Capítulo 4

PRINCIPIOS DEL LENGUAJE JAVA

1. INTRODUCCIÓN

En los capítulos 2 y 3 se estudiaron los elementos básicos para el manejo de datos, tales como tipos, variables, constantes, operadores, etc., y las estructuras de programación desde el punto de vista metodológico. Se aprendió lo que significa orden físico y lógico, la sintaxis y la conveniencia de una u otra estructura en razón del contexto del problema a resolver.

Java tiene una gramática muy corta, en cuanto a simbología y reglas. Suministra un número pequeño de tipos básico, operadores, etc. y herramientas para controlar el orden de ejecución de las instrucciones, es decir, para gestionar el flujo de control del programa.

En este capítulo se estudiará como implementa Java todo esto, con el objetivo de codificar en este lenguaje todos los algoritmos resueltos hasta ahora en pseudocódigo.

1.1. Orígenes del lenguaje

James Gosling, conocido como el padre del lenguaje de programación Java, trabajando para Sun Microsystems, donde fue vicepresidente hasta que ésta fue comprada por Oracle, pensó que las ventajas aportadas por la eficiencia de C++ no compensaban el gran coste de la prueba y depuración de aplicaciones C++. Estuvo trabajando durante algún tiempo en un lenguaje de programación que él había llamado Oak, que, aun partiendo de la sintaxis de C++, intentaba remediar las deficiencias que Gosling iba observando.

Bill Joy, cofundador de Sun y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podía llegar a ser el terreno adecuado para disputar a Microsoft su primacía casi absoluta en el terreno del software y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre, al estar Oak ya registrado como marca, Java fue presentado en sociedad en mayo de 1995.

Java es heredero directo de C++ y es por tanto un lenguaje orientado a objetos. Fue pensado en principio como un lenguaje sencillo para escribir programas de control de microprocesadores para electrodomésticos. Sin embargo, ha sido con el desarrollo de internet y en particular con el crecimiento de las páginas WEB cuando el lenguaje ha alcanzado su máxima difusión, hasta llegar hoy en día a ser uno de los lenguajes más demandados.

1.2. Características generales de JAVA

ALGUNAS VENTAJAS

Es multiplataforma

Al compilar un programa en Java, el código resultante es un tipo de código binario conocido como Java *bytecodes*. Este código es interpretado por diferentes computadoras de igual manera, por lo que únicamente hay que implementar un intérprete para cada plataforma. De esa manera Java logra ser un lenguaje que no depende de una arquitectura de ordenador específica. Como el código compilado de Java es interpretado, un programa compilado de Java puede ser utilizado por cualquier computadora que tenga implementado el intérprete de Java.

Es seguro

Pueden forzarse restricciones sobre las operaciones permitidas (los applets no acceden directamente al hardware de la máquina).

Al cargar un programa en memoria, la máquina virtual Java verifica los ***bytecodes*** de la aplicación.

Robusto.

Java simplifica la gestión de la memoria dinámica (recolector de basura). Por ejemplo, ya no es necesario la liberación explícita, el intérprete de Java lo lleva acabo automáticamente cuando detecta que una variable dinámica ya no es usada por el programa. Por otra parte, impide que un puntero Java apunte a una dirección de memoria no válida, los punteros (referencias) Java son seguros y deterministas: o bien apuntan a un elemento correctamente alojado en memoria o bien tienen el valor nulo. Finalmente el acceso a la memoria es supervisado por el intérprete de tal manera que no es posible acceder a zonas de memoria no autorizadas sin provocar un error. Por ejemplo, no es posible escribir fuera de los límites de un vector.

Multitarea (Multithreaded)

Un lenguaje que soporta múltiples *threads*, hilos o tareas, es un lenguaje que puede ejecutar diferentes líneas de código al mismo tiempo. El soporte y la programación de hilos en Java está integrado en la propia sintaxis del lenguaje.

Dinámico

En Java no es necesario cargar completamente el programa en memoria sino que las clases compiladas pueden ser cargadas bajo demanda en tiempo de ejecución (*dynamic binding*). Esto proceso permite la carga de código bajo demanda, lo que es especialmente importante en los *applets*.

Tiene un amplio conjunto de bibliotecas estándar

Bibliotecas para trabajar con colecciones y otras estructuras de datos, ficheros, acceso a bases de datos (JDBC), interfaces gráficas de usuario (JFC/Swing), redes de ordenadores (RMI, Jini), aplicaciones distribuidas (EJB), interfaces web (servlets/JSP), hebras, compresión de datos, criptografía...

Simplifica algunos aspectos a la hora de programar

- Comprobación estricta de tipos
- Sintaxis simplificada con respecto a C++.
  - No se manejan punteros explícitamente (todo son punteros en realidad).
  - No hay que crear makefiles (como en C/C++).
  - No hay que mantener ficheros de cabecera aparte (como en C/C++).
  - No existen macros (#define en C/C++), ya que son propensas a errores.
  - No existe herencia múltiple.

ALGUNAS DESVENTAJAS

Java también tiene su "lado oscuro". Defectos y problemas que podemos resumir en los siguientes puntos:

Poca velocidad.

Comparado C e incluso con C++, la ejecución de programas en Java es habitualmente más lenta. Esto es debido en parte a la propia metodología orientada a objetos y a algunas características específicas del lenguaje como la "recogida de basura", y en parte a que cada vez que se ejecuta un programa hay que convertir los *bytecodes* en código de la máquina, proceso más costoso informáticamente hablando que ejecutar código máquina directamente.

No es adecuado para controlar hardware.

En efecto, Java intenta, como hemos dicho, ser independiente del procesador de la máquina en la que se está ejecutando. Por ello no dispone de instrucciones de acceso a los puertos de entrada/salida del ordenador y no permite controlar periféricos o acceder a hardware. Es, por tanto, poco adecuado para hacer aplicaciones que tengan que ver con, por ejemplo, robótica.

Aprendizaje difícil

El aprendizaje es, al comienzo, más difícil que en otros lenguajes. Incluso el programa más corto incluye desde el principio una gran cantidad de términos extraños. Al principio debemos aceptarlos sin más porque no puede aprenderse todo a la vez, aunque poco a poco iremos estudiándolos todos.

MITOS Y REALIDADES DE JAVA

Mito: Java es un lenguaje de programación para la web.

Realidad: Java es un lenguaje de programación de propósito general.

Uso estimado de Java:

- 5% applets (clientes web)
- 45% aplicaciones de escritorio (PCs)
- 50% aplicaciones en el servidor (servlets)

Mito: "Write once, run anywhere"

Realidad: Se puede conseguir, aunque se debe comprobar.

Motivos: Las aplicaciones Java pueden ejecutar código local (nativo), las interfaces gráficas pueden comportarse de forma ligeramente distinta en distintas plataformas.

Mito: La seguridad y la independencia de la máquina "son gratis".

Realidad: Aplicaciones un 20% más lentas que en C++.

Mito: Java acabará con X (donde X puede ser Microsoft, C++...)

Realidad: Siempre existen ventajas y desventajas.

Microsoft tiene su propia alternativa: la plataforma .NET

Determinadas aplicaciones es mejor escribirlas en otros lenguajes:

- Utilidades simples y eficientes en ANSI C,
- Sistemas complejos de altas prestaciones en C++,
- Aplicaciones para Windows con Visual Basic .NET o C#...

JAVA ES UN LOO

Java es un lenguaje **orientado a objetos**. En el aprendizaje de la metodología orientada a objetos usaremos el lenguaje Java para concretar las ideas explicadas.

Los Lenguajes Orientados a Objetos (LOO) están basados en la modelización del mundo real o incluso virtual mediante objetos. Estos objetos tendrán la **información** y las **capacidades** necesarias para resolver los problemas en que

participen. Ejemplo: Una persona tiene un nombre, una fecha de nacimiento, unos estudios, etc. El interés en la información que se desee tratar sobre la persona dependerá del entorno dónde se desarrolle nuestras aplicaciones. No es el igual la información que desea tratar un profesor respecto a sus alumnos que la que necesita el Instituto Nacional de la Seguridad Social.

1.3. Mecanismo de creación de un programa Java

En este aspecto la principal originalidad de Java estriba en que es a la vez compilado e interpretado. Con el compilador de Java, el programa fuente con extensión `.java` es traducido a un lenguaje intermedio, no es código máquina, llamado Java *bytecodes*, generándose un programa compilado almacenado en un archivo con extensión `.class`. Este archivo puede ser posteriormente interpretado y ejecutado por el intérprete de Java, lo que se conoce como la **Máquina Virtual Java** o *Java Virtual Machine* (JVM). Por eso Java es multiplataforma, ya que existe un intérprete para cada máquina diferente. Por tanto, la compilación se produce una vez y la interpretación cada vez que el programa se ejecuta.

