Tema 1:

Introducción a Java

***Objetivos del tema**: este tema realiza una introducción rápida al lenguaje de programación Java, apoyándonos para ello en lo que hemos aprendido en la asignatura de Programación del primer semestre.*

INDICE:

1 Java: el lenguaje

La mayor parte de las personas que han empleado alguna vez un ordenador probablemente en alguna ocasión hayan oído hablar de algunos de estos lenguajes de programación: C, Pascal, Cobol, Visual Basic, Java, Fortran... y a una persona ya más introducida en ese mundillo posiblemente haya oído hablar de muchos otros: Oak, Prolog, Python, Dbase, JavaScrip, Delphi, Simula, Smalltalk, Modula, Oberon, Ada, BCPL, Common LISP, Scheme... En la actualidad se podrían recopilar del orden de varios cientos de lenguajes de programación distintos, sino miles.

Cabe hacerse una pregunta: ¿Para qué tanto lenguaje de programación?. Toda esta multitud de nombres puede confundir a cualquier no iniciado que haya decidido aprender un lenguaje, quien tras ver las posibles alternativas no sabe cuál escoger, al menos entre los del primer grupo, que por ser más conocidos deben estar más extendidos.

El motivo de esta disparidad de lenguajes es que cada uno ha sido creado para una determinada función, está especialmente diseñado para facilitar la programación de un determinado tipo de problemas, para garantizar seguridad de las aplicaciones, para obtener una mayor facilidad de programación, para conseguir un mayor aprovechamiento de los recursos del ordenador... Estos objetivos son muchos de ellos excluyentes: el adaptar un lenguaje a un tipo de problemas hará más complicado abordar mediante este lenguaje la programación de otros problemas distintos de aquellos para los que fue diseñado. El facilitar el aprendizaje al programador disminuye el rendimiento y aprovechamiento de los recursos del ordenador por parte de las aplicaciones programadas en este lenguaje; mientras que primar el rendimiento y aprovechamiento de los recursos del ordenador tiende a dificultar labor del programador.

La pregunta que viene a continuación es evidente: ¿Para qué fue pensado Java?. A continuación haremos una pequeña relación de las características del lenguaje, que nos ayudarán a ver para que tipo de problemas está pensado Java:

- **Simple**: es un lenguaje sencillo de aprender. Su sintaxis es la de C++ "simplificada". Los creadores de Java partieron de la sintaxis de C++ y trataron de eliminar de este todo lo que resultase complicado o fuente de errores en este lenguaje. La herencia múltiple, la aritmética de punteros, o la gestión de memoria dinámica (que en Java se elimina de modo transparente para el programador gracias al recogedor de basura) son ejemplos de "tareas complicadas" de C++ que en Java se han eliminado o simplificado.
- **Orientado a objetos**: en Java todo, a excepción de los tipos fundamentales de variables (int, char, long...) es un objeto.

- **Distribuido:** Java está muy orientado al trabajo en red, soportando protocolos como TCP/IP, UDP, HTTP y FTP. Por otro lado el uso de estos protocolos es bastante sencillo comparándolo con otros lenguajes que los soportan.
- **Robusto:** el compilador Java detecta muchos errores que otros compiladores solo detectarían en tiempo de ejecución o incluso nunca. Por ejemplo," if(a=b) then ... " o " int i; h=i*2; " son dos ejemplos en los que el compilador Java no nos dejaría compilar este código; sin embargo un compilador C compilaría el código y generaría un ejecutable que ejecutaría esta sentencia sin dar ningún tipo de error).
- **Seguro:** sobre todo en el caso de los Applet y las aplicaciones Java Web Start. Estos son programas diseñados para ser ejecutados en una página web. Java garantiza que ningún Applet puede escribir o leer de nuestro disco o mandar información del usuario que accede a la página a través de la red (como, por ejemplo, la dirección de correo electrónico). En general no permite realizar cualquier acción que pudiera dañar la máquina o violar la intimidad del que visita la página web.
- **Portable**: en Java no hay aspectos dependientes de la implementación, todas las implementaciones de Java siguen los mismos estándares en cuanto a tamaño y almacenamiento de los datos. Esto no ocurre así en C, por ejemplo. En éste un entero, por ejemplo, puede tener un tamaño de 16, 32 o más bits, siendo lo única limitación que el entero sea mayor que un short y menor que un long int. Así mismo C bajo UNIX almacena los datos en formato little endian, mientas que bajo Windows lo hace en big endian. Java lo hace siempre en little edian para evitar confusiones.
- **Arquitectura neutral**: el código generado por el compilador Java es independiente de la arquitectura: podría ejecutarse en un entorno UNIX, Mac o Windows. El motivo de esto es que el que realmente ejecuta el código generado por el compilador no es el procesador del ordenador directamente, sino que este se ejecuta mediante una máquina virtual. Esto permite que los Applets de una web pueda ejecutarlos cualquier máquina que se conecte a ella independientemente de que sistema operativo emplee (siempre y cuando el ordenador en cuestión tenga instalada una máquina virtual de Java).
- **Rendimiento medio**: actualmente la velocidad de procesado del código Java es semejante a la de C++, hay ciertos pruebas estándares de comparación (benchmarks) en las que Java gana a C y viceversa. Esto es así gracias al uso de compiladores just in time, compiladores que traducen los bytecodes de Java en código para una determinada CPU, que no precisa de la máquina virtual para ser ejecutado, y guardan el resultado de dicha conversión, volviéndolo a llamar en caso de volverlo a necesitar, con lo que se evita la sobrecarga de trabajo asociada a la interpretación del bytecode. No obstante por norma general el programa Java consume bastante más memoria

que el programa C, ya que no sólo ha de cargar en memoria los recursos necesarios para la ejecución del programa, sino que además debe simular un sistema operativo y hardware virtuales (la máquina virtual).

