INDICE

UNIDAD DIDACTICA 1ª. Introducción a la Programación Orientada a Objetos.

UNIDAD DIDACTICA 2ª. Iniciación en JAVA.

UNIDAD DIDACTICA 3ª. Profundizando en JAVA.

UNIDAD DIDACTICA 1ª. Introducción a la programación Orientada a Objetos.

Tema 1: Introducción a la programación de ordenadores.

1.1.- Programación de ordenadores.

Un **ordenador** es un dispositivo electrónico que es capaz de ejecutar un conjunto de instrucciones previamente almacenadas, que constituye lo que llamamos **programa**.

Los componentes físicos de un ordenador, la circuitería y la tecnología relacionada con el ordenador, considerado una máquina, recibe el nombre de **Hardware**. Los programas de ordenador y todo lo relacionado con su creación y desarrollo se llama **Software**.

La **Programación de Ordenadores** es un conjunto de métodos, técnicas y reglas para poder construir programas de ordenador legibles, correctos y eficientes.

Un **Programa** es una secuencia de instrucciones escrita en un lenguaje determinado que el ordenador es capaz de comprender.

Un **Lenguaje de Programación** es un lenguaje utilizado para construir programas de ordenador. En un programa se suele diferenciar entre la sintaxis (cómo escribir las sentencias) y la semántica (significado de las sentencias).

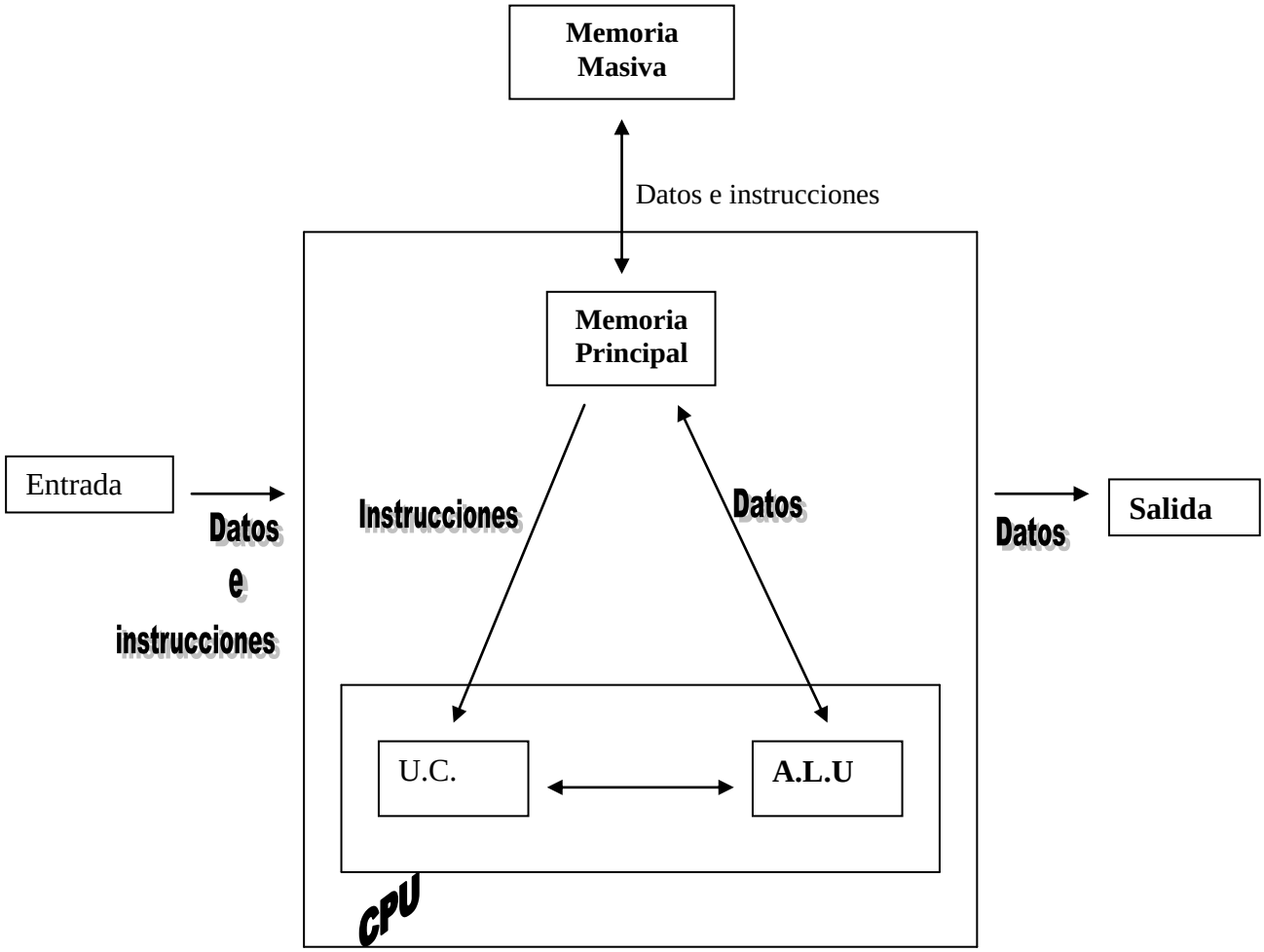
Un **algoritmo** es una secuencia de pasos, sin ambigüedad, que permite obtener la solución a un problema en un número finito de pasos y en un tiempo finito.

La expresión de un algoritmo mediante un lenguaje de programación recibe el nombre de **codificación**.

1.2.- El ordenador: Hardware y Software.

1.2.1.- Hardware.

La estructura funcional típica de un ordenador es la siguiente:



En un ordenador se distinguen fundamentalmente los siguientes componentes: Unidad Central de Proceso (CPU), dispositivos de Entrada/Salida, Memoria Principal. y dispositivos de Memoria Auxiliar.

La **Unidad de Entrada** es el dispositivo por el que se introducen datos al ordenador.

La **Unidad de Salida** es el dispositivo por el que obtenemos información por parte del ordenador.

La **Memoria Principal** es la unidad donde se almacenan los datos y los programas en ejecución. En la M.P. diferenciamos:

- Memoria RAM (Random Access Memory). Esta memoria es volátil donde se puede leer y escribir. En ella se almacenan los datos y los programas de aplicación.

- Memoria ROM (Read Only Memory). Esta memoria es permanente, pero no se puede escribir en ella. Normalmente almacena programas del sistema ó datos de configuración del ordenador.

La **Memoria Auxiliar** está formada por los dispositivos que permiten almacenar datos y programas de forma permanente en sistema de almacenamiento masivo, como por ejemplo los discos duros.

La **CPU** o procesador está formada por la UC y la ALU. La primera es la que controla y coordina a los restantes componentes del ordenador para la ejecución de las instrucciones. La ALU se encarga de realizar las operaciones lógicas y aritméticas entre los datos.

1.2.2.- Software. A un nivel muy básico se suele distinguir entre software de aplicación y software de sistemas.

- **Software de Aplicación**: son los programas que se desarrollan para solucionar las necesidades de un grupo de usuarios muy concretos.

- **Software de Sistemas**: programas de base que tratan directamente sobre la máquina, como los sistemas operativos, compiladores, etc.

1.3.- Lenguajes de programación. Todo lo relacionado con las reglas y los símbolos que pueden utilizarse para construir programas constituyen *un lenguaje de programación*. Los lenguajes de programación se pueden dividir en:

- Máquina.
- Bajo nivel o ensambladores.
- Alto nivel.

El lenguaje que puede comprender directamente la máquina es un lenguaje formado por órdenes codificadas en binario, denominado **lenguaje máquina.** Un lenguaje máquina tiene grandes inconvenientes, a saber, depende de la máquina sobre la que se va a ejecutar, son poco legibles, el repertorio de instrucciones suele ser muy reducido y además, las instrucciones y los datos se expresan mediante sucesiones de ceros y unos.

Para solucionar el problema de la dificultad de aprendizaje, escritura y corrección de los lenguajes máquinas se desarrollaron los **lenguajes ensambladores.** En éstos las instrucciones se representan utilizando, en lugar de códigos binarios, una notación nemotécnica para representar los códigos de operación. Las direcciones de memoria de los datos son simbólicas y no absolutas, es decir, a los datos se accede por su nombre y no por su dirección.

Es necesaria la traducción de un programa en lenguaje ensamblador a un programa en lenguaje máquina. Para ello se desarrollaron los **programas ensambladores**.

Aunque los lenguajes ensambladores supusieron un gran avance en la comprensión de los programas de ordenador, todavía resultaban demasiado lejanos para el hombre. Por ello se desarrollaron los **lenguajes de alto nivel** que proporcionan un repertorio de instrucciones amplio, potente y fácilmente asimilable. Pero se plantea el mismo problema que con el lenguaje ensamblador:

es necesario traducir o transformar estos programas escritos en un lenguaje de alto nivel a programas en lenguaje máquina. Para esta tarea se han desarrollado los **programas traductores**.

Un **traductor** es un programa que realiza la conversión de un programa escrito en un lenguaje de alto nivel a un programa en lenguaje máquina. El proceso de conversión se denomina **traducción.** Esta traducción puede realizarse de dos formas muy diferentes:

- por *interpretación*.
- por *compilación*.

Al programa escrito en el lenguaje de alto nivel se le denomina programa fuente.

Un **intérprete** es un programa que toma el programa fuente y lo trata instrucción a instrucción. Así, para cada instrucción realiza una comprobación léxica para comprobar si la sintaxis es correcta. Si es correcta se traduce la instrucción a código máquina y se ejecuta dicha instrucción.

Un **compilador** es un programa que toma un programa fuente y genera un programa en lenguaje máquina, llamado *programa objeto*. El compilador analiza las instrucciones del programa y comprueba la sintaxis y la semántica de todas ellas. Si el programa es correcto, entonces se produce la traducción a lenguaje máquina, generando un programa completo.

Tema 2: Introducción a la Programación Orientada a Objetos.

2.1.- Introducción.

La POO (Programación Orientada a Objetos) es un nuevo paradigma de la programación. Esto significa que es una nueva forma de organizar el conocimiento y de pensar sobre lo que significa programar y sobre cómo se estructura la información dentro del ordenador.

Tendremos entidades a las que llamaremos **objetos**. Los objetos se comunican con otros objetos mediante **mensajes**. Un mensaje le dice a un objeto qué acción debe iniciar, es decir, qué **método** debe ejecutar. Los objetos tienen unas características y un comportamiento y entenderán unos mensajes determinados. Agruparemos en lo que llamaremos **clases** a todos aquellos objetos con características y comportamiento similares.

Podremos hacer especializaciones de tal forma que tendremos una jerarquía de clases y las clases de niveles más bajos podrán usar datos y comportamiento de las clases más altas. Es lo que llamaremos **herencia**.

Podremos crear cuantos objetos y clases deseemos, asignándoles unas características y un comportamiento. No tendremos que conocer con detalle los pasos o el método que sigue un objeto para dar respuesta a un mensaje. Hablamos entonces de **abstracción de datos** y de **encapsulación y ocultamiento de la información**.

2.2.- Conceptos orientados a objetos.

Los principales conceptos relacionados con la Programación Orientada a Objetos son:

- abstracción de datos
- objetos
- clases
- atributos
- mensajes y métodos
- encapsulación
- polimorfismo o sobrecarga de funciones y operadores
- herencia

2.2.1.- Abstracción de datos.

La abstracción consiste en particionar un área de conocimiento y centrar nuestra atención en algo concreto.

En los lenguajes de programación se usan dos *mecanismos para la abstracción*:

a) **Abstracción por parametrización:** se generaliza un procedimiento que se aplica muchas veces a objetos diferentes y se representa en un solo procedimiento al que se le pasan unos parámetros.

b) **Abstracción por especificación:** consiste en considerar sólo el comportamiento que debe tener un procedimiento. A una especificación le podemos asociar infinitas implementaciones a las que supondremos un comportamiento similar.

Tenemos dos *tipos de abstracción* según el elemento de programa al que afecten:

a) **Procedimental:** se considera que un procedimiento se comporta como una operación simple.

b) **De datos:** es un nivel más alto y se considera como una unidad a una serie de datos junto con una serie de operaciones que permiten manejarlos. Se habla de **tipo abstracto de datos** (TAD). Los TAD se pueden instanciar.

Con estos dos tipos de abstracción se usa simultáneamente parametrización y especificación.

En P.O.O. se utilizan TAD con abstracción por especificación. La idea es poder cambiar la implementación de un TAD sin tener que cambiar nada en los programas u otros TDA que lo usen.

2.2.2.-Objeto.

Se corresponde con una entidad del mundo real. Tiene dos partes: una estructura de datos y unas operaciones que manejan esa estructura, llamadas *métodos*. Un objeto contiene atributos, a los que llamaremos *variables de instancia*. El objeto tiene una parte visible y una parte oculta. En la parte visible, llamada *interfaz* o protocolo del objeto, sólo vemos los nombres de los *mensajes* que el objeto entiende. En la parte oculta están la implementación de esos métodos y los atributos.

2.2.3.-Atributos.

Dos objetos tendrán, en principio, distinto valor en sus atributos. No se accede a los atributos directamente, sino con los métodos del protocolo del objeto. Un atributo puede tomar un valor o un conjunto de valores. Los atributos son también objetos.

Un objeto tiene una identidad: aunque cambien los valores de sus atributos, el objeto sigue siendo siempre el mismo.

2.2.4.- Mensajes y métodos.

Un objeto se comunica con otros objetos mediante mensajes. Un mensaje implica la realización de una acción, es decir, la ejecución de un método. Los métodos operan sobre los valores de un atributo y determinan el comportamiento de un objeto. Un objeto puede entender varios mensajes.

2.2.5.- Encapsulación.

Es el mecanismo que agrupa el código y los datos que maneja y los mantiene protegidos frente a cualquier interferencia y mal uso. Un objeto es por tanto, el dispositivo que soporta encapsulación. Un objeto oculta parte de su información a otros objetos. Sólo se puede acceder a un objeto a través de los mensajes de su propia interfaz. Los atributos y métodos están integrados en un objeto de tal forma que un objeto sólo puede modificar sus propios atributos con sus propios métodos. Por esto, los atributos y los métodos están almacenados físicamente en un mismo módulo de programación, es decir, encapsulados.

2.2.6.- Clases.

Todos los objetos con los mismos atributos y que entienden los mismos mensajes se agrupan en una clase. Las instancias o ejemplares de una clase son los objetos.

Una clase es también un objeto, y por tanto debe ser instancia, a su vez, de una clase a la que llamaremos *metaclase*. También tiene variables y métodos que se llaman variables de clase y métodos de clase. Las variables de clase son compartidas por todos los objetos de una clase. Entre los métodos de clase destacan los de creación de instancias de una clase.

2.2.7.- Herencia de clases.

Se establece una jerarquía de clases en la que una clase hereda de una clase superior sus atributos y sus métodos. A la clase superior se le llama clase *base* y a la clase que hereda se le llama clase *derivada*. Es una relación de especialización, llamada relación ES-UN. Significa esto que los objetos de una clase entenderán los mismos mensajes y tendrán los mismos atributos que los objetos de su clase base (*superclase*).

Además, a una clase puede añadirse nuevos métodos y nuevos atributos, o anular los métodos de su clase base, o redefinir dichos métodos.

La relación de herencia es ascendente: una clase hereda de su clase base que a su vez también hereda de su clase base y así sucesivamente.

Para visualizar la relación de herencia utilizaremos grafos o árboles de herencia.

C++ tiene herencia múltiple, esto quiere decir que una clase puede heredar atributos y métodos de más de una clase base. Java sin embargo, tiene herencia simple.

2.2.8.- Polimorfismo.

Un mismo mensaje puede ser válido para más de una clase porque su implementación puede ser distinta. En realidad podremos tener varios métodos para un mismo mensaje. Cuando ocurre esto decimos que hay *polimorfismo* o sobrecarga de funciones. Lo mismo ocurre con el nombre de los

atributos. El polimorfismo se puede aplicar tanto a funciones como a operadores, en el segundo caso hablaremos de sobrecarga de operadores.

Todos los lenguajes de POO, incluyendo C++, comparten las tres características siguientes: encapsulación, polimorfismo y herencia.

Ejemplo:

```
#include <iostream.h>
class Rectángulo{
    int ancho, largo;
public:
    void intro(int a, int b);
    int área ();
    int perímetro ();
};

void Rectángulo :: intro(int a, int b) {
    ancho=a;largo=b;
}

int Rectángulo :: área () {
    return (ancho * largo);
}

int Rectángulo :: perímetro () {
    return (2 * (ancho + largo));
}

int main (void)
{
    Rectángulo rect;
    rect.intro(5,4);
    cout << rect.área ();
    cout << rect.perímetro ();
    return 0;
}
```

En un objeto los atributos y los métodos están almacenados físicamente en un mismo módulo de programación, es decir, encapsulados.

En un objeto los atributos y los métodos, o ambos, pueden ser privados para ese objeto o públicos. Los atributos y/o los métodos privados sólo los conoce y son accesibles por otra parte del objeto. Cuando los métodos y/o los atributos son públicos, son accesibles desde cualquier punto del programa.

Para todos los propósitos un objeto es una variable de un tipo definido por el usuario. Cada vez que se define un nuevo objeto se está creando un nuevo tipo de dato. Cada instancia específica de este tipo de datos es una variable compuesta.

UNIDAD DIDACTICA 2ª. Iniciación en JAVA.

Tema 3: Introducción a JAVA.

3.1.- Introducción.

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Desarrollan un código "neutro" que no depende del tipo de electrodoméstico, el cual se ejecuta sobre una "*máquina hipotética o virtual*" denominada ***Java Virtual Machine*** (***JVM***). Es la ***JVM*** quien interpreta el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "*Write Once, Run Everywhere*".

A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje. ***Java***, como lenguaje de programación para computadores, se introdujo a finales de 1995. La clave fue la incorporación de un intérprete ***Java*** en el programa Netscape Navigator, versión 2.0 , produciendo una verdadera revolución en Internet. ***Java 1.1*** apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje.

Al programar en ***Java*** no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de clases preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el ***API*** o ***Application Programming Interface*** de ***Java***). ***Java*** incorpora muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso es un lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje ***Java*** es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor de ***Web***, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo (se está convirtiendo en un macro-lenguaje: ***Java 1.0*** tenía 12 packages; ***Java 1.1*** tenía 23 y ***Java 1.2*** tiene 59). En cierta forma casi todo depende de casi todo. Por ello, hay que aprenderlo de modo iterativo: primero una visión muy general, que se va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

La compañía ***Sun*** describe el lenguaje ***Java*** como "*simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y*

dinámico". Además de una serie de halagos por parte de *Sun* hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje ***Java***. Algunas de las anteriores ideas se irán explicando a lo largo de este manual.

3.2.- Qué es JAVA 2.

Java 2 (antes llamado ***Java 1.2*** o ***JDK 1.2***) es la tercera versión importante del lenguaje de programación ***Java***. No hay cambios conceptuales importantes respecto a ***Java 1.1*** (en ***Java 1.1*** sí los hubo respecto a ***Java 1.0***), sino extensiones y ampliaciones, lo cual hace que a muchos efectos sea casi lo mismo trabajar con ***Java 1.1*** o con ***Java 1.2***.

Los programas desarrollados en ***Java*** presentan diversas ventajas frente a los desarrollados en otros lenguajes como C/C++. La ejecución de programas en ***Java*** tiene muchas posibilidades: ejecución como aplicación independiente (***Stand-alone Application***), ejecución como ***applet***, ejecución como ***servlet***, etc.. Un ***applet*** es una aplicación especial que se ejecuta dentro de un navegador o browser (por ejemplo *Netscape Navigator* o *Internet Explorer*) al cargar una página HTML desde un servidor ***Web***. El ***applet*** se descarga desde el servidor y no requiere instalación en el ordenador donde se encuentra el browser. Un ***servlet*** es una aplicación sin interface gráfica que se ejecuta en un servidor de Internet. La ejecución como aplicación independiente es análoga a los programas desarrollados con otros lenguajes.

Además de incorporar la ejecución como ***Applet***, ***Java*** permite fácilmente el desarrollo tanto de arquitecturas cliente-servidor como de aplicaciones distribuidas, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, ***Java*** incorpora en su propio ***API*** estas funcionalidades.

3.3.- Características de JAVA.

Las principales características de JAVA son: es simple, Orientado a Objetos, distribuido, robusto, seguro, portable, interpretado, multihebras, dinámico y de arquitectura neutral.

- *Simple*. Basado en C++ elimina: la aritmética de punteros, las referencias, las estructuras y uniones, las definiciones de tipos y macros y la necesidad de liberar memoria.

- *Orientado a Objetos*. Incluye:

    - Encapsulación, herencia y polimorfismo.
    - Interfaces para suplir la carencia de herencia múltiple.
    - Resolución dinámica de métodos.

- Distribuido.

    - Extensas capacidades de comunicaciones.
    - Permite actuar con http y ftp.
    - El lenguaje no es distribuido, pero incorpora facilidades para construir programas distribuidos.
    - Se están incorporando características para hecerlo distribuido.

- *Robusto*. Realiza continuos chequeos en tiempo de compilación y en tiempo de ejecución.

- *Seguro*.

    - No hay punteros
    - El cast hacia lo general es ímplicito.
    - Los bytecodes pasan varios test antes de ser ejecutados.

- *Portable*.

    - Mismo código en distintas arquitecturas.
    - Define la longitud de sus tipos independientemente de la plataforma.
    - Construye sus interfaces en base a un sistema abstracto de ventanas.

- *Interpretado*.

    - Para conseguir la independencia del S.O. genera bytecodes.
    - El intérprete toma cada bytecode y lo interpreta.
    - El mismo intérprete corre en distintas arquitecturas.

- *Multiebras*.

    - Permite crear tareas o hebras.
    - Sincronización de métodos.
    - Comunicación de tareas.

- *Dinámico*.

    - Conecta los módulos que intervienen en una aplicación en el momento de su ejecución
    - No hay necesidad de enlazar previamente.

3.4.- Características eliminadas de C++.

- No hay *typedef*, *defines* ni preprocesamiento.

- No hay estructuras ni uniones ni enumeraciones.

- No hay funciones (sólo métodos en clases).

- No hay herencia múltiple.

- No hay goto.

- No hay sobrecarga de operadores.

- No hay conversión automática.
- No hay punteros.
- No hay templates.
- No hay que destruir los objetos inservibles.

3.5.- El entorno de desarrollo de JAVA. Existen distintos programas comerciales que permiten desarrollar código ***Java***. La compañía ***Sun***, creadora de ***Java***, distribuye gratuitamente el *Java(tm) Development Kit* (***JDK***). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en ***Java***. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (es el denominado ***Debugger***). Cualquier programador con un mínimo de experiencia sabe que una parte muy importante (muchas veces la mayor parte) del tiempo destinado a la elaboración de un programa se destina a la detección y corrección de errores. Existe también una versión reducida del ***JDK***, denominada ***JRE*** (*Java Runtime Environment*) destinada únicamente a ejecutar código ***Java*** (no permite compilar).

Los ***IDE***s (*Integrated Development Environment*), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código ***Java***, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar ***Debug*** gráficamente, frente a la versión que incorpora el ***JDK*** basada en la utilización de una consola (denominada habitualmente ventana de comandos de MS-DOS, en ***Windows NT/95/98***) bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con ***componentes*** ya desarrollados, los cuales se incorporan al proyecto o programa. Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas y ficheros resultantes de mayor tamaño que los basados en clases estándar.

3.5.1.- El compilador de Java. Se trata de una de las herramientas de desarrollo incluidas en el ***JDK***. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de ***Java*** (con extensión ****.java***). Si no encuentra errores en el código genera los ficheros compilados (con extensión ****.class***). En otro caso muestra la línea o líneas erróneas. En el ***JDK*** de ***Sun*** dicho compilador se llama ***javac.exe***. Tiene numerosas opciones, algunas de las cuales varían de una versión a otra. Se aconseja consultar la documentación de la versión del ***JDK*** utilizada para obtener una información detallada de las distintas posibilidades.

3.5.2.- La Java Virtual Machine. Tal y como se ha comentado al comienzo del capítulo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de ***Sun*** a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se plantea la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código "neutro" el cual estuviera preparado para ser ejecutado sobre una "*máquina hipotética o virtual*", denominada ***Java Virtual***

Machine (***JVM***). Es esta ***JVM*** quien ***interpreta*** este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La ***JVM*** es el intérprete de ***Java***. Ejecuta los "***bytecodes***" (ficheros compilados con extensión ****.class***) creados por el compilador de ***Java*** (***javac.exe***). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado ***JIT*** (*Just-In-Time Compiler*), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

3.5.3.- Las variables PATH y CLASSPATH. El desarrollo y ejecución de aplicaciones en ***Java*** exige que las herramientas para compilar (***javac.exe***) y ejecutar (***java.exe***) se encuentren accesibles. El ordenador, desde una ventana de comandos de MS-DOS, sólo es capaz de ejecutar los programas que se encuentran en los directorios indicados en la variable ***PATH*** del ordenador. Si se desea compilar o ejecutar código en ***Java*** en estos casos el directorio donde se encuentran estos programas (***java.exe*** y ***javac.exe***) deberán encontrarse en el ***PATH***. Tecleando ***set PATH*** en una ventana de comandos de MS-DOS se muestran los nombres de directorios incluidos en dicha variable de entorno.

Java utiliza además una nueva variable de entorno denominada ***CLASSPATH***, la cual determina dónde buscar tanto las clases o librerías de ***Java*** (el ***API*** de ***Java***) como otras clases de usuario. A partir de la versión 1.1.4 del JDK no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho ***JDK***. La variable ***CLASSPATH*** puede incluir la ruta de directorios o ficheros ****.zip*** o ****.jar*** en los que se encuentren los ficheros ****.class***. En el caso de los ficheros ****.zip*** hay que indicar que los ficheros en él incluidos no deben estar comprimidos. En el caso de archivos ****.jar*** existe una herramienta (***jar.exe***), incorporada en el ***JDK***, que permite generar estos ficheros a partir de los archivos compilados ****.class***. Los ficheros ****.jar*** son archivos comprimidos y por lo tanto ocupan menos espacio que los archivos ****.class*** por separado o que el fichero ****.zip*** equivalente.

Una forma general de indicar estas dos variables es crear un fichero ***batch*** de MS-DOS (****.bat***) donde se indiquen los valores de dichas variables. Cada vez que se abra una ventana de MS-DOS será necesario ejecutar este fichero ****.bat*** para asignar adecuadamente estos valores. Un posible fichero llamado ***jdk117.bat***, podría ser como sigue:

```
set JAVAPATH=C:\jdk1.1.7
set PATH=.;%JAVAPATH%\bin;%PATH%
set CLASSPATH=.\;%JAVAPATH%\lib\classes.zip;%CLASSPATH%
```

lo cual sería válido en el caso de que el ***JDK*** estuviera situado en el directorio ***C:\jdk1.1.7***. Si no se desea tener que ejecutar este fichero cada vez que se abre una consola de MS-DOS es necesario indicar estos cambios de forma "permanente". La forma de hacerlo difiere entre Windows 95/98 y Windows NT. En Windows 95/98 es necesario modificar el fichero ***Autoexec.bat*** situado en C:\, añadiendo las líneas antes mencionadas. Una vez rearrancado el ordenador estarán presentes en cualquier consola de MS-DOS que se cree. La modificación al fichero ***Autoexec.bat*** en ***Windows 95/98*** será la siguiente:

```
set JAVAPATH=C:\jdk1.1.7
set PATH=.;%JAVAPATH%\bin;%PATH%
set CLASSPATH=
```

En el caso de utilizar ***Windows NT*** se añadirá la variable ***PATH*** en el cuadro de diálogo que se abre con *Start -> Settings -> Control Panel -> System -> Environment -> User Variables for NombreUsuario*:

También es posible utilizar la opción ***–classpath*** en el momento de llamar al compilador ***javac.exe*** o al intérprete ***java.exe***. Los ficheros ****.jar*** deben ponerse con el nombre completo en el ***CLASSPATH***: no basta poner el ***PATH*** o directorio en el que se encuentra. Por ejemplo, si se desea compilar y ejecutar el fichero ***ContieneMain.java***, y éste necesitara la librería de clases ***G:\MyProject\OtherClasses.jar***, además de las incluidas en el ***CLASSPATH***, la forma de compilar y ejecutar sería:

javac -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain.java
java -classpath .\;G:\MyProject\OtherClasses.jar ContieneMain

Se aconseja consultar la ayuda correspondiente a la versión que se esté utilizando, debido a que existen pequeñas variaciones entre las distintas versiones del JDK.

Cuando un fichero ***filename.java*** se compila y en ese directorio existe ya un fichero ***filename.class***, se comparan las fechas de los dos ficheros. Si el fichero ***filename.java*** es más antiguo que el ***filename.class*** no se produce un nuevo fichero ***filename.class***. Esto sólo es válido para ficheros ****.class*** que se corresponden con una clase ***public***.

3.6.- Nomenclatura habitual en la programación en JAVA.

Los nombres de ***Java*** son sensibles a las letras mayúsculas y minúsculas. Así, las variables ***masa***, ***Masa*** y ***MASA*** son consideradas variables completamente diferentes. Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas de ordenador. Se recomienda seguir las siguientes instrucciones:

1. En ***Java*** es habitual utilizar nombres con minúsculas, con las excepciones que se indican en los puntos siguientes.

2. Cuando un nombre consta de varias palabras es habitual poner una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue a otra (Ejemplos:*elMayor()*, v*entanaCerrable*, r*ectanguloGrafico*, *addWindowListener()*).

3. Los nombres de ***clases*** e ***interfaces*** comienzan siempre por mayúscula (Ejemplos: *Geometria*, *Rectangulo*, *Dibujable*, *Graphics*, *Vector*, *Enumeration*).

4. Los nombres de ***objetos***, los nombres de ***métodos*** y ***variables miembro***, y los nombres de las ***variables locales*** de los métodos, comienzan siempre por minúscula (Ejemplos: *main()*, *dibujar()*, *numRectangulos*, *x*, *y*, *r*).

5. Los nombres de las ***variables finales***, es decir de las constantes, se definen siempre con mayúsculas (Ejemplo: *PI*)

3.7.- Estructura general de un programa JAVA. La estructura habitual de un programa realizado en cualquier lenguaje ***orientado a objetos*** u ***OOP*** (*Object Oriented Programming*), y en particular en el lenguaje ***Java*** es la siguiente: aparece una clase que contiene el programa principal (aquel que contiene la función ***main()***) y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal. Los ficheros fuente tienen la extensión ****.java***, mientras que los ficheros compilados tienen la extensión ****.class***.

Un fichero fuente (****.java***) puede contener más de una clase, pero sólo una puede ser ***public***. El nombre del fichero fuente debe coincidir con el de la clase ***public*** (con la extensión ****.java***). Si por ejemplo en un fichero aparece la declaración (***public class MiClase {...}***) entonces el nombre del fichero deberá ser ***MiClase.java***. Es importante que coincidan mayúsculas y minúsculas ya que ***MiClase.java*** y ***miclase.java*** serían clases diferentes para ***Java***. Si la clase no es ***public***, no es necesario que su nombre coincida con el del fichero. Una clase puede ser ***public*** o ***package*** (default), pero no ***private*** o ***protected***. Estos conceptos se explican posteriormente.

De ordinario una aplicación está constituida por varios ficheros ****.class***. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función ***main()*** (sin la extensión ****.class***). Las clases de ***Java*** se agrupan en ***packages***, que son librerías de clases. Si las clases no se definen como pertenecientes a un ***package***, se utiliza un ***package*** por defecto (*default*) que es el directorio activo. Los ***packages*** se estudian con más detenimiento en siguientes apartados.