Actualmente las máquinas virtuales modernas realizan una compilación JIT (Just In Time) en donde el *bytecode* no es interpretado, sino que se compila directamente a código máquina en tiempo de ejecución de acuerdo con la arquitectura (procesador y sistema operativo) en la que se ejecuta la máquina virtual. Esto permite conseguir velocidades de ejecución similares al C. En la práctica las máquinas virtuales suelen utilizar técnicas mixtas de interpretación/compilación JIT normalmente según la frecuencia de paso por un *bytecode* concreto.

Un programa Java puede funcionar como una aplicación independiente de consola o gráfica o como un *applet* (contracción de la expresión *little application*). Los applets de Java no se ejecutan independientemente sino que se pueden incrustar en una página de Web (empleando el lenguaje HTML). El programa resultante puede ser ejecutado por cualquier persona que tenga un navegador compatible con Java.

También se pueden construir un tercer tipo de aplicaciones, los denominados *servlets* (contracción de la expresión *server application*), que se ejecutan en servidores web conectados a intranets o a internet.

FASES EN LA CREACIÓN DE UN PROGRAMA

Fase I: Edición

- Se crea un programa con la ayuda de un editor
- Se almacena en un fichero con extensión *.java*

Fase II: Compilación

- El compilador lee el código Java (fichero *.java*)
- Si se detectan errores sintácticos, el compilador nos informa de ello.
- Si no hay errores se generan y almacenan los *bytecodes* en un fichero *.class*

Fase III: Cargado de clases

- El cargador de clases lee el ficheros *.class* e incluye las librerías necesarias.

Fase IV: Verificación de bytecodes

- El verificador de *bytecodes* comprueba que son válidos y no violan las restricciones de seguridad de la máquina virtual Java.

Fase V: Interpretación de bytecodes o compilación JIT(*just-in-time)*

- La máquina virtual Java (JVM) lee los *bytecodes* y los traduce a lenguaje máquina.
- Esta técnica, **compilación JIT** o **compilación en tiempo de ejecución**, se conoce también como **traducción dinámica**.

El proceso se esquematiza en la siguiente figura.

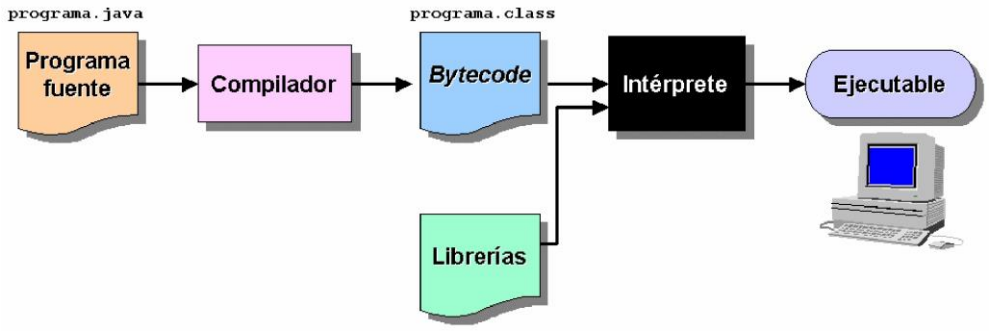


Figura 1.1. Esquema del proceso de creación de un programa con Java

2. ESTRUCTURA Y FUNCIONAMIENTO DE UN PROGRAMA EN JAVA

En este apartado, se hablará de algunos conceptos que permitirán ir familiarizándose con el modo de construir y de funcionar el lenguaje. El objetivo es poner en contacto al lector con algunos aspectos del lenguaje, por lo que no es imprescindible entenderlos en su totalidad en este momento, se irán comprendiendo mejor a medida que se avanza en la materia.

La estructura de un programa Java es similar a la de un programa de C/C++. Por su diseño, permite a los programadores de cualquier otro lenguaje leer código en Java sin mucha dificultad. Java emplea siempre la Programación Orientada a Objetos (POO), por lo que todo el código se incluye dentro de las clases. Aunque ya se explicarán detenidamente más adelante, las clases son combinaciones de datos (constantes y variables) y funcionalidades (métodos).

Un programa Java puede construirse empleando varias clases. En el caso más simple se utilizará una única clase. Esta clase contiene el programa o método principal **main()** y en éste se incluyen las sentencias del programa principal.

La estructura de un programa simple en Java es la siguiente:

```
<importaciones>
public  class  ClasePrincipal
{
     <Otras Declaraciones que pertenecen a ClasePrincipal>
     public static void main(String[] args)
     {
       <Declaraciones de objetos de datos de main>
       sentencia_1;

       sentencia_2;

       //  comentarios
       sentencia_N;
     }

}
```

Como primer ejemplo sencillo de programa escrito en Java utilizaremos uno que muestra un mensaje por la pantalla del ordenador. Por ejemplo, el típico HolaMundo.java:

```
// Mi primer programa JAVA HolaMundo.java
/**
La  clase  HolaMundo  construye  un  programa  que
muestra  en  pantalla el  mensaje  Hola mundo
*/
/*
* También muestra información sobre el nombre y la edad de una
* persona.
*/

import java.io.*; //no es necesario, se usa como ejemplo
public class HolaMundo
{
     //no tiene más entes, sólo main
     public static void main(String[] args)
     {
       int edad = 19;
       System.out.println("Hola mundo");
       System.out.print("Me llamo Sandra Yero Nevares  ");
       System.out.print("tengo  "+edad+ "  años ");
       System.out.println("y este es mi primer programa java");
     }
}
```

Vamos a ir analizando el programa línea a línea:

Todas las líneas que aparecen antes de `import` son distintas formas de comentarios que se verán más adelante.

`import java.io.*;` Las clases en java están organizadas en **paquetes**. La palabra reservada *import* se usa para importar clases de un paquete en particular. Se puede importar una clase individual, por ejemplo `import java.awt.Font;`

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un arterisco (*) para reemplazar los nombres de clase individuales, como es nuestro caso en el ejemplo `import java.io.*;`

`public class HolaMundo` En esta línea se indica el nombre de la clase que estamos definiendo. `public class` son palabras reservadas que indican que la clase se va a poder utilizar desde otras. A estas *palabras reservadas* les sigue el nombre que queremos poner a la clase, en este caso `HolaMundo`.

Cada fichero `.java` contiene una única clase pública, cuyo nombre corresponde con el del fichero sin la extensión. O lo que es igual el nombre de la clase que se define y la del fichero .java deben ser iguales.

Aunque en los temas posteriores veremos con detalle los conceptos de clases y paquetes no está de más que veamos una primera, incompleta y burda aproximación. Las **clases** representan conjuntos de objetos que quedan representados por unas características (*atributos*) y un comportamiento (dado por sus *métodos*) determinados.

La unidad de código de más alto nivel en Java es el **paquete**. Cada paquete agrupa una o más clases lógicamente relacionadas. Es habitual que las clases que forman un programa se agrupen en varios paquetes. Normalmente para cada paquete se creará un directorio o carpeta (con el nombre coincidente con el paquete) que contendrá todas las clases (ficheros .java) del paquete. A diferencia de las clases es "tradición" poner a los paquetes nombres que empiecen por la letra minúscula.

Cuando la aplicación es muy complicada interesa ampliar la jerarquía, y hacer paquetes que contienen otros paquetes y así repetidas veces hasta llegar a los paquetes más simples que sólo contienen clases. Aunque nosotros no haremos esto en nuestros ejemplos al ser éstos demasiado simples, conviene conocer esta posibilidad por las clases del API de Java que pudieran ser utilizadas y que sí están organizadas en paquetes y subpaquetes.

Si no asignamos una clase a un paquete particular, Java la asigna a un paquete genérico por defecto. Este es el caso de la clase HolaMundo de nuestro ejemplo.

Tras la declaración del nombre de clase viene el símbolo `{}` que indican el comienzo y fin de la declaración de los componentes de la clase.

`public static void main(String [] Args)` Así comienza la declaración del método **main**, único componente de la clase `HolaMundo`. El método **main** siempre debe ser declarado con la misma cabecera:

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

Podemos explicar brevemente cada una de las partes de esta declaración:

`public` Palabra reservada que significa que el método puede utilizarse desde fuera de la clase. Pero...¿tendría sentido escribir un método que no pudiera utilizarse fuera de la clase? Sí, cuando se trata de métodos auxiliares que serán utilizados desde otros métodos de la clase pero no queremos, normalmente por seguridad, que puedan ser "vistos" fuera de la clase.