- Multithread: Soporta de modo nativo los threads, sin necesidad del uso de librerías específicas (como es el caso de C). Esto le permite además que cada Thread de una aplicación java pueda correr en una CPU (o core) distinta, si la aplicación se ejecuta en una máquina que posee varias CPU.

2 El JDK, Java Development Kit

El jdk es el entorno de desarrollo para Java que distribuye directamente Oracle y que, de un modo similar al compilador gcc, se basa en comandos de consola. Aunque nosotros emplearemos un entorno de desarrollo visual, de un modo similar a como hicimos con C, es útil entender las herramientas que hay detrás del IDE que empleamos. En esta sección veremos las principales.

2.1 Javac

Es el comando compilador de Java. Su sintaxis es:

```
javac ejemplo.java
```

La entrada de este comando ha de ser necesariamente un fichero que contenga código escrito en lenguaje Java y con extensión .Java. El comando nos creará un fichero .class por cada clase que contenga el fichero Java (más adelante veremos exactamente qué es una clase). Los ficheros .class contienen código bytecode, el código que es interpretado por la máquina virtual Java.

2.2 Java

Es el intérprete de Java. Permite ejecutar aplicaciones que previamente hayan sido compiladas y transformadas en ficheros .class. Su sintaxis es:

```
java ejemplo
```

No es necesario aquí suministrar la extensión del fichero, ya que siempre ha de ser un fichero .class.

2.3 Appletviewer

Se trata de un comando que verifica el comportamiento de un applet. La entrada del comando ha de ser una página web que contenga una referencia al applet que deseamos probar. Su sintaxis es:

```
Appletviewer mipagina.html
```

El comando ignora todo el contenido de la página web que no sean applets y se limita a ejecutarlos. Un ejemplo de página web "mínima" para poder probar un applet llamado myapplet.class sería:

```
<HTML>
<TITLE>My Applet </TITLE>
<BODY>
<APPLET CODE="myapplet.class" WIDTH=180 HEIGHT=180>
</APPLET>
</BODY>
</HTML>
```

2.4 Javadoc

Este comando permite generar documentación en formato html sobre el contenido de ficheros con extensión .Java. Su sintaxis es:

```
javadoc ejemplo.java
```

En la documentación generada por este comando se puede ver que métodos y constructores posee una determinada clase, junto con comentarios sobre su uso, si posee inner classes, la versión y el autor de la clase.... Una explicación detallada de la sintaxis de los comentarios de javadoc, que aquí no abordaremos, se encuentra en el capítulo 2 página 122 del libro Thinking in Java de Bruce Eckel (**http://www.bruceeckel.com/**).

A continuación ponemos un ejemplo de código Java donde se puede ver la sintaxis de varios comentarios de Javadoc

```
//: c02:HelloDate.Java
import java.util.*;

/**Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
*/
public class HelloDate {
  /** Sole entry point to class & application
   * @param args array of string arguments
   * @return No return value
   * @exception exceptions No exceptions thrown
  */
  public static void main(String[] args) {
//Esta línea imprime por consola la cadena de caracteres
//“Hello it’s”
    System.out.println("Hello, it's: ");
//Esta sentencia imprime la fecha actual del equipo
    System.out.println(new Date());
  }
} ///:~
```

3 Tipos de datos primitivos en Java

En este tema trataremos las estructuras básicas de Java de datos, sus tipos y las variables, operadores. Aquellos que estén familiarizados con C, o C++, no encontrarán prácticamente nada nuevo en este tema, ya que, como se ha dicho en el primer tema, Java hereda toda su sintaxis de C, pudiendo considerarse la sintaxis de Java una versión simplificada de la de C. Sólo las operaciones con Strings pueden resultar un poco novedosas.

3.1 Tipos de datos

En Java toda variable declarada ha de tener su tipo, y además antes de poder emplearla hemos de inicializarla a un valor, si no es compilador se quejará y no generará los archivos .class. Esto por ejemplo en C no es necesario, siendo fuente de muchos errores al emplearse en operaciones variables que nos hemos olvidado de inicializar. A continuación pasamos a describir los tipos de datos primitivos soportados en Java.

Enteros

Almacenan como su propio nombre indica números enteros, sin parte decimal. Cabe destacar, como ya se indicó en el primer tema, que por razones de portabilidad todos los datos en Java tienen el mismo tamaño y formato. En Java hay cuatro tipos de enteros:

Tabla 1: tipo de datos enteros en Java

Tipo	Tamaño (bytes)	Rango
Byte	1	-128 a 127
Short	2	-32768 a 32767
Int	4	-2147483648 a 2147483647
Long	8	-9223372036854775808 a 9223372036854775807

Para indicar que una constante es de tipo long lo indicaremos con una L: 23L.

Reales

Almacenan número reales, es decir números con parte fraccionaria. Hay dos tipos:

Tabla 2: tipos de datos reales en Java

Tipo	Tamaño (bytes)	Rango
Float	4	± 3.40282347E+38
Double	8	± 17976931348623157E+308

Si queremos indicar que una constante es flotante: ej: 2.3 hemos de escribir: 2.3F, sino por defecto será double.