Ejemplo:

```
// fichero EjCadena.java
public class EjCadena {
  public static void main(String args[]) {
     Cadena cad=new Cadena(args[0]);
     cad.invierteCadena();
     cad.visualizaCadena();
     cad.encriptaCadena();
     cad.visualizaCadena();
     cad.desencriptaCadena();
     cad.visualizaCadena();
  }
}

// fichero Cadena.java
public class Cadena {
  private String cadena="";
  public Cadena(String cadena) {
     this.cadena=cadena;
  }
  public int ordinal(char c) {
     return ((int) c);
  }
```

```
  public char ascii(int i) {
     return ((char) i);
  }
  public void invierteCadena() {
     String cadena2="";
     for(int i=cadena.length()-1;i>=0;i--)
       cadena2=cadena2+cadena.charAt(i);
     cadena=cadena2;
  }
  public void encriptaCadena() {
     String cadena2="";
     char c;
     for(int i=0;i<cadena.length();i++) {
       c=cadena.charAt(i);
       cadena2=cadena2+ascii(ordinal(c)+3);
     }
     cadena=cadena2;
  }
  public void desencriptaCadena() {
     String cadena2="";
     char c;
     for(int i=0;i<cadena.length();i++) {
       c=cadena.charAt(i);
       cadena2=cadena2+ascii(ordinal(c)-3);
     }
     cadena=cadena2;
  }
  public void visualizaCadena() {
     System.out.println(cadena);
  }
}
```

Tema 4: Programación en JAVA.

4.1.- Variables. Una ***variable*** es un ***nombre*** que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en ***Java*** hay dos tipos principales de variables:

1. Variables de ***tipos primitivos***. Están definidas mediante un valor único.

2. Variables ***referencia***. Las variables referencia son referencias o nombres de una información más compleja: ***arrays*** u ***objetos*** de una determinada clase.

Desde el punto de vista de su papel en el programa, las variables pueden ser:

1. Variables ***miembro*** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser *tipos primitivos* o *referencias*.

2. Variables ***locales***: Se definen dentro de un método o más en general *dentro de cualquier bloque* entre llaves {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también *tipos primitivos* o *referencias*.

4.1.1.- Nombres de Variables. Los nombres de variables en ***Java*** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por ***Java*** como operadores o separadores (,.+-*/ etc.).

Existe una serie de ***palabras reservadas*** las cuales tienen un significado especial para ***Java*** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

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

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje ***Java***.

4.1.2.- Tipos Primitivos de Variables. Se llaman ***tipos primitivos*** de variables de ***Java*** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolean*, *caracteres* y *valores numéricos* enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores ***true*** y ***false*** (***boolean***); un tipo para almacenar caracteres (***char***), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (***byte***, ***short***, ***int*** y ***long***) y dos para valores reales de punto flotante (***float*** y ***double***). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la siguiente tabla:

Tipo de variable	Descripción
boolean	1 byte. Valores true y false
char	2 bytes. Unicode. Comprende el código ASCII
byte	1 byte. Valor entero entre -128 y 127
short	2 bytes. Valor entero entre -32768 y 32767
int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
long	8 bytes. Valor entre –9.223.372.036.854.775.808 7 9.223.372.036.854.775.807
float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Doublé	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Los tipos primitivos de ***Java*** tienen algunas características importantes que se resumen a continuación:

1. El tipo ***boolean*** no es un valor numérico: sólo admite los valores ***true*** o ***false***. El tipo ***boolean*** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser ***boolean***.

2. El tipo ***char*** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.

3. Los tipos ***byte***, ***short***, ***int*** y ***long*** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en ***Java*** no hay enteros ***unsigned***.

4. Los tipos ***float*** y ***double*** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.

5. Se utiliza la palabra ***void*** para indicar la ausencia de un tipo de variable determinado.

6. A diferencia de C/C++, los tipos de variables en ***Java*** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un ***int*** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.

7. Extensiones de ***Java 1.2*** para aprovechar la arquitectura de los procesadores ***Intel***, que permiten realizar operaciones con una precisión extendida de 80 bits.

4.1.3.- Cómo se definen e inicializan las variables. Una variable se define especificando el tipo y el nombre de la variable. Estas variables pueden ser tanto de tipos ***primitivos*** como ***referencias*** a objetos de alguna clase perteneciente al ***API*** de ***Java*** o generada por el usuario. Las variable ***primitivas*** se inicializan a cero (salvo *boolean* y *char*, que se inicializan a ***false*** y '\0') si no se especifica un valor en su declaración. Análogamente las variables de tipo ***referencia*** son inicializadas por defecto a un valor especial: ***null***.

Es importante distinguir entre la ***referencia*** a un objeto y el ***objeto*** mismo. Una ***referencia*** es una variable que indica dónde está en la memoria del ordenador un objeto. Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración) luego se le asigna el valor ***null***. Si se desea que esta ***referencia*** apunte a un nuevo objeto es necesario utilizar el operador ***new***. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la ***referencia*** declarada a un objeto existente previamente.

Un tipo particular de referencias son los ***arrays*** o vectores, sean estos de variables primitivas (por ejemplo vector de enteros) o de objetos. En la declaración de una referencia de tipo ***array*** hay que incluir los ***corchetes*** []. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos ***MyClass***. A su vez se garantiza que los elementos del vector son inicializados a ***null*** o a cero (según el tipo de dato) en caso de no indicar otro valor.

Ejemplos de declaración e inicialización de variables:

```
int x;                          // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;                      // Declaración de la variable primitiva y. Se inicializa a 5
MyClass unaRef;                 // Declaración de una referencia a un objeto MyClass.
                                // Se inicializa a null
unaRef = new MyClass();         // La referencia "apunta" al nuevo objeto creado
                                // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;    //Declaración de una referencia a un objeto // Se
                                inicializa al mismo valor que unaRef
int [] vector;                  // Declaración de un array. Se inicializa a null
vector = new int[10];           // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1}; // Declaración e inicialización de un vector de 3
                                // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5];// Se crea un vector de 5 referencias a objetos
                                // Las 5 referencias son inicializadas a null
lista[0] = unaRef;              // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass();       // Se asigna a lista[1] la referencia al nuevo objeto
                                // El resto (lista[2]…lista[4] siguen con valor null
```

En el ejemplo mostrado las referencias ***unaRef***, ***segundaRef*** y ***lista[0]*** actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

4.1.4.- Visibilidad y vida de las variables.

Se entiende por ***visibilidad***, ***ámbito*** o ***scope*** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en cualquier expresión. En ***Java*** todos las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves **{}**, es decir dentro de un ***bloque***, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque ***if*** no serán válidas al finalizar las sentencias correspondientes a dicho ***if*** y las variables miembro de una ***clase*** (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como ***public*** son accesibles a través de una ***referencia*** a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como ***private*** no son accesibles directamente desde otras clases. Las ***funciones miembro*** de una clase tienen acceso directo a ***todas*** las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase ***B*** derivada de otra ***A***, tienen acceso a todas las variables miembro de ***A*** declaradas como ***public*** o ***protected***, pero no a las declaradas como ***private***. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como ***public*** o ***protected***. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador ***this***.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. La forma de crear nuevos ***objetos*** es utilizar el operador ***new***. Cuando se utiliza el operador ***new***, la variable de tipo ***referencia*** guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo ***referencia*** es apuntado. La eliminación de los objetos la realiza el denominado ***garbage collector***, quien automáticamente libera o borra la memoria ocupada por un ***objeto*** cuando no existe ninguna ***referencia*** apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

4.2.- Operadores de JAVA.

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

4.2.1.- Operadores aritméticos.

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: ***suma*** (+), ***resta*** (-), ***multiplicación*** (*), ***división*** (/) y ***resto de la división*** (%).

4.2.2.- Operadores de asignación.

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el ***operador igual*** (=). La forma general de las sentencias de asignación con este operador es:

variable = expression;

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La siguiente tabla muestra estos operadores y su equivalencia con el uso del ***operador igual*** (=).

Operador	Utilización	Expresión equivalente
+=	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

4.2.3.- Operadores unarios. Los operadores ***más*** (+) y ***menos*** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en ***Java*** es el estándar de estos operadores.

4.2.4.- Operador instanceof. El operador ***instanceof*** permite saber si un objeto pertenece a una determinada clase o no. Es un operador binario cuya forma general es,

objectName instanceof ClassName

y que devuelve ***true*** o ***false*** según el objeto pertenezca o no a la clase.

4.2.5.- Operador condicional ?. Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

booleanExpression ? res1 : res2

donde se evalúa ***booleanExpression*** y se devuelve ***res1*** si el resultado es ***true*** y ***res2*** si el resultado es ***false***. Es el único operador ternario (tres argumentos) de ***Java***. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

x=1 ; y=10; z = (x<y)?x+3:y+8;

asignarían a ***z*** el valor 4, es decir x+3.

4.2.6.- Operadores incrementales. ***Java*** dispone del operador ***incremento*** (++) y ***decremento*** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. *Precediendo a la variable* (por ejemplo: ***++i***). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.

2. *Siguiendo a la variable* (por ejemplo: ***i++***). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles ***for*** es una de las aplicaciones más frecuentes de estos operadores.

4.2.7.- Operadores relacionales. Los ***operadores relacionales*** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor ***boolean*** (***true*** o ***false***) según se cumpla o no la relación considerada. La siguiente tabla muestra los operadores relacionales de ***Java***.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Estos operadores se utilizan con mucha frecuencia en las ***bifurcaciones*** y en los ***bucles***, que se verán en próximos apartados de este capítulo.

4.2.8.- Operadores lógicos. Los operadores lógicos se utilizan para construir ***expresiones lógicas***, combinando valores lógicos (***true*** y/o ***false***) o los resultados de los operadores ***relacionales***. La siguiente tabla muestra los operadores lógicos de ***Java***. Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser ***true*** y el primero es ***false*** ya se sabe que la condición de que ambos sean ***true*** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	op1 && op2	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
||	OR	op1 || op2	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	NOT	¡ op	true si op es false y false si es true
&	AND	op1 & op2	true si op1 y op2 son true. Siempre se evalua op2
|	OR	op1 | op2	true si op1 u op2 son true. Siempre se evalúa op2

4.2.9.- Operador de concatenación de cadenas de caracteres (+). El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método ***println()***. La variable numérica ***result*** es convertida automáticamente por ***Java*** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

4.2.10.- Operadores que actúan a nivel de bits. ***Java*** dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o ***flags***, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 2.5 muestra los operadores de ***Java*** que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	op1 >> op2	Desplaza los bits de op1 a la derecha una distancia op2
<<	op1 << op2	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	op1 >>> op2	Desplaza los bits de op1 a la derecha una distancia op2(positiva)
&	op1 & op2	Operador AND a nivel de bits
|	op1 | op2	Operador OR a nivel de bits
^	op1 ^ op2	Operador XOR a nivel de bits
~	op1 ~ op2	Operador complemento

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000), utilizando sólo 8 bits. La suma de estos números permite construir una variable ***flags*** con los bits activados que se deseen. Por ejemplo, para construir una variable ***flags*** que sea 00010010 bastaría hacer ***flags***=2+16. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

if (flags & 2 == 2) {...}

4.2.11.- Precedencia de operadores. El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de *x/y*z* depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de ***mayor a menor*** precedencia:

operadores postfijos	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creacion o cast	new (type)expr
multiplicación / división	* / %
suma / resta	+ -
shift	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
lógico AND	&&
lógico OR	||
condicional	? :
asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=

En ***Java***, todos los operadores binarios, excepto los operadores de asignación, se evalúan de ***izquierda a derecha***. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la izquierda se copia sobre la variable de la derecha.

4.3.- Estructuras de programación. En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto no se explican en profundidad los conceptos que aparecen.

Las ***estructuras de programación*** o ***estructuras de control*** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados ***bifurcaciones*** y ***bucles***. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a *concepto*, aunque su *sintaxis* varía de un lenguaje a otro. La sintaxis de ***Java*** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

4.3.1.- Sentencias o expresiones. Una ***expresión*** es un conjunto variables unidos por ***operadores***. Son órdenes que se le dan al ordenador para que realice una tarea determinada.

Una ***sentencia*** es una ***expresión*** que acaba en ***punto y coma*** (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j;// Línea compuesta de tres sentencias
```

4.3.2.- Comentarios. Existen dos formas diferentes de introducir comentarios entre el código de ***Java*** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremendamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras "**//**" en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos **/*…*/**. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta línea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una línea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada línea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado de líneas ya que
sólo requiere modificar el comienzo y el final. */
```

En ***Java*** existe además una forma especial de introducir los comentarios (utilizando /**…*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las ***clases*** y ***packages*** desarrollados por el programador. Una vez introducidos los comentarios, el programa ***javadoc.exe*** (incluido en el ***JDK***) genera de forma automática la información de forma similar a la presentada en la propia documentación del ***JDK***. La sintaxis de estos comentarios y la forma de utilizar el programa ***javadoc.exe*** se puede encontrar en la información que viene con el ***JDK***.

4.3.3.- Bifurcaciones. Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el *flujo de ejecución* de un programa. Existen dos bifurcaciones diferentes: ***if*** y ***switch***.

4.3.3.1.- Bifurcación if. Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor ***true***). Tiene la forma siguiente:

```
if (booleanExpression) {
   instrucciones;
}
```

Las ***llaves*** {} sirven para agrupar en un ***bloque*** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del ***if***.

4.3.3.2.- Bifurcación if else. Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el ***else*** se ejecutan en el caso de no cumplirse la expresión de comparación (***false***),

```
if (booleanExpression) {
   instrucciones1;
} else {
  instrucciones2;
}
```

4.3.3.3.- Sentencia switch. Se trata de una alternativa a la bifurcación ***if elseif else*** cuando se compara la ***misma expresión*** con distintos valores. Su forma general es la siguiente:

```
switch (expression) {
  case <value1>: statements1; break;
  case <value2>: statements2; break;
  case <value3>: statements3; break;
  case <value4>: statements4; break;
  case <value5>: statements5; break;
  case <value6>: statements6; break;
  [default: statements7;]
}
```

Las características más relevantes de ***switch*** son las siguientes:

1. Cada sentencia ***case*** se corresponde con un único valor de ***expression***. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos.

2. Los valores no comprendidos en ninguna sentencia ***case*** se pueden gestionar en ***default***, que es opcional.

3. En ausencia de ***break***, cuando se ejecuta una sentencia ***case*** se ejecutan también todas las que van a continuación, hasta que se llega a un ***break*** o hasta que se termina el ***switch***.

Ejemplo:

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras minúsculas
System.out.println("La letra " + c );
switch (c) {
    case 'a': // Se compara con la letra a
    case 'e': // Se compara con la letra e
    case 'i': // Se compara con la letra i
    case 'o': // Se compara con la letra o
    case 'u': // Se compara con la letra u
            System.out.println(" Es una vocal "); break;
    default:
            System.out.println(" Es una consonante ");
}
```

4.3.4.- Bucles. Un ***bucle*** se utiliza para realizar un proceso repetidas veces. Se denomina también ***lazo*** o ***loop***. El código incluido entre las ***llaves*** {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (***booleanExpression***) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

4.3.4.1.-Bucle while. Las sentencias ***statements*** se ejecutan mientras ***Expression*** sea ***true***.

```
while (Expression) {
    statements;
}
```

4.3.4.2.- Bucle for. La forma general del bucle ***for*** es la siguiente:

```
for (initialization; Expression; increment) {
  statements;
}
```

que es equivalente a utilizar ***while*** en la siguiente forma,

```
initialization;
while (Epression) {
   statements;
   increment;
}
```

La sentencia o sentencias ***initialization*** se ejecuta al comienzo del ***for***, e ***increment*** después de ***statements***. La ***booleanExpression*** se evalúa al comienzo de cada iteración; el bucle termina cuando la expresión de comparación toma el valor ***false***. Cualquiera de las tres partes puede estar vacía. La ***initialization*** y el ***increment*** pueden tener varias expresiones separadas por comas.

4.3.4.3.- Bucle do while. Es similar al bucle ***while*** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los ***statements***, se evalúa la condición: si resulta ***true*** se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a ***false*** finaliza el bucle.

```
do {
    statements
} while (Expression);
```

4.3.4.4.- Sentencias break y continue. La sentencia ***break*** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

La sentencia ***continue*** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

4.3.4.5.- Sentencias break y continue con etiquetas. Las ***etiquetas*** permiten indicar un lugar donde continuar la ejecución de un programa después de un ***break*** o ***continue***. El único lugar donde se pueden incluir etiquetas es ***justo delante de un bloque*** de código entre llaves {} (*if*, *switch*, *do...while*, *while*, *for*) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (*break*) o continuar con la siguiente iteración (*continue*) de un bucle que no es el actual.

La sentencia ***break labelName*** por lo tanto finaliza el bloque que se encuentre a continuación de ***labelName***. Por ejemplo, en las sentencias,

```
bucle1:                                    // etiqueta o label
for( int i = 0, j = 0; i < 100 ; i++){
        while ( true ) {
        if( (++j) > 5) { break bucleI; }   // Finaliza ambos bucles
        else { break; }                    // Finaliza el bucle interior ( while)
        }
}
```

la expresión break bucleI; finaliza los dos bucles simultáneamente, mientras que la expresión break; sale del bucle ***while*** interior y seguiría con el bucle ***for*** en ***i***. Con los valores presentados ambos bucles finalizarán con i = 5 y j = 6 (se invita al lector a comprobarlo).

La sentencia ***continue*** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

continue bucle1;

transfiere el control al bucle ***for*** que comienza después de la etiqueta ***bucle1:*** para que realice una nueva iteración:

```
bucle1:
for (int i=0; i<n; i++) {
        bucle2:
        for (int j=0; j<m; j++) {
        ...
        if (expression) continue bucle1; then continue bucle2;
        ...
        }
}
```

4.3.4.6.- Sentencia return. Otra forma de salir de un bucle (y de un método) es utilizar la sentencia ***return***. A diferencia de ***continue*** o ***break***, la sentencia ***return*** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del return (return value;).

4.3.4.7.- Bloque try {...} catch {...} finally {...}. ***Java*** incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje ***Java***, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son ***fatales*** y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras ***excepciones***, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser ***recuperables***. En este caso el programa debe dar al usuario la oportunidad de corregir el error (dando por ejemplo un nuevo ***path*** del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase ***Throwable***, pero los que tiene que chequear un programador derivan de ***Exception*** (***java.lang.Exception*** que a su vez deriva de ***Throwable***). Existen algunos tipos de excepciones que ***Java*** obliga a tener en cuenta. Esto se hace mediante el uso de bloques ***try*** ,***catch*** y ***finally***.

El código dentro del bloque ***try*** está "vigilado": Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque ***catch*** que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques ***catch*** como se desee, cada uno de los

cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque ***finally***, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una ***Exception*** y no se desee incluir en dicho método la gestión del error (es decir los bucles ***try/catch*** correspondientes), es necesario que el método pase la ***Exception*** al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra ***throws*** seguida del nombre de la ***Exception*** concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exception***. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una ***IOException*** relacionada con la lectura ficheros y una ***MyException*** propia. El primero de ellos (*metodo1*) realiza la gestión de las excepciones y el segundo (*metodo2*) las pasa al siguiente método.

```
void metodo1() {
        ...
        try {
                ... // Código que puede lanzar las excepciones IOException y MyException
        } catch (IOException e1) {
                // Se ocupa de IOException simplemente dando aviso
                System.out.println(e1.getMessage());
        }

        catch (MyException e2) {
                // Se ocupa de MyException dando un aviso y finalizando la función
                System.out.println(e2.getMessage()); return;
        }
        [finally {
                // Sentencias que se ejecutarán en cualquier caso
                ...
        }]
        ...
} // Fin del metodo1

void metodo2() throws IOException, MyException {
        ...
        // Código que puede lanzar las excepciones IOException y MyException
        ...
} // Fin del metodo2
```

El tratamiento de excepciones se desarrollará con más profundidad en capítulos posteriores.

4.4. Ejercicios propuestos.

4.1. Realizad un programa para calcular el valor de la hipotenusa de un triángulo rectángulo conocidos los 2 catetos aplicando el teorema de Pitágoras.

4.2. Para sacar al mercado un determinado producto intervienen 4 personas, una que lo diseña y 3 que lo fabrican. Diseñad un programa que calcule cuanto cobra cada uno de ellos, sabiendo que el diseñador cobra el doble que los fabricantes y conocidos el nº de unidades vendidas y el precio de venta de cada unidad.

4.3. Diseñad un programa que a partir de las longitudes de los lados de un triángulo, calcule el área del mismo, de acuerdo con la siguiente fórmula:

$$\text{Area} = \sqrt{T*(T-S1)*(T-S2)*(T-S3)}$$

donde T = S1+S2+S3 / 2 y S1, S2, S3 son las longitudes de los lados del triángulo.

4.4. Haced un programa para que leído desde el teclado un número nos diga si dicho número es positivo o negativo.

4.5. Haced un programa para ver si un número introducido por teclado es par o impar.

4.6. Realizad un programa que lea dos valores desde teclado y diga si cualquiera de ellos divide de forma entera al otro.

4.7. Realizad un programa para deducir el mayor de tres valores introducidos por teclado.

4.8. Diseñad un programa para leer las longitudes de los lados de un triángulo y determine que tipo de triángulo es, de acuerdo a los siguientes casos: suponiendo que A es el mayor de los lados y que B y C corresponden a los otros 2 lados:

Si $A >= B + C$ No es un triángulo
Si $A^2 = B^2 + C^2$ Triángulo rectángulo
Si $A^2 > B^2 + C^2$ Triángulo obtusángulo
Si $A^2 < B^2 + C^2$ Triángulo acutángulo

4.9. Haced un programa que nos sume los números naturales, comprendidos entre dos números introducidos por teclado.

4.10. Haced un programa que nos sume los números naturales que sean pares y sean menores que un número introducido por teclado.

4.11. Haced un programa para calcular el factorial de un número natural positivo.

4.12. Haced un programa para calcular el factorial de cualquier número.

4.13. Haced un programa que imprima, sume y cuente los números pares que hay entre dos números determinados.

4.14. Haced un programa para mostrar por pantalla 100 veces de una forma alternativa: Hola, Adiós, utilizando un switch.

4.15. Haced un programa para mostrar por pantalla los números múltiplos de 3 que hay entre dos números determinados, de forma alternativa.

4.16. Realizad un programa para calcular el valor máximo y el mínimo de una lista de n números que se introducen por teclado.

4.17. Haced un programa para calcular el valor máximo y el valor mínimo de una lista de números que se introducen por teclado, se terminará la operación cuando se introduzca el número 0.

4.18. Haced un programa que calcule el valor de un número combinatorio a partir de dos valores, A y B, que se introducen por teclado, aplicando la siguiente fórmula:

$$\binom{a}{b} = \frac{a!}{b!*(a-b)!}$$

4.19. Realizad un programa que cuente los números positivos y negativos que aparezcan en una lista de números que se introducen por teclado. El proceso finalizará introduciendo el número 0.

4.20. Realizad un programa que calcule la media aritmética de una lista de números que se introducen por teclado. El proceso finalizará con la introducción del número 0.

4.21. Realizad un programa para imprimir las tablas de multiplicar del uno al diez.

4.22. Haced un program para calcular si un número es primo o no.

4.23. Haced un programa para que introducido el número de mes lo visualice en letra. Repetir el proceso cuantas veces se quiera.

4.24. Introducid la nota (real) de una asignatura por teclado, que esté comprendida entre 0 y 10 y escribir la nota en letra, atendiendo a:

0 >= Nota < 3	Muy deficiente
3 >= Nota < 5	Insuficiente
5 >= Nota < 6	Suficiente
6 >= Nota < 7	Bien
7 >= Nota < 9	Notable
9 >= Nota =< 10	Sobresaliente

4.25. Haced un programa para que nos calcule la estadística de una serie de notas (entero) introducidas por teclado. La serie finalizará con la introducción del 0, sabiendo que :

1 > Nota < 5	Deficiente
5 >= Nota < 6	Suficiente
6 >= Nota < 7	Bien
7 >= Nota < 9	Notable
9 >= Nota =< 10	Sobresaliente

4.26. Utilizando números aleatorios simulad el lanzamiento de una moneda al aire y visualizad por pantalla si ha salido cara o cruz. Repetid el proceso tantas veces como se quiera.

4.27. Simulad 100 tiradas de un dado y contar las veces que aparece el nº 6.

4.28. Simulad 100 tiradas de 2 dados y contar las veces que entre los dos suman 10.

4.29. Generad aleatoriamente una quiniela de una columna, si la probabilidad de que salga 1 es del 50%, la x es del 30% y la del 2 es del 20%.

4.30. Haced un programa para adivinar un número entre 1 y 100 generado aleatoriamente por el ordenador, indicando en cada momento el intervalo en el que se encuentra el número y sabiendo que se cuenta con un máximo de 5 intentos para adivinarlo. Repetir el proceso tantas veces como se quiera.

Tema 5: Clases en JAVA.

5.1.- CONCEPTOS BÁSICOS.

5.1.1.- Concepto de Clase. Una *clase* es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos.

```
[public] class Classname {
// definición de variables y métodos
...
}
```

donde la palabra ***public*** es opcional: si no se pone, la clase sólo es visible para las demás clases del ***package***. Todos los métodos y variables deben ser definidos dentro del ***bloque*** {...} de la clase. En definitiva una *clase* es una plantilla para crear objetos que tienen los mismos atributos y responden a los mismos mensajes.

Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase. Las ***clases*** son como tipos de variables, mientras que los ***objetos*** son como variables concretas de un tipo determinado.

```
Classname unObjeto;
Classname otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de ***Java*** deben pertenecer a una clase. No hay variables y funciones globales.

2. Si una clase deriva de otra (***extends***), hereda todas sus variables y métodos.

3. ***Java*** tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

4. Una clase sólo puede heredar de una única clase (en ***Java*** no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de ***Object***. La clase ***Object*** es la base de toda la jerarquía de clases de ***Java***.

5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase ***public***. Este fichero se debe llamar como la clase ***public*** que contiene con extensión ****.java***. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.

6. Si una clase contenida en un fichero no es ***public***, no es necesario que el fichero se llame como la clase.

7. Los métodos de una clase pueden referirse de modo global al ***objeto*** de esa clase al que se aplican por medio de la referencia ***this***.

8. Las clases se pueden agrupar en ***packages***, introduciendo una línea al comienzo del fichero (***package packageName;***). Esta agrupación en ***packages*** está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

5.1.2.- Concepto de Interface. Una ***interface*** es un conjunto de declaraciones de funciones. Si una clase implementa (***implements***) una ***interface***, debe definir ***todas*** las funciones especificadas por la ***interface***. Las interfaces pueden definir también ***variables finales*** (constantes). Una ***clase*** puede implementar más de una ***interface***, representando una forma alternativa de la herencia múltiple.

En algunos aspectos las ***interfaces*** pueden utilizarse en lugar de las ***clases***. Por ejemplo, las ***interfaces*** sirven para definir ***referencias*** a cualquier objeto de las clases que implementan esa ***interface***. Este es un aspecto importante del ***polimorfismo***.

Una ***interface*** puede derivar de otra o incluso de varias ***interfaces***, en cuyo caso incorpora las declaraciones de todos los métodos de las ***interfaces*** de las que deriva.

5.2.- Ejemplo de definición de una clase.

```
// fichero Circulo.java

public class Circulo extends Geometria {
	static int numCirculos = 0;
	public static final double PI=3.14159265358979323846;
	public double x, y, r;
	public Circulo(double x, double y, double r) {
		this.x=x; this.y=y; this.r=r;
		numCirculos++;
	}
	public Circulo(double r) { this(0.0, 0.0, r); }
	public Circulo(Circulo c) { this(c.x, c.y, c.r); }
	public Circulo() { this(0.0, 0.0, 1.0); }
	public double perimetro() { return 2.0 * PI * r; }
	public double area() { return PI * r * r; }
	// método de objeto para  comparar círculos
	public Circulo elMayor(Circulo c) {
		if (this.r>=c.r) return this;
		else return c; }
	// método de clase para comparar círculos
	public static Circulo elMayor(Circulo c, Circulo d) {
		if (c.r>=d.r) return c;
		else return d; }
} // fin de la clase Circulo
```

En este ejemplo se ve cómo dentro de la clase se definen las variables miembro y los métodos, que pueden ser ***de objeto*** o ***de clase*** (***static***). Se puede ver también cómo el nombre del fichero coincide con el de la clase ***public*** con la extensión ****.java***.

5.3.- VARIABLES MIEMBRO. A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está ***centrada en los datos***. Una clase son unos ***datos*** y unos ***métodos*** que operan sobre esos datos.

5.3.1.- Variables miembro de objeto. Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas ***campos***) pueden ser de ***tipos primitivos*** (*boolean*, *int*, *long*, *double*, …) o referencias a ***objetos*** de otra clase (*composición*).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de ***tipos primitivos*** se inicializan siempre de modo automático, incluso antes de llamar al ***constructor*** (***false*** para ***boolean***, la cadena vacía para ***char*** y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la ***declaración***, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

long nDatos = 100;

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada ***objeto*** que se crea de una clase tiene ***su propia copia*** de las variables miembro. Por ejemplo, cada objeto de la clase ***Circulo*** tiene sus propias coordenadas del centro ***x*** e ***y***, y su propio valor del radio ***r***. Se puede aplicar un método a un objeto concreto poniendo el nombre del objeto y luego el nombre del método separados por un punto. Por ejemplo, para calcular el área de un objeto de la clase ***Circulo*** llamado ***c1*** se escribe: ***c1.area();***

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: ***public***, ***private***, ***protected*** y ***package*** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (***public*** y ***package***), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

5.3.2.- Variables miembro de clase (static). Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama ***variables de clase*** o variables ***static***. Las variables ***static*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Circulo***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Las variables de clase son lo más parecido que ***Java*** tiene a las ***variables globales*** de C/C++.

Las variables de clase se crean anteponiendo la palabra ***static*** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, ***Circulo.numCirculos*** es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro ***static*** se inicializan con los valores por defecto (***false*** para ***boolean***, la cadena vacía para ***char*** y cero para los tipos numéricos) para los tipos primitivos, y con ***null*** si es una referencia.

Las variables miembro ***static*** se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método ***static*** o en cuanto se utiliza una variable ***static*** de dicha clase. Lo importante es que las variables miembro ***static*** se inicializan siempre antes que cualquier objeto de la clase.

5.4.- VARIABLES FINALES.

Una variable de un tipo primitivo declarada como ***final*** no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una ***constante***, y equivale a la palabra ***const*** de C/C++.

Java permite separar la ***definición*** de la ***inicialización*** de una variable ***final***. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable ***final*** así definida es ***constante*** (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Pueden ser ***final*** tanto las variables miembro, como las variables locales o los propios argumentos de un método.

Declarar como ***final*** un objeto miembro de una clase hace ***constante*** la ***referencia***, pero no el propio objeto, que puede ser modificado. En ***Java*** no es posible hacer que un objeto sea constante.

5.5.- MÉTODOS (FUNCIONES MIEMBRO).

5.5.1.- Métodos de objeto.

Los ***métodos*** son funciones definidas dentro de una clase. Salvo los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del ***operador punto*** (.). Dicho objeto es su ***argumento implícito***. Los métodos pueden además tener otros ***argumentos explícitos*** que van entre paréntesis, a continuación del nombre del método. La primera línea de la definición de un método se llama ***declaración*** o ***header***; el código comprendido entre las ***llaves*** {…} es el ***cuerpo*** o ***body*** del método. Considérese el siguiente método tomado de la clase ***Circulo***:

```
public Circulo elMayor(Circulo c) { // header y comienzo del método
        if (this.r>=c.r) // body
            return this; // body
        else // body
            return c; // body
} // final del método
```

El ***header*** consta del cualificador de acceso (***public***, en este caso), del tipo del valor de retorno (***Circulo*** en este ejemplo, ***void*** si no tiene), del ***nombre de la función*** y de una lista de ***argumentos explícitos*** entre paréntesis, separador por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen ***visibilidad directa*** de las variables miembro del objeto que es su ***argumento implícito***, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia ***this***, de modo discrecional (como en el ejemplo anterior con ***this.r***) o si alguna variable local o argumento las oculta.