`static` Palabra reservada que significa que para utilizar este método no necesitamos declarar un objeto de la clase. Veremos que en general para utilizar una clase es necesario tener un objeto de dicha clase, es decir una variable de ese tipo, dicho con otras palabras, pero el método **main** es un método especial ya que es el punto de comienzo de la aplicación.

`void` es un tipo de dato de Java. Todos los métodos deben devolver un resultado de algún tipo. En la cabecera hay que indicar dicho tipo. Sin embargo algunos métodos, como este, realmente no necesitan devolver ningún valor. En este caso se declaran como de tipo **void**.

`String[] args` Los métodos pueden recibir argumentos, es decir valores que se le pasan como en el caso de las funciones matemáticas seno y coseno, que tienen como argumento el valor del que queremos calcular el seno o el coseno. En el caso de **main** este argumento es una lista, representada en Java por [], de cadenas de caracteres (`String` en Java). Estas cadenas de caracteres se utilizan para que el usuario pueda pasar información al programa al inicio antes de ser ejecutado.

En Java, igual que en C y C++ se utilizan las llaves { } para agrupar código, en este caso el cuerpo del método. Todo método en Java está compuesto de cabecera y cuerpo. La cabecera de **main** la vimos en un párrafo anterior y en este caso su cuerpo está formado por las instrucciones:

```
int edad = 19;
System.out.println("Hola mundo");
System.out.print("Me llamo Sandra Yero Nevares  ");
System.out.print("tengo  "+edad+"  años ");
System.out.println("y este es mi primer programa java");
```

`int edad = 19;` Declaración e inicialización de variable

`System.out.println("Hola mundo");` Instrucción que, como todas en Java, terminan en ;. En este caso se llama al método `println` que escribe por pantalla una cadena de caracteres y hace un salto de línea. El método pertenece al objeto `out` definido en la clase `System` que siempre puede usarse sin necesidad de utilizar una declaración `import`.

`System.out.print("Me llamo Sandra Yero Nevares  ");` Se llama al método `println` que como `print` escribe por pantalla pero no hace salto de línea.

3. ALGUNOS COMPONENTES LÉXICOS DEL LENGUAJE

Los componentes léxicos básicos del lenguaje son comentarios, identificadores, palabras reservadas, símbolos de puntuación, caracteres de "espacio blanco" constantes literales y operadores. De los tres últimos, se hablará más adelante.

a) **Comentarios**. El comentario se utiliza normalmente para explicar cuestiones concretas dentro del código fuente. Los comentarios en JAVA se escriben con alguno de los siguientes formatos:

- Estilo de C. Es una cadena de caracteres insertada entre los símbolos /* y */. Esta cadena puede ser tan larga como se desee, ocupando varias líneas si es necesario.

  Por ejemplo, `/*Esto es un comentario */`

```
/************** Este es un comentario de
dos líneas. ************************************/
/*
* Se suele incluir un asterisco
* al principio de cada línea del
* comentario para facilitar su lectura
*/
```

- Estilo de C++. La cadena que forma el comentario se escribe tras el símbolo //

  Por ejemplo: // `Comentario de una sola línea en Java`

  Un comentario que comienza por // tiene una longitud, como máximo, hasta final de la línea en que se encuentre el símbolo.

- Comentarios de documentación. Comenzando con /** y terminando con */. Estos comentarios son utilizados por la herramienta *javadoc* para generar documentación sobre la clase.

b) **Identificadores**. Los identificadores son nombres que permiten hacer referencia a los objetos de los que se debe hacer uso en un programa: constantes, variables, funciones y otros objetos definidos por el usuario.

Un identificador está compuesto por una serie de caracteres y para su construcción en Java se siguen las reglas siguientes:

- Un identificador comienza por una letra, un carácter de subrayado (_) o un carácter de dólar ($). Aunque no se recomienda emplear el carácter $, ya que el compilador suele utilizarlos de forma interna para crear identificadores propios.
- El resto de caracteres pueden ser también dígitos, pero no pueden emplearse espacios en blanco u otros caracteres como el signo de interrogación (?) o el signo del tanto por ciento (%).
- No hay límite máximo de caracteres.

- En los identificadores del código fuente de un programa en Java se distinguen las mayúsculas de las minúsculas. Por ejemplo, nombre, NOMBRE y Nombre son tres identificadores diferentes.
- Pueden incluir caracteres Unicode.
- No puede emplearse el identificador de una variable o cualquier otro elemento del código fuente del programa para otro ya existente en el mismo bloque.

c) **Palabras reservadas**. Las palabras reservadas en Java se escriben siempre en minúsculas. Podemos verlas a continuación.

```
abstract   do         implements  protected    throw
boolean    double     import      public       throws
break      else       instanceof  rest         transient
byte       extends    int         return       true
case       false      interface   short        try
catch      final      long        static       void
char       finally    native      strictfp     volatile
class      float      new         super        while
const      for        null        switch
continue   goto       package     synchronized
default    if         private     this
```

d) **Símbolos de puntuación** o **delimitadores**. Son signos especiales que permiten al compilador separar y reconocer las diferentes unidades sintácticas del lenguaje. Los principales signos delimitadores son los siguientes:

; es considerado como **terminador** de sentencia y es necesario cuando finaliza una instrucción simple o una declaración de objetos.

Ejemplo: `int var;`

, separa dos elementos consecutivos de una lista.

Ejemplo: `int var, numero, notas;`

( ) Delimita expresiones de grupo, expresiones condicionales, y listas de parámetros.

Ejemplo: `total = (cantidad * precio) - descuento;`

Ejemplo: `for (i = 0; i < num; i++)`

{ } Delimita un bloque de instrucciones o una lista de valores iniciales.

Ejemplo:
```
{
 var = 25;
 System.out.println("var vale: "+var);
}
```

[] En la definición de arrays, delimita sus dimensiones.

Ejemplo: `int array [3]= {1,2,3};`

4. CONVENCIONES JAVA. REGLAS DE ESTILO

Razones para mantener convenciones:

- El 80% del código de un programa necesita de un posterior mantenimiento y/o adaptación a nuevas necesidades.
- Ningún software es mantenido durante toda su vida por el autor original.
- Las convenciones mejoran la lectura del software, permitiendo entender el código nuevo mucho más rápidamente y mejor.
- Si comercias tu código fuente como un producto, necesitas asegurarte de que está bien hecho y presentado como cualquier otro producto.

Reglas para nombrar interfaces y clases:

- Los nombres de interfaces deben ser sustantivos.
- Cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas.
- Se deben mantener los nombres de interfaces simples y descriptivos.
- Usar palabras completas, evitar acrónimos y abreviaturas (salvo que ésta sea mucho más conocida que el nombre completo, como DNI, URL, HTML…)
- Las clases tendrán el nombre de la interfaz principal que implementan y terminada en Impl (Implementación).
- Si hay más implementaciones se añadirá Impl1, Impl2… o algún sufijo explicativo, p.e. MatrizImplEnteros, MatrizImplPersona.

Reglas para nombrar métodos, variables, constantes:

- Cualquier nombre compuesto tendrá la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula.

- Los nombres no deben empezar por los caracteres "_" o "$", aunque ambos estén permitidos por el lenguaje.
- Los nombres deben ser cortos pero con significado, de forma que sean un mnemotécnico que indique la función que realiza en el programa.
- Deben evitarse nombres de variables de un solo carácter salvo para índices temporales, como son i, j, k, m, y n para enteros; c, d, y e para caracteres.
- Las constantes se escriben con todas las letras en mayúsculas separando las palabras con un guión bajo ("_")

5. TIPOS DE DATOS

En Java existen dos categorías de tipos de datos, tipos **primitivos** y tipos **referenciados**.

5.1 Tipos primitivos

Se llaman tipos primitivos de datos en Java a los tipos sencillos más habituales. En Java casi todo ente de dato es un objeto, pero los tipos primitivos no lo son, por tanto se tratan de forma especial. Se pueden ver en la siguiente tabla:

Tipo	Significado	Tamaño en Bytes	Rango
boolean	Lógico	1	true, false
char	Un carácter	2	Unicode [0, 65535]
byte	entero	1	[-128 ,127]
short	entero	2	$[-2^{15}, 2^{15}-1]$
int	entero	4	$[-2^{31}, 2^{31}-1]$
long	entero	8	$[-2^{63}, 2^{63}-1]$
float	Real simple precisión	4	$[-3.4^{38}, -3.4^{-38}]$ y $[3.4^{-38}, 3.4^{+38}]$
double	Real doble precisión	8	$[-1.7^{308}, -1.7^{-308}]$ y $[1.7^{-308}, 1.7^{+308}]$
void	Sin valor	0	Sin valor

El tipo de dato **void** indica ausencia de valor y se utiliza:

- Como tipo devuelto en aquellos subprogramas (funciones o métodos) que no devuelven nada, es decir, no tienen ningún valor asociado al nombre.
- En la lista de parámetros de funciones y procedimientos cuando está vacía.