Caracteres

En Java hay un único tipo de carácter: **char**. Cada carácter en Java esta codificado en un formato denominado Unicode, en este formato cada carácter ocupa dos bytes, frente a la codificación en ASCII, dónde cada carácter ocupaba un solo byte.

Unicode es una extensión de ASCII, ya que éste último al emplear un byte por carácter sólo daba acogida a 256 símbolos distintos. Para poder aceptar todos los alfabetos (chino, japonés, ruso...) y una mayor cantidad de símbolos se creó el formato Unicode.

En Java al igual que en C se distingue la representación de los datos char frente a las cadenas de caracteres. Los char van entre comillas simples: char ch = 'a', mientras que las cadenas de caracteres usan comillas dobles.

Boolean

Se trata de un tipo de dato que solo puede tomar dos valores: "**true**" y "**false**". Es un tipo de dato bastante útil a la hora de realizar chequeos sobre condiciones. En C, hasta C99, no había un dato equivalente y para suplir su ausencia muchas veces se emplean enteros con valor 1 si "true" y 0 si "false". Otros lenguajes como Pascal sí tienen este tipo de dato.

3.2 Definición de variables

Al igual que C, Java requiere que se declaren los tipos de todas las variables empleadas. La sintaxis de inicialización es la misma que C:

```
int i;
```

Sin embargo, y a diferencia que en C, se requiere inicializar todas las variables locales antes de usarlas, si no el compilador genera un error y aborta la compilación. Se puede inicializar y asignar valor a una variable en una misma línea:

```
int i = 0;
```

Asignación e inicialización pueden hacerse en líneas diferentes:

```
int i ;
i = 0;
```

Es posible iniciar varias variables en una línea:

```
int i, j, k;
```

Después de cada línea de código, bien sea de inicialización o de código, al igual que en C va un ";".

En Java, al igual que en todo lenguaje de programación hay una serie de palabras reservadas que no pueden ser empleadas como nombres de variables (if, int, char, else, goto....); alguna de estas se emplean en la sintaxis del lenguaje, otras, como goto no se emplean en la actualidad pero se han reservado por motivos de compatibilidad por si se emplean en el futuro.

Los caracteres aceptados en el nombre de una variable son los comprendidos entre "A-Z", "a-z", _, $ y cualquier carácter que sea una letra en algún idioma, no necesariamente sólo del idioma inglés, como sucede en C. No obstante, es recomendable limitarse a emplear caracteres del alfabeto inglés para curarse en salud.

4 Conversión entre tipos numéricos

Las normas de conversión entre tipos numéricos son las habituales en un lenguaje de programación: si en una operación se involucran varios datos numéricos de distintos tipos todos ellos se convierten al tipo de dato que permite una mayor precisión y rango de representación numérica; así, por ejemplo:

-Si cualquier operando es double todos se convertirán en double.

-Si cualquier operando es float y no hay ningún double todos se convertirán a float.

-Si cualquier operando es long y no hay datos reales todos se convertirán en long.

-Si cualquier operando es int y no hay datos reales ni long se convertirán en int.

-En cualquier otro caso el resultado será también un int.

Java solo tiene dos tipos de operadores enteros: uno que aplica para operar datos de tipo long, y otro que emplea para operar datos de tipo int. De este modo cuando operemos un byte con un byte, un short con un short o un short con un byte Java empleará para dicha operación el operador de los datos tipo int, por lo que el resultado de dicha operación será un int siempre.

La "jerarquía" en las conversiones de mayor a menor es:

double <- float <- long <- int <- short <- byte

Estas conversiones sólo nos preocuparán a la hora de mirar en qué tipo de variable guardamos el resultado de la operación; esta ha de ser de una jerarquía mayor o igual a la jerarquía de la máxima variable involucrada en la operación. Si es de rango superior no habrá problemas.

Es posible convertir un dato de jerarquía "superior" a uno con jerarquía "inferior", arriesgándonos a perder información en el cambio. Este tipo de operación (almacenar el contenido de una variable de jerarquía superior en una de jerarquía inferior) se denomina *cast*. Veamos un ejemplo para comprender su sintaxis:

```
public class Ejemplo1 {
  public static void main(String[] args) {
    int entero = 9,entero2;
    float real = 49.99F;
    System.out.println("entero:  "  +  entero  +  "  real:  "
+real);
    entero2 = (int)real; //empleo de un cast
    System.out.println("real: " + real + " : " +entero2);
    real = entero2; //no necesita cast
    System.out.println("real: " + real + " : " +entero2);
    int enteroMasGrande = 0x7FFFFFFF;
    System.out.println("El   entero   más   grande:   "   +
enteroMasGrande);
    enteroMasGrande++;//overflow
    System.out.println("El  entero  más  grande  +  1:  "  +
enteroMasGrande);
    float real2 = 2.3F;
    //float m = 2.3; daria error al compilar.
    System.out.println("m: "+real2);
  }
}
```

5 Operadores

Los operadores básicos de Java son + , - , * , / para suma, resta, producto y división. El operador / representa la división de enteros si ambos operandos son enteros. Su módulo puede obtenerse mediante el operador % (7/4= 1; 7% 4=3).