El ***valor de retorno*** puede ser un valor de un ***tipo primitivo*** o una ***referencia***. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de ***interface***. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir ***variables locales***. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

5.5.2.- Métodos sobrecargados (overloaded).

Al igual que C++, ***Java*** permite métodos ***sobrecargados*** (***overloaded***), es decir métodos distintos con ***el mismo nombre*** que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase ***Circulo*** presenta dos métodos sobrecargados: los cuatro constructores y los dos métodos ***elMayor()***.

A la hora de llamar a un método sobrecargado, ***Java*** sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.

2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.

3. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, *long* en vez de *int*), el programador debe hacer un ***cast*** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.

4. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la ***sobrecarga*** de métodos es la ***redefinición***. Una clase puede ***redefinir*** (***override***) un método heredado de una superclase. ***Redefinir*** un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la ***herencia***.

5.5.3.- Paso de argumentos a métodos. En ***Java*** los argumentos de los ***tipos primitivos*** se pasan siempre ***por valor***. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. No hay otra forma de modificar un argumento de un tipo primitivo dentro de un método y que incluirlo en una clase como variable miembro. Las ***referencias*** se pasan también ***por valor***, pero a través de ellas se pueden modificar los objetos referenciados.

.

En ***Java*** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar como argumentos punteros a función). Lo que se puede hacer en ***Java*** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear ***variables locales*** de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método. Los argumentos formales de un método (las variables que aparecen en el ***header*** del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

5.5.4.- Métodos de clase (static). Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama ***métodos de clase*** o ***static***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin(), cos(), exp(), pow(),* etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra ***static***. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que ***Java*** tiene a las funciones y variables globales de C/C++ o Visual Basic.

Ejemplo: Clase punto.

```
// fichero Punto.java
import java.lang.Math.*;
public class Punto {
        private double x, y;
        public void valorInicial(double vx, double vy) {
                x=vx;
                y=vy;
        }
```

```
        public double distancia(Punto q) {
                double dx=x-q.x;
                double dy=y-q.y;
                return Math.sqrt(dx*dx+dy*dy);
        }
}

// fichero UsoPunto.java
public class UsoPunto {
        public static void main(String arg []) {
                Punto p, q;
                double d;
                p=new Punto();
                p.valorInicial(5, 4);
                q=new Punto();
                q.valorInicial(2,2);
                d=p.distancia(q);
                System.out.println("Distancia: " + d);
        }
}
```

5.5.5.- Constructores. Un ***constructor*** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del ***constructor*** es reservar memoria e inicializar las variables miembro de la clase.

Los ***constructores*** no tienen valor de retorno (ni siquiera ***void***) y su ***nombre*** es el mismo que el de la clase. Su ***argumento implícito*** es el objeto que se está creando.

De ordinario una clase tiene ***varios constructores***, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos ***sobrecargados***). Se llama ***constructor por defecto*** al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un ***constructor*** de una clase puede llamar a ***otro constructor previamente definido*** en la misma clase por medio de la palabra ***this***. En este contexto, la palabra ***this*** sólo puede aparecer en la ***primera sentencia*** de un ***constructor***.

El ***constructor*** de una ***sub-clase*** puede llamar al constructor de su ***super-clase*** por medio de la palabra ***super***, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El ***constructor*** es tan importante que, si el programador no prepara ***ningún constructor*** para una clase, el ***compilador*** crea un ***constructor por defecto***, inicializando las variables de los tipos primitivos a su valor por defecto, los ***Strings*** a la cadena vacía y las ***referencias*** a objetos a ***null***. Si hace falta, se llama al ***constructor*** de la ***super-clase*** para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los ***constructores*** pueden tener también los modificadores de acceso ***public***, ***private***, ***protected*** y ***package***. Si un ***constructor*** es ***private***, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos ***public*** y ***static*** (*factory methods*) que llamen al ***constructor*** y devuelvan un objeto de esa clase.

Dentro de una clase, los ***constructores*** sólo pueden ser llamados por otros ***constructores*** o por métodos ***static***. No pueden ser llamados por los ***métodos de objeto*** de la clase.

Ejemplo: Clase Punto.

```
// fichero punto.java
import java.lang.Math.*;
class Punto {
        double x, y;
        Punto(double x, double y) {this.x=x; this.y=y;}
        void valorInicial(double vx, double vy) {
                x=vx;
                y=vy;
        }
        double distancia(Punto q) {
                double dx=x-q.x;
                double dy=y-q.y;
                return Math.sqrt(dx*dx+dy*dy);
        }
}

// fichero usoPunto.java
class UsoPunto {
        public static void main(String arg []) {
                Punto p, q;
                double d;
                p=new Punto(5, 4);
                q=new Punto(2, 2):
                d=p.distancia(q);
                System.out.println("Distancia: " + d);
        }
}
```

5.5.6.- Inicializadores. Por motivos que se verán más adelante, ***Java*** todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los ***inicializadores***, que pueden ser ***static*** (para la clase) o ***de objeto***.

5.5.6.1.- Inicializadores static. Un ***inicializador static*** es un método (un bloque {…} de código) que se llama automáticamente al crear la clase (al utilizarla por primera vez). Se diferencia del ***constructor*** en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los ***inicializadores static*** se crean dentro de la clase, como métodos sin nombre y sin valor de retorno, con tan sólo la palabra ***static*** y el código entre llaves {...}. En una clase pueden definirse ***varios inicializadores static***, que se llamarán en el orden en que han sido definidos.

Los ***inicializadores static*** se pueden utilizar para dar valor a las variables ***static***. Además se suelen utilizar para llamar a ***métodos nativos***, esto es, a métodos escritos por ejemplo en C/C++ (llamando a los métodos ***System.load()*** o ***System.loadLibrary()***, que leen las librerías nativas). Por ejemplo:

```
static{
    System.loadLibrary("MyNativeLibrary");
}
```

5.5.6.2.- Inicializadores de objeto. A partir de ***Java 1.1*** existen también ***inicializadores de objeto***, que no llevan la palabra ***static***. Se utilizan para las ***clases anónimas***, que por no tener nombre no tienen constructor. En este caso se llaman cada vez que se crea un objeto de la clase anónima.

5.5.7.- Resumen del proceso de creación de un objeto.

El proceso de creación de objetos de una clase es el siguiente:

1. Al crear el primer objeto de la clase o al utilizar el primer método o variable ***static*** se localiza la clase y se carga en memoria.

2. Se ejecutan los ***inicializadores static*** (sólo una vez).

3. Cada vez que se quiere crear un ***nuevo objeto***:

    · se comienza reservando la memoria necesaria
    · se da valor por defecto a las variables miembro de los tipos primitivos
    · se ejecutan los inicializadores de objeto
    · se ejecutan los constructores

5.5.8.- Destrucción de objetos (liberación de memoria).

En ***Java*** no hay ***destructores*** como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han ***perdido la referencia***, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que habían sido definidos, porque a la ***referencia*** se le ha asignado el valor ***null*** o porque a la ***referencia*** se le ha asignado la dirección de otro objeto. A esta característica de ***Java*** se le llama ***garbage collection*** (recogida de basura).

En ***Java*** es normal que varias variables de tipo referencia apunten al mismo objeto. ***Java*** lleva internamente un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar ésta a ***null***, haciendo por ejemplo:

```
ObjetoRef = null;
```

En ***Java*** no se sabe exactamente cuándo se va a activar el ***garbage collector***. Si no falta memoria es posible que no se llegue a activar en ningún momento. No es pues conveniente confiar en él para la realización de otras tareas más criticas.

Se puede llamar explícitamente al ***garbage collector*** con el método **System.gc()**, aunque esto es considerado por el sistema sólo como una "sugerencia" a la JVM.

5.5.9.- Finalizadores. Los ***finalizadores*** son métodos que vienen a completar la labor del ***garbage collector***. Un ***finalizador*** es un método que se llama automáticamente cuando se va a destruir un objeto (antes de que la memoria sea liberada de modo automático por el sistema). Se utilizan para ciertas ***operaciones de terminación*** distintas de liberar memoria (por ejemplo: cerrar ficheros, cerrar conexiones de red, liberar memoria reservada por funciones nativas, etc.). Hay que tener en cuenta que el ***garbage collector*** sólo libera la memoria reservada con ***new***. Si por ejemplo se ha reservado memoria con funciones nativas en C (por ejemplo, utilizando la función ***malloc()***), esta memoria hay que liberarla explícitamente con el método ***finalize()***.

Un ***finalizador*** es un método de objeto (no ***static***), sin valor de retorno (***void***), sin argumentos y que siempre se llama ***finalize()***. Los ***finalizadores*** se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un ***finalizador*** debería terminar siempre llamando al ***finalizador*** de su ***super-clase***.

Tampoco se puede saber el momento preciso en que los ***finalizadores*** van a ser llamados. En muchas ocasiones será conveniente que el programador realice esas operaciones de finalización de modo explícito mediante otros métodos que él mismo llame.

El método ***System.runFinalization()*** "sugiere" a la JVM que ejecute los ***finalizadores*** de los objetos pendientes (que han perdido la referencia). Parece ser que para que este método se ejecute hay que llamar primero a ***gc()*** y luego a ***runFinalization()***.

5.6.- PACKAGES.

5.6.1.- Qué es un package. Un ***package*** es una agrupación de clases. En la API de ***Java 1.1*** había 22 ***packages***; en Java 1.2 hay 59 ***packages***, lo que da una idea del "crecimiento" experimentado por el lenguaje.

Además el usuario puede crear sus propios ***packages***. Para que una clase pase a formar parte de un ***package*** llamado ***pkgName***, hay que introducir en ella la sentencia:

package pkgName;

que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los ***packages*** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un ***package*** puede constar de varios nombres unidos por puntos (los propios ***packages*** de ***Java*** siguen esta norma, como por ejemplo ***java.awt.event***).

Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio. Los nombres compuestos de los ***packages*** están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los ***nombres de las clases*** de ***Java*** sean únicos en ***Internet***. Es el nombre del ***package*** lo que permite obtener esta característica. Una forma de conseguirlo es incluir el ***nombre del dominio*** (quitando quizás el país), como por ejemplo en el ***package*** siguiente:

es.ceit.jgjalon.infor2.ordenar

Las clases de un ***package*** se almacenan en un directorio con el mismo nombre largo (***path***) que el ***package***. Por ejemplo, la clase,

es.ceit.jgjalon.infor2.ordenar.QuickSort.class

debería estar en el directorio,

CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de ***Java*** (clases del sistema o de usuario), en este caso la posición del directorio ***es*** en los discos locales del ordenador.

Los ***packages*** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.

2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de ***Java*** es la ***Internet***). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del ***package***.

3. Para ayudar en el control de la accesibilidad de clases y miembros.

5.6.2.- Cómo funcionan los packages. Con la sentencia ***import packname;*** se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, ***Java*** da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del ***package***.

El importar un ***package*** no hace que se carguen todas las clases del ***package***: sólo se cargarán las clases ***public*** que se vayan a utilizar. Al importar un ***package*** no se importan los ***sub-packages***. Éstos deben ser importados explícitamente, pues en realidad son ***packages*** distintos. Por ejemplo, al importar ***java.awt*** no se importa ***java.awt.event***.

Es posible guardar en jerarquías de directorios diferentes los ficheros ****.class*** y ****.java***, con objeto por ejemplo de no mostrar la situación del código fuente. Los ***packages*** hacen referencia a los ficheros compilados ****.class***.

En un programa de ***Java***, una clase puede ser referida con su nombre completo (el nombre del ***package*** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia ***import*** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del ***package*** importado. Se importan por defecto el ***package java.lang*** y el ***package*** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar ***import***: para ***una clase*** y para ***todo un package***:

import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.;*

que deberían estar en el directorio:

classpath\es\ceit\jgjalon\infor2\ordenar

5.7.- HERENCIA.

5.7.1.- Concepto de herencia. Se puede construir una clase a partir de otra mediante el mecanismo de la ***herencia***. Para indicar que una clase deriva de otra se utiliza la palabra ***extends***, como por ejemplo:

*class CirculoGrafico **extends** Circulo {...}*

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser ***redefinidas*** (***overridden***) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la ***sub-clase*** (la clase derivada) "contuviera" un objeto de la ***super-clase***; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de ***Java*** creadas por el programador tienen una ***super-clase***. Cuando no se indica explícitamente una ***super-clase*** con la palabra ***extends***, la clase deriva de ***java.lang.Object***, que es la clase raíz de toda la jerarquía de clases de ***Java***. Como consecuencia, todas las clases tienen algunos métodos que han heredado de ***Object***.

La ***composición*** (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la ***herencia*** en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace ***private***).

Ejemplo: Clase Pixels.

```
// fichero pixels.java
class Pixels extends Punto {
        byte color;
        Pixel(double vx, double vy, byte vc) {
                super(vx, vy);          // debe ser la primera sentencia
                color=vc;
        }
        .......
}
```

5.7.2 La clase Object. Como ya se ha dicho, la clase ***Object*** es la raíz de toda la jerarquía de clases de ***Java***. Todas las clases de ***Java*** derivan de ***Object***.

La clase ***Object*** tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

1. Métodos que pueden ser redefinidos por el programador:

   clone() Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de ***Object*** lanza una *CloneNotSupportedException*. Si se desea poder clonar una clase hay que implementar la interface ***Cloneable*** y redefinir el método ***clone()***. Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador ***new*** ni a los constructores.

   equals() Indica si dos objetos son o no iguales. Devuelve ***true*** si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.

   toString() Devuelve un ***String*** que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirlo o exportarlo.

   finalize() Este método ya se ha visto al hablar de los ***finalizadores***.

2. Métodos que no pueden ser redefinidos (son métodos ***final***):
   getClass() Devuelve un objeto de la clase ***Class***, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.
   notify(), ***notifyAll()*** y ***wait()*** Son métodos relacionados con las ***threads.***

5.7.3.- Redefinición de métodos heredados. Una clase puede ***redefinir*** (volver a definir) cualquiera de los métodos heredados de su ***super-clase*** que no sean ***final***. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Los métodos de la ***super-clase*** que han sido redefinidos pueden ser todavía accedidos por medio de la palabra ***super*** desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ***ampliar los derechos de acceso*** de la ***super-clase*** (por ejemplo ser ***public***, en vez de ***protected*** o ***package***), pero nunca restringirlos.

Los ***métodos de clase*** o ***static*** no pueden ser redefinidos en las clases derivadas.

Ejemplo:

```
class Pixels extends Punto {
        .....
        double distancia(double cx, double cy) {
                double dx=Math.abs(x-cx);
                double dy=Math.abs(y-cy);
                return dx+dy;          // distancia manhattan
        }
}
```

5.7.4.- Clases y métodos abstractos. Una ***clase abstracta*** (***abstract***) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra ***abstract***, como por ejemplo,

public abstract class Geometria { ... }

Una clase ***abstract*** puede tener métodos declarados como ***abstract***, en cuyo caso no se da definición del método. Si una clase tiene algún método ***abstract*** es obligatorio que la clase sea ***abstract***. En cualquier ***sub-clase*** este método deberá bien ser redefinido, bien volver a declararse como ***abstract*** (el método y la ***sub-clase***).

Una clase ***abstract*** puede tener métodos que no son ***abstract***. Aunque no se puedan crear objetos de esta clase, sus ***sub-clases*** heredarán el método completamente a punto para ser utilizado.

Como los métodos ***static*** no pueden ser redefinidos, un método ***abstract*** no puede ser ***static***.

Ejemplo:

```
abstract class Figura {
        public abstract double perimetro();
        public abstract double area();
}
class Cuadrado extends Figura {
        double lado;
        Cuadrado(double l) {lado=l;}
        double perimetro() {return lado*4;}
```

```
        double area() {return lado*lado;}
}

class Circulo extends Figura {
        double radio;
        Circulo(double r) {radio=r;}
        double perimetro() {return 2*Math.PI*radio;}
        double area() {return Math.PI*radio*radio;}
}
class Ejherencia {
        static public void main(String arg []) {
                Figura [] f=new Figura[2];
                f[0]=new Cuadrado(12);
                f[1]=new Circulo(7);
                for(int i=0;i<2;i++) {
                        System.out.println("Figura "+i);
                        System.out.println("\t"+f[i].perimetro());
                        System.out.println("\t"+f[i].area());
                }
        }
}
```

5.7.5.- Constructores en clases derivadas. Ya se comentó que un ***constructor*** de una clase puede llamar por medio de la palabra ***this*** a otro ***constructor*** previamente definido en la misma clase. En este contexto, la palabra ***this*** sólo puede aparecer en la primera sentencia de un ***constructor***.

De forma análoga el ***constructor*** de una clase derivada puede llamar al ***constructor*** de su ***super-clase*** por medio de la palabra ***super()***, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la ***super-clase***. De esta forma, un ***constructor*** sólo tiene que inicializar directamente las variables no heredadas.

La llamada al ***constructor*** de la ***super-clase*** debe ser la ***primera sentencia del constructor*** , excepto si se llama a otro constructor de la misma clase con ***this()***. Si el programador no la incluye, ***Java*** incluye automáticamente una llamada al ***constructor por defecto*** de la ***super-clase***, ***super()***. Esta llamada en cadena a los ***constructores de las super-clases*** llega hasta el origen de la jerarquía de clases, esto es al constructor de ***Object***.

Como ya se ha dicho, si el programador no prepara un ***constructor por defecto***, el compilador crea uno, inicializando las variables de los ***tipos primitivos*** a cero, los ***Strings*** a la cadena vacía y las ***referencias*** a objetos a ***null***. Antes, incluirá una llamada al constructor de la ***super-clase***.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con ***new***, de la que se encarga el ***garbage collector***), es importante llamar a los ***finalizadores*** de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el ***finalizador de la sub-clase*** deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la

forma ***super.finalize()***. Los métodos ***finalize()*** deben ser al menos ***protected***, ya que el método ***finalize()*** de ***Object*** lo es, y no está permitido reducir los permisos de acceso en la herencia.

5.8.- CLASES Y MÉTODOS FINALES.

Una ***clase*** declarada ***final*** no puede tener clases derivadas. Esto se puede hacer por motivos de ***seguridad*** y también por motivos de ***eficiencia***, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un ***método*** declarado como ***final*** no puede ser redefinido por una clase que derive de su propia clase.

5.9.- INTERFACES.

5.9.1.- Concepto de interface.

Una ***interface*** es un conjunto de ***declaraciones de métodos*** (sin ***definición***). También puede definir ***constantes***, que son implícitamente ***public***, ***static*** y ***final***, y deben siempre inicializarse en la declaración. Estos métodos definen un *tipo de conducta*. Todas las clases que implementan una determinada ***interface*** están obligadas a proporcionar una definición de los métodos de la ***interface***, y en ese sentido adquieren una ***conducta*** o ***modo de funcionamiento***.

Una ***clase*** puede ***implementar*** una o varias ***interfaces***. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las ***interfaces***, separados por comas, detrás de la palabra ***implements***, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo
        implements Dibujable, Cloneable {
        ...
}
```

¿Qué diferencia hay entre una ***interface*** y una ***clase abstract***? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase ***abstract*** puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas ***diferencias importantes***:

1. Una clase no puede heredar de dos clases ***abstract***, pero sí puede heredar de una clase ***abstract*** e implementar una ***interface***, o bien implementar dos o más ***interfaces***.

2. Una clase no puede heredar métodos -definidos- de una ***interface***, aunque sí ***constantes***.

3. Las ***interfaces*** permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, inpendientemente de su situación en la jerarquía de clases de ***Java***.

4. Las ***interfaces*** permiten "publicar" el comportamiento de una clase desvelando un mínimo de información.

5. Las ***interfaces*** tienen una ***jerarquía*** propia, independiente y más flexible que la de las clases, ya que tienen permitida la ***herencia múltiple***.

6. De cara al ***polimorfismo,*** las ***referencias*** de un tipo ***interface*** se pueden utilizar de modo similar a las clases ***abstract***.

5.9.2.- Definición de interfaces. Una ***interface*** se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface ***Dibujable***:

```
// fichero Dibujable.java
import java.awt.Graphics;
public interface Dibujable {
        double PI=3.141615;
        public void setPosicion(double x, double y);
        public void dibujar(Graphics dw);
}
```

Cada ***interface public*** debe ser definida en un fichero ****.java*** con el mismo nombre de la ***interface***. Los ***nombres de las interfaces*** suelen comenzar también con ***mayúscula***.

Las ***interfaces*** no admiten más que los modificadores de acceso ***public*** y ***package***. Si la ***interface*** no es ***public*** no será accesible desde fuera del ***package*** (tendrá la accesibilidad por defecto, que es ***package***). Los métodos declarados en una ***interface*** son siempre ***public*** y ***abstract***, de modo implícito.

5.9.3.- Herencia en interfaces. Entre las ***interfaces*** existe una ***jerarquía*** (independiente de la de las clases) que permite ***herencia simple y múltiple***. Cuando una ***interface*** deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una ***interface*** puede derivar de varias ***interfaces***. Para la herencia de ***interfaces*** se utiliza asimismo la palabra ***extends***, seguida por el nombre de las ***interfaces*** de las que deriva, separadas por comas.

Una ***interface*** puede ocultar una constante definida en una ***super-interface*** definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una ***super-interface***.

Las ***interfaces*** no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha ***interface*** dejarán de funcionar, a menos que implementen el nuevo método.

5.9.4.- Utilización de interfaces. Las ***constantes*** definidas en una ***interface*** se pueden utilizar en cualquier clase (aunque no implemente la ***interface***) precediéndolas del nombre de la ***interface***, como por ejemplo (suponiendo que PI hubiera sido definida en ***Dibujable***):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que ***implementan*** la ***interface*** las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces para agrupar constantes simbólicas relacionadas (en este sentido pueden en parte suplir las variables ***enum*** de C/C++).

De cara al ***polimorfismo***, el nombre de una ***interface*** se puede utilizar como un ***nuevo tipo de referencia***. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la ***interface***. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Ejemplo:

```
interface Agrandable {
        void zoom(int i);
}
class Circulo extends Figura implements Agrandable {
        ........
        public void zoom(int i) {
                radio*=i;  }
        .....
}
class Punto implements Agrandable {
        .........
        public void zoom(int i) {
                x*=i;
                y*=i;   }
        ..........
}
class EjInterfaz {
        static public void main(String arg []) {
                Agrandable [] f=new Agrandable [2];
                f[0]=new Circulo(12);
                f[1]=new Punto(7, 4);
                for(int i=0;i<2;i++)
                        f[i].zoom(3);
        }
}
```

5.10.- PERMISOS DE ACCESO EN JAVA.

Una de las características de la *Programación Orientada a Objetos* es la ***encapsulación***, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una determinada tarea. Los permisos de acceso de ***Java*** son una de las herramientas para conseguir esta finalidad.

5.10.1.- Accesibilidad de los packages.

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un ***package*** es accesible si sus directorios y ficheros son accesibles (si están en un

ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos ***packages*** que se encuentren en la variable ***CLASSPATH*** del sistema.

5.10.2.- Accesibilidad de clases o interfaces. En principio, cualquier ***clase*** o i***nterface*** de un ***package*** es accesible para todas las demás clases del ***package***, tanto si es ***public*** como si no lo es. Una clase ***public*** es accesible para cualquier otra clase siempre que su ***package*** sea accesible. Recuérdese que las ***clases*** e ***interfaces*** sólo pueden ser ***public*** o ***package*** (la opción por defecto cuando no se pone ningún modificador).

5.10.3.- Accesibilidad de las variables y métodos miembros de una clase.

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia ***this***) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.

2. Los miembros ***private*** de una clase sólo son accesibles para la propia clase.

3. Si el ***constructor*** de una clase es ***private***, sólo un método ***static*** de la propia clase puede crear objetos.

Desde una ***sub-clase***:

1. Las ***sub-clases*** heredan los miembros ***private*** de su ***super-clase***, pero sólo pueden acceder a ellos a través de métodos ***public***, ***protected*** o ***package*** de la super-clase.

Desde otras clases del ***package***:

1. Desde una clase de un ***package*** se tiene acceso a todos los miembros que no sean ***private*** de las demás clases del ***package***.

Desde otras clases fuera del ***package***:

1. Los métodos y variables son accesibles si la clase es ***public*** y el miembro es ***public***.

2. También son accesibles si la clase que accede es una ***sub-clase*** y el miembro es ***protected***.

La siguiente tabla muestra un resumen de los permisos de acceso de **Java**.

Visibilidad	**public**	**protected**	**private**	**default**
Desde la propia clase	SI	SI	SI	SI
Desde otra clase en el propio package	SI	NO	NO	SI
Desde otra clase fuera del package	SI	NO	NO	NO

Desde una sub-clase en el propio package	SI	SI	NO	SI
Desde una sub-clase fuera del propio package	SI	SI	NO	NO

5.11.- TRANSFORMACIONES DE TIPO: CASTING. En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

La conversión entre tipos primitivos es más sencilla. En ***Java*** se realizan de modo automático conversiones implícitas ***de un tipo a otro de más precisión***, por ejemplo de ***int*** a ***long***, de ***float*** a ***double***, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son ***conversiones inseguras*** que pueden dar lugar a errores (por ejemplo, para pasar a ***short*** un número almacenado como ***int***, hay que estar seguro de que puede ser representado con el número de cifras binarias de ***short***). A estas conversiones explícitas de tipo se les llama ***cast***. El ***cast*** se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en ***Java*** no se puede convertir un tipo numérico a ***boolean***.

La conversión de ***Strings*** (texto) a números se verá en el tema siguiente.

5.12.- POLIMORFISMO. El ***polimorfismo*** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama ***vinculación*** (*binding*). La ***vinculación*** puede ser ***temprana o estática*** (en tiempo de compilación) o ***tardía o dinámica*** (en tiempo de ejecución).

La *vinculación temprana* se realiza mediante funciones sobrecargadas. En el *polimorfismo por sobrecarga* los nombres de las funciones miembro de una clase pueden repetirse si varía el número o el tipo de los argumentos.

Ejemplo: Polimorfismo por vinculación temprana (por sobrecarga de funciones).

```
// fichero punto.java
import java.lang.Math.*;
class Punto {
        double x, y;
        Punto(double x, double y) {this.x=x; this.y=y;}
        void valorInicial(double vx, double vy) {
                x=vx;
```

```
            y=vy;
        }
        double distancia(Punto q) {
            double dx=x-q.x;
            double dy=y-q.y;
            return Math.sqrt(dx*dx+dy*dy);
        }
        double distancia(double cx, double cy) {
            double dx=x-cx;
            double dy=y-cy;
            return Math.sqrt(dx*dx+dy*dy);
        }
}
```

Con funciones redefinidas en ***Java*** se utiliza siempre ***vinculación tardía***, excepto si el método es ***final***. La ***vinculación tardía*** hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea ***el tipo de objeto*** y ***no el tipo de la referencia*** lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el ***polimorfismo*** necesita evaluación tardía.

El ***polimorfismo*** puede hacerse con referencias de ***super-clases abstract***, ***super-clases normales*** e ***interfaces***. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las ***interfaces*** permiten ampliar muchísimo las posibilidades del polimorfismo.

Ejemplo: Polimorfismo por vinculación tardía (vinculación dinámica).

```
class PruebaPixel2
{
    public static void main(String arg [])
    {
        Punto x=new Pixel(4, 3, (byte)2);
        Punto p=new Punto(4, 3);
        double d1, d2;
        d1=x.distancia(1, 1);      // método de Pixel
        d2=p.distancia(1, 1);      // método de Punto
        System.out.println("Distancia pixel: " + d1);
        System.out.println("Distancia punto: ") + d1);
    }
}
```

Si se desea acceder a algún método oculto de la clase base se ha de utilizar *super*.

Ejemplo: Acceso al método oculto de la clas base.

```
class Punto {
        .......
        void trasladar(double cx, double cy) {
                x+=cx;
                y+=cy;
        }
}

class Pixel extends Punto {
        ........
        void trasladar(double cx, double cy) {
                super.trasladar(cx, cy);
                color=(byte)0;
        }
}
```

5.13.- Ejercicicios propuestos.

1. Se dispone de una urna con un número de bolas blancas y otro número de negras. Se pretende realizar el siguiente experimento: mientras quede más de una bola en la urna, sacar dos bolas y mirar su color. Si ambos son del mismo color, meter en la urna una bola negra; si por el contrario son de distinto color, meter en la urna una bola blanca. Extraer la última bola y mirar su color. Crear la clase **urna** que responda a este comportamiento y realizar un programa en JAVA para averiguar de que color es la ultima bola extraida de la urna.

2. A partir de la urna anterior implementar la clase **urnaTrampa** con el siguiente comportamiento: cada vez que se saque una bola, hay una probabilidad del 0.2 % de que una de las bolas que quedan dentro cambie de color (para ello ha de haber de ambos colores en la urna).

3. Cread la clase **Mate** con los siguientes métodos de clase (***static***): factorial, primo, perfecto, amigos, primos, euler y potencia.

Tema 6. Paquetes en Java. Clases de utilidad.

6.1.- Introducción. En Java los paquetes son una forma de agrupar clases e interfaces asociadas.Un *paquete* es un conjunto de clases almacenadas todas en un mismo directorio. La jerarquía de directorios es la misma que la jerarquía de paquetes. Los nombres de los paquetes deben coincidir con los nombres de los directorios. El nombre completo de una clase es el del paquete seguido por el nombre de la clase, separados por un punto.

Java proporciona una librería completa de clases en forma de paquetes. A saber:

- lang: para funciones del lenguaje (por defecto)
- util: para utilidades adicionales
- io: para entrada y salida
- text: para formato especializado
- awt: para diseño gráfico e interfaz de usuario
- awt.event: para gestionar eventos
- applet: para crear aplicaciones de red
- net: para comunicaciones
- otras más que van estandarizándose según versiones

6.2.- ARRAYS. Los ***arrays*** de ***Java*** (vectores, matrices, hiper-matrices de más de dos dimensiones) se tratan como objetos de una clase predefinida. Los ***arrays*** son ***objetos***, pero con algunas características propias.

Algunas de las características más importantes de los ***arrays*** son las siguientes:

1. Los ***arrays*** se crean con el operador ***new*** seguido del tipo y número de elementos.

2. Se puede acceder al número de elementos de un array con la variable miembro implícita ***length*** (por ejemplo, ***vect.length***).

3. Se accede a los elementos de un ***array*** con los ***corchetes []*** y un ***índice*** que varía de 0 a ***length-1***.

4. Se pueden crear ***arrays*** de objetos de cualquier tipo. En principio un ***array*** de objetos es un ***array de referencias*** que hay que completar llamando al operador ***new***.

5. Los elementos de un ***array*** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, la cadena vacía para ***Strings***, ***false*** para ***boolean***, ***null*** para referencias).

6. Como todos los objetos, los ***arrays*** se pasan como argumentos a los métodos ***por referencia***.

7. Se pueden crear ***arrays anónimos*** (por ejemplo, crear un nuevo array como argumento actual en la llamada a un método).

Inicialización de ***arrays***:

1. Los ***arrays*** se pueden inicializar con valores entre llaves {...} separados por comas.

2. También los ***arrays de objetos*** se pueden inicializar con varias llamadas a ***new*** dentro de unas llaves {...}.

3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.

4. Creación de una ***referencia*** a un array. Son posibles dos formas:

   double[] x; // preferible
   double x[];

5. Creación del ***array*** con el operador ***new***:

   x = new double[100];

6. Las dos etapas 4 y 5 se pueden unir en una sola:

   double[] x = new double[100];

Ejemplos: Creación de arrays unidimensionales.