El rango de depende de la cantidad de bytes que ocupe cada uno de ellos y puede verlo especificado en la tabla anterior.

5.2 Tipos referenciados

Es un tipo de dato característico de Java. Son referencias de una información más compleja, por ejemplo, arrays u objetos de una determinada clase o interfaces. Por ejemplo, en Java no existen datos primitivos que permitan almacenar una cadena de caracteres, por lo que hay que recurrir a un objeto que desempeñe esa labor, este objeto pertenece a una clase de tipo *String*. Volveremos a hablar de ellos más adelante.

NOTA: Java no posee punteros. Aparece en escena el concepto de **manejador**. Un manejador es como una referencia de C++, con la diferencia de que en Java todos los objetos deben ser controlados obligatoriamente a través de un manejador.

5.3 La clase `String`

En Java no existe un tipo de datos primitivo que sirva para la manipulación de cadenas de caracteres. En su lugar se utiliza una clase definida en la API que es la clase *String*. Esto significa que en Java las cadenas de caracteres son, a todos los efectos, objetos que se manipulan como tales, aunque existen ciertas operaciones, como la creación de *Strings*, para los que el lenguaje tiene soporte directo, con lo que se simplifican algunas operaciones.

La clase *String* forma parte del *package java.lang* y se describe completamente en la documentación del API del JDK.

Creación de Strings

Los strings u objetos de la clase *String* se pueden crear implícitamente o explícitamente. Para crear un *string* implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, Java crea un objeto de la clase *String* automáticamente cuando se escribe.

```
System.out.println("El primer programa");
```

Un *String* puede crearse explícitamente como se crea cualquier otro objeto de cualquier clase mediante el operador new.

```
String s = new String("Esto es un objeto cadena");
```

Sin embargo, también es posible crear un *String* explícitamente, sin usar el operador ***new***, haciendo una asignación simple como si se tratara de un dato primitivo.

```
String s = "Esto es un objeto cadena";
```

Ambas expresiones conducen al mismo objeto.

Los *Strings* no se modifican una vez que se les ha asignado valor, o sea son inmutables. Quiere decir que no se pueden modificar los caracteres individuales de la cadena. Si se produce una reasignación se crea un nuevo objeto *String* con el nuevo contenido. Veremos más acerca de *String* más adelante.

6. VARIABLES

Recuerda que una variable es una posición de memoria representada mediante un identificador, donde va a almacenarse un valor que puede variar durante la ejecución de un programa.

6.1 Declaración de variables

Todas las variables deben declararse antes de ser utilizadas. La declaración se usa para asociar un tipo de dato a un identificador de una dirección de memoria. El tipo de dato representa el intervalo de valores, dominio, que puede tomar la variable y el conjunto de operaciones definidas sobre la misma. Por convenio se escriben en minúsculas. Mediante la declaración indicamos que la variable guardará un valor del tipo declarado. Mediante una asignación podemos dar un nuevo valor a la variable.

La información elemental de una variable se compone de:

- El tipo, que define el conjunto de valores posibles del dominio. Por ejemplo, *int*.
- Un valor, que es el elemento del dominio especificado por el tipo, que tiene la variable en un momento dado durante la ejecución del programa. Por ejemplo, *5* si el tipo es entero.
- El identificador, que es el nombre con el que se refiere la información contenida en la zona de memoria correspondiente, cualquiera que sea su valor en un momento de ejecución del programa. Por ejemplo, *nota*.

  Ejemplo: `int nota = 5;`

Formato de declaración de una variable en C:

```
tipo lista_variables;
```

Tipo por ahora será cualquier tipo primitivo, pero puede ser cualquier tipo de datos no nativo o definido por el usuario (se verán más adelante).

Lista de variables está formada por una serie de identificadores de variables del mismo tipo separados por comas.

Ejemplos:

```
int numero, resultado;
char caracter;
float potencia;
double radio, longitud;
String nombre = "  ";;
Bombilla b = new Bombilla();
```

Para los tipos no primitivos es necesario crear un manejador. En nuestro ejemplo necesitamos un manejador de `String` y de `Bombilla`, simplemente

declarando `nombre` y `b`. Sin embargo, sólo con la declaración todavía no existe ningún objeto `string` ni `Bombilla`. Es necesario crear un objeto `String` o `Bombilla` inicializando directamente o mediante el operador ***new***, y asociar el objeto con el manejador.

En Java no se manejan los objetos directamente, sino a través de manejadores. Todos los objetos se crean obligatoriamente con ***new***. Los mensajes se le envían a los manejadores, y éstos los redirigen al objeto concreto al que apuntan.

6.2 ¿Dónde se declaran las variables en Java?

Java tiene tres tipos de variables:

- De instancia o miembro: Las variables de instancia o miembros dato como veremos más adelante, se usan para guardar los atributos de un objeto particular. Pueden ser de tipos primitivos o referencias y también variables o constantes.

- De clase: Las variables de clase o miembros dato estáticos (*static*) son similares a las variables de instancia, con la excepción de que los valores que guardan son los mismos para todos los objetos de una determinada clase.

- Locales: Las variables locales se utilizan dentro de las funciones miembro o métodos. Se definen dentro de un método o, en general, dentro de cualquier bloque de sentencias entre llaves {}. La variable desaparece una vez finalizada la ejecución del método o del bloque de sentencias. También pueden ser de tipos primitivos o referenciado.

En el siguiente ejemplo, *PI* es una variable de clase y *radio* es una variable de instancia. *PI* guarda el mismo valor para todos los objetos de la clase *Circulo*, pero el radio de cada círculo puede ser diferente.

```
class Circulo{
        static double PI=3.14;
        double radio;
//...
}
```

En el siguiente ejemplo *area* es una variable local a la función *calcularArea* en la que se guarda el valor del área de un objeto de la clase *Circulo*. Una variable local existe desde el momento de su definición hasta el final del bloque en el que se encuentra.

```
class Circulo{
//...
        double calcularArea(){
                double area=PI*radio*radio;
                return area;
        }
}
```

Veamos algunos ejemplos de declaración de algunas variables

```
int x=0;
String nombre="Daniel Rivas";
double a=3.5, b=0.0, c=-2.4;
boolean bIdentNuevo=true;
int[] datos;
float precio= 0.0;
char estadoCivil= 'S';
```

6.3 Inicialización de las variables

Es una buena costumbre inicializar las variables en el momento en el que son declaradas. El sólo hecho de declarar la variable no le proporciona un valor. En Java se puede dar valores a las variables a la vez que se declaran, o bien, después de la declaración en cualquier punto del programa.

Formato de inicialización al declararlas:

```
tipo  variable1 = cte, variable2= cte2, variableN=cteN;
```

Donde la constante debe ser del mismo tipo de dato que la variable o de un tipo compatible.

Ejemplos:

```
char respuesta;
int numero, suma;
…
respuesta = 's';
numero = 1;
suma = 0;
```

o bien,

```
char respuesta = 's';
int numero = 1, suma = 0;
```

7. CONSTANTES

Las constantes en Java son variables que no pueden modificarse. Se declaran anteponiendo la palabra reservada *final* a la declaración de la variable.

Una constante puede ser de cualquiera de los tipos primitivos o cualquier tipo de dato definido por el usuario.

Su formato de declaración es el siguiente:

```
final tipoVariable nombreVariable = valorInicial;
```

Es necesario incluir la inicialización junto con la declaración, porque la operación de asignación solo está permitida al declararlas y por convenio se hace con todas sus letras en mayúsculas.

```
final int  DIAS_SEMANA  =7;
final char LETRA = 'a';
final double IVA = 16.50;
final double PI = 3.141592;
final String  TITULO  =  "Técnico Superior DAM";
```

Como son elementos del lenguaje que permiten guardar y referenciar datos que van a permanecer invariables durante la ejecución del código dada la siguiente declaración de variable:

```
double dob;
dob = PI +1; // esto es válido
PI = dob +1; //esto NOOOO!!!!
```

7.1 Literales

Una constante literal es un valor constante formado por una secuencia de caracteres. No se declaran, se usan directamente en el código. Pueden ser:

- **Booleanas** : son las palabras reservadas *true* y *false*.
- **Enteros cortos** y **largos** (se pone detrás del número la letra L):

```
123       //literal int
123L      //literal long
```

- **Reales**. *P*unto o coma flotante. Es necesario colocar el punto:

```
123.456   //literal double
123.456F  //literal float
```

- **Carácter**. Entre comillas simples:

```
'a ', 'X', '$', '+'
```

- **Cadena**. Las cadenas en Java son instancias de la clase *String*. Las literales de cadena se representan por una secuencia de caracteres entre dobles comillas:

```
"hola",   "cadena123",   "12345"
```

8. OPERADORES

Los **operadores** en Java se aplican a operandos de tipos de datos primitivos incorporados, devolviendo un valor determinado también de un tipo primitivo y siguiendo unas reglas definidas por el propio lenguaje.