Además existen los operadores decremento e incremento: -- y ++, respectivamente. La operación que realizan son incrementar y decrementar en una unidad a la variable a la que se aplican. Su acción es distinta según se apliquen antes (++a) o después (a++) de la variable. El siguiente programa ilustra estos distintos efectos:

```
public class Ejemplo2 {

    public static void main(String[] args) {
        int entero = 1;
```

```
        prt("entero : " + entero);
        prt("++entero : " + ++entero); // Pre-incremento
        prt("entero++ : " + entero++); // Post-incremento
        prt("entero : " + entero);
        prt("--entero : " + --entero); // Pre-decremento
        prt("entero-- : " + entero--); // Post-decremento
        prt("entero : " + entero);
    }

//esto nos ahorrara teclear
    static void prt(String s) {
        System.out.println(s);
    }
}
```

5.1 Exponenciación

En Java a diferencia de en otros lenguajes no existe el operador exponenciación. Tampoco existen los operadores Seno o Coseno. Si se desea realizar una operación de exponenciación se deberá invocar el método correspondiente de la clase Math de Java.lang. Estos métodos son estáticos, por lo que no es necesario crear un objeto de dicha clase. Su sintaxis general es:

```
        Math.metodo(argumentos);
```

En el siguiente código aparecen ejemplos. Si alguna vez deseamos realizar algún otro tipo de operación y queremos ver si la soporta Java podemos hacerlo consultando la ayuda on-line de la clase Math, o bien la documentación que podemos descargar los desde la misma página que el jdk.

```
public class Ejemplo3 {

    public static void main(String[] args) {
        double dato = Math.PI, dato2 = 2;
        prt("Cos dato : " + Math.cos(dato));
        prt("Sin dato : " + Math.sin(dato));
        prt("dato2^dato : " + Math.pow(dato2, dato));
    }
    //esto nos ahorrara teclear
```

```
    static void prt(String s) {
        System.out.println(s);
    }
}
```

5.2 Operadores lógicos

En la tabla a continuación recogemos los operadores lógicos disponibles en Java:

Tabla 3: operadores lógicos

Operador	Operación que realiza
!	Not lógico
==	Test de igualdad
!=	Test de desigualdad
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
&&	And lógico
||	Or lógico

Con el siguiente ejemplo se muestra la función de estos operadores:

```
import java.util.*;
```

```
public class Ejemplo4 {
  public static void main(String[] args) {
//Creamos un objeto de tipo Random, almecenado un puntero a
//el en la variable rand. En el tema 5 se detallará como se
//crean objetos.
    Random rand = new Random();
//el método nextInt() del objeto Random creado (se invoca
//como rand.nextInt()) genera un número aleatorio entero. En
//el tema 5 se explica que son los métodos y como emplearlos.
//El módulo 100 de un entero aleatorio será un entero
//aleatorio entre 0 y 100.
    int i = rand.nextInt() % 100;
    int j = rand.nextInt() % 100;
//Imprime I y j por consola
    prt("i = " + i);
    prt("j = " + j);
//Imprime diversas operaciones binarias sobre i y j, junto
//con su resultado.
    prt("i > j es " + (i > j));
    prt("i < j es " + (i < j));
    prt("i >= j es " + (i >= j));
    prt("i <= j es " + (i <= j));
    prt("i == j es " + (i == j));
    prt("i != j es " + (i != j));
    prt("(i < 10) && (j < 10) es "
       + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) es "
       + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Una posible salida de este programa es:

```
i = 85
```

```
j = 4
i > j es true
i < j es false
i >= j es true
i <= j es false
i == j es false
i != j es true
//true && false = false
(i < 10) && (j < 10) es false
//true || false = true
(i < 10) || (j < 10) es true
```

6 Cadenas de caracteres

En Java no hay un tipo predefinido para cadenas de caracteres, en su lugar hay una clase, String, que es la que soporta las distintas operaciones con cadenas de caracteres. La definición de un String es:

```
   String cadena;   //no inicializado
   String cadena =””;   //cadena vacía
   String cadena = “Hola”;  //inicialización y asignación
juntas.
```

A continuación veremos algunas operaciones básicas soportadas por la clase String:

6.1 Concatenación

La concatenación en Java es muy sencilla: se realiza con el operador +, es decir “sumando” cadenas de caracteres obtenemos la concatenación de estas. Lo ilustraremos con un ejemplo:

```
   String saludo = “hola”;
   String nombre = “Pepe”;
   String saludaPepe = “”;
   saludaPepe = saludo + nombre;// saluda_pepe toma el valor
holaPepe
```

La sencillez de Java en el manejo de cadenas de caracteres llega incluso más alla: si una cadena la intentamos encadenar con otro tipo de variable automáticamente se convierte la otra variable a String, de tal modo que es perfectamente correcto hacer:

```
   String saludo = “hola”;
   int  n = 5;
   saludo = saludo + “ “ + n;// saludo toma el valor “hola 5”
```

6.2 Subcadenas

En la clase String hay un método que permite la extracción de una subcadena de caracteres de otra. Su sintaxis es:

```
nombreString.substring((int)posiciónInicial,(int)posiciónFinal);
```

Donde posiciónInicial y posiciónFinal son respectivamente la posición del primer carácter que se desea extraer y del primer carácter que ya no se desea extraer.

