de la primera expresión, y en caso contrario, devuelve el valor de la segunda expresión.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo

? :	Operador condicional	a = 4;
		b = a == 4 ? a+5 : 6-a;
		b = a > 4 ? a*7 : a+8;
			b vale 9
			b vale 12

Ejemplo:

```
int vble1 = 5;
int vble2 = 4;
int mayor;
mayor = (vble1 > vble2)?vble1:vble2;//mayor contiene el mayor de los dos, es
decir, 5
```

El operador ternario condicional da error si no se utiliza, es decir, hay que asignárselo a alguien o meterlo por ejemplo en un System.out.println. No pasa lo mismo con las funciones.

15.8 Operador concatenación de cadenas

El operador concatenación `+` es un operador binario que devuelve una cadena resultado de concatenar las dos cadenas que actúan como operandos. Si solo uno de los operandos es de tipo cadena, el otro operando se convierte implícitamente en tipo cadena.

Operador	Descripción	Ejemplo de expresión	Resultado del ejemplo
+	Operador concatenación	"Hola" + "Juan" "Hola" + 5	"HolaJuan" "Hola5"

15.9 Operadores a nivel de bits

Manipulan los bits de los números.

Operador	Descripción
&	AND
|	OR
~	NOT
^	XOR
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda
>>>	Desplazamiento a la derecha con relleno de ceros
<<<	Desplazamiento a la izquierda con relleno de ceros

16. Evaluación de cortocircuito

La evaluación de cortocircuito denota la semántica de algunos operadores booleanos en algunos lenguajes de programación en los cuales si con la evaluación de la primera expresión ya se conoce el resultado, ya no se evalúan el resto de expresiones. En Java, se utiliza la evaluación de cortocircuito tanto en el producto lógico (AND) como en la suma lógica (OR).

Por ejemplo, veamos la siguiente expresión que utiliza operadores `OR`:

```
8 <= 13 || 12 < 9 || 5 > 1
```

Se evalúa la primera expresión `8 <= 13` dando *true*. Como el resultado va a ser *true* independientemente del resultado de la segunda y tercera expresión, entonces no se evalúan ni la segunda `12 < 9` ni la tercera expresión `5 > 1`, sino que solamente se evalúa la primera, dando como resultado *true*.

En el caso de utilizar operadores `OR`, es conveniente colocar la condición más propensa a ser verdadera en el término de la izquierda.

Lo mismo ocurre con el operador `AND`:

```
12 < 9 && 5 > 1 && 8 <= 13
```

Se evalúa la primera expresión `12 < 9` dando *false*. Como el resultado va a ser *false* independientemente del resultado de la segunda y tercera expresión, entonces no se evalúan ni la segunda `5 > 1` ni la tercera expresión `8<= 13`, sino que solamente se evalúa la primera, dando como resultado *false*.

En el caso de utilizar operadores `AND`, es conveniente situar la condición más propensa a ser falsa en el término de la izquierda.

Estas técnicas reducen el tiempo de ejecución del programa y ayudan al programador a evitar ciertos errores. Por ejemplo, `5/b==2`, si *b* es una variable de tipo entero y su valor es 0, se genera la excepción *ArithmeticException*. Para evitar este problema, el programador puede hacer lo siguiente: `b!=0 && 5/b==2`. Si *b* contiene el valor 0, `b!=0` dará *false*, entonces `5/b==2` no se evalúa, evitando así la generación de la excepción.