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.

En Metodología Orientada a Objetos, se verá que los operadores pueden sobrecargarse, es decir, pueden ser aplicados a operandos con tipos de datos definidos

por el propio programador, que serán tratados en líneas generales como tipos de datos incorporados al lenguaje. Sin embargo, la sobrecarga no puede modificar las reglas que tienen estos operadores para ser aplicados a tipos para los cuales están definidos por el propio lenguaje, es decir, la sobrecarga debe respetar el *modus operandi* del operador. Como ésta es una característica propia del paradigma orientado a objetos, no se hará uso de ella en el paradigma estructurado aunque el lenguaje lo permite.

8.1 Operadores aritméticos

Se aplican a datos numéricos y sirven para realizar operaciones aritméticas. Son los siguientes:

*	producto
/	división
%	resto de división entera
+	suma
–	resta (como operador unario es el cambio de signo)

El operador / tiene como resultado un entero si los dos operandos son enteros y un real si, al menos, uno de los dos operandos es real.

Es conveniente tener en cuenta las siguientes peculiaridades:

- La división entera se trunca hacia 0. La división o el resto de dividir por cero es una operación válida que genera una excepción *ArithmeticException* que puede dar lugar a un error de ejecución y la consiguiente interrupción de la ejecución del programa.

- La aritmética real (en coma flotante) puede desbordar al infinito (demasiado grande, *overflow*) o hacia cero (demasiado pequeño, *underflow*).

- El resultado de una expresión inválida, por ejemplo, dividir infinito por infinito, no genera una excepción ni un error de ejecución: es un valor *NaN* (*Not a Number*).

8.2. Operadores relacionales

Operan sobre elementos de diferentes tipos y tienen como resultado un valor de tipo lógico. Son los siguientes:

<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
!=	distinto
==	igual que (No confundir con el operador de asignación =)

8.3. Operadores lógicos

Se usan para realizar operaciones lógicas y dan como resultado un valor de tipo lógico, verdadero o falso. Los operadores lógicos son:

`&&`	operador Y
`||`	operador O
!	operador NO

Los operadores lógicos los evalúa el compilador en **cortocircuito**, los operadores de la izquierda de *&&* y || se evalúan en primer lugar. Si el valor del operando de la izquierda determina el valor de la expresión, el operando de la derecha no se evalúa. Es decir, si el operando a la izquierda de *&&* es falso o el operando a la izquierda de || es verdadero, el operando de la derecha no se evalúa, pues ya se sabe el resultado, falso en el primer caso y verdadero en el segundo. Esta característica se llama **evaluación en cortocircuito**.

8.4. Otros operadores

ASIGNACIÓN =

La operación de asignación consiste en asociar el valor de una expresión a una variable. La sintaxis es la siguiente:

```
nombre_de_variable = expresión;
```

La semántica del operador de asignación es la siguiente, en primer lugar se evalúa la expresión a la derecha de =, es decir, se obtiene su resultado, y, a continuación, se almacena el valor obtenido en la posición de memoria correspondiente a la variable cuyo identificador aparece a la izquierda del símbolo igual.

Por ejemplo, dadas dos variables *x* e *y*, tal que *x* vale 7 e *y* vale 3, después de las siguientes asignaciones:

```
y = x + 1;
x = x + y;
```

el contenido de *y* será 8 y el contenido de *x* pasará a ser 15.

Observe cómo esta última asignación es correcta por la forma en que el compilador realiza la operación, primero se evalúa la expresión *x + y* con el valor actual de la variable *x* y, posteriormente, se almacena el resultado en la dirección de memoria representada por la propia variable *x*.

Como se ve, igual que se explicó en el apartado de pseudocódigo, el efecto de la asignación es **destructivo**, cuando se asigna un nuevo valor a una variable se pierde el valor que anteriormente tenía.

Otros ejemplos de asignación:

```
contador = 0;
media = (a + b) / 2;
```

```
char  Seguir = 'S';
kilometros = 1.8854 * millasMarinas;
suma = 0;
i = i + 1;
```

Se puede abreviar esta sentencia con el operador abreviado +=, de la siguiente manera:

```
i += 1;
```

La siguiente tabla muestra los operadores abreviados de asignación y sus equivalentes largos:

Operador	Uso	Equivalente a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2

Tabla 16: Operadores abreviados de asignación

INCREMENTO Y DECREMENTO: + +, - -

La asignación `i = i + 1;` puede escribirse de forma más simple utilizando el operador de incremento, `++` de la siguiente forma: `i++;`

Igual ocurre con el operador de decremento **--** . La asignación `i--;` equivale a `i = i-1;`

El operando puede ser numérico o de tipo *char* y el resultado es del mismo tipo que el operando.

Ambos operadores se pueden usar en *pre* y en *post* incremento o decremento. La utilización correcta es crítica en situaciones donde el valor de la sentencia es utilizado en mitad de un cálculo más complejo.

```
X = 10;
Y = ++ X;
```

El resultado en este caso sería:
Primero se incrementa el valor de X y después se asigna.

Y = 11
X = 11

```
X = 10;
Y = X++;
```

En este otro: Primero se asigna el valor de X y a continuación se incrementa X.

Y = 10
X = 11

También pueden utilizarse dentro de una expresión compleja, por ejemplo,

```
X = 10 + Y++;
```

Esta asignación equivale a estas otras dos:

```
X = 10 + Y;
Y++;
```

Operador	Uso	Descripción
++	op++	Incrementa op en 1; se evalúa al valor anterior al incremento
++	++op	Incrementa op en 1; se evalúa al valor posterior al incremento
--	op--	Decrementa op en 1; se evalúa al valor anterior al incremento
--	--op	Decrementa op en 1; se evalúa al valor posterior al incremento

Tabla: Operaciones con "++" y "--"

EL OPERADOR CONCATENACIÓN +

El operador + puede utilizarse como operador binario para concatenar cadenas y devuelve la cadena resultado de concatenar las dos cadenas que actúan como operandos. si sólo uno de los operandos es de tipo cadena, el otro operando se convierte implícitamente en tipo cadena.

Ejemplo:

```
"Hola"  +  "Juan"
```

Resultado

```
"HolaJuan"
```

OTROS OPERADORES

Existen otros operadores en Java, para aritmética de bits, aritméticos combinados, condicional, que no se usarán.

OTROS CÁLCULOS ARITMÉTICOS:

La clase *Math* ofrece un conjunto de métodos adecuados para el cálculo del valor absoluto, máximo, mínimo, potencia, raíz cuadrada, funciones trigonométricas, etc. Todos para distintos tipos de datos primitivos.

La clase *Math* forma parte del *package java.lang* y se describe completamente en la documentación del API del JDK.

Algunas funcionalidades que contiene y que podrán ser usadas en nuestros programas:

```
package java.lang;

public class Math

{

 //Existen versiones para double, float, int y long

 ...

 public static double abs(double a);

 public static int max(int a, int b);

 public static long min(long a, long b);

 public static double pow(double b, double e);

 public static double random();

 public static double sqrt(double a);

...

}
```

8.5 Precedencia y asociatividad de operadores

La prioridad en la precedencia de los operadores, así como, su asociatividad en Java, determinan el orden en que se deben evaluarse dentro de una expresión.

Los paréntesis pueden utilizarse para alterar el orden de ejecución de las operaciones, ya que las subexpresiones que están dentro de los paréntesis se evalúan en primer lugar.

Por ejemplo: Dada la expresión `1 + 2 * 3`.

Como el operador ∗ tiene mayor prioridad que el operador +, se ejecuta primero la multiplicación y, después, la suma; por tanto, el valor de la expresión es 7. Si colocamos paréntesis de la forma (1 + 2) ∗ 3, ahora la expresión tiene el valor 9.

Sea ahora la expresión `6 / 3 * 2`.