```
   String saludo = “hola”;
   String subsaludo = “”;
   Subsaludo = saludo.substring(0,2);// subsaludo toma el valor “ho”
```

Puede extraerse un char de una cadena, para ello se emplea el método charAt(posición), siendo posición la posición del carácter que se desea extraer.

6.3 Comparación de cadenas

Para comparar dos cadenas de caracteres se emplea otro método de String: equals. Su sintaxis es:

```
   cadena1.equals(cadena2);
```

Devuelve true si son iguales y false si son distintos. El siguiente ejemplo permitirá ilustrar estas operaciones con Strings:

```
public class Ejemplo5 {

    public static void main(String[] args) {
        String saludo = "Hola";
        String saludo2 = "hola";
        int entero = 5;
        prt(saludo.substring(0, 2));
        prt(saludo + " " + entero);
        prt("saludo == saludo2 " + saludo.equals(saludo2));
    }

    static void prt(String s) {
        System.out.println(s);
    }
}
```

Consulta el javadoc de la clase java.lang.String para ver qué otros métodos útiles tiene esta clase.

7 Arrays

En Java los arrays son un objeto. Como tales se crean mediante el comando new (se verá su uso más adelante). La sintaxis en la definición de un array es la siguiente:

```
        Tipo_datos[] nombre_array = new
Tipo_datos[tamaño_array];
```

Tipo_datos es el tipo de los datos que se almacenarán en el array (int, char, String... o cualquier objeto). Tamaño_array es el tamaño que le queremos dar a este array. Veamos un ejemplo:

```
            int[] edades = new int[10];
```

En este ejemplo hemos definido un array llamado edades, en el que podremos almacenar 10 datos tipo entero. El primer elemento de un array se sitúa en la posición 0, exactamente igual que en C. Una diferencia que hay respecto a C es que el objeto que representa al array tiene un campo denominado "length" que contiene el tamaño del array, de tal modo que no es necesario estar trabajando por un lado con el array y por otro lado con un entero que representa su dimensión. Veamos un ejemplo de código Java que trabaja con arrays:

```
public class Ejemplo6 {

    public static void main(String[] args) {
        int[] edades = new int[10];
        for (int i = 0; i < edades.length; i++) {
            edades[i] = i;
            System.out.println("Elemento " + i + ": "+
edades[i]);
        }
        int suma = 0;
        for (int i = 0; i < 10; i++) {
            suma = suma + edades[i];
        }
        System.out.println("Suma " + suma);

        float matriz[][] = new float[4][5];
        for (int i = 0; i < matriz.length; i++) {
            for (int j = 0; j < matriz[0].length; j++) {
               matriz[i][j]= i *100 + j;
               System.out.println("Elemento " + i + ", " + j
+ ": "+ matriz[i][j]);
            }
        }
    }
}
```

8 Tipos enumerados

Esta característica del lenguaje sólo está disponible en Java 5 y superior. Si estás empleando Java 1.4.X o inferior no podrás emplearla. Cuando escribo este tutorial la versión actual de Java es la 7.

Los tipos de datos enumerados son un tipo de dato definido por el programador (no como ocurre con los tipos de datos primitivos). En su definición el programador debe indicar un conjunto de valores finitos sobre los cuales las variables de tipo enumeración deberán tomar valores. La principal funcionalidad de los tipos de datos enumerados es incrementar la legibilidad del programa. La mejor forma de comprender lo qué son es viéndolo; para definir un tipo de dato enumerado se emplea la sintaxis:

```
modificadores enum NombreTipoEnumerado{ VALOR1,VALOR2,.. }
```

Los posibles valores de "modificadores" serán vistos más adelante. El caso más habitual es que modificadores tome el valor "public". El nombre puede ser cualquier nombre válido dentro de Java. Entre las llaves se ponen los posibles valores que podrán tomar las variables de tipo enumeración, valores que habitualmente se escriben en letras mayúsculas. Un ejemplo de enumeración podría ser:

```
public enum Semana {LUNES, MARTES, MIÉRCOLES, JUEVES,
VIERNES, SÁBADO, DOMINGO}
```

Las definiciones de los tipos enumerados deben realizarse fuera del método main y, en general, fuera de cualquier método; es decir, deben realizarse directamente dentro del cuerpo de la clase. Más adelante se explicará detalladamente qué es una clase y qué es un método.

Para definir una variable de la anterior enumeración se emplearía la siguiente sintaxis:

```
Semana variable;
```

y para darle un valor a las variables de tipo enumeración éstas deben asignarse a uno de los valores creados en su definición. El nombre del valor debe ir precedido del nombre de la propia enumeración:

```
variable = Semana.DOMINGO;
```

Veamos un ejemplo de programa que emplea enumeraciones:

```
public class Ejemplo7 {
    //definimos un tipo enumerado
    //los tipos enumerados deben definirse siempre fuera
    //del main y fuera de cualquier metodo

    public enum Semana {
        LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO,
DOMINGO
    };

    public static void main(String[] args) {
        //definimos una variable que pertenece al tipo
enumerado Semana
        //y le damos el valor que representa el dia martes
        Semana hoy = Semana.MARTES;
        //si el dia se cayese en el fin de semana no hay que
trabajar
        //observese como gracias a la numeracion del programa
es facil del entender
        if (hoy == Semana.DOMINGO || hoy == Semana.SABADO) {
            System.out.println("Hoy toca descansar");
        } else {
            System.out.println("Hoy toca trabajar");
        }
    }
}
```

9 Control de flujo

El modo de ejecución de un programa en Java en ausencia de elementos de control de flujo es secuencial, es decir una instrucción se ejecuta detrás de otra y sólo se ejecuta una vez. Esto nos permite hacer programas muy limitados; para evitarlo se introducen estructuras de control de flujo. Las estructuras de control de flujo de Java son las típicas de cualquier lenguaje de programación, por lo que supondremos que todos estáis familiarizados con ellas y se explicaran con poco detenimiento.

9.1 Sentencias Condicionales

Ejecutan un código u otro en función de que se cumpla o no una determinada condición. Pasemos a ver sus principales tipos.

if then else

Su modo más simple de empleo es:

```
if(condicion) {
     Grupo de sentencias
}
```

Condición es un valor tipo boolean; a diferencia de C no es posible emplear cualquier expresión numérica como condición de un condicional tipo if (o de cualquier otra estructura de control de flujo) en Java. El grupo de sentencias se ejecuta solo si la condición toma un valor true. En caso contrario se sigue ejecutando ignorando el Grupo de sentencias.

```
if(condicion) {
     Grupo de sentencias
}
else  {
     Grupo2 de sentencias
}
```

Si condición toma el valor true se ejecuta Grupo de sentencias, en caso contrario se ejecuta Grupo2 de sentencias. En ambos casos se continúa ejecutando el resto del código.

```
if(condicion) {
     Grupo de sentencias
}
else if (condicion2){
     Grupo2 de sentencias
}
else if (condicion3){
     Grupo3 de sentencias
}
…

else{
     Grupo_n de sentencias
```

```
    }
```

Si condición toma el valor true se ejecuta Grupo de sentencias, si condicion2 toma el valor true se ejecuta Grupo2 de sentencias... y así sucesivamente hasta acabarse todas las condiciones. Si no se cumple ninguna se ejecuta Grupo_n de sentencias. Este último else es opcional. En ambos casos se continúa ejecutando el resto del código.

Ilustraremos esta estructura de control de flujo con el siguiente ejemplo:

```
public class Ejemplo8 {

    static int test(int val, int val2) {
        int result;
        if (val > val2) {
            result = +1;
        } else if (val < val2) {
            result = -1;
        } else {
            result = 0;
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println(test(10, 5));
        System.out.println(test(5, 10));
        System.out.println(test(5, 5));
    }
}
```

Switch

Los creadores de Java trataron de hacer de este lenguaje una versión simplificada y mejorada del lenguaje de C++. Su trabajo fue bastante bueno, pero no perfecto. Prueba de ello es esta sentencia: es casi tan poco flexible como en C++.

Expliquemos su sintaxis antes de dar los motivos de esta crítica:

```
switch(selector) {
   case valor1 : Grupo de sentencias1; break;
   case valor2 : Grupo de sentencias2; break;
   case valor3 : Grupo de sentencias3; break;
   case valor4 : Grupo de sentencias4; break;
   case valor5 : Grupo de sentencias5; break;
             // ...
   default: statement;
}
```

Se compara el valor de selector con sentencias_n. Si el valor coincide se ejecuta su respectivo grupo de sentencias. Si no se encuentra ninguna coincidencia se ejecutan las sentencias de default. Si no se pusieran los break una vez que se encontrase un valor que coincida con el selector se ejecutarían todos los grupos de sentencias, incluida la del default.

Ha llegado el momento de justificar la crítica hecha a los creadores de Java. Este tipo de estructura tiene sus posibilidades muy limitadas, ya que en las condiciones sólo se admite la igualdad, no ningún otro tipo de condición (sería fácil pensar ejemplos dónde, por poner un caso, se le sacaría partido a esta sentencia si aceptase desigualdades). Además para colmo esta comparación de igualdad sólo admite valores tipo char o cualquier tipo de valores enteros menos long (si estás empleando Java 5 o posterior también se puede emplear un tipo enumerado). No podemos comparar contra reales, Strings....

También se le podría criticar el hecho de que una vez cumplida una condición se ejecuten todas las sentencias si no hay instrucciones break que lo impidan. Esto es en muchas ocasiones fuente de errores, aunque también hay que reconocer que a veces se le puede sacar partido, de hecho en el ejemplo que empleamos para ilustrar esta sentencia aprovechamos esta característica:

```
public class Ejemplo9 {

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            char caracter = (char) (Math.random() * 26 +
'a');
            System.out.print(caracter + ": ");
            switch (caracter) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
```

```
                case 'u':
                    System.out.println("vocal");
                    break;
                default:
                    System.out.println("consonante");
            }
        }
    }
}
```

9.2 Bucles

Son instrucciones que nos permiten repetir un bloque de código mientras se cumpla una determinada condición. Pasemos a ver sus tipos.

Bucle while

Cuando en la ejecución de un código se llega a un bucle while se comprueba si se verifica su condición, si se verifica se continua ejecutando el código del bucle hasta que esta deje de verificarse. Su sintaxis es:

```
   while(condición){
        Grupo de sentencias}
```

Ilustramos su funcionamiento con un ejemplo:

```
public class Ejemplo10 {

    public static void main(String[] args) {
        double r = 0;
        while (r < 0.99d) {
            r = Math.random();
            System.out.println(r);
        }
    }
```

```
}
```

Bucle do while

Su comportamiento es semejante al bucle while, sólo que aquí la condición va al final del código del bucle, por lo que tenemos garantizado que el código se va a ejecutar al menos una vez. Dependerá del caso concreto si es más conveniente emplear un bucle while o do while. La sintaxis de do while es:

```
do {
     Grupo de sentencias;
}while(condición);
```

Obsérvese como el ejemplo 10 implementado mediante un bucle do while independientemente del valor de r ejecutará al menos una vez el código de su cuerpo:

```
public class Ejemplo11 {

    public static void main(String[] args) {
        double r;
        do {
            r = Math.random();
            System.out.println(r);
        } while (r < 0.99d);
    }
}
```

Bucle for

Su formato es el siguiente:

```
for(expresion1;expresion2;expresion3){
     Grupo de sentecias;
}
```

Expresion1 es una asignación de un valor a una variable, la variable-condición del bucle. Expresion2 es la condición que se le impone a la variable del bucle y expresion3 indica una

operación que se realiza en cada iteración a partir de la primera (en la primera iteración el valor de la variable del bucle es el que se le asigna en expresion1) sobre la variable del bucle. A diferencia de C (y al igual que C++), Expresion1 además de una asignación de valor a una variable también puede definir una variable cuyo ámbito será igual al cuerpo del bucle.

```
public class Ejemplo12 {

    public static void main(String[] args) {

        for (char caracter = 0; caracter < 128; caracter++) {
            System.out.println(
                    "valor: " + (int) caracter + " caracter:
" + caracter);
        }
    }
}
```

Bucle for-each

Esta característica sólo está disponible en Java 5 y versiones posteriores. Se trata de un bucle diseñado con el propósito de recorrer un conjunto de objetos. Por lo de ahora la única estructura de datos que hemos visto que puede contener un conjunto de objetos es un array; más adelante en este curso veremos otras estructuras de datos que pueden contener varios objetos. Su sintaxis es:

```
   for(Tipo elemento: colecciónElementos){
        Grupo de sentecias;}
```

Tipo es el tipo de dato de los elemento del conjunto; elemento es una variable a auxiliar que la primera vez que se ejecute el bucle tomará el valor del primer elemento del conjunto, la segunda vez tomará el valor del segundo elemento del conjunto y así sucesivamente. La colecciónElementos es el conjunto de elementos sobre los cuales queremos iterar (un array para nosotros). El bucle se repetirá una vez para cada elemento de la colección. Veamos un ejemplo:

```
public class Ejemplo13 {

    public static void main(String[] args) {
        //Este arrays sera la coleccion de elementos por la
que iteraremos
        int array[] = new int[10];
        int suma = 0;
        //con este bucle damos valores a los elementos del
array
        for (int i = 0; i < array.length; i++) {
            array[i]= 2*i;
        }
        for (int entero : array){  //para cada elemento del
array
            suma = suma + entero;
        }
        System.out.println(suma);
    }
}
```

9.3 Break y continue

No se tratan de un bucle, pero sí de sentencias íntimamente relacionadas con estos. El encontrarse una sentencia break en el cuerpo de cualquier bucle detiene la ejecución del cuerpo del bucle y sale de este, continuándose ejecutando el código que hay tras el bucle.

Esta sentencia también se puede usar para forzar la salida del bloque de ejecución de una instrucción condicional (esto ya se vio con switch).

Continue también detiene la ejecución del cuerpo del bucle, pero en esta ocasión no se sale del bucle, sino que se pasa a la siguiente iteración de este.

Observaremos el funcionamiento de ambos en un mismo ejemplo:

```
public class Ejemplo14 {


    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            if (i == 74) {
                break; // teminamos aqui el bucle
            }
            if (i % 9 != 0) {
                continue;//Salto a la siguiete iteracion
            }
```

```
            System.out.println(i);//Solo se imprimen los
multiplos de nueve menores que 74
        }
        int entero = 0;
        // Lazo infinito del cual se sale con break:
        while (true) {
            entero++;
            int j = entero * 27;
            if (j == 1269) {
                break; // Salimos del lazo
            }
            if (entero % 10 != 0) {
                continue; //Salto a la siguiete iteracion
            }
            System.out.println(entero);
        }
    }
}
```

9.4 return

Sus funciones son las mismas que en C. Cuando se llama a un procedimiento (que en OOP se denominó método) al encontrarse con una sentencia return se pasa el valor especificado al código que llamó a dicho método y se devuelve el control al código invocador. Su misión tiene que ver con el control de flujo: se deja de ejecutar código secuencialmente y se pasa al código que invocó al método.

Esta sentencia también está profundamente relacionada con los métodos, ya que es la sentencia que le permite devolver al método un valor. Podíamos haber esperado ha hablar de métodos para introducir esta sentencia, pero hemos decidido introducirla aquí por tener una función relacionada, entre otras cosas, con control de flujo. En el ejemplo 7 ya se ha ejemplificado su uso.

10 Lectura de datos de consola

En Java no hay una función equivalente a "scanf" que nos permita leer de un modo sencillo datos de la consola, sino que para leer datos de la consola tenemos que emplear directamente las librerías de entrada y salida de Java. Más adelante en esta asignatura

cubriremos esas librerías en detalle, pero ahora daremos una serie de recetas para conseguir obtener entrada del usuario de la consola.

Lo primero que tenemos que hacer es "envolver" la entrada estándar de la consola ("System.in") en un BufferedReader que nos va a permitir leer una línea de texto de la consola:

```
BufferedReader consola = new BufferedReader(new
                                InputStreamReader(System.in));
```

A continuación podremos leer líneas de texto empleando el método readLine ():