```
// crea un array de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];

// otra forma de hacer esto mismo sería:
int v[];                    // define un array de enteros
v=new int[10];              // reserva memoria para 10 elementos

// array de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a
                                     // null
for( i = 0 ; i < 5;i++)
      listaObj[i] = new MiClase(...);

// crear arrays inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves", “viernes”, "sabado", "domingo"};

// array anónimo
obj.metodo( new String[]={"uno", "dos", "tres"});
```

6.2.1.- Arrays bidimensionales. No existen como tal en Java. En ***Java*** una ***matriz*** es un ***vector*** de ***vectores fila***, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

int [][] mat = new int[3][4];

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la ***referencia*** indicando con un doble corchete que es una ***referencia a matriz***,

   int[][] mat;

2. Crear el vector de referencias a las filas,

   mat = new int[nfilas][];

3. Reservar memoria para los vectores correspondientes a las filas,

   ```
   for (int i=0; i<nfilas; i++);
       mat[i] = new int[ncols];
   ```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},{4, 5, 6}};      // esta coma es permitida
int c[][] = new int [3][];                    // se crea el array de referencias a arrays
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

En el caso de una matriz ***b***, ***b.length*** es el número de filas y ***b[0].length*** es el número de columnas (de la fila 0). Por supuesto, los arrays bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.

6.3.- El paquete java.lang. Contiene las siguientes clases: Object, System, Math, Character, String, StringBuffer y las clases envoltorios de tipos básicos. Es el paquete por defecto.

6.3.1. La clase Object. Es la clase superior de toda la jerarquía de clases de Java: si una definición de clase no extiende a otra, entonces extiende a Object. Los métodos más importantes de esta clase son:

- *boolean equals(Object)*
- *String toString()*
- *void finalize()*

6.3.2.- La clase System. Maneja particularidades del sistema. Dispone de tres variables de clase públicas:

- *PrintStream out, err*
- *InputStream in*

y de varios métodos de clase públicos, de entre los cuales destacan:

- *void exit(int)*
- *void gc()*
- *void runFinalizersOnExit(boolean)*

6.3.3.- La clase Math. Incorpora como *miembros estáticos* constantes y funciones matemáticas, a saber:

- Constantes: E, PI
- Métodos: abs(), sin(), cos(), tan(), asin(), acos(), atan(), max(), min(), exp(), pow(), sqrt(), round(), random(),.....

Para utilizar cualquiera de estos métodos no hay que importar la clase *Math* puesto que pertenece al paquete *java.lang* (paquete que se carga por defecto):

System.out.println("raíz cuadrada de " + x + "= " + Math.sqrt(x));

La siguiente tabla muestra los métodos soportados por esta clase.

Métodos	**Significado**	**Métodos**	**Significado**
abs()	Valor absoluto	sin(double)	Calcula el seno
acos()	Arcocoseno	tan(double)	Calcula la tangente
asin()	Arcoseno	exp()	Calcula la función exponencial
atan()	Arcotangente entre - PI/2 y PI/2	log()	Calcula el logaritmo natural (base e)
atan2(,)	Arcotangente entre -PI y PI	max(,)	Máximo de dos argumentos
ceil()	Entero más cercano en dirección a infinito	min(,)	Mínimo de dos argumentos
floor()	Entero más cercano en dirección a -infinito	cos(double)	Calcula el coseno
random()	Número aleatorio entre 0.0 y 1.0	round()	Entero más cercano al Argumento
power(,)	Devuelve el primer argumento elevado al segundo	rint(double)	Devuelve el entero más próximo

sqrt()	Devuelve la raíz cuadrada	IEEEremainder (double,double)	Calcula el resto de la división
toDegrees(double)	Pasa de radianes a grados (Java 2)	toRadians()	Pasa de grados a radianes (Java 2)

6.3.4.- La clase String. Sirve para manipular cadenas de caracteres constantes. La clase ***String*** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar. La clase ***StringBuffer*** permite que el programador cambie la cadena insertando, borrando, etc. La primera es más eficiente, mientras que la segunda permite más posibilidades.

Ambas clases pertenecen al package ***java.lang***, y por lo tanto no hay que importarlas. Hay que indicar que el ***operador de concatenación*** (+) entre objetos de tipo ***String*** utiliza internamente objetos de la clase ***StringBuffer*** y el método ***append()***.

Los objetos de la clase ***String*** se pueden crear a partir de cadenas constantes o *literales*, definidas entre dobles comillas, como por ejemplo: "Hola". ***Java*** crea siempre un objeto ***String*** al encontrar una cadena entre comillas. A continuación se describen dos formas de crear objetos de la clase ***String***,

```
String str1 = "Hola";                 // el sistema más eficaz de crear Strings
String str2 = new String("Hola");  // también se pueden crear con un constructor
```

El primero de los métodos expuestos es el más eficiente, porque como al encontrar un texto entre comillas se crea automáticamente un objeto ***String***, en la práctica utilizando ***new*** se llama al constructor dos veces. También se pueden crear objetos de la clase ***String*** llamando a otros constructores de la clase, a partir de objetos ***StringBuffer***, y de arrays de ***bytes*** o de ***chars***.

La siguiente tabla muestra los métodos más importantes de la clase ***String***.

Métodos de String	**Función que realizan**
String(...)	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
String(String str) y String(StringBuffer sb)	Costructores a partir de un objeto String o StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
indexOf(String, [int])	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
lastIndexOf(String, [int])	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
length()	Devuelve el número de caracteres de la cadena
replace(char, char)	Sustituye un carácter por otro en un String
startsWith(String)	Indica si un String comienza con otro String o no

substring(int, int)	Devuelve un String extraído de otro
toLowerCase()	Convierte en minúsculas (puede tener en cuenta el locale)
toUpperCase()	Convierte en mayúsculas (puede tener en cuenta el locale)
trim()	Elimina los espacios en blanco al comienzo y final de la cadena
valueOf()	Devuelve la representación como String de sus argumento. Admite Object, arrays de caracteres y los tipos primitivos

Un punto importante a tener en cuenta es que hay métodos, tales como ***System.out.println()***, que exigen que su argumento sea un objeto de la clase ***String***. Si no lo es, habrá que utilizar algún metodo que lo convierta en ***String***.

El ***locale*** citado en la tabla anterior, es la forma que java tiene para adaptarse a las peculiaridades de los idiomas distintos del inglés: acentos, caracteres especiales, forma de escribir las fechas y las horas, unidades monetarias, etc.

6.3.5.- La clase StringBuffer. Sirve para manipular cadenas de caracteres permitiendo su actualización. La clase ***StringBuffer*** se utiliza prácticamente siempre que se desee modificar una cadena de caracteres. Completa los métodos de la clase ***String*** ya que éstos realizan sólo operaciones sobre el texto que no conllevan un aumento o disminución del número de letras del ***String***.

La siguiente tabla muestra los métodos más importantes de la clase ***StringBuffer***.

Métodos de StringBuffer	**Función que realizan**
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
capacity()	Devuelve el espacio libre del StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
insert(int,)	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
length()	Devuelve el número de caracteres de la cadena
reverse()	Cambia el orden de los caracteres
setCharAt(int, char)	Cambia el carácter en la posición indicada
setLength(int)	Cambia el tamaño de un StringBuffer
toString()	Convierte en objeto de tipo String

6.3.6.- Las clases envoltorios (WRAPPERS). Los ***Wrappers*** (*envoltorios*) son clases diseñadas para ser un ***complemento*** de los ***tipos primitivos***. En efecto, los tipos primitivos son los únicos elementos

de ***Java*** que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la ***eficiencia***, pero algunos inconvenientes desde el punto de vista de la ***funcionalidad***. Por ejemplo, los ***tipos primitivos*** siempre se pasan como argumento a los métodos ***por valor***, mientras que los ***objetos*** se pasan ***por referencia***. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada. Una forma de conseguir esto es utilizar un ***Wrapper***, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases ***Wrapper*** también proporcionan métodos para realizar otras tareas con los tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase ***Wrapper*** para cada uno de los tipos primitivos numéricos, esto es, existen las clases ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double*** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de ***Java***). A continuación se van a ver dos de estas clases: ***Double*** e ***Integer***. Las otras cuatro son similares y sus características pueden consultarse en la documentación on-line.

6.3.6.1 Clase Double. La clase ***java.lang.Double*** deriva de ***Number***, que a su vez deriva de ***Object***. Esta clase contiene un valor primitivo de tipo ***double***. La siguiente tabla muestra algunos métodos y constantes predefinidas de la clase ***Double***.

Métodos	**Función que desempeñan**
Double(double) y Double(String)	Los constructores de esta clase
doubleValue(), floatValue(), longValue(), intValue(),shortValue(), byteValue()	Métodos para obtener el valor del tipo primitivo
String toString(), Double valueOf(String)	Conversores con la clase String
isInfinite(), isNaN()	Métodos de chequear condiciones
equals(Object)	Compara con otro objeto
MAX_DOUBLE,MIN_DOUBLE, POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, TYPE	Constantes predefinidas. TYPE es el objeto Class representando esta clase

El *Wrapper* ***Float*** es similar al *Wrapper* ***Double***.

6.3.6.2.- Clase Integer. La clase ***java.lang.Integer*** tiene como variable miembro un valor de tipo ***int***. La siguiente tabla muestra los métodos y constantes de la clase ***Integer***.

Métodos	**Función que desempeñan**
Integer(int) y Integer(String)	Constructores de la clase
doubleValue(), floatValue(), longValue(),intValue(), shortValue(), byteValue()	Conversores con otros tipos primitivos
Integer decode(String), Integer parseInt(String), String toString(), Integer valueOf(String)	Conversores con String
String toBinaryString(int), String toHexString(int), String toOctalString(int)	Conversores a cadenas representando enteros en otros sistemas de numeración
Integer getInteger(String)	Determina el valor de una propiedad del sistema a partir del nombre de dicha propiedad
MAX_VALUE, MIN_VALUE, TYPE	Constantes predefinidas

Los *Wrappers* ***Byte***, ***Short*** y ***Long*** son similares a ***Integer***.

Los *Wrappers* son utilizados para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un fichero de texto, etc. Los ejemplos siguientes muestran algunas conversiones:

```
String numDecimalString = "8.9";
float numFloat=Float.valueOf(numDecimalString).floatValue();
double numDouble=Double.valueOf(numDecimalString).doubleValue();
String numIntString = "1001";
int numInt=Integer.valueOf(numIntString).intValue();
```

En el caso de que el texto no se pueda convertir directamente al tipo especificado se lanza una excepción de tipo ***NumberFormatException***, por ejemplo si se intenta convertir directamente el texto "4.897" a un número entero. El proceso que habrá que seguir será convertirlo en primer lugar a un número ***float*** y posteriormente a número entero.

6.4.- El paquete java.util. Contiene clases de utilidad, a saber:

- Las colecciones
- La clase StringTokenizer
- La clase Random
- Otras (ver documentación)

6.4.1.- Colecciones. ***Java*** dispone también de clases e interfaces para trabajar con colecciones de objetos. Entre otras tenemos las clases ***Vector, Stack*** y ***Hashtable***, así como la interface ***Enumeration***.

6.4.1.1- Clase Vector. La clase ***java.util.Vector*** deriva de ***Object***, implementa ***Cloneable*** (para poder sacar copias con el método ***clone()***) y ***Serializable*** (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, ***Vector*** representa un ***array de objectos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos de forma dinámica. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes [].

El método ***capacity()*** devuelve el tamaño o número de elementos que puede tener el vector. El método ***size()*** devuelve el número de elementos que realmente contiene, mientras que ***capacityIncrement*** es una variable que indica el salto que se dará en el tamaño cuando se necesite crecer. La siguiente tabla muestra los métodos más importantes de la clase ***Vector***. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Métodos	**Función que realizan**
Vector(), Vector(int), Vector(int, int)	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
void addElement(Object obj)	Añade un objeto al final
boolean removeElement(Object obj)	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
void removeAllElements()	Elimina todos los elementos
Object clone()	Devuelve una copia del vector
void copyInto(Object anArray[])	Copia un vector en un array
void trimToSize()	Ajusta el tamaño a los elementos que tiene
void setSize(int newSize)	Establece un nuevo tamaño
int capacity()	Devuelve el tamaño (capacidad) del vector
int size()	Devuelve el número de elementos
boolean isEmpty()	Devuelve true si no tiene elementos
Enumeration elements()	Devuelve una Enumeración con los elementos
boolean contains(Object elem)	Indica si contiene o no un objeto
int indexOf(Object elem, int index)	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
int lastIndexOf(Object elem, int index)	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
Object elementAt(int index)	Devuelve el objeto en una determinada posición
Object firstElement()	Devuelve el primer elemento

Object lastElement()	Devuelve el último elemento
void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void removeElementAt(int index)	Elimina el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición

Además de ***capacityIncrement***, existen otras dos variables miembro: ***elementCount***, que representa el número de componentes válidos del vector, y ***elementData[]*** que es el array de ***Objects*** donde realmente se guardan los elementos del objeto ***Vector*** (***capacity*** es el tamaño de este array). Las tres variables citadas son ***protected***.

6.4.1.2.- Interface Enumeration. La interface ***java.util.Enumeration*** define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface ***Enumeration*** declara dos métodos:

1. ***public boolean hasMoreElements()***. Indica si hay más elementos en la colección o si se ha llegado ya al final.

2. ***public Object nextElement()***. Devuelve el siguiente objeto de la colección. Lanza una *NoSuchElementException* si se llama y ya no hay más elementos.

Todas las colecciones implementan el método *Enumeration elements()*

que devuelve una Enumeración con los elementos de la colección correspondiente.

Ejemplo: Para imprimir los elementos de un vector ***vec*** se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); ) {
        System.out.println(e.nextElement());
}
```

donde, como puede verse en la tabla de los métodos de la clase Vector, el método ***elements()*** devuelve precisamente una referencia de tipo ***Enumeration***. Con los métodos ***hasMoreElements()*** y ***nextElement()*** y un bucle ***for*** se pueden ir imprimiendo los distintos elementos del objeto ***Vector***.

6.4.1.3.- Clase Hashtable. La clase ***java.util.Hashtable*** extiende ***Dictionary*** (***abstract***) e implementa ***Cloneable*** y ***Serializable***. Una ***hashtable*** es una tabla que relaciona una ***clave*** con un ***valor***. Cualquier objeto distinto de ***null*** puede ser tanto ***clave*** como ***valor***.

La clase a la que pertenecen las ***claves*** debe implementar los métodos ***hashCode()*** y ***equals()***, con objeto de hacer búsquedas y comparaciones. El método ***hashCode()*** devuelve un entero único y distinto

para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método ***equals()***, el método ***hashCode()*** devuelve el mismo entero.

Cada objeto de ***hashtable*** tiene dos variables: ***capacity*** y ***load factor*** (entre 0.0 y 1.0). Cuando el número de elementos excede el producto de estas variables, la ***hashtable*** crece llamando al método ***rehash()***. Un ***load factor*** más grande apura más la memoria, pero será menos eficiente en las búsquedas. Es conveniente partir de una ***hashtable*** suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de ***hashtable***:

```
import java.util.Hashtable;
class EjHT {
        static public void main() {
                Hashtable h = new Hashtable();
                String s;
                h.put("casa", “home”);
                h.put("copa", “cup”);
                h.put("lápiz", “pen”);
                s=(String) h.get(“copa”);
                System.out.println(h.get(“copa”));
                System.out.println(h);
        }
}
```

donde se ha hecho uso del método ***put()***. La tabla siguiente muestra los métodos de la clase ***Hashtable***.

Métodos	**Función que realizan**
Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)	Constructores
int size()	Devuelve el tamaño de la tabla
boolean isEmpty()	Indica si la tabla está vacía
Enumeration keys()	Devuelve una Enumeration con las claves
Enumeration elements()	Devuelve una Enumeration con los valores
boolean contains(Object value)	Indica si hay alguna clave que se corresponde con el valor
boolean containsKey(Object key)	Indica si existe esa clave en la tabla
Object get(Object key)	Devuelve un valor dada la clave
void rehash()	Amplía la capacidad de la tabla
Object put(Object key, Object value)	Establece una relación clave-valor
Object remove(Object key)	Elimina un valor por la clave
void clear()	Limpia la tabla
Object clone()	Hace una copia de la tabla
String toString()	Devuelve un string representando la tabla

Java 2 ha añadido muchas clases e interfaces nuevas para tratar colecciones de datos. En la documentación del package ***java.util*** se puede encontrar mucha más información al respecto.

6.4.2.- La clase StringTokenizer. Permite la realización de un pequeño analizador para una cadena de caracteres dada. En el constructor se proporcionan la cadena y opcionalmente, los delimitadores. Luego se maneja a través de los métodos.

- *boolean hasMoreTokens()*
- *String nextToken()*

Ejemplo:

```
import java.util.StringTokenizer;

class ST {
    static public void main (String arg []) {
        StringTokenizer st=new StringTokenizer("Esto es una prueba de
        StringTokenizer");
        while ( st.hasMOreTokens())
            System.out.println(st.nextToken());}}
```

6.4.3.- Otras clases del package java.util. El package ***java.util*** tiene otras clases interesantes para aplicaciones de distinto tipo, entre ellas:

1. ***Date***, que representa un instante de tiempo dado con precisión de milisegundos. Permite obtener información del año, mes, día, horas, minutos y segundos.

2. ***DateFormat*** (***Java 2***, package ***java.text***), que permite dar formato a fechas y horas de acuerdo con distintos criterios locales. Permite también convertir ***Strings*** representando fechas y horas en objetos de la clase ***Date***.

3. ***GregorianCalendar***, que maneja fechas según dos eras, antes y después del ***Nacimiento de Jesucristo***.

4. ***Random,*** que permite generar números aleatorios de diversas formas:

   - ***float nextFloat()***
   - ***double nextDouble()***
   - ***double nextGaussian()***

6.5.- Ejercicios propuestos.

1. Utilizando la clase *Vector* construid la clase *Lista* con la siguiente interface:

   - *Lista*(). Constructor de la clase Lista. Crea una lista vacía.

- *void ponPrimero(Object ob)*. Crea un nuevo nodo del tipo NodoLista, donde almacena *ob,* y lo coloca como primer nodo de nuestra lista.
- *void ponUltimo(Object ob)*. Crea un nuevo nodo del tipo NodoLista, donde almacena *ob*, y lo coloca como último nodo de nuestra lista.
- *boolean estaVacia().* Devuelve verdadero si la lista está vacía y falso en caso contrario.
- *Object extraePrimero().* Devuelve el primer elemento almacenado en la lista, eliminándolo de esta.
- *Object extraeUltimo().* Devuelve el último elemento almacenado en la lista, eliminándolo de esta.
- *void visualizaElementos.* Visualiza todos los elemntos almacenados en la lista.

Haced un programa para comprobar el buen funcionamiento de dicha clase.

2. Utilizando la clase *Lista*, anteriormente creada, construid la clase *Pila* con la siguiente interface: *pop*(), *push(Object ob), vacia() y cima().* Haced un programa para comprobar el buen funcionamiento de dicha clase.

3. Utilizando la clase *Lista*, anteriormente creada, construid la clase *Cola* con la siguiente interface: *extraeDeCola(), ponEnCola(Object ob), vacia() y frente().* Haced un programa para comprobar el buen funcionamiento de dicha clase.

4. Cread la clae carácter con los dos métodos estáticos siguientes:

   - int ordinal(char c).
   - char ascii(int i).

5. Cread la clase Cadena de forma que nos permita almacenar una cadena de caracteres, con al menos la siguiente interface:

   - inviertecadena(). Invierte la cadena miembro.
   - encriptaCadena(). Encripta la cadena miembro.
   - desencriptaCadena(). Desencripta la cadena miembro.
   - visualizaCadena(). Visualiza la cadena miembro.

UNIDAD DIDACTICA 3ª. Profundizando en JAVA.

TEMA 7: CONSTRUCCION DE GUI EN JAVA.

7.1 EL AWT (ABSTRACT WINDOWS TOOLKIT). El AWT (Abstract Windows Toolkit) es la parte de ***Java*** que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en ***Java*** desde la versión 1.0, la versión 1.1representó un cambio notable, sobre todo en lo que respecta al ***modelo de eventos***. La versión 1.2 ha incorporado un modelo distinto de componentes llamado ***Swing***, que también está disponible en la versión 1.1 como package adicional. En este Capítulo se seguirá el AWT de ***Java 1.1***, también soportado por la versión 1.2.

Para construir una interface gráfica de usuario hace falta:

1. Un "contenedor" o ***Container***, que es la ventana o parte de la ventana donde se situarán los componentes (botones, barras de desplazamiento, etc.) y donde se realizarán los dibujos. Se correspondería con un ***formulario*** o una ***picture box*** de ***Visual Basic***.

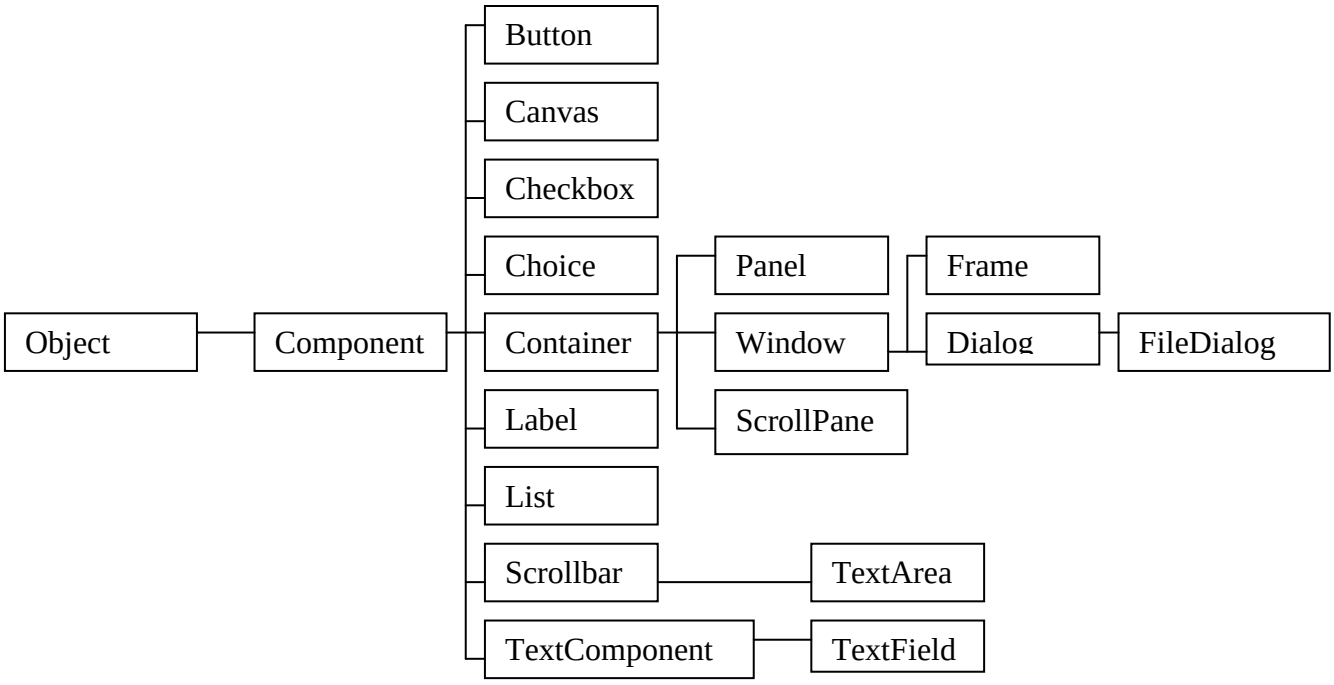
2. Los ***componentes***: menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc. Se corresponderían con los ***controles*** de ***Visual Basic***.

3. El ***modelo de eventos***. El usuario controla la aplicación actuando sobre los componentes, de ordinario con el ratón o con el teclado. Cada vez que el usuario realiza una determinada acción, se produce el ***evento*** correspondiente, que el sistema Operativo transmite al AWT. El AWT crea un ***objeto*** de una determinada clase de evento, derivada de ***AWTEvent***. Este ***evento*** es transmitido a un determinado ***método*** para que lo gestione. En ***Visual Basic*** el entorno de desarrollo crea automáticamente el procedimiento que va a gestionar el evento (uniendo el nombre del control con el tipo del evento mediante el carácter _) y el usuario no tiene más que introducir el código. En ***Java*** esto es un poco más complicado: el componente u objeto que recibe el evento debe "registrar" o indicar previamente qué objeto se va a hacer cargo de gestionar ese evento.

En los siguientes apartados se verán con un cierto detalle estos tres aspectos del AWT. Hay que considerar que el AWT es una parte muy extensa y complicada de ***Java***, sobre la que existen libros con muchos cientos de páginas.

Las capacidades gráficas del AWT resultan pobres y complicadas en comparación con lo que se puede conseguir con ***Visual Basic***, pero tienen la ventaja de poder ser ejecutadas casi en cualquier ordenador y con cualquier sistema operativo.

7.2.- Componentes soportados por el AWT de Java. Como todas las clases de ***Java***, los componentes utilizados en el AWT pertenecen a una determinada jerarquía de clases, que es muy importante conocer. Esta jerarquía de clases se muestra en la siguiente figura. Todos los componentes descienden de la clase ***Component***, de la que pueden ya heredar algunos métodos interesantes. El ***package*** al que pertenecen estas clases se llama ***java.awt***.



A continuación se resumen algunas características importantes de los componentes mostrados en la figura anterior:

1. Todos los ***Components*** (excepto ***Window*** y los que derivan de ella) deben ser añadidos a un ***Container***. También un ***Container*** puede ser añadido a otro C***ontainer***.

2. Para añadir un ***Component*** a un ***Container*** se utiliza el método ***add()*** de la clase ***Container***:

   containerName.add(componentName);

3. Los ***Containers*** de máximo nivel son las ***Windows*** (***Frames*** y ***Dialogs***). Los ***Panels*** y ***ScrollPanes*** deben estar siempre dentro de otro ***Container***.

4. Un ***Component*** sólo puede estar en un ***Container***. Si está en un ***Container*** y se añade a otro, deja de estar en el primero.

5. La clase ***Component*** tiene una serie de funcionalidades básicas comunes (variables y métodos) que son heredadas por todas sus ***sub-clases***.

7.2.1.- Clase Component. La clase ***Component*** es una clase ***abstract*** de la que derivan todas las clases del AWT, según el diagrama mostrado previamente. Los métodos de esta clase son importantes porque son heredados por todos los componentes del AWT. La siguiente tabla muestra algunos de los métodos más utilizados de la clase ***Component***. En las declaraciones de los métodos de dicha clase aparecen las clases ***Point***, ***Dimension*** y ***Rectangle***. La clase ***java.awt.Point*** tiene dos variables miembro *int* llamadas ***x*** e ***y***. La clase ***java.awt.Dimension*** tiene dos variables miembro *int*: ***height*** y ***width***. La

clase ***java.awt.Rectangle*** tiene cuatro variables *int*: ***height***, ***width***, ***x*** e ***y***. Las tres son sub-clases de ***Object***.

Métodos de Component	**Función que realizan**
boolean isVisible(), void setVisible(boolean)	Permiten chequear o establecer la visibilidad de un componente
boolean isShowing()	Permiten saber si un componente se está viendo. Para ello tanto el componente debe ser visible, y su container debe estar mostrándose
boolean is Enabled(), void setEnabled(boolean)	Permiten saber si un componente está activado y activarlo o desactivarlo
Point getLocation(), Point getLocationScreen()	Permiten obtener la posición de la esquina superior izquierda de un componente
void setLocation(Point), void setLocation(int x, int y)	Desplazan un componente a la posición especificada respecto al container o respecto a la pantalla
Dimension getSize(), void setSize(int w, int h), void setSize(Dimension d)	Permiten obtener o establecer el tamaño de un componente
Rectangle getBounds(), void setBounds(Rectangle), void setBounds(int x, int y, int width, int height)	Obtienen o establecen la posición y el tamaño de un componente
Invalidate(), validate() invalidate()	marca un componente y sus contenedores para indicar que se necesita volver a aplicar el Layout Manager.
validate()	se asegura que el Layout Manager está bien aplicado
setBackground(Color), setForeground(Color)	Métodos para establecer los colores por defecto
paint(Graphics), repaint() y update(Graphics)	Métodos gráficos para dibujar en la pantalla

Además de los métodos mostrados en esta tabla , la clase ***Component*** tiene un gran número de métodos básicos cuya funcionalidad puede estudiarse mediante la documentación on-line de ***Java***. Entre otras funciones, permite controlar los ***colores***, las ***fonts*** y los ***cursores***.

7.2.2.- Clase Container. La clase ***Container*** es también una clase muy general. De ordinario, nunca se crea un objeto de esta clase, pero los métodos de esta clase son heredados por las clases ***Frame*** y ***Panel***, que sí se utilizan con mucha frecuencia para crear objetos.

Los containers mantienen una ***lista de los objetos*** que se les han ido añadiendo. Cuando se añade un nuevo objeto se incorpora al final de la lista, salvo que se especifique una posición determinada. En esta clase tiene mucha importancia todo lo que tiene que ver con los ***Layout Managers***, que se explicarán más adelante. La siguiente tabla muestra algunos métodos de la clase ***Container***.

Métodos de Container	**Función que realizan**
add()	Añade un componente al container
doLayout()	Ejecuta el algoritmo de ordenación del layout manager
getComponent(int)	Obtiene el n-ésimo componente en el container
getComponentAt(int, int), getComponentAt(Point)	Obtiene el componente que contine un determinado punto
getComponentCount()	Obtiene el número de componentes en el container
getComponents()	Obtiene los componentes en este container.
remove(Component), remove(int), removeAll()	Elimina el componente especificado.
setLayout(LayoutManager)	Determina el layout manager para este container

7.3.- Contrucción de un GUI. Para la construcción de una interface gráfica de usuario, deberemos dar los siguientes pasos:

I. Crear un contenedor.
II. Seleccionar un *gestor de esquemas* para la inserción de los componentes.
III. Crear los componentes adecuados.
IV. Agregarlos al contenedor.
V. Dimensionar el contenedor(opcional):
VI. Pedir el ajuste de los componentes al contenedor.
VII. Mostrar el contenedor.

7.3.1.-Crear un contenedor. Hay dos clases de contenedores:

- *Frame*. Ventana de nivel superior con bordes y título, cuyos métodos principales son:

  pack(), setTitle(), getTitle(), setIconImage().

- *Panel*. No puede utilizarse de forma aislada. Necesita estar contenida en otro contenedor.

Un contenedor puede contener, a su vez, a otros contenedores o a otros componentes mediante la utilización del método *add*() de la clase Component.

Un *Frame*(ventana*)* puede contener a varios paneles y componentes. Un *Panel* puede contener a otros paneles y componentes.

7.3.2.-Seleccionar un gestor de esquemas. Un ***Layout Managers*** (Gestor de esquemas) es un objeto que controla cómo los ***Components*** (componentes) se situan en un ***Container*** (contenedor). Determinan, por tanto, como encajan los componentes dentro de los contenedores.

Cada contenedor tiene su propio gestor de esquemas por defecto. Por defecto, un *Frame* tiene un *BorderLayout* y un *Panel* tiene un *FlowLayout*.

Los gestores existentes son: *FlowLayout*, *BorderLayout*, *GridLayout*, *GridBagLayout* y *CardLayout*.

Para asignar un gestor de esquemas a un Container se utiliza el método:

setLayout(Layout Managers)

por ejemplo,

```
setLayout(new FlowLayout());
```

7.3.3.-Crear componentes. Cada componente es una clase que se debe instanciar, creando un objeto de dicha clase. Por ejemplo:

```
Button bSi = new Button("SI");
Label l = new Label("Nombre");
```

7.3.4.-Agregar componentes al contenedor. Se hace a través del método add() de la clase Container. Por ejemplo:

```
Frame f = new Frame("Ejemplo de GUI");
f.setLayout(new FlowLayout());
Button bSi = new Button("SI");
Button bNo = new Button("NO");
Label l = new Label("Nombre");
f.add(l);
f.add(bSi);
f.add(bNo);
```

7.3.5.-Dimensionar el contenedor. Es opcional. Si no se indica, el ajuste se hace para que quepan todos los componentes (excepto para la clase Canvas).