Como los operadores / y ∗ tienen la misma prioridad, para determinar la forma de evaluar la expresión se usa la regla de asociatividad, que en estos operadores es de izquierda a derecha. Esto significa que las operaciones se ejecutan en ese orden. El valor de la expresión anterior es, pues,

`(6/3) * 2 = 2 * 2 = 4.`

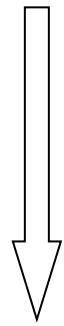
Si queremos que se realice el producto antes que la división, debemos colocar paréntesis de la forma siguiente: `6 / (3 * 2)`

Con lo cual el valor es ahora `1`, ya que los paréntesis tienen la máxima prioridad y se evalúan en primer lugar.

La tabla siguiente resume las reglas de precedencia y asociatividad de los operadores más usuales en C. Los operadores que están en la misma línea tienen la misma precedencia, y las filas están en orden de precedencia decreciente.

Operador	Asociatividad
`( )`	Izquierda a derecha, primero los internos
`! ++ --` - (unario)	derecha a izquierda
`* / %`	izquierda a derecha
`+ -` (binario)	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`=`	Derecha a izquierda

Mayor Precedencia



Menor Precedencia

El uso de paréntesis modifica el orden de aplicación de los operadores. Si no hay paréntesis el orden de aplicación de los operadores viene definido por la prioridad y la asociatividad de los mismos definida arriba.

9. EXPRESIONES

Una **expresión** puede ser una constante, una variable, la llamada a una función o una **secuencia bien construida** de constantes, variables, funciones y operadores que especifica un cálculo. Se entiende por secuencia bien construida que sea sintácticamente correcta, según la gramática de Java, en la que los tipos de los valores que aparecen son los adecuados para los operadores utilizados. En definitiva, una expresión bien formada es una secuencia de símbolos que tiene siempre un tipo y una vez evaluada devuelve un valor de ese tipo.

Las expresiones son evaluadas de acuerdo a las reglas de precedencia de los operadores utilizados, de los paréntesis, si los hay y de los tipos de datos de los operandos.

- Ejemplos de expresiones aritméticas:

```
porcentaje = (votos/electores) * 100
media = totalNotas/ numeroAlumnos
horas*3600 + minutos*60 + segundos
```

- Ejemplos de expresiones relacionales:

```
x >= (y+z)
Numero%2 == 0
```

```
encontrado != Cierto
```

- Ejemplos de expresiones lógicas:

```
(numero < MINIMO) || (numero > MAXIMO)
(numero >= MINIMO) && (numero <= MAXIMO)
!encontrado
```

9.1. Conversiones de tipo

Java no incorpora restricciones en cuanto a la utilización de objetos de distintos tipos de datos en operaciones aritméticas, así que cuando se mezclan variables de diferentes tipos en expresiones aritméticas, Java realiza conversiones automáticas de tipos antes de calcular el resultado de la expresión, a no ser que el programador haya hecho conversiones explícitamente.

CONVERSIONES AUTOMÁTICAS

Son aquellas que realiza el compilador sin que el programador intervenga. En general, las conversiones automáticas transforman un operando de menor tamaño en el de tamaño mayor, con el objetivo de aumentar la precisión y solidez del resultado.

Conversiones automáticas en operaciones

Cuando en una expresión se mezclan constantes, variables, objetos de diferentes tipos de datos, el compilador va convirtiendo, operación a operación, todos los operandos hacia el tipo del operando de mayor tamaño, según la siguiente regla llamada de **promoción integral**. Los tipos de datos primitivos tienen unas reglas sencillas de conversión.

- Si los operandos tienen distintos tipos para cada par de operandos, la lista siguiente determina a qué tipo se convertirá cada operando.

  byte → short → int → long → float → double

  Como puede observarse, la lista anterior está ordenada, de menor a mayor cantidad de bytes de memoria que ocupa cada tipo de datos, de manera que el tipo de menor tamaño se transforma en el tipo que ocupa más bytes.

- Java no acepta conversiones de tipos de menor rango a tipos de mayor rango.

Conversiones automáticas en asignaciones

Una vez evaluada la expresión a la derecha del símbolo de asignación, el compilador debe determinar qué ocurre si los tipos del resultado de la expresión a la derecha y a la izquierda de = son diferentes. En estos casos, el tipo del dato de la parte derecha del símbolo = se convierte al tipo de dato de la parte izquierda si ocupa menos bytes en memoria. En caso contrario dará error de compilación. Por ejemplo

- Si el tipo a la derecha es **char** y el izquierdo es **int**, la conversión ocurre sin problemas. Al contrario dará error en compilación.

- Si el tipo a la derecha es **int** y a la izquierda es **float**, la conversión ocurre sin problemas. Al contrario dará error de compilación.

```
double d= 1.0e24;
int i= d;      // Error durante la compilación

float f= 1.0f;
int i= f;      // Error durante la compilación
short s= i;    // Error durante la compilación
```

Se aconseja si hay dudas respecto a los resultados producidos por expresiones donde se mezclan distintos tipos de datos, que se realice una conversión forzada.

CONVERSIONES FORZADAS

Son conversiones que realiza el programador para forzar una expresión a ser de un tipo determinado llamadas *cast* (máscara o molde). Java acepta cualquier conversión usando *casts*

El formato es el siguiente, siendo *tipo* cualquier tipo de dato de C:

```
(tipo) expresión
```

Java no avisa en caso de errores de conversión numéricos.

Ejemplo:

```
int i= 256;
byte b= (byte)i; // b==0!!!, pero no da advertencia.
```

9.2. Espaciado y paréntesis

En Java pueden añadirse en la escritura del código caracteres de espacios en blanco, tabuladores y saltos de línea cuantos se desee. Se conocen como **delimitadores, separadores** o "espacios en blanco" en general y sirven para separar identificadores, palabras reservadas o cualquier otro objeto del programa, con el objetivo de que ayuden a clarificar la lectura del código.

También, se pueden usar cuantos paréntesis se estimen conveniente, su exceso o redundancia no implica errores ni aumento del tiempo de ejecución del programa.

Los caracteres de espacio blanco, son ignorados por el compilador, aunque su uso es imprescindible para hacer la escritura y lectura del código sencilla y amigable.

FALTAN LAS FUNCIONES DE ENTRADA/SALIDA

Se hará con presentaciones

Cuando en ejecución al esperar un carácter en una variable declarada como tal leemos un número entero con una funcionalidad de Scanner tal que así:
numero = leeTeclado.nextInt();
Nos dará un error en ejecución
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at opcionA.main(opcionA.java:81)
Presione una tecla para continuar . . .

Este error lo gestiona la clase del mismo nombre de java.util
Class InputMismatchException
Thrown by a `Scanner` to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.
http://docs.oracle.com/javase/1.5.0/docs/api/java/util/InputMismatchException.html

11. SENTENCIAS

Se define una **sentencia** o **proposición** o **instrucción** en Java como un enunciado que controla la secuencia de ejecución de las órdenes que componen un programa. Pueden ser de dos tipos:

11.1. Simples

Una instrucción o sentencia simple representa la tarea más sencilla que se puede realizar en un programa. Suelen llamarse también **sentencias de expresión**. Son las definidas por una expresión que acaba en punto y coma.

Su sintaxis es

```
Sentencia_expresión;
```

El *;* es el terminador de sentencia y es obligatorio.

El proceso consiste en evaluar la expresión y realizar todas los acciones que provoque antes de ejecutarse la siguiente sentencia.

Existen cuatro tipos de sentencias de expresión:

a) **De declaración**. Se ha aplicado anteriormente a la declaración de variables.

b) **De asignación**.

```
nombreObjeto = expresión;
```

Donde *nombreObjeto* debe ser una variable modificable, y *expresión* se ajusta a lo que Java entiende como expresión bien construida.

El valor de *expresión*, una vez calculado, se convierte al tipo de *Nombre-objeto* y se almacena en él.

Ejemplos:

```
int x = 0;
i = j++;
media = total/numAlumno;
```

c) De llamada a métodos (se verá con detalle en el capítulo 6)

```
NombreClase.NombreMetodo ([parámetros-actuales]);
```

Ejemplos:

```
System.out.println ("Esta es la suma " + sum);
car = System.in.read();
```

d) De incremento o decremento

```
++expresión; | expresión++;
--expresión; | expresión--;
```

Ejemplos:

```
++ i;
j --;
```

11.2. Sentencia compuesta o bloque

Una **sentencia compuesta** o **bloque** está formado por una lista de sentencias agrupadas entre llaves {}. Una sentencia compuesta puede utilizarse en cualquier parte del programa donde se permita usar una sentencia simple. Su sintaxis es:

```
{
      lista-de-sentencias
}
```

Donde *lista-de-sentencias* puede contener a su vez sentencias simples u otros bloques, o incluso puede ser una sentencia vacía o una sentencia nula.

No puede aparecer ningún punto y coma tras la llave de cierre, puesto que las llaves son delimitadores y no sentencias simples.