```
stringLeido = consola.readLine();
```

Si el dato que queremos obtener del usuario era una cadena de texto, con esto hemos terminado nuestro trabajo. Si el dato es un número entero o un número real, por ejemplo, tendremos que transformar esa cadena de texto que contiene ese número entero o número real al dato correspondiente. Para ello en Java y una serie de clases que suelen tener el mismo nombre que el tipo de dato primitivo correspondiente, pero que se escriben con la primera letra en mayúscula, y que proporcionan una serie de métodos de utilidad (.parseXXX) para realizar conversiones de una cadena de caracteres al tipo de dato correspondiente o viceversa. Por ejemplo, si la cadena de caracteres que hemos leído en la consola realmente contiene un número real debemos escribir:

```
float real = Float.parseFloat(stringLeido);
```

Donde stringLeido es la cadena de caracteres que leímos de la consola.

Todo el código que accede al sistema de entrada y salida en Java lanza excepciones; es decir, a diferencia de en C donde los fallos en las operaciones de entrada y salida se indicaban con un tipo especial de retorno, aquí se termina la ejecución del código en la línea que ha dado el error; por ello el código que realicé lecturas de la consola deberá estar dentro de un bloque try-catch

```
try {
     //código que accede a la consola
```

```
} catch (IOException ex) {
            System.out.println("Ha habido un error" + ex);
}
```

A continuación veremos un ejemplo donde se lee de la consola números enteros, reales de simple y doble precisión, y un carácter.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Ejemplo15 {

    public static void main(String[] args) {
        try {
            System.out.println("Escribe una cadena de texto :
");
            String stringLeido;
            BufferedReader consola = new BufferedReader(new
InputStreamReader(
                    System.in));
            stringLeido = consola.readLine();
            System.out.println("La cadena de texto leida es:"
+ stringLeido);

            System.out.println("Escribe un entero : ");

            stringLeido = consola.readLine();
            int entero = Integer.parseInt(stringLeido);
            System.out.println("El entero leido es:" +
entero);

            System.out.println("Escribe un float : ");
            stringLeido = consola.readLine();
            float real = Float.parseFloat(stringLeido);
            System.out.println("El real leido es:" + real);
```

```
            System.out.println("Escribe un double : ");
            stringLeido = consola.readLine();
            double realGrande =
Double.parseDouble(stringLeido);
            System.out.println("El real  largo leido es:" +
realGrande);

            System.out.println("Escribe un caracter : ");
            stringLeido = consola.readLine();
            char caracter = stringLeido.charAt(0);
            System.out.println("La caracter leido es:" +
caracter);
        } catch (IOException ex) {
            System.out.println("Se ha producido un error" +
ex);
        }
    }
}
```

11 Ejercicios

1. Escribir un programa que defina variables que representen el número de días de un año, el número de horas que tiene un día, el número de minutos que tiene una hora y el número de segundos que tiene un minuto. Emplear las variables que ocupen el mínimo espacio de memoria posible. A continuación, calcular el número de segundos que tiene un año y almacenar el valor del cálculo en otra variable.
2. Escribir un programa que muestre por consola los mayores números enteros que se pueden representar mediante un char, short e int.
3. Calcular la suma de todos los múltiplos de 5 comprendidos entre 1 y 100. Calcular además cuantos hay y visualizar cada uno de ellos.
4. Escribe un programa que calcule el mínimo y el máximo de una lista de números enteros positivos introducidos por el usuario. La lista finalizará cuando se introduzca un número negativo.

5. Escribe un programa que visualice por pantalla la tabla de multiplicar de los 10 primeros números naturales.
6. Escribe un programa que muestre por pantalla la lista de los 100 primeros números primos.
7. Escribe un programa que lea un número entero de teclado y lo descomponga en factores primos; por ejemplo 40 = 2 * 2 * 2 * 5.
8. Empleando un array, escribir un programa que pida al usuario números enteros hasta que se introduzca el número 0. A continuación, calcular la media, el mínimo y el máximo de los datos introducidos.
9. Escribir un programa que solicite al usuario dos vectores de N elementos y que imprima su producto escalar.
10. Escribir un programa que rellene una matriz cuadrada (las dimensiones de la matriz serán un parámetro que se pida al usuario) con números aleatorios de tal modo que la matriz sea simétrica. Imprimir la matriz por pantalla.
11. Escribir un programa que multiplique dos matrices. Sus dimensiones y valores deben de solicitarse al usuario por teclado y tras realizar la multiplicación debe visualizarse en pantalla ambas matrices y el resultado de la multiplicación.
12. Escribe un programa que acepte una cadena de caracteres (que podrá contener cualquier carácter a excepción del retorno de carro) y que diga cuántas vocales contiene.
13. Escribe un programa que devuelva el número de bases de ADN que hay entre la primera y la última aparición de una base determinada en una cadena de ADN. Tanto la cadena de ADN como la base se le pedirán al usuario. Por ejemplo, la cadena puede ser AACGTGTCACGTGGTAC y la base de la G, con lo que la respuesta sería 9 (AACG**TGTCACGTG**GTAC).
14. Escribe un programa que pida dos cadenas de ADN al usuario y cuente cuántas veces está contenida la segunda cadena de ADN en la primera.
15. Ejercicio voluntario: escribir un programa que genere una cadena de ADN aleatoria con un tamaño de 10,000,000 de pares de bases. A continuación el usuario introducirá una tripleta de pares de bases (por ejemplo CGC) y el programa deberá indicar cuántas veces sucede esa tripleta en la cadena.