El método a llamar es setSize() de la clase Component(). Por ejemplo:

```
f.setSize(int anchura, int altura);
```

7.3.6.-Ajuste del contenedor. Calcula la posición de los componentes teniendo en cuenta:

- El gestor de esquemas seleccionado.
- El número y orden de los componentes añadidos.
- La dimensión dada o calculada.

El método a llamar es *pack*() de la clase Container. Por ejemplo:

```
f.pack();
```

7.3.7.-Mostrar el contenedor. Para hacer visible o invisible un contenedor se utiliza el método setVisible(boolean) de la clase Component. Este método es válido para mostrar u ocultar componentes y contenedores. Por ejemplo:

```
f.setVisible(true);
```

7.3.8.-Ejemplo de GUI.

```
import java.awt.*;
class GUI01 {
        public static void main(String args []) {
                Frame f = new Frame("Ejemplo de GUI");
                f.setLayout(new FlowLayout());
                Button bSi = new Button("SI");
                Button bNo = new Button("NO");
                Label l = new Label("Nombre");
                f.add(l);
                f.add(bSi);
                f.add(bNo);
                f.pack();
                f.setVisible(true);
        }
}
```

7.4.-Gestores de esquemas. La portabilidad de *Java* a distintas plataformas y distintos sistemas operativos necesita flexibilidad a la hora de situar los *Components* (*Buttons*, *Canvas*, *TextAreas*, etc.) en un *Container* (*Window*, *Panel*, …). Un *Layout Manager*(gestor de esquemas) es un objeto que controla cómo los *Components* se sitúan en un *Container*.

El AWT define cinco ***Layout Managers***: dos muy sencillos (***FlowLayout*** y G***ridLayout***), dos más especializados (***BorderLayout*** y ***CardLayout***) y uno muy general (***GridBagLayout***). Además, los usuarios pueden escribir su propio ***Layout Manager***, implementando la interface ***LayoutManager***, que especifica 5 métodos. ***Java*** permite también posicionar los ***Components*** de ***modo absoluto***, sin ***Layout Manager***, pero de ordinario puede perderse la portabilidad y algunas otras características.

Todos los ***Containers*** tienen un ***Layout Manager*** por defecto, que se utiliza si no se indica otra cosa: Para ***Panel***, el defecto es un objeto de la clase ***FlowLayout***. Para ***Window*** (***Frame*** y ***Dialog***), el defecto es un objeto de la clase ***BorderLayout***.

Se debe elegir el ***Layout Manager*** que mejor se adecúe a las necesidades de la aplicación que se desea desarrollar. Recuérdese que cada ***Container*** tiene un ***Layout Manager*** por defecto. Si se desea

utilizar el ***Layout Manager*** por defecto basta crear el ***Container*** (su constructor crea un objeto del ***Layout Manager*** por defecto e inicializa el ***Container*** para hacer uso de él).

Para utilizar un ***Layout Manager*** diferente hay que crear un objeto de dicho ***Layout Manager*** y pasárselo al constructor del container o decirle a dicho container que lo utilice por medio del método ***setLayout()***, en la forma:

unContainer.setLayout(new GridLayout());

La clase ***Container*** dispone de métodos para manejar el ***Layout Manager*** (ver Tabla siguiente).

Si se cambia de modo indirecto el tamaño de un ***Component*** (por ejemplo cambiando el tamaño del ***Font***), hay que llamar al método ***invalidate()*** del ***Component*** y luego al método ***validate()*** del ***Container***, lo que hace que se ejecute el método ***doLayout()*** para reajustar el espacio disponible.

Métodos de Container para manejar Layout Managers	**Función que realizan**
add()	Permite añadir Components a un Container
remove() y removeAll()	Permiten eliminar Components de un Container
doLayout(), validate() doLayout()	se llama automáticamente cada vez que hay que redibujar el Container y sus Components. Se llama también cuando el usuario llama al método validate()

7.4.1.-FlowLayout. ***FlowLayout*** es el ***Layout Manager*** por defecto para ***Panel***. ***FlowLayout*** coloca los componentes en una fila, de izquierda a derecha y de arriba a abajo, en la misma forma en que procede un procesador de texto. Los componentes se añaden en el mismo orden en que se ejecutan los métodos ***add()***. Si se cambia el tamaño de la ventana los componentes se redistribuyen de modo acorde, ocupando más filas si es necesario.

La clase ***FlowLayout*** tiene tres constructores:

FlowLayout();
FlowLayout(int alignement);
FlowLayout(int alignement, int horizontalGap, int verticalGap);

Se puede establecer la alineación de los componentes (centrados, por defecto), por medio de las constantes FlowLayout.LEFT, FlowLayout.CENTER y FlowLayout.RIGHT.

Es posible también establecer una distancia horizontal y vertical entre componentes (el ***gap***, en pixels). El valor por defecto son 5 pixels.

7.4.2.-BorderLayout. ***BorderLayout*** es el ***Layout Manager*** por defecto para ***Windows*** y ***Frames***. ***BorderLayout*** define cinco áreas: ***North***, ***South***, ***East***, ***West*** y ***Center***. Si se aumenta el tamaño de la

ventana todas las zonas se mantienen en su mínimo tamaño posible excepto ***Center***, que absorbe casi todo el crecimiento. Los componentes añadidos en cada zona tratan de ocupar todo el espacio disponible. Por ejemplo, si se añade un botón, el botón se hará tan grande como la celda, lo cual puede producir efectos muy extraños. Para evitar esto se puede introducir en la celda un panel con ***FlowLayout*** y añadir el botón al panel y el panel a la celda.

Los constructores de ***BorderLayout*** son los siguientes:

BorderLayout();
BorderLayout(int horizontalGap, int verticalGap);

Por defecto ***BorderLayout*** no deja espacio entre componentes. Al añadir un componente a un ***Container*** con ***BorderLayout*** se debe especificar la zona como primer argumento:

miContainer.add("North", new Button("Norte"));

Ejercicio. Construir el siguiente GUI:

7.4.3.- GridLayout.

Con ***GridLayout*** las componentes se colocan en una matriz de celdas. Todas las celdas tienen el mismo tamaño. Cada componente utiliza todo el espacio disponible en su celda, al igual que en ***BorderLayout***.

GridLayout tiene dos constructores:

GridLayout(int nfil, int ncol);
GridLayout(int nfil, int ncol, int horizontalGap, int verticalGap);

Al menos uno de los parámetros ***nfil*** y ***ncol*** debe ser distinto de cero. El valor por defecto para el espacio entre filas y columnas es cero pixels.

7.4.4.- CardLayout.

CardLayout permite disponer distintos componentes (de ordinario ***Panels***) que comparten la misma ventana para ser mostrados sucesivamente. Son como transparencias, diapositivas o cartas de baraja que van apareciendo una detrás de otra.

El ***orden*** de las "cartas" se puede establecer de los siguientes modos:

1. Yendo a la primera o a la última, de acuerdo con el orden en que fueron añadidas al container.

2. Recorriendo las cartas hacia delante o hacia atrás, de una en una.

3. Mostrando una carta con un nombre determinado.

Los ***constructores*** de esta clase son:

CardLayout()
CardLayout(int horizGap, int vertGap)

Para añadir componentes a un container con ***CardLayout*** se utiliza el método:

Container.add(Component comp, int index)

donde ***index*** indica la posición en que hay que insertar la carta. Los siguientes métodos de ***CardLayout*** permiten controlar el orden en que aparecen las cartas:

void first(Container cont);
void last(Container cont);
void previous(Container cont);
void next(Container cont);
void show(Container cont, String nameCard);

7.4.5.- GridBagLayout. El ***GridBagLayout*** es el ***Layout Manager*** más completo y flexible, aunque también el más complicado de entender y de manejar. Al igual que el ***GridLayout***, el ***GridBagLayout*** parte de una matriz de celdas en la que se sitúan los componentes. La diferencia está en que las filas pueden tener distinta altura, las columnas pueden tener distinta anchura, y además en el ***GridBagLayout*** un componente puede ocupar varias celdas contiguas.

La posición y el tamaño de cada componente se especifican por medio de unas "restricciones" o ***constraints***. Las restricciones se establecen creando un objeto de la clase ***GridBagConstraints***, dando valor a sus propiedades (variables miembro) y asociando ese objeto con el ***GridBagLayout*** por medio del método ***setConstraints()***.

Las ***variables miembro*** de ***GridBagConstraints*** son las siguientes:

- ***gridx*** y ***gridy***. Especifican la fila y la columna en la que situar la esquina superior izquierda del componente (se empieza a contar de cero). Con la constante GridBagConstraints.RELATIVE se indica que el componente se sitúa relativamente al anterior componente situado (es la condición por defecto).

- ***gridwidth*** y ***gridheight***. Determinan el número de columnas y de filas que va a ocupar el componente. El valor por defecto es una columna y una fila. La constante GridBagConstraints.REMAINDER indica que el componente es el último de la columna o de la

fila, mientras que GridBagConstraints.RELATIVE indica que el componente es el penúltimo de la fila o columna.

- ***fill***. En el caso en que el componente sea más pequeño que el espacio reservado, esta variable indica si debe ocupar o no todo el espacio disponible. Los posibles valores son: GridBagConstraints.NONE (no lo ocupa; defecto), GridBagConstraints.HORIZONTAL (lo ocupa en dirección horizontal), GridBagConstraints.VERTICAL (lo ocupa en vertical) y GridBagConstraints.BOTH (lo ocupa en ambas direcciones).

- ***ipadx*** y ***ipady***. Especifican el espacio a añadir en cada dirección al tamaño interno del componente. Los valores por defecto son cero. El tamaño del componente será el tamaño mínimo más dos veces el ***ipadx*** o el ***ipady***.

- ***insets***. Indican el espacio mínimo entre el componente y el espacio disponible. Se establece con un objeto de la clase ***java.awt.Insets***. Por defecto es cero.

- ***anchor***. Se utiliza para determinar dónde se coloca el componente, cuando éste es menor que el espacio disponible. Sus posibles valores vienen dados por las constantes de la clase ***GridBagConstraints***: CENTER (el valor por defecto), NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST y NORTHWEST.

- ***weightx*** y ***weighty***. Son unos coeficientes entre 0.0 y 1.0 que sirven para dar más o menos "peso" a las distintas filas y columnas. A más "peso" más probabilidades tienen de que se les dé más anchura o más altura.

A continuación se muestra una forma típica de crear un container con ***GridBagLayout*** y de añadirle componentes:

```
GridBagLayout unGBL = new GridBagLayout();
GridBagConstraints unasConstr = new GridBagConstraints();
unContainer.setLayout(unGBL);
// Ahora ya se pueden añadir los componentes
//...Se crea un componente unComp
//...Se da valor a las variables del objeto unasConstr
// Se asocian las restricciones con el componente
unGBL.setConstraints(unComp, unasConstr);
// Se añade el componente al container
unContainer.add(unComp);
```

Ejercicio. Construir el siguiente GUI:

7.5.-Componentes.

7.5.1.-Clase Panel. Un ***Panel*** es un ***Container*** de propósito general. Se puede utilizar tal cual para contener otras componentes, y también crear una sub-clase para alguna finalidad más específica. Por defecto, el ***Layout Manager*** de ***Panel*** es ***FlowLayout***. Los ***Applets*** son sub-clases de ***Panel***. La siguiente tabla muestra los métodos más importantes que se utilizan con la clase ***Panel***, que son algunos métodos heredados de ***Component*** y ***Container***, pues la clase ***Panel*** no tiene métodos propios.

Métodos de Panel	**Función que realiza**
Panel(), Panel(LayoutManager miLM)	Constructores de Panel
Métodos heredados de Container y Component	
add(Component), add(Component, int)	Añade componentes al panel
setLayout(LayoutManager lm), getLayout()	Establece o permite obtener el layout manager utilizado
validate(), doLayout()	Para reorganizar los componentes después de algún cambio. Es mejor utilizar validate()
remove(int), remove(Component), removeAll()	Para eliminar componentes
getMaximumSize(), gettMinimumSize(), getPreferredSize()	Permite obtener los tamaños máximo, mínimo y preferido
Insets getInsets()	

Un ***Panel*** puede contener otros ***Panel***. Esto es una gran ventaja respecto a los demás tipos de containers, que son containers de máximo nivel y no pueden introducirse en otros containers.

Insets es una clase que deriva de ***Object***. Sus variables son ***top***, ***left***, ***botton***, ***right***. Representa el *espacio que se deja libre* en los bordes de un ***Container***. Se establece mediante la llamada al adecuado constructor del ***Layout Manager***.

Ejemplo:

```
Panel p = new Panel();
```

7.5.2.- Clase Button. El aspecto de un ***Button*** depende de la plataforma (PC, Mac, Unix), pero la funcionalidad siempre es la misma. Se puede cambiar el ***texto*** y la ***font*** que aparecen en el ***Button***, así

como el ***foreground*** y ***backgroung color***. También se puede establecer que esté activado o no. La siguienteTabla muestra los métodos más importantes de la clase ***Button***.

Métodos de la clase Button	**Función que realiza**
Button(String label) y Button()	Constructores
setLabel(String str), String getLabel()	Permite establecer u obtener la etiqueta del Button
addActionListener(ActionListener al), removeActionListener(ActionListener al)	Permite registrar el objeto que gestionará los eventos, que deberá implementar ActionListener
setActionCommand(String cmd), String getActionCommand()	Establece y recupera un nombre para el objeto Button independiente del label y del idioma

Ejemplo: Button b1 = new Button("Texto");

7.5.3.-Clases Checkbox y CheckboxGroup. Los objetos de la clase c***heckbox*** son ***botones de opción o de selección*** con dos posibles valores: ***on*** y ***off***. Al cambiar la selección de un ***Checkbox*** se produce un ***ItemEvent***.

La clase ***CheckboxGroup*** permite la opción de agrupar varios ***Checkbox*** de modo que uno y sólo uno esté en ***on*** (al comienzo puede que todos estén en ***off***).Se corresponde con los botones de opción de Visual Basic. La siguiente tabla muestra los métodos más importantes de estas clases.

Métodos de Checkbox	**Función que realizan**
Checkbox(), Checkbox(String), Checkbox(String, boolean), Checkbox(String, boolean, CheckboxGroup), Checkbox(String, CheckboxGroup, boolean)	Constructores de Checkbox. Algunos permiten establecer la etiqueta, si está o no seleccionado y si pertenece a un grupo
addItemListener(ItemListener), removeItemListener(ItemListener)	Registra o elimina los objetos que gestionarán los eventos ItemEvent
setLabel(String), String getLabel()	Establece u obtiene la etiqueta del componente
setState(boolean), boolean getState()	Establece u obtiene el estado (true o false, según esté seleccionado o no)
setCheckboxGroup(CheckboxGroup), CheckboxGroup getCheckboxGroup()	Establece u obtiene el grupo al que pertenece el Checkbox
Métodos de CheckboxGroup	**Función que realizan**
CheckboxGroup()	Constructores de CheckboxGroup
Checkbox getSelectedCheckbox(), setSelectedCheckbox(Checkbox box)	Obtiene o establece el componente seleccionado de un grupo

Cuando el usuario actúa sobre un objeto ***Checkbox*** se ejecuta el método ***itemStateChanged()***, que es el único método de la interface ***ItemListener***. Si hay varias ***checkboxes*** cuyos eventos se gestionan en

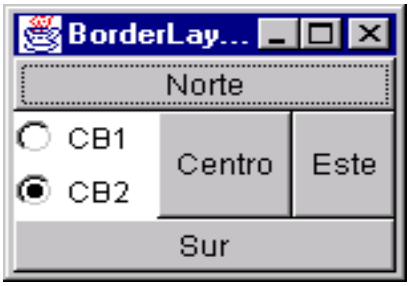
un mismo objeto y se quiere saber cuál es la que ha recibido el evento, se puede utilizar el método ***getSource()*** del evento ***ItemEvent***. También se puede utilizar el método ***getItem()*** de ***ItemEvent***, cuyo valor de retorno es un ***Object*** que contiene el ***label*** del componente (para convertirlo a ***String*** habría que hacer un ***cast***).

Para crear un ***grupo*** o conjunto de botones de opción (de forma que uno y sólo uno pueda estar activado), se debe crear un objeto de la clase ***CheckboxGroup***. Este objeto no tiene datos: simplemente sirve como identificador del grupo. Cuando se crean los objetos ***Checkbox*** se pasa a los ***constructores*** el objeto ***CheckboxGroup*** del grupo al que se quiere que pertenezcan.

Cuando se selecciona un ***Checkbox*** de un grupo se producen dos eventos: uno por el elemento que se ha seleccionado y otro por haber perdido la selección el elemento que estaba seleccionado anteriormente. Al hablar de evento ***ItemEvent*** y del método ***itemStateChanged()*** se verán métodos para determinar los ***checkboxes*** que han sido seleccionados o que han perdido la selección.

```
Ejemplo:  Checkbox cb = new Checkbox("Mostrar", true);
          CheckboxGroup cbg = new CheckboxGroup();
          Checkbox cb1 = new Checkbox("Hombres", cbg, true);
          Checkbox cb2 = new Checkbox("Mujeres", cbg, true);
```

Ejercicio. Construir la siguiente interface gráfica de usuario:



7.5.4.-Clase Choice.

La clase ***Choice*** permite elegir un item de una lista desplegable. Los objetos ***Choice*** ocupan menos espacio en pantalla que los ***Checkbox***. Al elegir un item se genera un ***ItemEvent***. Un ***index*** permite determinar un elemento de la lista (se empieza a contar desde 0). La siguiente tabla muestra los métodos más habituales de esta clase.

Métodos de Choice	**Función que realizan**
Choice()	Constructor de Choice
addItemListener(ItemListener), removeItemListener(ItemListener)	Establece o elimina un ItemListener
add(String), addItem(String)	Añade un elemento a la lista
insert(String label, int index)	Inserta un elemento con un label an la posición incicada

int getSelectedIndex(), String getSelectedItem()	Obtiene el index o el label de un elemento de la lista
int getItemCount()	Obtiene el número de elementos
String getItem(int)	Obtiene el label a partir del index
select(int), select(String)	Selecciona un elemento por el index o el label
removeAll(), remove(int), remove(String)	Elimina todos o uno de los elementos de la lista

La clase ***Choice*** genera el evento ***ItemEvent*** al seleccionar un item de la lista. En este sentido es similar a las clases ***Checkbox*** y ***CheckboxGroup***, así como a los menús de selección.

Ejemplo: Choice c = new Choice();
c.add("Elección 1");
c.add("Elección 2");
c.add("Elección 3");

7.5.5.-Clase Canvas. Una ***Canvas*** es una zona rectangular de pantalla en la que se puede dibujar y en la que se pueden generar eventos. Las ***Canvas*** permiten realizar dibujos, mostrar imágenes y crear componentes a medida, de modo que muestren un aspecto similar en todas las plataformas. La siguiente tabla muestra los métodos de la clase ***Canvas***.

Métodos de Canvas	**Función que realiza**
Canvas()	Es el único constructor de esta clase
void paint(Graphics g);	Dibuja un rectángulo con el color de background. Lo normal es que las sub-clases de Canvas redefinan este método.

Desde los objetos de la clase ***Canvas*** se puede llamar a los métodos ***paint()*** y ***repaint()*** de la super-clase ***Component***. Con frecuencia conviene redefinir los siguientes métodos de ***Component***: ***getPreferredSize()***, ***getMinimumSize()*** y ***getMaximumSize()***, que devuelven un objeto de la clase ***Dimension***. El ***LayoutManager*** se encarga de utilizar estos valores.

La clase ***Canvas*** no tiene eventos propios, pero puede recibir los eventos ***ComponentEvent*** de su super-clase ***Component***.

Ejemplo: Canvas c = new Canvas();

7.5.6.-Clase Label. La clase ***Label*** introduce en un container un ***texto no seleccionable y no editable***, que por defecto se alinea por la izquierda. La clase ***Label*** define las constantes Label.CENTER, Label.LEFT y Label.RIGHT para determinar la alineación del texto. La siguiente tabla muestra algunos métodos de esta clase.

Métodos de Label	**Función que realizan**
Label(String lbl), Label(String lbl, int align)	Constructores de Label

setAlignement(int align), int getAlignement()	Establecer u obtener la alineación del texto
setText(String txt), String getText()	Establecer u obtener el texto del Label

La elección del *font*, de los *colores*, del tamaño y posición de ***Label*** se realiza con los métodos heredados de la clase ***Component***: ***setFont(Font f)***, ***setForeground(Color)***, ***setBackground(Color)***, ***setSize(int, int)***, ***setLocation(int, int)***, ***setVisible(boolean)***, etc.

La clase ***Label*** no tiene más ***eventos*** que los de su super-clase ***Component***.

Ejemplo:

```
Label lb = new Label("Nombre del concursante");
Label lb2 = new Label ("Nombre", Label.LEFT);
```

7.5.7.-Clases TextArea y TextField. Ambas componentes heredan de la clase ***TextComponent*** y muestran texto seleccionable y editable. La diferencia principal es que ***TextField*** sólo puede tener una línea, mientras que ***TextArea*** puede tener varias líneas. Además, ***TextArea*** ofrece posibilidades de edición de texto adicionales.

Se pueden especificar el *font* y los *colores de foreground y background*. Ambas componentes generan ***ActionEvents***. La siguiente tabla muestra algunos métodos de las clases ***TextComponent***, ***TextField*** y ***TextArea***. No se pueden crear objetos de la clase ***TextComponent*** porque su ***constructor*** no es ***public***; por eso su constructor no aparece en dicha tabla.

Métodos heredados de TextComponent	**Función que realizan**
String getText() y setText(String str)	Permiten establecer u obtener el texto del componente
setEditable(boolean b), boolean isEditable()	Hace que el texto sea editable o pregunta por ello
setCaretPosition(int n), int getCaretPosition()	Fija la posición del punto de inserción o la obtiene
String getSelectedText(), int etSelectionStart() int getSelectionEnd()	Obtiene el texto seleccionado y el comienzo y el final de la selección
selectAll(), select(int start, int end)	Selecciona todo o parte del texto
Métodos de TextField	**Función que realizan**
TextField(), TextField(int ncol), TextField(String s),TextField(String s, int ncol)	Constructores de TextField
int getColumns()	Establece el número de columnas del TextField
setEchoChar(char c), char getEchoChar(), boolean echoCharIsSet()	Establece, obtiene o pregunta por el carácter utilizado para passwords, de forma que no se pueda leer lo tecleado por el usuario
Métodos de TextArea	**Función que realizan**
TextArea(), TextArea(int nfil, int ncol),	Constructores de TextArea

TextArea(Stringtext), TextArea(String text, int nfil, int ncol)	
setRows(int), setColumns(int), int getRows(), int getColumns()	Establecer y/u obtener los números de filas y de columnas
append(String str), insert(String str, int pos), replaceRange(String s, int i, int f)	Añadir texto al final, insertarlo en una posición determinada y reemplazar un texto determinado

La clase ***TextComponent*** recibe eventos ***TextEvent***, y por lo tanto también los reciben sus clases derivadas ***TextField*** y ***TextAreas***. Este evento se produce cada vez que se modifica el texto del componente. La caja ***TextField*** soporta también el evento ***ActionEvent***, que se produce cada vez que el usuario termina de editar la única línea de texto pulsando ***Intro***.

Como es natural, las cajas de texto pueden recibir también los eventos de sus ***super-clases***, y más en concreto los eventos de ***Component***: ***FocusEvent***, ***MouseEvent*** y sobre todo ***KeyEvent***. Estos eventos permiten capturar las teclas pulsadas por el usuario y tomar las medidas adecuadas. Por ejemplo, si el usuario debe teclear un número en un ***TextField***, se puede crear una función que vaya capturando los caracteres tecleados y que rechace los que no sean numéricos.

Cuando se cambia desde programa el número de filas y de columnas de un TextField o TextArea, hay que llamar al método validate() de la clase Component, para que vuelva a aplicar el LayoutManager correspondiente. De todas formas, los tamaños fijados por el usuario tienen el carácter de "recomendaciones" o tamaños "preferidos", que el LayoutManager puede cambiar si es necesario.

La clase TextArea define cuatro constantes: SCROLLBARS_BOTH, SCROLLBARS_VERTICAL_ONLY, SCROLLBARS_HORIZONTAL_ONLY y SCROLLBARS_NONE.

Ejemplo:

```
TextArea ta=new TextArea("texto",10,20,TextArea.SCROLLBARS_BOTH);
TextField tf=new TextField("texto", 30);
```

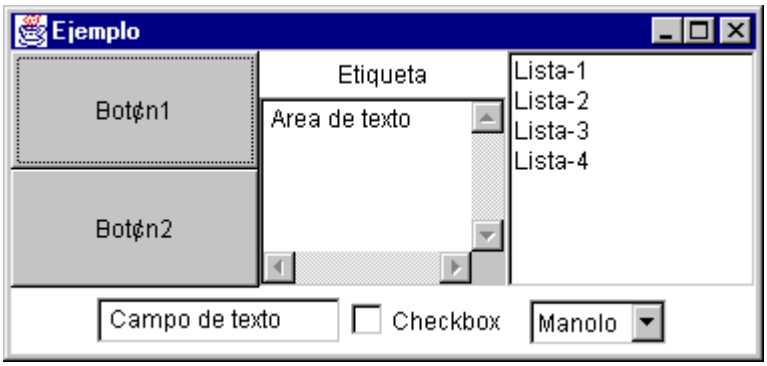
7.5.8.-Clase List. La clase ***List*** viene definida por una zona de pantalla con varias líneas, de las que se muestran sólo algunas, y entre las que se puede hacer una ***selección simple o múltiple***. Las ***List*** generan eventos de la clase ***ActionEvents*** (al ***clicar dos veces*** sobre un item o al pulsar ***return***) e ***ItemEvents*** (al seleccionar o deseleccionar un item). Al gestionar el evento ***ItemEvent*** se puede preguntar si el usuario estaba pulsando a la vez alguna tecla (***Alt***, ***Ctrl***, ***Shift***), por ejemplo para hacer una selección múltiple.

Las ***List*** se diferencian de las ***Choices*** en que muestran varios items a la vez y que permiten hacer selecciones múltiples. La siguiente tabla muestra los principales métodos de la clase ***List***.

Métodos de List	**Función que realiza**
List(), List(int nl), List(int nl, boolean mult)	Constructor: por defecto 1 línea y selección simple
add(String), add(String, int), addItem(String),	Añadir un item. Por defecto se añaden

addItem(String, int)	al final
addActionListener(ActionListener), addItemListener(ItemListener)	Registra los objetos que gestionarán los dos tipos de eventos soportados
insert(String, int)	Inserta un nuevo elemento en la lista
replaceItem(String, int)	Sustituye el item en posición int por el String
void delItem(int), remove(int), remove(String), removeAll()	Eliminar uno o todos los items de la lista
int getItemCount(), int getRows()	Obtener el número de items o el número de items visibles
String getItem(int), String[] getItems()	Obtiene uno o todos los elementos de la lista
int getSelectedIndex(), String getSelectedItem(), int [] getSelectedIndexes(), String[] getSelectedItems()	Obtiene el/los elementos seleccionados
void select(int), void deselect(int)	Selecciona o elimina la selección de un elemento
boolean isIndexSelected(int), boolean isItemSelected(String)	Indica si un elemento está seleccionado o no
boolean isMultipleMode(), setMultipleMode(boolean)	Pregunta o establece el modo de selección múltiple
int getVisibleIndex(), makeVisible(int)	Indicar o establecer si un item es visible

Ejercicio. Construir el siguiente GUI.



7.5.9.-Clase Dialog.

Un ***Dialog*** es una ventana que depende de otra ventana (de una ***Frame***). Si una ***Frame*** se cierra, se cierran también los ***Dialog*** que dependen de ella; si se iconifica, sus ***Dialog*** desaparecen; si se restablece, sus ***Dialog*** aparecen de nuevo. Este comportamiento se obtiene de forma automática.

Las ***Applets*** estándar no soportan ***Dialogs*** porque no son ***Frames*** de ***Java***. Las ***Applets*** que abren ***Frames*** sí pueden soportar ***Dialogs***.

Un ***Dialog modal*** require la atención inmediata del usuario: no se puede hacer ninguna otra cosa hasta no haber cerrado el ***Dialog***. Por defecto, los ***Dialogs*** son ***no modales***. La siguiente tabla muestra los métodos más importantes de la clase ***Dialog***. Se pueden utilizar también los métodos heredados de sus super-clases.

Métodos de Dialog	**Función que realiza**
Dialog(Frame fr), Dialog(Frame fr, boolean mod), Dialog(Frame fr, String title), Dialog(Frame fr, String title, boolean mod)	Constructores
String getTitle(), setTitle(String)	Permite obtener o determinar el título
boolean isModal(), setModal(boolean)	Pregunta o determina si el Dialog es modal o no
boolean isResizable(), setResizable(boolean)	Pregunta o determina si se puede cambiar el tamaño
show()	Muestra y trae a primer plano el Dialog

Ejemplo:

```
Frame f = new Frame();
Dialog d = new Dialog(f, "Ventana modal", true);
```

7.5.10.-Clase FileDialog. La clase ***FileDialog*** muestra una ventana de diálogo en la cual se puede seleccionar un fichero. Esta clase deriva de ***Dialog***. Las constantes enteras LOAD (abrir ficheros para lectura) y SAVE (abrir ficheros para escritura) definen el ***modo*** de apertura del fichero. La siguiente tabla muestra algunos métodos de esta clase.

Métodos de la clase FileDialog	**Función que realizan**
FileDialog(Frame parent), FileDialog(Frame parent, String title), FileDialog(Frame parent, String title, int mode)	Constructores
int getMode(), void setMode(int mode)	Modo de apertura (SAVE o LOAD)
String getDirectory(), String getFile()	Obtiene el directorio o fichero elegido
void setDirectory(String dir), void setFile(String file)	Determina el directorio o fichero elegido
FilenameFilter getFilenameFilter(), void setFilenameFilter(FilenameFilter filter)	Determina o establece el filtro para los ficheros

Las clases que implementan la interface ***java.io.FilenameFilter*** permiten filtrar los ficheros de un directorio. Para más información, ver la documentación on-line.

7.5.11.-Clase Scrollbar. Una ***Scrollbar*** es una barra de desplazamiento con un cursor que permite introducir y modificar valores, entre unos valores mínimo y máximo, con pequeños y grandes incrementos. Las ***Scrollbars*** de ***Java*** se utilizan tanto como "sliders" o barras de desplazamiento aisladas (al estilo de Visual Basic), como unidas a una ventana en posición vertical y/u horizontal para mostrar una cantidad de información superior a la que cabe en la ventana.

La clase ***Scrollbar*** tiene dos constantes, Scrollbar.HORIZONTAL y Scrollbar.VERTICAL, que indican la posición de la barra. El cambiar el valor de la ***Scrollbar*** produce un ***AdjustementEvent***. La siguiente tabla muestra algunos métodos de esta clase.

Métodos de Scrollbar	**Función que realizan**
Scrollbar(), Scrollbar(int pos), Scrollbar(int pos, int val, int vis, int min, int max)	Constructores de Scrollbar
addAdjustmentListener(AdjustmentListener)	registra el objeto que gestionará los eventos
int getValue(), setValue(int)	Permiten obtener y fijar el valor
setMaximum(int), setMinimum(int)	Establecen los valores máximo y mínimo
setVisibleAmount(int), int getVisibleAmount()	Establecen y obtienen el tamaño del área visible
setUnitIncrement(int), int getUnitIncrement()	Establecen y obtienen el incremento pequeño
setBlockIncrement(int), int getBlockIncrement()	Establecen y obtienen el incremento grande
setOrientation(int), int getOrientation()	Establecen y obtienen la orientación
setValues(int value, int vis, int min, int max)	Establecen los parámetros de la barra

En el constructor general, el parámetro ***pos*** es la posición de la barra (horizontal o vertical); el ***rango*** es el intervalo entre los valores mínimo ***min*** y máximo ***max***; el parámetro ***vis*** (de ***visibleAmount***) es el tamaño del área visible en el caso en que las ***Scrollbars*** se utilicen en ***TextAreas***. En ese caso, el tamaño del cursor representa la relación entre el ***área visible*** y el ***rango***, como es habitual en ***Netscape***, ***Word*** y tantas aplicaciones de ***Windows***. El valor seleccionado viene dado por la variable ***value***. Cuando ***value*** es igual a ***min*** el área visible comprende el inicio del rango; cuando ***value*** es igual a ***max*** el área visble comprende el final del ***rango***. Cuando la ***Scrollbar*** se va a utilizar aislada (como ***slider***), se debe hacer ***visibleAmount*** igual a cero.