Ejemplos:

```
for (cont = 0; cont < = N; cont++)
{
```

```
            System.out.print("El contador es: "+cont);
            suma = suma + cont;
        }
        ............
```

Por supuesto dentro de un bloque puede haber uno o más bloques y dentro de estos otros, etc. Esto forma una *jerarquía de bloques* con sub-bloques anidados, de manera que desde un bloque externo parece que todo lo que está dentro de llaves se ejecuta "como si fuera" una sentencia y existe independiente de lo que está fuera de él.

11.3. Ámbito

El concepto de ámbito está estrechamente relacionado con el concepto de bloque y es muy importante cuando se trabaja con variables en Java. El ámbito se refiere a cómo las secciones de un programa, bloques o subprogramas, afectan al tiempo de vida de las variables, entendiendo por tiempo de vida el tiempo en que son conocidas, es decir, existen.

Toda variable u objeto tiene un ámbito, en el que es conocida y por tanto, puede ser usada, lo que viene determinado por los bloques. Una variable definida en un bloque interno no es visible por el bloque externo, pero si al contrario.

Aunque declararemos todas las variables u otros objetos de datos al comienzo del programa para que sean conocidas por todo él, el siguiente ejemplo sirve para entender el concepto de ámbito.

```
{
  // Bloque externo
  int x = 1;
  {
    // Bloque interno, variables invisibles en el exterior
    int y = 2;
    if (y > x) //no es error porque x es conocida en el bloque interno
       System.out.print("y es mayor que x");
  }
  x = y* 2; // Da error: Y está fuera de ámbito
}
```

12. ESTRUCTURA SECUENCIAL

La **estructura de programación secuencial** se implementa en Java como una sucesión de proposiciones o sentencias simples.

Ejemplo:

```
Scanner teclado = new Scanner (System.in);
double radio = 0.0;
System.out.print("Introduzca el radio: ");
radio =teclado.nextDouble();
```

EJERCICIO PRÁCTICO DE APLICACIÓN

Realiza un programa para calcular el área de un cilindro.

```
/* Nombre del programa: AreaCilindro.java */
import java.util.Scanner;
import java.io.*;
public class AreaCilindro
{
  public static void main (String[] args)throws java.io.IOException
    {
       //Declaraciones de variables, constantes y otros objetos
       final double PI = 3.1416;
       double radio, altura, area;
       Scanner teclado = new Scanner (System.in);

       //Obtener los datos
       System.out.print("Datos del cilindro:"+'\n'+"Radio:  ");
       radio =teclado.nextDouble();
       System.out.print("Altura:  ");
       altura = teclado.nextDouble();

       //Calcular área
       area = PI*radio*radio*altura;

       //pintar resultados
       System.out.println("el área del cilindro es:  "+area);
       teclado.close();
    }//fin main
}
```

Observe que el pseudocódigo generalizado se ha incrustado como comentario.

13. ESTRUCTURAS DE SELECCIÓN SÍMPLE

Las estructuras de selección simple, al igual que estudiamos en pseudocódigo, permiten elegir diferentes alternativas, realizando acciones distintas según el valor que tomen las expresiones que la acompañan. En Java se implementa con *if-else*.

if – else es una sentencia de selección que, en función de una expresión condicional, ejecuta una acción entre dos posibles opciones.

```
if (expresión-condicional)
   proposición1
[else]
   proposición2
//línea siguiente
.........
```

Donde *expresión-condicional* puede ser una expresión simple o compleja.

Las proposiciones *proposición1* y *proposición2* pueden ser simples o compuestas. **else** es opcional y, si no existe, tampoco existirá *proposición2*.

¿Cómo se evalúa el proceso?: El compilador evalúa la **expresión-condicional**, si es cierta, se realizará **proposición1** y acaba el *if* en el punto y coma si esta proposición fuera simple o en la llave de cierre del bloque si fuera compuesta. A continuación, pasa el control del flujo a la *línea siguiente*. Si la expresión-condicional es falsa y existe *else*, se efectuará **proposición2** y, a continuación, irá a la *línea siguiente*; si la **expresión-condicional** es falsa y no existe *else*, el proceso de ejecución pasará directamente a la *línea siguiente*.

Ejemplo: Escribir el fragmento de programa que implemente si un número es positivo o negativo y muestre los mensajes correspondientes.

```
if (numero < 0)
    System.out.println("Número negativo);//Sentencia simple
else
{
    System.out.print("Número negativo);//Bloque de sentencias
    System.out.print();
}
```

Ejemplo: Escribir el fragmento de programa que implemente si la variable *numero1* es mayor que *numero2*, e imprima un mensaje en caso afirmativo.

Como en caso negativo no es necesario realizar ninguna instrucción, se utilizará una sentencia *if* sin *else*.

```
if (numero1 > numero2)
    System.out.println("numero1 es mayor que numero2);
```

13.1. If anidados

Al igual que en pseudocódigo, en Java se pueden anidar los *if* :

```
if (expresión1)
      if (expresión2)
            if (expresión3)
                  proposición1
            else
                  proposición2
      else
            proposición3
else
      proposición4
```

En caso de que se omita algún *else*, podría darse algún tipo de ambigüedad, ya que el compilador asocia cada *else* al *if* más cercano que no tenga *else*.

Ejemplo: Vamos a observar el siguiente fragmento de código:

```
if (n > 0)
```

```
      if (a > b)
            z = a;
   else
      z = b;
```

¿A que *if* pertenece el *else* del código anterior?. Aunque se haya escrito de manera que parece pertenecer al primer *if*, lo cierto es que pertenece al segundo. Si realmente el ejercicio requiere que este *else* pertenezca al primer *if*, deberemos escribir el código del modo siguiente:

```
if (n > 0)
{
   if (a > b)
     z = a;
}
else
    z = b;
```

Puesto que las llaves son delimitadores de bloque, estamos delimitando con ellas la sentencia compuesta que acompaña al primer *if*, por tanto, el *else* será suyo.

13.2. If-else-if

Ésta es la forma más general de escribir una sentencia de decisión múltiple donde las condiciones se ejecutan escalonadamente. Las expresiones se evalúan en orden y, si alguna de ellas es cierta, se ejecuta la proposición asociada. Si por el contrario, ninguna de ellas es cierta, se ejecutará el bloque o proposición correspondiente al *else* del último *if*, si es que existe, puesto que es opcional.

Formato:

```
if (expresión1)
  proposición1
else
  if (expresión2)
    proposición2
  else
    if (expresión3)
      proposición3
    [else]
      proposición4
```

Ejemplo: Escribir el fragmento de programa que compruebe si *numero1* es mayor, menor o igual que *numero2*, e imprima los mensajes correspondientes.

```
if (numero1 > numero2)
    System.out.println("numero1 es mayor que numero2);
else
    if (numero1 < numero2)
      System.out.println("numero1 es menor que numero2);

    else
```

```
System.out.println("numero1 es igual que numero2.");
```

EJERCICIO PRÁCTICO DE APLICACIÓN

Codificación en Java del ejercicio del apartado 3.2 del capítulo 3.

```
/*Nombre del programa: ProductoSuma.cpp */
import java.util.Scanner;
import java.io.*;
public class ProductoSuma
{
Public static void main (String[] args)throws java.io.IOException
   {
    //Declaraciones de variables y otros objetos
    int num1, num2, num3, multiplica= 0, suma =0;
    Scanner teclado = new Scanner (System.in);

    /*Obtener Datos */
    System.out.print("Introduzca tres números: ");
    num1 =teclado.nextInt();
    num2 =teclado.nextInt();
    num3 =teclado.nextInt();
    /*Fin Obtener Datos*/

    /*ejecutar operaciones */
    if (num1 < 0)
    {
       multiplica = num1 * num2 * num3;
       System.out.println("Producto de los números: "+ multiplica);
    }
    else
    {
       suma = num1+num2+num3;
       System.out.println("Suma de los números: "+suma);
    }
    /*Fin ejecutar operaciones */
   }//fin main
}
```

14. ESTRUCTURAS REPETITIVAS

Las tres estructuras iterativas estudiadas en pseudocódigo, que permiten repetir un conjunto de instrucciones en función de cierta condición especificada previamente, se realizan en C con las sentencias de iteración: **while**, **do-while** y **for**.

14.1. Sentencia de iteración while

Equivale al *Mientras* de pseudocódigo. Su sintaxis es:

```
while (expresión-condicional)
```

```
        proposición
    //línea siguiente
```

La sentencia *while* ejecuta reiteradamente la o las sentencias correspondientes a **proposición**, mientras la **expresión-condicional** sea cierta. **Proposición** puede ser incluso una sentencia nula. Cuando la **expresión-condicional** sea falsa continuará el orden de ejecución por la *línea siguiente*.

Ejemplo: Escribir un fragmento de código para leer y contar caracteres hasta que se introduzca por teclado un 0.

```
 ..................
int numero, contNumeros=0;
Scanner teclado = new Scanner (System.in);

System.out.println("Escriba un número (0 para finalizar)");
num1 = teclado.nextInt();
 while(numero != 0)
  {
   contNumeros = contNumeros + 1;
   System.out.println("Escriba un número (0 para finalizar)");
   num1 = teclado.nextInt();
  }
 ...................
```

EJERCICIO PRÁCTICO DE APLICACIÓN

Codificación en Java del ejercicio del apartado 4 del capítulo 3.

```
/*Nombre del programa: Sumar.java*/
import java.util.Scanner;
import java.io.*;
public class Sumar
{
   public static void main (String[] args)throws java.io.IOException
   {
    int num1,num2,suma;
    char respuesta ='a';
    Scanner teclado = new Scanner (System.in);

    //Creamos un buffer de entrada de datos y
    //lo que se lea se le asigna a la variable 'entrada'
    BufferedReader entrada = new BufferedReader(new
    InputStreamReader(System.in));

    //¿Desea realizar suma de números?
    System.out.print("¿Quiere realizar la operacion de sumar S/N?");
    //Siempre que se lean datos de teclado se hace con String,
    //despues hay que hacer la conversion a char
    respuesta = entrada.readLine().charAt(0);
```

```
    //Fin DeseaRS

    while (respuesta =='S')
    {
       //Obtener Datos
       System.out.print("Introduzca dos números: ");
       num1 =teclado.nextInt();
       num2 =teclado.nextInt();
       //Fin Obtener datos

       //Realizar Suma
       suma = num1+ num2;
       //Fin Realizar Suma

       //Mostrar en pantalla resultado...
       System.out.println("El resultado de la suma es: "+suma);
       //Fin Mostrar....

       //¿Desea realizar la suma de números?
       System.out.print("¿Quiere realizar la operacion de sumar S/N?");
       respuesta = entrada.readLine().charAt(0);
       //Fin DeseaRS.
    }
   }
}
```

14.2. Proposición do while

Equivale a *Repetir-mientras* de pseudocódigo. Su sintaxis es:

```
do
 proposición
while (expresión-condicional);
//línea siguiente
```

La sentencia *do while* ejecuta iterativamente la o las instrucciones correspondientes a **proposición**, mientras que **expresión-condicional** sea cierta (distinta de cero, o NULL en el caso de expresiones punteros). Como se estudió en pseudocódigo, el cuerpo del bucle se ejecuta al menos una vez. Cuando **expresión-condicional** sea falsa el control de ejecución pasará a la *línea siguiente*.

Ejemplo: Escribir un fragmento de código para leer y contar caracteres hasta que se introduzca por teclado un asterisco. Resolver el ejercicio con un *do while*.

```
int contCarac;
char caracter;
BufferedReader entrada;
entrada = new BufferedReader(new InputStreamReader(System.in));
...................
do
```

```
  {
   System.out.print("Escriba un carácter (* para acabar)");
   caracter = entrada.readLine().charAt(0);
   contCarac = contCarac + 1;
  }
while(caracter != '*');
..................
```

14.3. Proposición for

Equivale a la estructura *Para* de pseudocódigo. Su sintaxis es:

```
for ([exp1]; [exp2]; [exp3])
   proposición
//línea siguiente
```

Las expresiones *exp1*, *exp2*, *exp3*, pueden ser cualquier tipo de expresiones bien construidas. Son opcionales y pueden ser de cualquier tipo, pero habitualmente se utilizan para lo siguiente:

exp1. Corresponde a la inicialización de la Variable de Control de Bucle contadora. Si existen varias inicializaciones aparecerán separadas por comas.

exp2. Es la condición de iteración del bucle y puede ser simple o compuesta.

exp3. Sentencia de actualización de la VCB (incremento o decremento). Cuando existen varias actualizaciones se separarán por comas.

El bloque de **proposición** puede ser cualquier tipo de sentencia simple o compuesta, e incluso sentencia nula.

El proceso se realiza del siguiente modo: en primer lugar se realiza si existe **exp1**, que sólo se evalúa una vez al comienzo del *for*. A continuación, se evalúa la **exp2** si existe y es cierta, se ejecuta la instrucción o instrucciones correspondientes a **proposicion** y, seguidamente, se realiza *exp3* si existe. Finalizada esta iteración, se vuelve a evaluar *exp2* comenzando con ello una iteración nueva, el proceso acaba cuando *exp2* sea falsa, pasando el control a la *línea siguiente*.

Ejemplo:

```
for (;;);// bucle infinito
```

Ejemplo: Escribir un fragmento de código para leer diez caracteres por teclado.

Dadas las anteriores declaraciones y

```
int cont;
..................
for (cont = 0; cont < 10; cont++)
{
 System.out.print("Escriba un carácter ");
 caracter = entrada.readLine().charAt(0);
}
```

```
.................
```

15. ESTRUCTURA DE SELECCIÓN MÚLTIPLE.

La estructura **switch** es equivalente al SEGÚN. Ejecuta cero, una o varias proposiciones posibles dependiendo del valor que tome la expresión que le acompaña.

Formato:

```
switch (expresión)
{
 case const 1:
      proposición1
      [break;]
 case const 2:
      proposición2
      [break;]
      …
 case const n:
      proposición n
      [break;]
      [default:]
       proposición
       [break;]
} /* Fin de switch. */
```

La sentencia *default* (similar a *else*) es opcional.

Las constantes sólo pueden ser números enteros o carácter (un sólo carácter). Las constantes pueden estar en cualquier orden, pero no está permitido repetir alguno de sus valores en un mismo *switch*.

Evaluación de la instrucción: Se evalúa la **expresión**, se compara con la constante de cada caso (*case*) hasta encontrar una que coincida con el valor de **expresión**, se ejecuta el bloque de proposiciones correspondientes. Si no hay entre las constantes ningún valor igual al de **expresión** y existe *default*, se ejecutará la proposición que le corresponde a este. Si no hay ninguna constante igual y no existe default, pasará a fin del *switch*.

La finalización ocurre cuando encuentra la llave de cierre del *switch* o cuando encuentra un *break*. Los *break* son opcionales, pero deben usarse si se desea que se detenga la ejecución en el *case* correspondiente. Si un *case* no incluye *break* la ejecución sigue hasta encuentrar un *break*, si existe, o la llave de cierre del *switch*.

Ejemplo: Realizar un fragmento de código para elegir entre las operaciones sumar o restar dos números.

```
int resultado, num1, num2;
char opcion;
BufferedReader entrada;
entrada = new BufferedReader(new InputStreamReader(System.in));
```

```
.................
menu();
opcion = entrada.readLine().charAt(0);
while (opcion != 'F')
{
 switch (opcion)
 {
       case 'S':
        resultado = sumar (num1,num2);
        break;

       case 'R':
        //System.out.print("\nEn construcción!!"); Ver NOTA1

        resultado = restar (num1,num2);
        break;

       default:
        System.out.print("\nError!!");
 }
       menu();
       opcion = entrada.readLine().charAt(0);
}
.................
```

EJERCICIO PRÁCTICO DE APLICACIÓN

Codificación en Java del ejercicio del apartado 11.2 del capítulo 3.

NOTA1:

Como técnica a la hora de desarrollar ejercicios más complicados se aconseja que se vaya desarrollando los módulos uno a uno dejando los demás "en construcción" . Esto ayuda a la verificación y validación de módulos haciendo posible que focalicemos solo en el módulo que nos disponemos a desarrollar y no en los demás.
La técnica no solo ayuda a depurar y validar de manera más sencilla, sino que además cara a nuestro cliente/profesor hará que nuestras aplicaciones/ejercicios estén siempre casi terminados y podamos ir enseñando en ejecución lo que se lleva desarrollado.

Nota2:

Recordad añadir *print* al código con objeto de ir observando en ejecución los valores que van teniendo las distintas variables del programa. Esto ayudará a verificar y depurar código.
Es una buena técnica dejar en comentario todos aquellos *print o código de relleno* que nos ha servido para la depuración. Esto nos ayudará más adelante en nuestro aprendizaje a recordar los pasos y dificultades que tuvimos en el desarrollo de cualquier ejercicio.

Nota3:

No olvidéis documentar cualquier problema relevante que nos haya ocurrido durante el proceso de desarrollo de cualquier ejercicio. Esto también nos ayudará en el proceso de aprendizaje.