Las variables ***Unit Increment*** y ***Block Increment*** representan los incrementos pequeño y grande, respectivamente. Por defecto, ***Unit Increment*** es "1" y ***Block Increment*** es "10", mientras que ***min*** es "0" y ***max*** es "100".

Cada vez que cambia el valor de una *Scrollbar* se genera un evento *AdjustementEvent* y se ejecuta el único método de la interface *AdjustmentListener*, que es *adjustmentValueChanged()*.

7.5.12.-Clase ScrollPane. Un ***ScrollPane*** es como una ventana de tamaño limitado en la que se puede mostrar un componente de mayor tamaño con dos ***Scrollbars***, una horizontal y otra vertical. El componente puede ser una imagen, por ejemplo. Las ***Scrollbars*** son visibles sólo si son necesarias (por defecto). Las constantes de la clase ***ScrollPane*** son (su significado es evidente): SCROLLBARS_AS_NEEDED, SCROLLBARS_ALWAYS, SCROLLBARS_NEVER. Los ***ScrollPanes*** no generan eventos. La siguiente tabla muestra algunos métodos de esta clase.

Métodos de ScrollPane	**Función que realizan**
ScrollPane(), ScrollPane(int scbs)	Constructores que pueden incluir las ctes.
Dimension getViewportSize(), int getHScrollbarHeight(), int getVScrollbarWidth()	Obtiene el tamaño del ScrollPane y la altura y anchura de las barras de desplazamiento
setScrollPosition(int x, int y), setScrollPosition(Point p), Point getScrollPosition()	Permiten establecer u obtener la posición del componente
setSize(int, int), add(Component)	Heredados de Container, permiten establecer el tamaño y añadir un componente

En el caso en que no aparezcan scrollbars (SCROLLBARS_NEVER) será necesario desplazar el componente (hacer ***scrolling***) desde programa, con el método ***setScrollPosition()***.

Ejercicio. Construir el siguiente GUI.

7.5.13.-Clase Window. Los objetos de la clase ***Window*** son ventanas de máximo nivel, pero ***sin bordes*** y ***sin barra de menús***. En realidad son más interesantes las clases que derivan de ella: ***Frame*** y ***Dialog***. Los métodos más útiles, por ser heredados por las clases ***Frame*** y ***Dialog***, se muestran en la siguiente tabla.

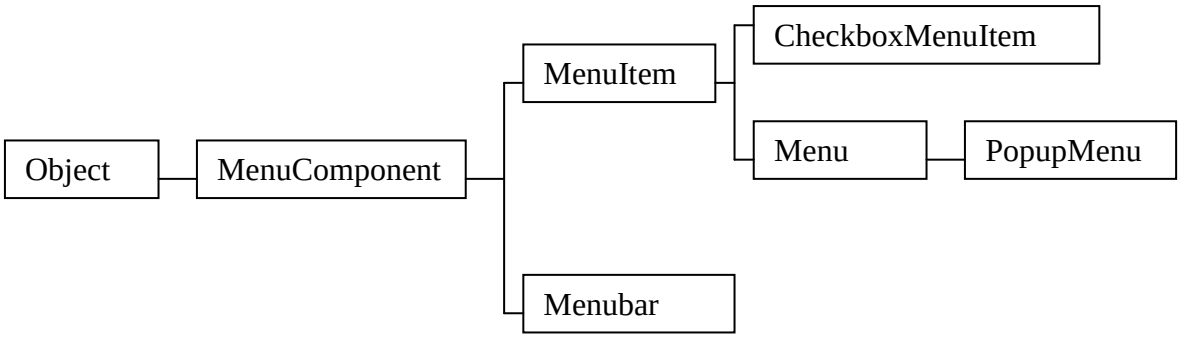
Métodos de Window	**Función que realizan**
toFront() y toBack()	Para desplazar la ventana hacia adelante y hacia atrás en la pantalla
show()	Muestra la ventana y la trae a primer plano
pack()	Hace que los componentes se reajusten al tamaño preferido

7.5.14.-Clase Frame. Es una ventana ***con un borde*** y que puede tener una ***barra de menús***. Si una ventana depende de otra ventana, es mejor utilizar una ***Window*** (ventana sin borde ni barra de menús) que un ***Frame***. La siguiente tabla muestra algunos métodos más utilizados de la clase ***Frame***.

Métodos de Frame	**Función que realiza**
Frame(), Frame(String title)	Constructores de Frame
String getTitle(), setTitle(String)	Obtienen o determinan el título de la ventana
MenuBar getMenuBar(), setMenuBar(MenuBar), remove(MenuComponent)	Permite obtener, establecer o eliminar la barra de menús
Image getIconImage(), setIconImage(Image)	Obtienen o determinan el icono que apareceré en la barra de títulos
setResizable(boolean), boolean isResizable()	Determinan o chequean si se puede cambiar el tamaño
dispose()	Método que libera los recursos utilizados en una ventana. Todos los componentes de la ventana son destruidos.

Además de los métodos citados, se utilizan mucho los métodos ***show()***, ***pack()***, ***toFront()*** y ***toBack()***, heredados de la super-clase ***Window***.

7.6.-MENUS. Los ***Menus*** de ***Java*** no descienden de ***Component***, sino de ***MenuComponent***, pero tienen un comportamiento similar, pues aceptan ***Events***. La siguiente figura muestra la jerarquía de clases de los ***Menus*** de ***Java 1.1***.



Para crear un ***Menú*** se debe crear primero una ***MenuBar***; después se crean los ***Menus*** y los***MenuItem***. Los ***MenuItems*** se añaden al ***Menu*** correspondiente; los ***Menus*** se añaden a la ***MenuBar*** y la ***MenuBar*** se añade a un ***Frame***.

También puede añadirse un ***Menu*** a otro **Menu** para crear un ***sub-menú***, del modo que es habitual en ***Windows***. La clase ***Menu*** es ***sub-clase*** de ***MenuItem***. Esto es así precisamente para permitir que un ***Menu*** sea añadido a otro ***Menu***.

7.6.1.-Clase MenuShortcut. La clase ***java.awt.MenuShortcut*** (derivada de ***Object***) representa las teclas aceleradoras que pueden utilizarse para activar los menús desde teclado, sin ayuda del ratón. Se establece un ***Shortcut*** con el método ***MenuShortcut(int VK_key)***, pasando un objeto ***MenuShortcut*** al constructor adecuado de ***MenuItem***. Para más información, consúltese la adocumentación on-line de ***Java***. La siguiente tabla muestra algunos métodos de esta clase. Los ***MenuShortcut*** de ***Java*** están restringidos al uso la tecla control (CTRL). Al definir el ***MenuShortcut*** no hace falta incluir dicha tecla.

Métodos de la clase MenuShortcut	**Función que realizan**
MenuShortcut(int key)	Constructor
int getKey()	Obtiene el código virtual de la tecla utilizada como shortcut

7.6.2.-Clase MenuBar. La siguiente tabla muestra algunos métodos de la clase ***MenuBar***. A una ***MenuBar*** sólo se pueden añadir objetos ***Menu***.

Métodos de MenuBar	**Función que realizan**
MenuBar()	Constructor
add(Menu), int getMenuCount(), Menu getMenu(int i)	Añade un menú, obtiene el número de menús y el menú en una posición determinada
MenuItem getShortcutMenuItem(MenuShortcut), deleteShortcut(MenuShortcut)	Obtiene el objeto MenuItem relacionado con un Shortcut y elimina el Shortcut especificado
remove(int index), remove(MenuComponent m)	Elimina un objeto Menu a partir de un índice o una referencia.
Enumeration shortcuts()	Obtiene un objeto Enumeration con todos los Shortcuts

7.6.3.-Clase Menu. El objeto ***Menu*** define las opciones que aparecen al seleccionar uno de los menús de la barra de menús. En un ***Menu*** se pueden introducir objetos ***MenuItem***, otros objetos ***Menu*** (para crear sub-menús), objetos ***CheckboxMenuItem***, y ***separadores***. La siguiente tabla muestra algunos métodos de la clase ***Menu***. Obsérvese que como ***Menu*** desciende de ***MenuItem***, el método ***add(MenuItem)*** permite añadir objetos ***Menu*** a otro ***Menu***.

Métodos de Menu	**Función que realizan**
Menu(String)	Constructor a partir de una etiqueta
int getItemCount()	Obtener el número de items
MenuItem getItem(int)	Obtener el MenuItem a partir de un índice
add(String), add(MenuItem), addSeparator(), insertSeparator(int index)	Añadir un MenuItem o un separador
remove(int index), remove(MenuComponent), removeAll()	Eliminar uno o todos los componentes
insert(String lbl, int index),	Insertar items en una posición dada

insert(MenuItem mnu, int index)	
String getLabel(), setLabel(String)	Obtener y establecer las etiquetas de los ítems

7.6.4.-Clase MenuItem. Los objetos de la clase ***MenuItem*** representan las distintas opciones de un menú. Al seleccionar, en la ejecución del programa, un objeto ***MenuItem*** se generan eventos del tipo ***ActionEvents***. Para cada item de un ***Menu*** se puede definir un ***ActionListener***, que define el método ***actionPerformed()***. La siguiente tabla muestra algunos métodos de la clase ***MenuItem***.

Métodos de MenuItem	**Función que realizan**
MenuItem(String lbl), MenuItem(String, MenuShortcut)	Constructores. El carácter (-) es el label de los separators
boolean isEnabled(), setEnabled(boolean)	Pregunta y determina si en item está activo
String getLabel(), setLabel(String)	Obtiene y establece la etiqueta del ítem
MenuShortcut getShortcut(), setShortcut(MenuShortcut), deleteShortcut(MenuShortcut)	Permiten obtener, establecer y borrar los MenuShortcuts
String getActionCommand(), setActionCommand(String)	Para obtener y establecer un identificador distinto del label

El método ***getActionCommand()***, asociado al ***getSource()*** del evento correspondiente, no permite identificar correctamente al ***item*** cuando éste se ha activado mediante el ***MenuShortcut*** (en ese caso devuelve ***null***).

7.6.5.-Clase CheckboxMenuItem. Son ***items*** de un ***Menu*** que pueden estar activados o no activados. La clase ***CheckboxMenuItem*** no genera un ***ActionEvent***, sino un ***ItemEvent***, de modo similar a la clase ***Checkbox*** . En este caso hará registrar un ***ItemListener***. La siguiente tabla muestra algunos métodos de esta clase.

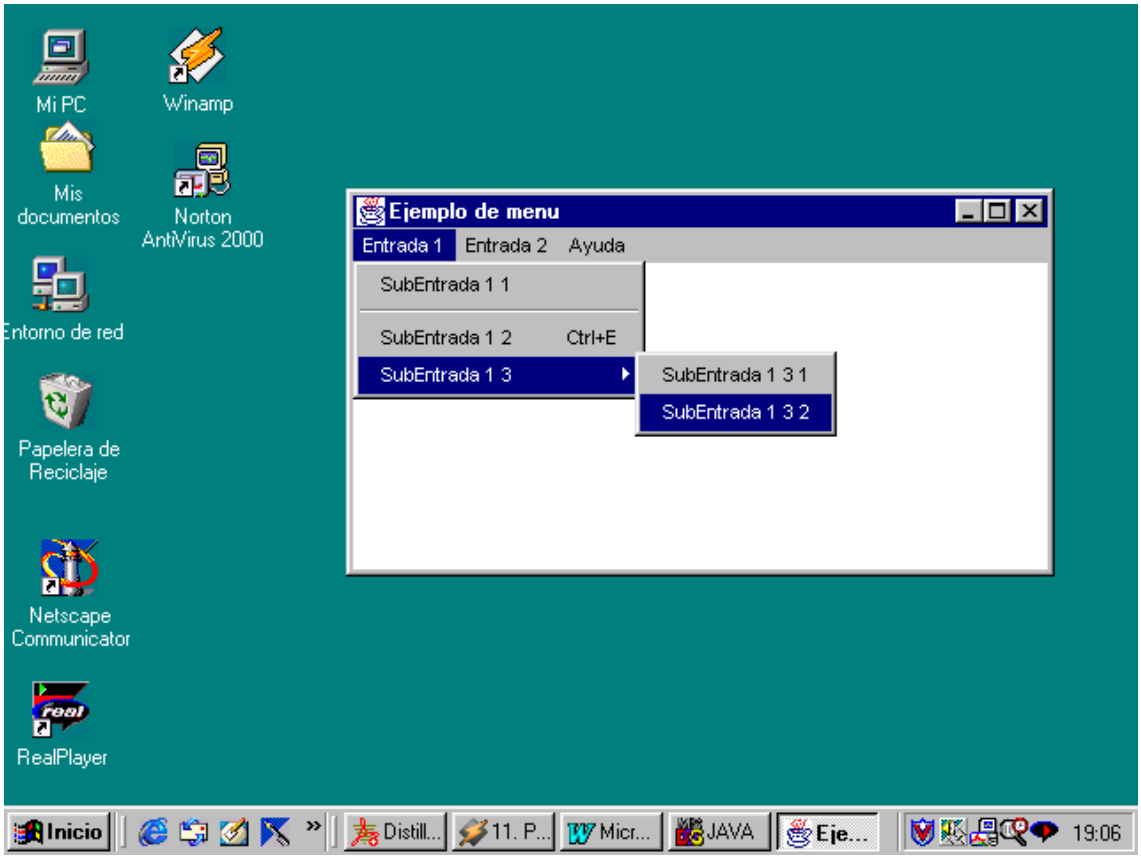
Métodos de la clase CheckboxMenuItem	**Función que realizan**
CheckboxMenuItem(String lbl), CheckboxMenuItem(String lbl, boolean state)	Constructores
Boolean getState(), setState(boolean)	Permiten obtener y establecer el estado del ChecboxMenuItem

7.6.6.- Menús pop-up. Los menús ***pop-up*** son menús que aparecen en cualquier parte de la pantalla al clicar con el botón derecho del ratón (***pop-up trigger***) sobre un componente determinado (***parent Component***). El menú ***pop-up*** se muestra en unas coordenadas relativas al ***parent Component***, que debe estar visible.

Métodos de PopupMenu	**Función que realizan**
PopupMenu(), PopupMenu(String title)	Constructores de PopupMenu
show(Component origin, int x, int y)	Muestra el pop-up menú en la posición indicada

Además, se pueden utilizar los métodos de la clase ***Menu***, de la que deriva ***PopupMenu***. Para hacer que aparezca el ***PopupMenu*** habrá que registrar el ***MouseListener*** y definir el método ***mouseClicked()***.

Ejercicio. Construir el siguiente menú.



7.7.-El Modelo de Eventos.

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos (***event sources***) "registran" los objetos que habrán de gestionarlos (***event listeners***), para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir. En los apartados siguientes se verán con más detalle los ***componentes*** que pueden recibir ***eventos***, los distintos tipos de ***eventos*** y los ***métodos*** de las interfaces ***Listener*** que hay que definir para gestionarlos. En este punto es muy importante ser capaz de buscar la información correspondiente en la documentación de ***Java***.

7.7.1.-Proceso a seguir para crear una aplicación interactiva (orientada a eventos).

Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario:

1. Crear una clase ***Ventana***, sub-clase de ***Frame***, que responda al evento ***WindowClosing()***.

2. Determinar los ***componentes*** que van a constituir la interface de usuario (botones, cajas de texto, menús, etc.).

3. Añadir al objeto ***Ventana*** todos los ***componentes*** y ***menús*** que deba contener.

4. Cada objeto que pueda recibir un evento (***event source***), deberá registrar el objeto que lo gestione (***event listener***).

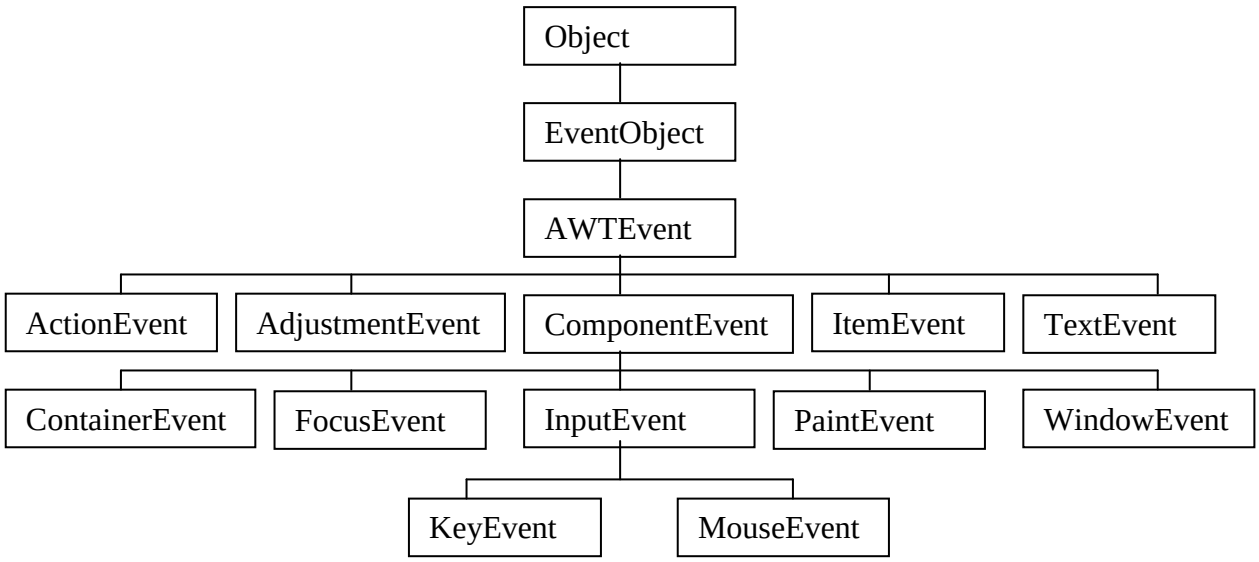
5. Definir los objetos ***Listener*** (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces ***Listener***) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto ***Ventana*** se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.

6. Implementar los métodos de las interfaces ***Listener*** que se vayan a hacer cargo de la gestión de los eventos.

7. Crear una ***clase*** para la aplicación que contenga la función ***main()***.

8. La función ***main()*** deberá crear un objeto de la clase ***Ventana*** (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.

7.7.2.-Jerarquía de eventos. Todos los eventos de ***Java 1.1*** y ***Java 1.2*** son objetos de clases que pertenecen a una determinada jerarquía de clases. La super-clase ***EventObject*** pertenece al package ***java.util***. De ***EventObject*** deriva la clase ***AWTEvent***, de la que dependen todos los eventos de AWT. La siguiente figura muestra la jerarquía de clases para los eventos de ***Java***. Por conveniencia, estas clases están agrupadas en el package ***java.awt.event***.



7.7.3.-Relación entre Componentes y Eventos. La siguiente tabla muestra los componentes del AWT y los eventos específicos de cada uno de ellos, así como una breve explicación de en qué consiste cada tipo de evento.

Component	**Eventos generados**	**Significado**
Button	ActionEvent	Clicar en el botón
Checkbox	ItemEvent	Seleccionar o deseleccionar un item
CheckboxMenuItem	ItemEvent	Seleccionar o deseleccionar un item
Choice	ItemEvent	Seleccionar o deseleccionar un item
Component	ComponentEvent	Mover, cambiar tamaño, mostrar u ocultar un componente
	FocusEvent	Obtener o perder el focus
	KeyEvent	Pulsar o soltar una tecla
	MouseEvent	Pulsar o soltar un botón del ratón; entrar o salir de un componente;mover o arrastrar el ratón (recordar que este evento tiene dos Listener)
Container	ContainerEvent	Añadir o eliminar un componente de un container
List	ActionEvent	Hacer doble click sobre un item de la lista List
	ItemEvent	Seleccionar o deseleccionar un item de la lista
MenuItem	ActionEvent	Seleccionar un item de un menú
Scrollbar	AdjustementEvent	Cambiar el valor de la scrollbar
TextComponent	TextEvent	Cambiar el texto
TextField	ActionEvent	Terminar de editar un texto pulsando Intro

La relación entre componentes y eventos indicada en la tabla anterior pueden inducir a engaño si no se tiene en cuenta que los eventos propios de una ***super-clase*** de componentes pueden afectar también a los componentes de sus ***sub-clases***. Por ejemplo, la clase ***TextArea*** no tiene ningún evento específico o propio, pero puede recibir los de su ***super-clase TextComponent***.

La siguiente tabla muestra los ***componentes*** del AWT y ***todos los tipos de eventos*** que se pueden producir sobre cada uno de ellos, teniendo en cuenta también los que son específicos de sus ***super-clases***. Entre ambas tablas se puede sacar una idea bastante precisa de qué tipos de eventos están soportados en ***Java*** y qué eventos concretos puede recibir cada componente del AWT. En la práctica, no todos los tipos de evento tienen el mismo interés.

AWT Components	Eventos que se pueden generar										
	ActionEvent	**AdjustementEvent**	**ComponentEvent**	**ContainerEvent**	**FocusEvent**	**ItemEvent**	**KeyEvent**	**MouseEvent**	**MousemotionEvent**	**TextEvent**	**WindowEvent**
Button	X		X		X		X	X	X		
Canvas					X		X	X	X		
Checkbox			X		X	X	X	X	X		
Checkbox-MenuItem						X					
Choice					X	X	X	X	X		
Component			X		X		X	X	X		
Container			X	X	X		X	X	X		
Dialog			X	X	X		X	X	X		X
Frame			X	X	X		X	X	X		X
Label			X		X		X	X	X		
List	X		X		X	X	X	X	X		
MenuItem	X										
Panel			X	X	X		X	X	X		
Scrollbar		X	X		X		X	X	X		
TextArea			X		X		X	X	X	X	
TextField	X		X		X		X	X	X	X	
Window			X	X	X		X	X	X		X

7.7.4-Interfaces Listener. Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de ***Java***:

1. Cada objeto que puede recibir un evento (***event source***), "registra" uno o más objetos para que los gestionen (***event listener***). Esto se hace con un método que tiene la forma,

   eventSourceObject.addEventListener(eventListenerObject);

   donde ***eventSourceObject*** es el objeto en el que se produce el evento, y ***eventListenerObject*** es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface ***Listener*** que la clase del ***eventListenerObject*** debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento. La siguiente tabla relaciona los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos. Se indican también los métodos declarados en cada interface.

Evento	Interface Listener	Métodos de Listener
ActionEvent	ActionListener	actionPerformed()
AdjustementEvent	AdjustementListener	adjustementValueChanged()
ComponentEvent	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
ContainerEvent	ContainerListener	componentAdded(),componentRemoved()
FocusEvent	FocusListener	focusGained(), focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
	MouseMotionListener	mouseDragged(), mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowClosed(), windowClosing(), windowIconified(), windowDeiconified(), windowOpened()

Es importante observar la correspondencia entre ***eventos*** e ***interfaces Listener***. Cada evento tiene su interface, excepto el ratón que tiene dos interfaces ***MouseListener*** y ***MouseMotionListener***. La razón de esta duplicidad de interfaces se encuentra en la peculiaridad de los eventos que se producen cuando el ratón se mueve. Estos eventos, que se producen con muchísima más frecuencia que los simples clicks, por razones de eficiencia son gestionados por una interface especial: ***MouseMotionListener***.

Obsérvese que el ***nombre de la interface*** coincide con el ***nombre del evento***, sustituyendo la palabra ***Event*** por ***Listener***.

2. Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface ***Listener***, se deben definir los métodos de dicha interface. Siempre hay que definir ***todos los métodos*** de la interface, aunque algunos de dichos métodos puedan estar "vacíos".

Ejemplo. El oyente es la propia ventana.

```
import java.awt.*;
import java.awt.event.*;

class GUI01c extends Frame implements ActionListener {
   Button bSi,bNo;
   Label l;

   GUI01c(String s){
      super(s);
```

```
        setLayout(new FlowLayout());
        bSi = new Button("SI");
        bNo = new Button("NO");
        l = new Label("Pulsaciones");
        bSi.addActionListener(this);
        bNo.addActionListener(this);
        add(l);
        add(bSi);
        add(bNo);
    }

    public static void main(String [] args) {
        GUI01c f = new GUI01c("Con actionListener");
        f.pack();
        f.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==bSi)
            l.setText("Si Pulsado");
        else
            l.setText("No pulsado");
    }
}
```

Ejemplo. El oyente es un objeto con visibilidad.

```
import java.awt.*;
import java.awt.event.*;

class GUI01c1 extends Frame {
    Button bSi,bNo;
    Label l;

    GUI01c1(String s) {
        super(s);
        setLayout(new FlowLayout());
        bSi = new Button("SI");
        bNo = new Button("NO");
        BotonControl bc = new BotonControl();
        l = new Label("Pulsaciones");
        bSi.addActionListener(bc);
        bNo.addActionListener(bc);
        add(l);
        add(bSi);
```

```
        add(bNo);   }

    public static void main(String [] args) {
        GUI01c1 f = new GUI01c1("Con ActionListener");
        f.pack();
        f.setVisible(true);
    }
    class BotonControl implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (e.getSource()==bSi)
                l.setText("Si Pulsado");
            else
                l.setText("No pulsado");
        }
    }
}
```

7.7.4.1- Clases EventObject y AWTEvent. Todos los métodos de las interfaces ***Listener*** relacionados con el AWT tienen como argumento único un objeto de alguna clase que desciende de la clase ***java.awt.AWTEvent***.

La clase ***AWTEvent*** desciende de ***java.util.EventObject***. La clase ***AWTEvent*** no define ningún método, pero hereda de ***EventObject*** el método ***getSource()***:

Object getSource();

que devuelve una referencia al objeto que generó el evento. Las clases de eventos que descienden de ***AWTEvent*** definen métodos similares a ***getSource()*** con unos valores de retorno menos genéricos. Por ejemplo, la clase ***ComponentEvent*** define el método ***getComponent()***, cuyo valor de retorno es un objeto de la clase ***Component***.

7.7.4.2.-Clase ComponentEvent. Los eventos ***ComponentEvent*** se generan cuando un ***Component*** de cualquier tipo se muestra, se oculta, o cambia de posición o de tamaño. Los eventos de *mostrar* u *ocultar* ocurren cuando se llama al método ***setVisible(boolean)*** del ***Component***, pero no cuando se *minimiza* la ventana.

Otro método útil de la clase ***ComponentEvent*** es ***Component getComponent()*** que devuelve el componente que generó el evento. Se puede utilizar en lugar de ***getSource()***.

7.7.4.3.-Clase ContainerEvent. Los ***ContainerEvents*** se generan cada vez que un ***Component*** se añade o se retira de un ***Container***. Estos eventos sólo tienen un *papel de aviso* y no es necesario gestionarlos para que se realize la operación.

Los métodos de esta clase son ***Component getChild()***, que devuelve el ***Component*** añadido o eliminado, y ***Container getContainer()***, que devuelve el ***Container*** que generó el evento.

7.7.4.4.-Clase ActionEvent. Los eventos ***ActionEvent*** se producen al clicar con el ratón en un botón (***Button***), al elegir un comando de un menú (***MenuItem***), al hacer doble clic en un elemento de una lista (***List***) y al pulsar ***Intro*** para introducir un texto en una caja de texto (***TextField***).

El método ***String getActionCommand()*** devuelve el texto asociado con la acción que provocó el evento. Este texto se puede fijar con el método ***setActionCommand(String str)*** de las clases ***Button*** y ***MenuItem***. Si el texto no se ha fijado con este método, el método ***getActionCommand()*** devuelve el texto mostrado por el componente (su etiqueta). Para objetos con varios items el valor devuelto es el nombre del item seleccionado.

El método ***int getModifiers()*** devuelve un entero representando una constante definida en ***ActionEvent*** (SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK). Estas constantes sirven para determinar si se pulsó una de estas teclas modificadores mientras se clicaba. Por ejemplo, si se estaba pulsando la tecla CTRL la siguiente expresión es distinta de cero:

actionEvent.getModifiers() & Action.Event.CTRL_MASK

Ejemplo.

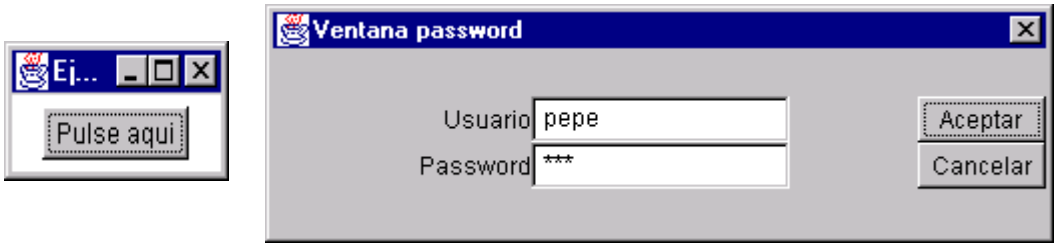
```
import java.awt.*;
import java.awt.event.*;

class GUI01c2 extends Frame {
   Button bSi,bNo;
   Label l;
   GUI01c2(String s)  {
      super(s);
      setLayout(new FlowLayout());
      bSi = new Button("SI");
      bNo = new Button("NO");
      BotonControl bc = new GUI01c2.BotonControl();
      l = new Label("Pulsaciones");
      bSi.addActionListener(bc);
      bSi.setActionCommand("SI");
      bNo.addActionListener(bc);
      bNo.setActionCommand("NO");
      add(l);
      add(bSi);
      add(bNo);
   }

   public static void main(String [] args) {
      GUI01c2 f = new GUI01c2("Con ActionListener");
      f.pack();
      f.setVisible(true);
    }
```

```
        class BotonControl implements ActionListener  {
           public void actionPerformed(ActionEvent e) {
              String comando = e.getActionCommand();
              if (comando.equals("SI"))
                        l.setText("Si Pulsado");
              else if (comando.equals("NO"))
                        l.setText("No pulsado");  }
}}
```

Ejercicio. Diseñar el siguiente GUI interactivo.



7.7.4.5.-Clase FocusEvent.

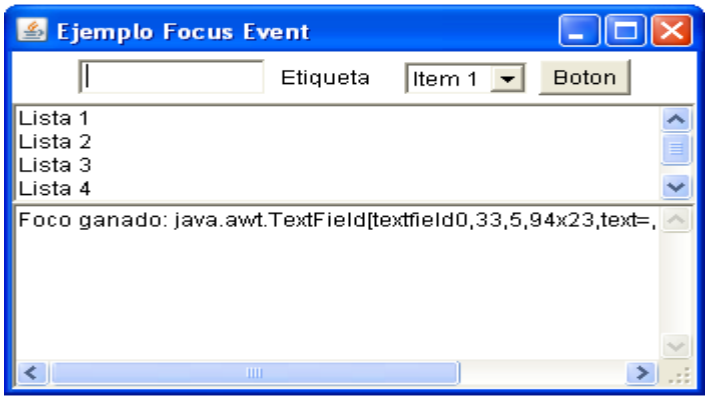
El ***Focus*** está relacionado con la posibilidad de sustituir al ratón por el teclado en ciertas operaciones. De los componentes que aparecen en pantalla, en un momento dado hay sólo uno que puede recibir las acciones del teclado y se dice que ese componente tiene el ***Focus***. El componente que tiene el ***Focus*** aparece diferente de los demás (resaltado de alguna forma). Se cambia el elemento que tiene el ***Focus*** con la tecla ***Tab*** o con el ratón. Se produce un ***FocusEvent*** cada vez que un componente gana o pierde el ***Focus***.

El método ***requestFocus()*** de la clase ***Component*** permite hacer desde el programa que un componente obtenga el ***Focus***.

El método ***boolean isTemporary()***, de la clase ***FocusEvent***, indica si la pérdida del ***Focus*** es o no temporal (puede ser temporal por haberse ocultado o dejar de estar activa la ventana, y recuperarse al cesar esta circunstancia).

El método ***Component getComponent()*** es heredado de ***ComponentEvent***, y permite conocer el componente que ha ganado o perdido el ***Focus***. Las constantes de esta clase ***FOCUS_GAINED*** y ***FOCUS_LOST*** permiten saber el tipo de evento ***FocusEvent*** que se ha producido.

Ejemplo.



7.7.4.6.-Clase ItemEvent. Se produce un ***ItemEvent*** cuando ciertos componentes (***Checkbox***, ***CheckboxMenuItem***, ***Choice*** y ***List***) cambian de estado (on/off). Estos componentes son los que implementan la interface ***ItemSelectable***. La siguiente tabla muestra algunos métodos de esta clase.

Métodos de la clase ItemEvent	**Función que realizan**
Object getItem()	Devuelve el objeto donde se originó el evento
ItemSelectable getItemSelectable()	Devuelve el objeto ItemSelectable donde se originó el evento
int getStateChange()	Devuelve una de las constantes SELECTED o DESELECTED definidas en la clase ItemEvent

La clase ***ItemEvent*** define las constantes enteras SELECTED y DESELECTED, que se pueden utilizar para comparar con el valor devuelto por el método ***getStateChange()***.

7.7.4.7.-Clase KeyEvent. Se produce un ***KeyEvent*** al pulsar sobre el teclado. Como el teclado no es un componente del AWT, es el ***objeto que tiene el focus*** en ese momento quien genera los eventos ***KeyEvent*** (que es el *event source*).

Hay dos tipos de ***KeyEvents***:

1. ***key-typed***, que representa la introducción de un carácter Unicode.

2. ***key-pressed*** y ***key-released***, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como por ejemplo F1.

Para estudiar estos eventos son muy importantes las ***Virtual KeyCodes*** (VKC). Los VKC son unas constantes que se corresponden con las ***teclas físicas*** del teclado, sin considerar minúsculas (que no tienen VKC). Se indican con el prefijo VK_, como VK_SHIFT o VK_A. La clase ***KeyEvent*** (en el package ***java.awt.event***) define constantes VKC para todas las teclas del teclado.

Por ejemplo, para escribir la letra "A" mayúscula se generan 5 eventos: *key-pressed VK_SHIFT*, *key-pressed VK_A*, *key-typed "A"*, *key-released VK_A*, y *key-released VK_SHIFT*. La siguiente tabla muestra algunos métodos de la clase ***KeyEvent*** y otros heredados de ***InputEvent***.

Métodos de la clase KeyEvent	**Función que realizan**
int getKeyChar()	Obtiene el carácter Unicode asociado con el evento
int getKeyCode()	Obtiene el VKC de la tecla pulsada o soltada
boolean isActionKey()	Indica si la tecla del evento es una ActionKey (HOME, END, …)
String getKeyText(int keyCode)	Devuelve un String que describe el VKC, tal como "HOME", "F1" o "A". Estos Strings se definen en el fichero awt.properties
String getKeyModifiersText(int modifiers)	Devuelve un String que describe las teclas modificadoras, tales como"Shift" o "Ctrl+Shift" (fichero awt.properties)
Métodos heredados de InputEvent	**Función que realizan**
boolean isShiftDown(), boolean isControlDown(), boolean IsMetaDown(), boolean isAltDown(), int getModifiers()	Permiten identificar las teclas modificadoras

Las constantes de ***InputEvent*** permiten identificar las teclas modificadoras y los botones del ratón. Estas constantes son SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK. A estas constantes se pueden aplicar las operaciones lógicas de bits para detectar combinaciones de teclas o pulsaciones múltiples.

7.7.4.8.-Clases InputEvent, MouseEvent y MouseMotionEvent. De la clase ***InputEvent*** descienden los eventos del ratón y el teclado. Esta clase dispone de métodos para detectar si los botones del ratón o las teclas especiales han sido pulsadas. Estos botones y estas teclas se utilizan para cambiar o modificar el significado de las acciones del usuario. La clase ***InputEvent*** define unas constantes que permiten saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento, como son: SHIFT_MASK, ALT_MASK, CTRL_MASK, BUTTON1_MASK, BUTTON2_MASK y BUTTON3_MASK, cuyo significado es evidente. La siguiente tabla muestra algunos métodos de esta clase.

Métodos heredados de la clase InputEvent	**Función que realizan**
isShiftDown(), isAltDown(), isControlDown()	Devuelven un boolean con información sobre si esa tecla estaba pulsada o no
int getModifiers()	Obtiene información con una máscara de bits sobre las teclas y botones pulsados
long getWhen()	Devuelve la hora en que se produjo el evento

Se produce un ***MouseEvent*** cada vez que el cursor movido por el ratón entra o sale de un componente visible en la pantalla, al clicar, o cuando se pulsa o se suelta un botón del ratón. Los métodos de la interface ***MouseListener*** se relacionan con estas acciones, y son los siguientes : ***mouseClicked()***, ***mouseEntered()***, ***mouseExited()***, ***mousePressed()*** y ***mouseReleased()***. Todos son *void* y reciben como argumento un objeto ***MouseEvent***. La siguiente tabla muestra algunos métodos de la clase ***MouseEvent***.

Los Métodos de la clase MouseEvent	**Función que realizan**
int getClickCount()	Devuelve el número de clicks en ese evento
Point getPoint(), int getX(), int getY()	Devuelven la posición del ratón al producirse el evento
boolean isPopupTrigger()	Indica si este evento es el que dispara los menús popup

La clase ***MouseEvent*** define una serie de constantes ***int*** que permiten identificar los tipos de eventos que se han producido: MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED, BUTTON1, BUTTON2, BUTTON3.

Además, el método ***Component getComponent()***, heredado de ***ComponentEvent***, devuelve el componente sobre el que se ha producido el evento.

Los eventos ***MouseEvent*** disponen de una segunda interface para su gestión, la interface ***MouseMotionListener***, cuyos métodos reciben también como argumento un evento de la clase ***MouseEvent***. Estos eventos están relacionados con el ***movimiento del ratón***. Se llama a un método de la interface ***MouseMotionListener*** cuando el usuario utiliza el ratón (o un dispositivo similar) para mover el cursor o arrastrarlo sobre la pantalla. Los métodos de la interface ***MouseMotionListener*** son ***mouseMoved()*** y ***mouseDragged()***.

7.7.4.9.-Clase WindowEvent. Se produce un ***WindowEvent*** cada vez que se *abre*, *cierra*, *iconiza*, *restaura*, *activa* o *desactiva* una ventana. La interface ***WindowListener*** contiene los siete métodos siguientes, con los que se puede responder a este evento:

```
void windowOpened(WindowEvent we); // antes de mostrarla por primera vez
void windowClosing(WindowEvent we); // al recibir una solicitud de cierre
void windowClosed(WindowEvent we); // después de cerrar la ventana
void windowIconified(WindowEvent we);
void windowDeiconified(WindowEvent we);
void windowActivated(WindowEvent we);
void windowDeactivated(WindowEvent we);
```

El uso más frecuente de ***WindowEvent*** es para cerrar ventanas (por defecto, los objetos de la clase ***Frame*** no se pueden cerrar más que con ***Ctrl+Alt+Supr***. También se utiliza para detener ***threads*** y

liberar recursos al iconizar una ventana (que contiene por ejemplo animaciones) y comenzar de nuevo al restaurarla.

La clase WindowEvent define la siguiente serie de constantes que permiten identificar el tipo de evento:

WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED, WINDOW_ICONIFIED, WINDOW_DEICONIFIED, WINDOW_ACTIVATED, WINDOW_DEACTIVATED.

En la clase WindowEvent el método Window getWindow() devuelve la Window que generó el evento. Se utiliza en lugar de getSource(),

7.7.4.10.-Clase AdjustmentEvent. Se produce un evento ***AdjustementEvent*** cada vez que se cambia el valor (entero) de una ***Scrollbar***. Hay cinco tipos de ***AdjustementEvent***:

1. ***track***: se arrastra el cursor de la ***Scrollbar***.
2. ***unit increment***, ***unit decrement***: se clica en las flechas de la ***Scrollbar***.
3. ***block increment***, ***block decrement***: se clica encima o debajo del cursor.

Métodos de la clase AdjustementEvent	**Función que realizan**
Adjustable getAdjustable()	Devuelve el Component que generó el evento (implementa la interface Adjustable)
int getAdjustementType()	Devuelve el tipo de adjustement
int getValue()	Devuelve el valor de la Scrollbar después del cambio

La tabla anterior muestra algunos métodos de esta clase. Las constantes UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK permiten saber el tipo de acción producida, comparando con el valor de retorno de ***getAdjustementType()***.

7.7.4.11.-Clase TextEvent. Se produce un ***TextEvent*** cada vez que cambia algo en un ***TextComponent*** (***TextArea*** y ***TextField***). Se puede desear evitar ciertos caracteres y para eso hay que gestionar los eventos correspondientes.

La interface ***TextListener*** tiene un único método:

void textValueChanged(TextEvent te).

No hay métodos propios de la clase ***TextEvent***. Se puede utilizar el método ***Object getSource()***, que es heredado de ***EventObject***.

7.8.-Adaptadores. Es tedioso tener que implementar todos los métodos de un interfaz. Por ejemplo, de un WindowListener puede interesar sólo el método de cierre, es decir el método *windowClosing*. El problema radica en que si se implementa el interfaz hay que implementar todas las funciones, aunque estén vacías. Una solución son los adaptadores.

Java proporciona ayudas para definir los métodos declarados en las interfaces ***Listener***. Una de estas ayudas son las clases ***Adapter***, que existen para cada una de las interfaces ***Listener*** que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra "*Listener*" por "*Adapter*". Hay 7 clases ***Adapter***: *ComponentAdapter*, *ContainerAdapter*, *FocusAdapter*, *KeyAdapter*, *MouseAdapter*, *MouseMotionAdapter* y *WindowAdapter*.

Las clases ***Adapter*** derivan de ***Object***, y son clases predefinidas que contienen ***definiciones vacías*** para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface ***Listener***, basta crear una clase que derive de la clase ***Adapter*** correspondiente, y redefina sólo los métodos de interés. Por ejemplo, la clase ***VentanaCerrable*** se puede definir de la siguiente forma:

```
1. // Fichero VentanaCerrable.java
2. import java.awt.*;
3. import java.awt.event.*;
4. class VentanaCerrable extends Frame {
5.         // constructores
6.         public VentanaCerrable() { super(); }
7.         public VentanaCerrable(String title) {
8.                 super(title);
9.                 setSize(500,500);
10.                CerrarVentana cv = new CerrarVentana();
11.                this.addWindowListener(cv);
12.        }
13. } // fin de la clase VentanaCerrable
14. // definición de la clase CerrarVentana
15. class CerrarVentana extends WindowAdapter {
16.        void windowClosing(WindowEvent we) { System.exit(0); }
17. } // fin de la clase CerrarVentana
```

Las sentencias 15-17 definen una clase auxiliar (***helper class***) que deriva de la clase ***WindowAdapter***. Dicha clase hereda definiciones vacías de todos los métodos de la interface ***WindowListener***. Lo único que tiene que hacer es redefinir el único método que se necesita para cerrar las ventanas. El constructor de la clase ***VentanaCerrable*** crea un objeto de la clase ***CerrarVentana*** en la sentencia 10 y lo registra como ***event listener*** en la sentencia 11. En la sentencia 11 la palabra ***this*** es opcional: si no se incluye, se supone que el ***event source*** es el objeto de la clase en la que se produce el evento, en este caso la propia ventana.

7.9.-Gráficos, Texto e Imágenes.

En esta parte final del AWT se van a describir, también muy sucintamente, algunas clases y métodos para realizar dibujos y añadir texto e imágenes a la interface gráfica de usuario.

7.9.1.-Capacidades gráficas del AWT.

La clase ***Component*** tiene tres métodos muy importantes relacionados con gráficos: ***paint()***, ***repaint()*** y ***update()***. Cuando el usuario llama al método ***repaint()*** de

un componente, el AWT llama al método ***update()*** de ese componente, que por defecto llama al método ***paint()***.

7.9.1.1.-Método paint(Graphics g). El método ***paint()*** está definido en la clase ***Component***, pero ese método no hace nada y hay que redefinirlo en una de sus clases derivadas. El programador no tiene que preocuparse de llamar a este método: el sistema operativo lo llama al dibujar por primera vez una ventana, y luego lo vuelve a llamar cada vez que entiende que la ventana o una parte de la ventana debe ser re-dibujada (por
ejemplo, por haber estado tapada por otra ventana y quedar de nuevo a la vista).

7.9.1.2.-Método update(Graphics g). El método ***update()*** hace dos cosas: primero re-dibuja la ventana con el color de fondo y luego llama al método ***paint()***. Este método también es llamado por el AWT, y también puede ser llamado por el programador, quizás porque ha realizado algún cambio en la ventana y necesita que se dibuje de nuevo.

La propia estructura de este método -el comenzar pintando de nuevo con el color de fondo-hace que se produzca ***parpadeo*** (***flicker***) en las animaciones. Una de las formas de evitar este efecto es redefinir este método de una forma diferente, cambiando de una imagen a otra sólo lo que haya que cambiar, en vez de re-dibujar todo otra vez desde el principio. Este método no siempre propor-ciona los resultados buscados y hay que recurrrir al método del ***doble buffer***.

7.9.1.3.-Método repaint(). Este es el método que con más frecuencia es llamado por el programador. El método ***repaint()*** llama "lo antes posible" al método ***update()*** del componente. Se puede también especificar un número de milisegundos para que el método ***update()*** se llame transcurrido ese tiempo. El método ***repaint()***tiene las cuatro formas siguientes:

repaint()
repaint(long time)
repaint(int x, int y, int w, int h)
repaint(long time, int x, int y, int w, int h)

Las formas tercera y cuarta permiten definir una ***zona rectangular*** de la ventana a la que aplicar el método.

7.9.2.-Clase Graphics.

El único argumento de los métodos ***update()*** y ***paint()*** es un objeto de esta clase. La clase ***Graphics*** dispone de métodos para soportar dos tipos de gráficos:

1. Dibujo de ***primitivas gráficas*** (*texto, líneas, círculos, rectángulos*, .. …).
2. Presentación de ***imágenes*** en formatos ****.gif*** y ****.jpeg***.

Además, la clase ***Graphics*** mantiene un ***contexto gráfico***: un área de dibujo actual, un color de dibujo del background y otro del foreground, un font con todas sus propiedades, etc. El sistema de coordenadas utilizado en ***Java,*** como es habitual en Informática, tiene el origen en el vértice superior izquierdo (0,0).

7.9.3.-Primitivas gráficas. ***Java*** dispone de métodos para realizar dibujos sencillos, llamados a veces "primitivas" gráficas. Las coordenadas se miden en pixels, empezando a contar desde cero. La clase ***Graphics*** dispone de los métodos para primitivas gráficas reseñados en la siguiente tabla.

Método gráfico	**Función que realizan**
drawLine(int x1, int y1, int x2, int y2)	Dibuja una línea entre dos puntos
drawRect(int x1, int y1, int w, int h)	Dibuja un rectángulo (w-1, h-1)
fillRect(int x1, int y1, int w, int h)	Dibuja un rectángulo y lo rellena con el color actual
clearRect(int x1, int y1, int w, int h)	Borra dibujando con el background color
draw3DRect(int x1, int y1, int w, int h, boolean raised)	Dibuja un rectángulo resaltado (w+1, h+1)
fill3DRect(int x1, int y1, int w, int h, boolean raised)	Rellena un rectángulo resaltado (w+1, h+1)
drawRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Dibuja un rectángulo redondeado
fillRoundRect(int x1, int y1, int w, int h, int arcw, int arch)	Rellena un rectángulo redondeado
drawOval(int x1, int y1, int w, int h)	Dibuja una elipse
fillOval(int x1, int y1, int w, int h)	Dibuja una elipse y la rellena de un color
drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Dibuja un arco de elipse (ángulos en grados)
fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)	Rellena un arco de elipse
drawPolygon(int x[], int y[], int nPoints)	Dibuja y cierra el polígono de modo automático
drawPolyline(int x[], int y[], int nPoints)	Dibuja un polígono pero no lo cierra
fillPolygon(int x[], int y[], int nPoints)	Rellena un polígono

Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son ***w*** y ***h***. Los polígonos admiten un argumento de la clase ***java.awt.Polygon***.

Los métodos ***draw3DRect()***, ***fill3DRect()***, ***drawOval()***, ***fillOval()***, **drawArc()** y ***fillArc()*** dibujan objetos cuyo tamaño total es (w+1, h+1) pixels.

7.9.4.-Clases Graphics y Font. La clase ***Graphics*** permite "dibujar" texto, como alternativa al texto mostrado en los componentes ***Label***, ***TextField*** y ***TextArea***. Los métodos de esta clase para dibujar texto son los siguientes:

drawBytes(byte data[], int offset, int length, int x, int y);
drawChars(char data[], int offset, int length, int x, int y);
drawString(String str, int x, int y);

En estos métodos, los argumentos ***x*** e ***y*** representan las coordenadas de la ***línea base***. Cada tipo de letra está representado por un objeto de la clase ***Font***. Las clases ***Component*** y ***Graphics*** disponen de métodos ***setFont()*** y ***getFont()***. El constructor de ***Font*** tiene la forma:

Font(String name, int style, int size)

donde el ***style*** se puede definir con las constantes Font.PLAIN, Font.BOLD y Font.ITALIC. Estas constantes se pueden combinar en la forma: Font.BOLD | Font.ITALIC.

La clase ***Font*** tiene tres variables *protected*, llamadas ***name***, ***style*** y ***size***. Además tiene tres constantes enteras: PLAIN, BOLD e ITALIC. Esta clase dispone de los métodos ***String getName()***, ***int getStyle()***, ***int getSize()***, ***boolean isPlain()***,***boolean isBold()*** y ***boolean isItalic()***, cuyo significado es inmediato.

Para mayor portabilidad se recomienda utilizar nombres lógicos de fonts, tales como ***Serif*** (*Times New Roman*), ***SansSerif*** (*Arial*) y ***Monospaced*** (*Courier*).

7.9.5.-Clase Color. La clase ***java.awt.Color*** encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima.

En la clase ***Color*** existen constantes para colores predeterminados de uso frecuente: ***black***, ***white***, ***green***, ***blue***, ***red***, ***yellow***, ***magenta***, ***cyan***, ***orange***, ***pink***, ***gray***, ***darkGray***, ***lightGray***. La siguiente tabla muestra algunos métodos de la clase ***Color***.

Metodos de la clase Color	**Función que realizan**
Color(int), Color(int,int,int), Color(float,float,float)	Constructores de Color, con enteros entre 0 y 255 y float entre 0.0 y 1.0
Color brighter(), Color darker()	Obtienen una versión más o menos brillante de un color
Color getColor(), int getRGB()	Obtiene un color en los tres primeros bytes de un int
int getGreen(), int getRed(), int getBlue()	Obtienen las componentes de un color
Color getHSBColor()	Obtiene un color a partir de los valores de "hue", "saturation" y "brightness" (entre 0.0 y 1.0)
float[] RGBtoHSB(int,int,int,float[]), int HSBtoRGB(float,float,float)	Métodos static para convertir colores de un sistema de definición de colores a otro

7.9.6.-Imágenes. ***Java*** permite incorporar imágenes de tipo GIF y JPEG definidas en ficheros. Se dispone para ello de la clase ***java.awt.Image***. Para cargar una imagen hay que indicar la localización del fichero (URL) y cargarlo mediante los métodos ***Image getImage(String)*** o ***Image getImage(URL, String)***. Estos métodos existen en las clases ***java.awt.Toolkit*** y ***java.applet.Applet***.

Cuando estas imágenes se cargan en ***applets***, para obtener el URL pueden ser útiles las funciones ***getDocumentBase()*** y ***getCodeBase()***, que devuelven el URL del fichero HTML que llama al ***applet***, y el directorio que contiene el ***applet*** (en forma de ***String***).

Para cargar una imagen hay que comenzar creando un objeto ***Image***, y llamar al método ***getImage()***, pasándole como argumento el URL. Por ejemplo:

Image miImagen = getImage(getCodeBase(), "imagen.gif")

Una vez cargada la imagen, hay que representarla, para lo cual se redefine el método ***paint()*** y se utiliza el método ***drawImage()*** de la clase ***Graphics***. Dicho método admite varias formas, aunque casi siempre hay que incluir el nombre del objeto imagen creado, las dimensiones de dicha imagen y un objeto ***ImageObserver***. Para más información sobre dicho método dirigirse a la referencia de la API.

ImageObserver es una interface que declara métodos para observar el estado de la carga y visualización de la imagen. Si se está programando un ***applet***, basta con poner como ***ImageObserver*** la referencia ***this***, ya que en la mayoría de los casos, la implementación de esta interface en la clase ***Applet*** proporciona el comportamiento deseado.

La clase ***Image*** define ciertas constantes para controlar los algoritmos de cambio de escala:

SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH,
SCALE_REPLICATE, SCALE_AVERAGE.
La siguiente tabla muestra algunos métodos de la clase ***Image***.

Métodos de la clase Image	**Función que realizan**
Image()	Constructor
int getWidth(ImageObserver) int getHeight(ImageObserver)	Determinan la anchura y la altura de la imagen. Si no se conocen todavía, este método devuelve -1 y el objeto ImageObserver especificado será notificado más tarde
Graphics getGraphics()	Crea un contexto gráfico para poder dibujar en una imagen no visible en pantalla. Este método sólo se puede llamar para métodos no visibles en pantalla
Object getProperty(String, ImageObserver)	Obtiene una propiedad de una imagen a partir del nombre de la propiedad
Image getScaledInstance(int w, int h, int hints)	Crea una versión de la imagen a otra escala. Si w o h son negativas se utiliza la otra dimensión manteniendo la proporción. El último argumento es información para el algoritmo de cambio de escala

7.9.7.-ANIMACIONES. Las animaciones tienen un gran interés desde diversos puntos de vista. Una imagen vale más que mil palabras y una imagen en movimiento es todavía mucho más útil: para presentar o describir ciertos conceptos el movimiento animado es fundamental. Además, las animaciones o mejor dicho, la forma de hacer animaciones en ***Java*** ilustran mucho la forma en que dicho lenguaje realiza los gráficos. En estos apartados se va a seguir el esquema del ***Tutorial*** de ***Sun*** sobre el AWT.

Se pueden hacer animaciones de una forma muy sencilla: se define el método ***paint()*** de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior. De todas formas, recuérdese que el programador no llama directamente a este método. El programador llama al método ***repaint()***, quizás dentro de un bucle ***while*** que incluya una llamada al método ***sleep()*** de la clase ***Thread***, para esperar un cierto número de milisegundos entre dibujo y dibujo (entre *frame* y *frame*, utilizando la terminología de las animaciones). Recuérdese que ***repaint()*** llama a ***update()*** lo antes posible, y que ***update()*** borra todo redibujando con el color de fondo y llama a ***paint()***.

La forma de proceder descrita da buenos resultados para animaciones muy sencillas, pero produce ***parpadeo*** o ***flicker*** cuando los gráficos son un poco más complicados. La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor. La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 herzios. Eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo. Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de los siguientes pasos de refresco del monitor. Ésta es la causa del ***flicker***. A continuación se verán dos formas de reducirlo o eliminarlo.

7.9.7.1.-Eliminación del parpadeo o flicker redefiniendo el método update(). El problema del ***flicker*** se localiza en la llamada al método ***update()***, que borra todo pintando con el color de fondo y después llama a ***paint()***. Una forma de resolver esta dificultad es ***re-definir*** el método ***update()***, de forma que se adapte mejor al problema que se trata de resolver.

Una posibilidad es no re-pintar todo con el color de fondo, no llamar a ***paint()*** e introducir en ***update()*** el código encargado de realizar los dibujos, cambiando sólo aquello que haya que cambiar. A pesar de esto, es necesario re-definir ***paint()*** pues es el método que se llama de forma automática cuando la ventana de ***Java*** es tapada por otra que luego se retira. Una posible solución es hacer que ***paint()*** llame a ***update()***, terminando por establecer un orden de llamadas opuesto al de defecto. Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre *frame* y *frame* lo que sea necesario. Los métodos ***setClip()*** y ***clipRect()*** de la clase ***Graphics*** permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada. Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

7.9.7.2.-Técnica del doble buffer. La técnica del ***doble buffer*** proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada. La idea básica del ***doble buffer*** es realizar los dibujos en una imagen invisible, distinta de la que se está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase ***Image*** del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase ***Graphics*** que permita dibujar sobre la imagen invisible. Esto se hace con las sentencias,

```
Image imgInv;
Graphics graphInv;
Dimension dimInv;
Dimension d = size(); // se obtiene la dimensión del panel
```

en la clase que controle el dibujo (por ejemplo en una clase que derive de ***Panel***). En el método ***update()*** se modifica el código de modo que primero se dibuje en la imagen invisible y luego ésta se haga visible:

```
public void update (Graphics.g) {
  // se comprueba si existe el objeto invisible y si sus dimensiones son correctas
  if ((graphInv==null) || (d.width!=dimInv.width) || (d.height!=dimInv.height)) {
          dimInv = d;
        // se llama al método createImage de la clase Component
        imgInv = createImage(d.width, d.height);
        // se llama al método getGraphics de la clase Image
        graphInv = imgInv.getGraphics();
  }
  // se establecen las propiedades del contexto gráfico invisible,
  // y se dibuja sobre él
  graphInv.setColor(getBackground());
  ...
  // finalmente se hace visible la imagen invisible a partir del punto (0, 0)
  // utilizando el propio panel como ImageObserver
  g.drawImage(imgInv, 0, 0, this);
} // fin del método update()
```

Los gráficos y las animaciones son particularmente útiles en las applets. El ***Tutorial*** de ***Sun***tiene un ejemplo (un ***applet***) completamente explicado y desarrollado sobre las animaciones y los distintos métodos de eliminar el flicker o parpadeo.

Tema 8: APPLETS.

8.1.-Introducción.

Un ***applet*** es una mini-aplicación, escrita en ***Java***, que se ejecuta en un browser (***Netscape Navigator***, ***Microsoft Internet Explorer***, …) al cargar una página HTML que incluye información sobre el ***applet*** a ejecutar por medio de las ***tags*** <APPLET>... </APPLET>.

A continuación se detallan algunas características de las ***applets***:

1. Los ficheros de ***Java*** compilados (*.class) se descargan a través de la red desde un servidor de ***Web*** o servidor ***HTTP*** hasta el browser en cuya *Java Virtual Machine* se ejecutan. Pueden incluir también ficheros de imágenes y sonido.

2. Los ***applets*** no tienen ventana propia: se ejecutan en la ventana del browser (en un "***panel***").

3. Por la propia naturaleza "abierta" de Internet, los ***applets*** tienen importantes restricciones de seguridad, que se comprueban al llegar al browser: sólo pueden leer y escribir ficheros en el servidor del que han venido, sólo pueden acceder a una limitada información sobre el ordenador en el que se están ejecutando, etc. Con ciertas condiciones, los ***applets*** "de confianza" (***trusted applets***) pueden pasar por encima de estas restricciones.

Aunque su entorno de ejecución es un browser, los ***applets*** se pueden probar sin necesidad de browser con la aplicación ***appletviewer*** del JDK de ***Sun***.

8.2.-Características de los applets.

Las características de los ***applets*** se pueden considerar desde el punto de vista del programador y desde el del usuario. En este manual lo más importante es el punto de vista del programador:

· Los ***applets*** no tienen un método ***main()*** con el que comience la ejecución. El papel central de su ejecución lo asumen otros métodos que se verán posteriormente.

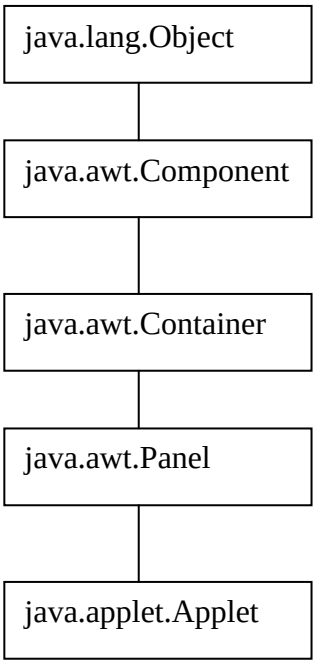
· Todos los ***applets*** derivan de la clase ***java.applet.Applet***. La siguiente figura muestra la jerarquía de clases de la que deriva la clase ***Applet***. Los ***applets*** deben redefinir ciertos métodos heredados de ***Applet*** que controlan su ejecución: ***init()***, ***start()***, ***stop()***, ***destroy()***.

· Se heredan otros muchos métodos de las super-clases de ***Applet*** que tienen que ver con la generación de interfaces gráficas de usuario (AWT). Así, los métodos gráficos se heredan de ***Component***, mientras que la capacidad de añadir componentes de interface de usuario se hereda de ***Container*** y de ***Panel***.

· Los ***applets*** también suelen redefinir ciertos métodos gráficos: los más importantes son ***paint()*** y ***update()***, heredados de ***Component*** y de ***Container***; y ***repaint()*** heredado de ***Component***.

· Los ***applets*** disponen de métodos relacionados con la obtención de información, como por ejemplo: ***getAppletInfo()***, ***getAppletContext()***, ***getParameterInfo()***, ***getParameter()***, ***getCodeBase()***, ***getDocumentBase()***, e ***isActive()***. El método ***showStatus()*** se utiliza para mostrar

información en la barra de estado del browser. Existen otros métodos relacionados con imágenes y sonido: ***getImage()***, ***getAudioClip()***, ***play()***, etc.



8.3.-Métodos que controlan la ejecución de un applet. Los métodos que se estudian en este apartado controlan la ejecución de los ***applets***. De ordinario el programador tiene que redefinir uno o más de estos métodos, pero no tiene que preocuparse de llamarlos: el browser se encarga de hacerlo.

8.3.1.-Método init(). Se llama automáticamente al método ***init()*** en cuanto el browser o visualizador carga el ***applet***. Este método se ocupa de todas las tareas de inicialización, realizando las funciones del ***constructor*** (al que el browser no llama). En este método es donde debemos construir la forma del applet.

8.3.2.-Método start(). El método ***start()*** se llama automáticamente en cuanto el ***applet*** se hace visible, después de haber sido inicializado. Se llama también cada vez que el ***applet*** se hace de nuevo visible después de haber estado oculto (por dejar de estar activa esa página del browser, al cambiar el tamaño de la ventana del browser, al hacer ***reload***, etc.).

Es habitual crear ***threads*** en este método para aquellas tareas que, por el tiempo que requieren, dejarían sin recursos al ***applet*** o incluso al browser. Las animaciones y ciertas tareas a través de Internet son ejemplos de este tipo de tareas.

8.3.3.- Método stop(). El método ***stop()*** se llama de forma automática al ocultar el ***applet*** (por haber haber dejado de estar activa la página del browser, por hacer ***reload*** o ***resize***, etc.).

Con objeto de no consumir recursos inútilmente, en este método se suelen parar las ***threads*** que estén corriendo en el ***applet***, por ejemplo para mostrar animaciones.

8.3.4.-Método destroy(). Se llama a este método cuando el ***applet*** va a ser descargado para liberar los recursos que tenga reservados (excepto la memoria). De ordinario no es necesario redefinir este método, pues el que se hereda cumple bien con esta misión.

8.3.5.-Métodos para dibujar el applet. Los ***applets*** son aplicaciones gráficas que aparecen en una zona de la ventana del browser. Por ello deben redefinir los métodos gráficos ***paint()*** y ***update()***. El método ***paint()*** se declara en la forma:

public void paint(Graphics g)

El objeto gráfico ***g*** pertenece a la clase ***java.awt.Graphics***, que siempre debe ser importada por el ***applet***. Este objeto define un contexto o estado gráfico para dibujar (métodos gráficos, colores, fonts, etc.) y es creado por el browser.

Todo el trabajo gráfico del ***applet*** (dibujo de líneas, formas gráficas, texto, etc.) se debe incluir en el método ***paint()***, porque este método es llamado cuando el ***applet*** se dibuja por primera vez y también de forma automática cada vez que el ***applet*** se debe redibujar.

En general, el programador crea el método ***paint()*** pero no lo suele llamar. Para pedir explícitamente al sistema que vuelva a dibujar el ***applet*** (por ejemplo, por haber realizado algún cambio) se utiliza el método ***repaint()***, que es más fácil de usar, pues no requiere argumentos. El método ***repaint()*** se encarga de llamar a ***paint()*** a través de ***update()***.

El método ***repaint()*** llama a ***update()***, que borra todo pintando de nuevo con el color de fondo y luego llama a ***paint()***. A veces esto produce parpadeo de pantalla o ***flickering***. Existen dos formas de evitar el ***flickering***:

1. Redefinir ***update()*** de forma que no borre toda la ventana sino sólo lo necesario.

2. Redefinir ***paint()*** y ***update()*** para utilizar ***doble buffer***.

Ambas formas fueron estudiadas en el tema anterior.

Ejemplo.

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class Applet1 extends java.applet.Applet {
    Font f = new Font("TimesRoman", Font.BOLD, 36);
    public void paint(Graphics pantalla) {
            pantalla.setFont(f);
            pantalla.setColor(Color.red);
            pantalla.drawString("Esto es una prueba de un Applet", 5, 40);
    }}
```

8.4.-Cómo incluir un Applet en una página HTML. Para llamar a un ***applet*** desde una página HTML se utiliza la tag doble <APPLET>…</APPLET>, cuya forma general es (los elementos opcionales aparecen entre corchetes[]):

```
<APPLET CODE="miApplet.class" [CODEBASE="unURL"] [NAME="unName"]
        WIDTH="wpixels" HEIGHT="hpixels"
        [ALT="TextoAlternativo"]>
        [texto alternativo para browsers que reconocen el tag <applet> pero no pueden ejecutar el
        applet]
        [<PARAM NAME="MyName1" VALUE="valueOfMyName1">]
        [<PARAM NAME="MyName2" VALUE="valueOfMyName2">]
</APPLET>
```

El atributo NAME permite dar un nombre opcional al ***applet***, con objeto de poder comunicarse con otras ***applets*** o con otros elementos que se estén ejecutando en la misma página. El atributo ARCHIVE permite indicar uno o varios ficheros Jar o Zip (separados por comas) donde se deben buscar las clases.

A continuación se señalan otros posibles atributos de <APPLET>:

· ARCHIVE="file1, file2, file3". Se utiliza para especificar ficheros JAR y ZIP.
· ALIGN, VSPACE, HSPACE. Tienen el mismo significado que el tag IMG de HTML.

Ejemplo. Fichero Applet1.html

```
<HTML>
<HEAD>
<TITLE> Mi primer ejemplo de Applet </TITLE>
</HEAD>
<BODY>
<APPLET CODE=”Applet1.class” WIDTH=600 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

8.5.-Ciclo de vida de un applet. Cada vez que visitamos una página *Web* que contiene una referencia a un *applet*, es el explorador el de su clase que se encarga de cargarlo. Una vez cargado, se crea un objeto de su clase, esto es, se crea un objeto *applet*. A continuación, el applet se inicia así mismo (ejecuta el método **init**) y comienza su ejecución (ejecuta el método **start**).

Una vez que el *applet* está en ejecución, puede ocurrir que visitemos otra página. En este caso, el *applet* detendrá su ejecución (ejecuta su método **stop**) que la reanudará en el momento en el que sea visitada su página de nuevo (vuelve a ejecutar su método **start**).

Cuando un applet es descargado, porque salimos del explorador, libera todos los recursos que hubiera adquirido y detiene su ejcución invocando al método **stop** y después al método **destroy**.

Cuando el explorador muestra una página *Web* con un *applet* y se pulsa el botón *Actualizar* del mismo, el *applet* es descargado y vuelto a cargar de nuevo.

8.6.-Paso de parámetros a un Applet. Los tags PARAM permiten pasar diversos parámetros desde el fichero HTML al programa ***Java*** del ***applet***, de una forma análoga a la que se utiliza para pasar argumentos a ***main()***.

Cada parámetro tiene un ***nombre*** y un ***valor***. Ambos se dan en forma de ***String***, aunque el valor sea numérico. El ***applet*** recupera estos parámetros y, si es necesario, convierte los ***Strings*** en valores numéricos. El valor de los parámetros se obtienen con el siguiente método de la clase ***Applet***:

String getParameter(String name)

La conversión de ***Strings*** a los tipos primitivos se puede hacer con los métodos asociados a los ***wrappers*** que ***Java*** proporciona para dichos tipo fundamentales (Integer.parseInt(String), Double.valueOf(String), …). Estas clases de wrappers se estudiaron en temas anteriores.

En los ***nombres*** de los parámetros no se distingue entre mayúsculas y minúsculas, pero sí en los ***valores***, ya que serán interpretados por un programa ***Java***, que sí distingue.

El programador del ***applet*** debería prever siempre unos ***valores por defecto*** para los parámetros del ***applet***, para el caso de que en la página HTML que llama al ***applet*** no se definan.

El método ***getParameterInfo()*** devuelve una matriz de ***Strings*** (*String[][]*) con información sobre cada uno de los parámetros soportados por el ***applet***: *nombre*, *tipo* y *descripción*, cada uno de ellos en un ***String***. Este método debe ser redefinido por el programador del ***applet*** y utilizado por la persona que prepara la página HTML que llama al ***applet***. En muchas ocasiones serán personas distintas, y ésta es una forma de que el programador del ***applet*** dé información al usuario.

8.7.-Carga de Applets.

8.7.1.-Localización de ficheros. Por defecto se supone que los ficheros ****.class*** del ***applet*** están en el mismo directorio que el fichero HTML. Si el ***applet*** pertenece a un ***package***, el browser utiliza el nombre del ***package*** para construir un ***path de directorio*** relativo al directorio donde está el HTML.

El atributo CODEBASE permite definir un URL para los ficheros que continen el código y demás elementos del ***applet***. Si el directorio definido por el URL de CODEBASE es *relativo*, se interpreta respecto al directorio donde está el HTML; si es *absoluto* se interpreta en sentido estricto y puede ser cualquier directorio de la Internet.

8.7.2.-Archivos JAR (Java Archives). Si un ***applet*** consta de varias clases, cada fichero ****.class*** requiere una conexión con el servidor de Web (servidor de protocolo HTTP), lo cual puede requerir

algunos segundos. En este caso es conveniente agrupar todos los ficheros en un archivo único, que se puede comprimir y cargar con una sola conexión HTTP.

Los archivos JAR están basados en los archivos ZIP y pueden crearse con el programa ***jar*** que viene con el JDK. Por ejemplo:

*jar cvf myFile.jar *.class *.gif*

crea un fichero llamado ***myFile.jar*** que contiene todos los ficheros ****.class*** y ****.gif*** del directorio actual. Si las clases pertenecieran a un package llamado ***es.ceit.infor2*** se utilizaría el comando:

jar cvf myFile.jar es\ceit\infor2.class *.gif*

8.8.- Mostrar una imagen. Para mostrar una imagen en un applet, lo primero es crear un objeto **URL** que indique la ubicación del fichero que contiene la imagen. La clase **URL** pertenece al paquete **java.net.** Por ejemplo:

URL url = new URL("c:/java/proyecto/practica17");

Lo siguiente es cargar la imagen y crear un objeto **Image**. La clase **Image** del paquete **java.awt** es la superclase de todas las clases que representan imágenes gráficas. Para cargar la imagen, el *applet* invocará a su método **getImage**. Por ejemplo:

Image imgTfno = getImage(url, "teléfono.gif");

Una vez cargada y lista para ser visualizada se invoca al método **drawImage** del objeto **Graphics**, para mostrala. Por ejemplo:

g.drawImage(ImgTfno, x, y, this);

Los argumentos de **drawImage** son el objeto **Image**, las coordenadas *x* e *y* de donde se quiere colocar la imagen dentro del *applet* y la referencia al *applet*.

Podemos obtener el URL invocando al método **getDocumentBase** o al método **getCodeBase**. El primero devuelve el URL de la carpeta desde la que se cargó el fichero html y el segundo devuelve el URL de la carpeta desde la que se cargó el fichero class. Por ejemplo:

Image imgTfno = getImage(getDocumentBase(), "teléfono.gif");

Podemos también, visualizar imágenes en determinados componentes como, por ejemplo, un *JLabel*. El método **setIcon** de *JLabel* permite asignar una imagen a una etiqueta. Tiene como argumento una referencia a un objeto de una clase que implemente la interfaz **Icon**, por ejemplo, un objeto de la clase **ImageIcon**.

La clase **ImageIcon** del paquete **java.swing** entre sus constructores, tiene uno que permite construir un objeto de esta clase a partir de un objeto **Image**. Por ejemplo:

```
Image imgTfno = getImage(getDocumentBase(), "teléfono.gif");
jlbTelefono.setIcon(new ImageIcon(imgTfno));
```

Otro de los constructores de **ImageIcon** permite construir un objeto **ImageIcon** a partir del **URL** que define la ruta del fichero que contiene la imagen. Ejemplo:

```
jlbTelefono.setIcon(new ImageIcon(new java.net.URL(getDocumentBase(), "imgTfno.gif");
```

8.9.-Reproduccion de un fichero de sonido. La forma más sencilla de reproducir un sonido es invocando al método **play** del *applet*. Por ejemplo:

```
play(getDocumentBase(), "sonidox.midi");
```

Si necesitamos tener control sobre la reproducción, podemos utilizar la funcionalidad proporcionada por la interfaz **AudioClip** del paquete **java.applet**, la cual proporciona los siguientes métodos:

play: inicia la reproducción de un fichero de sonido.
loop: repite indefinidamente la reproducción de un fichero de sonido.
stop: detiene la reproducción de un fichero de sonido.

Para utilizar estos métodos, es necesario crear un objeto **AudioClip** y cargar en él, el fichero de sonido, mediante el método **getAudioClip** del *applet*. Por ejemplo:

```
AudioClip sonido =  getAudioClip(getDocumentBase(), "sonidox.midi");
```

Si lo que necesitamos es incorporar sonido a una "aplicación", entonces utilizaremos el método **static** de **Applet** denominado **newAudioClip**. Por ejemplo:

```
java.net.URL url = null;
java.applet.AudioClip sonido = null;
try {
        url = new java.net.URL(getDocumentBase(), "sonidox.midi");
} catch(java.net.MalformedURLException e) {}
sonido = java.applet.Applet.newAudioClip(url);
```

8.10.-Comunicación del Applet con el Browser. La comunicación entre el applet y el browser en el que se está ejecutando se puede controlar mediante la interface ***AppletContext*** (package ***java.applet***). ***AppletContext*** es una interface implementada por el browser, cuyos métodos pueden ser utilizados por el ***applet*** para obtener información y realizar ciertas operaciones, como por ejemplo sacar ***mensajes breves*** en la ***barra de estado*** del browser. Hay que tener en cuenta que la barra de estado es

compartida por el browser y las ***applets***, lo que tiene el peligro de que el mensaje sea rápidamente sobre-escrito por el browser u otros ***applets*** y que el usuario no llegue a enterarse del mensaje.

Los mensajes breves a la barra de estado se producen con el método ***showStatus()***, como porejemplo,

getAppletContext().showStatus("Cargado desde el fichero " + filename);

Los mensajes más importantes se deben dirigir a la ***salida estándar*** o a la ***salida de errores***, que en ***Netscape Navigator*** es la ***Java Console*** (la cual se hace visible desde el menú ***Options*** en ***Navigator 3.0***, desde el menú ***Communicator*** en ***Navigator 4.0**** y desde ***Communicator/Tools*** en ***Navigator 4.5***). Estos mensajes se pueden enviar con las sentencias:

System.out.print();
System.out.println();
System.error.print();
System.error.println();

Para mostrar documentos HTML en una ventana del browser se pueden utilizar los métodos siguientes:

- ***showDocument(URL miUrl, [String target])***, que muestra un documento HTML en el *frame* del browser indicado por *target* (*name*, *_top*, *_parent*, *_blank*, *_self*).
- ***showDocument(URL miUrl)***, que muestra un documento HTML en la ventana actual del browser.

Un ***applet*** puede conseguir información de otros ***applets*** que están corriendo en la misma página del browser, enviarles mensajes y ejecutar sus métodos. El mensaje se envía invocando los métodos del otro ***applet*** con los argumentos apropiados.

Algunos browsers exigen, para que los ***applets*** se puedan comunicar, que los ***applets*** provengan del mismo browser o incluso del mismo directorio (que tengan el mismo *codebase*).

Por ejemplo, para obtener información de otros applets se pueden utilizar los métodos:

- ***getApplet(String name)***, que devuelve el ***applet*** llamada ***name*** (o ***null*** si no la encuentra). El nombre del ***applet*** se pone con el atributo opcional NAME o con el parámetro NAME.
- ***getApplets()***, que devuelve una *enumeración* con todas las ***applets*** de la página.

Para poder utilizar todos los métodos de un applet que se está ejecutando en la misma página HTML (y no sólo los métodos comunes heredados de ***Applet***), debe hacerse un ***cast*** del objeto de la clase ***Applet*** que se obtiene como valor de retorno de ***getApplet()*** a la clase concreta del ***applet***.

Para que pueda haber ***respuesta*** (es decir, comunicación en los dos sentidos), el primer applet que envía un mensaje debe enviar una referencia a sí misma por medio del argumento ***this***.

Ejemplo.

```
// fichero Applet2.java
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
public class Applet2 extends java.applet.Applet {
    Font f = new Font("TimesRoman", Font.BOLD, 36);
    String parametro;

    public void paint(Graphics pantalla) {
        pantalla.setFont(f);
        pantalla.setColor(Color.red);
        pantalla.drawString(parametro, 5, 40);
    }

    public void init() {
        parametro=getParameter("Saludo");
        if (parametro==NULL)
            parametro="Esto es un ejemplo de Applet";
    }
}

// fichero Applet2.html
<HTML>
<HEAD>
<TITLE> Mi segundo ejemplo de Applet </TITLE>
</HEAD>
<BODY>
<APPLET CODE="Applet1.class" CODEBASE= "/JDK.2/APPLETS" WIDTH=600
HEIGHT=100>
<PARAM NAME = parametro VALUE = "Esto es un ejemplo de Applet"
</APPLET>
</BODY>
</HTML>
```

8.11.-Obtención de las propiedades del Sistema. Un ***applet*** puede obtener información del sistema o del entorno en el que se ejecuta. Sólo algunas propiedades del sistema son accesibles. Para acceder a las propiedades del sistema se utiliza un método ***static*** de la clase ***System***:

String salida = System.getProperty("file.separator");

Los nombres y significados de las propiedades del sistema accesibles son las siguientes:

"file.separator" Separador de directorios (por ejemplo, "/" o "\")
"java.class.version" Número de version de las clases de ***Java***

"java.vendor" Nombre específico del vendedor de Java
"java.vendor.url" URL del vendedor de Java
"java.version" Número de versión Java
"line.separator" Separador de líneas
"os.arch" Arquitectura del sistema operativo
"os.name" Nombre del sistema operativo
"path.separator" Separador en la variable Path (por ejemplo, ":"

No se puede acceder a las siguientes propiedades del sistema: "java.class.path", "java.home", "user.dir", "user.home", "user.name".

8.12.-Utilización de THREADS en Applets.

Un ***applet*** puede ejecutarse con varias ***threads***, y en muchas ocasiones será necesario o conveniente hacerlo así. Hay que tener en cuenta que un ***applet*** se ejecuta siempre en un browser (o en la aplicación ***appletviewer***).

Así, las ***threads*** en las que se ejecutan los métodos mayores -***init()***, ***start()***, ***stop()*** y ***destroy()***- dependen del browser o del entorno de ejecución. Los métodos gráficos -***paint()***, ***update()*** y ***repaint()***- se ejecutan siempre desde una ***thread*** especial del AWT.

Algunos browsers dedican un ***thread*** para cada ***applet*** en una misma página; otros crean un grupo de ***threads*** para cada ***applet*** (para poderlas matar al mismo tiempo, por ejemplo). En cualquier caso se garantiza que todas las ***threads*** creadas por los métodos mayores pertenecen al mismo grupo.

Se deben introducir ***threads*** en ***applets*** siempre que haya tareas que consuman mucho tiempo (cargar una imagen o un sonido, hacer una conexión a Internet, …). Si estas tareas pesadas se ponen en el método ***init()*** bloquean cualquier actividad del ***applet*** o incluso de la página HTML hasta que se completan. Las tareas pesadas pueden ser de dos tipos:

· Las que sólo se hacen una vez.
· Las que se repiten muchas veces.

Un ejemplo de tarea que se repite muchas veces puede ser una animación. En este caso, la tarea repetitiva se pone dentro de un bucle ***while*** o ***do…while***, dentro del ***thread***. El ***thread*** se debería crear dentro del método ***start()*** del ***applet*** y destruirse en ***stop()***. De este modo, cuando el ***applet*** no está visible se dejan de consumir recursos.

Al crear el ***thread*** en el método ***start()*** se pasa una referencia al ***applet*** con la palabra ***this***, que se refiere al ***applet***. El applet deberá implementar la interface ***Runnable***, y por tanto debe definir el método ***run()***, que es el centro del ***Thread.***

Un ejemplo de tarea que se realiza una sola vez es la carga de imágenes ****.gif*** o ****.jpeg***, que ya se realiza automáticamente en un ***thread*** especial.

Sin embargo, los sonidos no se cargan en ***threads*** especiales de forma automática; los debe crear el programador para cargarlos en "background". Este es un caso típico de programa ***producer-consumer***: el

thread es el *producer* y el ***applet*** el *consumer*. Las ***threads*** deben estar ***sincronizadas***, para lo que se utilizan los métodos ***wait()*** y ***notifyAll()***.

A continuación se presenta un ejemplo de ***thread*** con tarea repetitiva:

```
public void start() {
        if (repetitiveThread == null) {
                repetitiveThread = new Thread(this); // se crea un nuevo thread
        }
        repetitiveThread.start(); // se arranca el thread creado: start() llama a run()
}
public void stop() {
        repetitiveThread = null; // para parar la ejecución del thread
}
public void run() {
        ...
        while (Thread.currentThread() == repetitiveThread) {
        ... // realizar la tarea repetitiva.
        }
}
```

El método ***run()*** se detendrá en cuanto se ejecute el método ***stop()***, porque la referencia al ***thread*** está a ***null***.

8.13.-Applets que también son aplicaciones. Es muy interesante desarrollar ***aplicaciones*** que pueden funcionar también como ***applets*** y viceversa. En concreto, para hacer que un ***applet*** pueda ejecutarse como ***aplicación*** pueden seguirse las siguientes instrucciones:

1. Se añade un método ***main()*** a la clase ***MiApplet*** (que deriva de ***Applet***).

2. El método ***main()*** debe crear un objeto de la clase ***MiApplet*** e introducirlo en un ***Frame***.

3. El método ***main()*** debe también ocuparse de hacer lo que haría el browser, es decir, llamar a los métodos ***init()*** y ***start()*** de la clase ***MiApplet***.

4. Se puede añadir también una ***static inner class*** que derive de ***WindowAdapter*** y que gestione el evento de cerrar la ventana de la aplicación definiendo el método ***windowClosing()***. Este método llama al método ***System.exit(0)***. Según como sea el ***applet***, el método ***windowClosing()*** previamente deberá también llamar a los métodos ***MiApplet.stop()*** y ***MiApplet.destroy()***, cosa que para las ***applets*** se encarga de hacer el browser. En este caso conviene que el objeto de ***MiApplet*** creado por ***main()*** sea ***static***, en lugar de una variable local.

   A continuación se presenta un ejemplo:

   ```
   public class MiApplet extends Applet {
   ```

```
        ...
        public void init() {...}
        ...
        // clase para cerrar la aplicación
                static class WL extends WindowsAdapter {
                public void windowClosing(WindowEvent e) {
                MiApplet.stop();
                MiApplet.destroy();
                System.exit(0);
        }
        } // fin de WindowAdapter

        // programa principal
        public static void main(String[] args) {
                static MiApplet unApplet = new MiApplet();
                Frame unFrame = new Frame("MiApplet");
                unFrame.addWindowListener(new WL());
                unFrame.add(unapplet, BorderLayout.CENTER);
                unFrame.setSize(400,400);
                unApplet.init();
                unApplet.start();
                unFrame.setVisible(true);
        }
} // fin de la clase MiApplet
```

TEMA 9. THREADS: PROGRAMAS MULTITAREA.

9.1.- Introducción. Los procesadores y los Sistemas Operativos modernos permiten la ***multitarea***, es decir, la realización simultánea de dos o más actividades (al menos aparentemente). En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez. Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU: reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo, operaciones de lectura de datos desde el teclado) para trabajar en la otra. En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un ***hilo*** o ***thread*** diferente.

Un ***proceso*** es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un Sistema Operativo multitarea es capaz de ejecutar más de un ***proceso*** simultáneamente. Un ***thread*** o ***hilo*** es un ***flujo secuencial simple*** dentro de un ***proceso***. Un único ***proceso*** puede tener varios ***hilos*** ejecutándose. Por ejemplo el programa ***Netscape*** sería un proceso, mientras que cada una de las ventanas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un ***hilo***.

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario. Sin el uso de ***threads*** hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Los ***threads*** o ***hilos*** de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un ***hilo*** de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente. Bastará con indicar lo que tiene que hacer en el método ***run()***, que es el que define la actividad principal de las ***threads***.

Los ***threads*** pueden ser ***daemon*** o ***no daemon***. Son ***daemon*** aquellos hilos que realizan en ***background*** (en un segundo plano) servicios generales, esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un ***thread daemon*** podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de ***Java*** finaliza cuando sólo quedan corriendo ***threads*** de tipo ***daemon***. Por defecto, y si no se indica lo contrario, los ***threads*** son del tipo ***no daemon***.

9.2.-Creación de THREADS. En ***Java*** hay dos formas de crear nuevos ***threads***. La primera de ellas consiste en crear una nueva clase que herede de la clase ***java.lang.Thread*** y redefinir el método ***run()*** de dicha clase. El segundo método consiste en declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como argumento al constructor el objeto creado de la nueva clase (la que implementa la interface ***Runnable***). Como ya se ha apuntado, tanto la clase ***Thread*** como la interface ***Runnable*** pertenecen al package ***java.lang***, por lo que no es necesario importarlas.

A continuación se presentan dos ejemplos de creación de ***threads*** con cada uno de los dos métodos citados.

9.2.1.-Creación de threads derivando de la clase Thread. Considérese el siguiente ejemplo de declaración de una nueva clase:

```
public class SimpleThread extends Thread {
        // constructor
        public SimpleThread (String str) {
                super(str);
        }
        // redefinición del método run()
        public void run() {
                for(int i=0;i<10;i++)
                        System.out.println("Este es el thread : " + getName());}
}
```

En este caso, se ha creado la clase ***SimpleThread***, que hereda de ***Thread***. En su constructor se utiliza un ***String*** (opcional) para poner nombre al nuevo ***thread*** creado, y mediante ***super()*** se llama al constructor de la super-clase ***Thread***. Asimismo, se redefine el método ***run()***, que define la principal actividad del ***thread***, para que escriba 10 veces el nombre del ***thread*** creado.

Para poner en marcha este nuevo ***thread*** se debe crear un objeto de la clase ***SimpleThread***, y llamar al método ***start()***,heredado de la super-clase ***Thread***, que se encarga de llamar a ***run()***. Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");
miThread.start();
```

9.2.2.-Creación de threads implementando la interface Runnable. Esta segunda forma también requiere que se defina el método ***run()***, pero además es necesario crear un objeto de la clase ***Thread*** para lanzar la ejecución del nuevo hilo. Al constructor de la clase ***Thread*** hay que pasarle una referencia del objeto de la clase que implementa la interface ***Runnable***. Posteriormente, cuando se ejecute el método ***start()*** del ***thread***, éste llamará al método ***run()*** definido en la nueva clase. A continuación se muestra el mismo estilo de clase que en el ejemplo anterior implementada mediante la interface ***Runnable***:

```
public class SimpleRunnable implements Runnable {
        // se crea un nombre
        String nameThread;
        // constructor
        public SimpleRunnable (String str) {
                nameThread = str;
        }
        // definición del método run()

        public void run() {
                for(int i=0;i<10;i++)
                        System.out.println("Este es el thread: " + nameThread);
        }
}
```

El siguiente código crea un nuevo ***thread*** y lo ejecuta por este segundo procedimiento:

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
// se crea un objeto de la clase Thread pasándolo el objeto Runnable como argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

Este segundo método cobra especial interés con las ***applets***, ya que cualquier ***applet*** debe heredar de la clase ***java.applet.Applet***, y por lo tanto ya no puede heredar de ***Thread***. Véase el siguiente ejemplo:

```
class ThreadRunnable extends Applet implements Runnable {
        private Thread runner=null;
        // se redefine el método start() de Applet
        public void start() {
                if (runner == null) {
                        runner = new Thread(this);
                        runner.start(); // se llama al método start() de Thread
                }
        }
        // se redefine el método stop() de Applet
        public void stop(){
                runner = null; // se libera el objeto runner
        }
}
```

En este ejemplo, el argumento ***this*** del constructor de ***Thread*** hace referencia al objeto ***Runnable*** cuyo método ***run()*** debería ser llamado cuando el hilo ejecutado es un objeto de ***ThreadRunnable***.

La elección de una u otra forma -derivar de ***Thread*** o implementar ***Runnable***- depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

9.3.-Ciclo de vida de un THREAD.

En el apartado anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. Un ***thread*** puede presentar cuatro estados distintos:

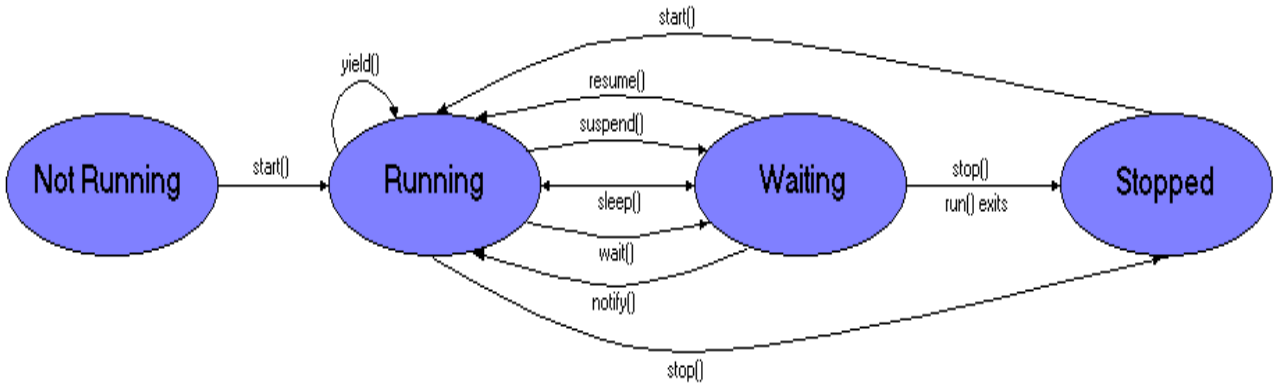
1. *Nuevo* (***New-Not Running***): El ***thread*** ha sido creado pero no inicializado, es decir, no se ha ejecutado todavía el método ***start()***. Se producirá un mensaje de error (***IllegalThreadStateException***) si se intenta ejecutar cualquier método de la clase ***Thread*** distinto de ***start()***.

2. *Ejecutable* (***Runnable-Running***): El ***thread*** puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado en beneficio de otro ***thread***.

3. *Bloqueado* (***Blocked*** o ***Not Runnable-Waiting***): El ***thread*** podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un t***hread*** está en este estado, no se le asigna tiempo de CPU.

4. *Muerto* (***Dead-Stopped***): La forma habitual de que un ***thread*** muera es finalizando el método ***run()***.

También puede llamarse al método ***stop()*** de la clase ***Thread***, aunque dicho método es considerado "peligroso" y no se debe utilizar.



A continuación se explicarán con mayor detenimiento los puntos anteriores.

9.3.1.-Ejecución de un nuevo thread. La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo. Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

El método ***start()*** se encarga de llamar al método ***run()*** de la clase ***Thread***. Si el nuevo ***thread*** se ha creado heredando de la clase ***Thread*** la nueva clase deberá redefinirir el método ***run()*** heredado. En el caso de utilizar una clase que implemente la interface ***Runnable***, el método ***run()***de la clase ***Thread*** se ocupa de llamar al método ***run()*** de la nueva clase.

Una vez que el método ***start()*** ha sido llamado, se puede decir ya que el ***thread*** está "corriendo" (***running***), lo cual no quiere decir que se esté ejecutando en todo momento, pues ese ***thread*** tiene que compartir el tiempo de la CPU con los demás ***threads*** que también estén ***running***. Por eso más bien se dice que dicha ***thread*** es ***runnable***.

9.3.2.-Detener un Thread temporalmente. El sistema operativo se ocupa de asignar tiempos de CPU a los distintos ***threads*** que se estén ejecutando simultáneamente. Aun en el caso de disponer de un ordenador con más de un procesador (2 ó más CPUs), el número de ***threads*** simultáneos suele siempre superar el número de CPUs, por lo que se debe repartir el tiempo de forma que parezca que todos los procesos corren a la vez (quizás más lentamente), aun cuando sólo unos pocos pueden estar ejecutándose en un instante de tiempo.

Los tiempos de CPU que el sistema continuamente asigna a los distintos ***threads*** en estado ***runnable*** se utilizan en ejecutar el método ***run()*** de cada ***thread***. Por diversos motivos, un ***thread*** puede en un determinado momento renunciar "voluntariamente" a su tiempo de CPU y otorgárselo al sistema para que se lo asigne a otro ***thread***. Esta "renuncia" se realiza mediante el método ***yield()***. Es importante que este método sea utilizado por las actividades que tienden a "monopolizar" la CPU. El método ***yield()*** viene a indicar que en ese momento no es muy importante para ese ***thread*** el ejecutarse continuamente y por lo tanto tener ocupada la CPU. En caso de que ningún ***thread*** esté requiriendo la CPU para una actividad muy intensiva, el sistema volverá casi de inmediato a asignar nuevo tiempo al ***thread*** que fue "generoso" con los demás. Por ejemplo, en un Pentium II 400 Mhz es posible llegar a más de medio millón de llamadas por segundo al método ***yield()***, dentro del método ***run()***, lo que significa que llamar al método ***yield()*** apenas detiene al ***thread***, sino que sólo ofrece el control de la CPU para que el sistema decida si hay alguna otra tarea que tenga mayor prioridad.

Si lo que se desea es parar o bloquear temporalmente un ***thread*** (pasar al estado ***Not Runnable***), existen varias formas de hacerlo:

1. A través del método **sleep(**nº ms**)**, lleva al hilo a un estado de "dormido", durante un número de milisegundos concreto. El hilo vuelve a ejecutarse cuando pase el nº de ms indicado. Este método lanza la excepción *InterruptedException*, por ello hay que incluirlo dentro de un bloque **try..catch** .

2. A través del método **wait()** de la clase *Objet*, el hilo queda esperando durante un nº de ms o hasta que sea despertado a través de *notify()* o *notifyAll()*. El método *wait()* hace que el *thread* de ejecución espere en estado dormido hasta que se le notifique que continúe. El método *notify()* informa de que un *thread* (no se sabe cuál) y sólo uno, en espera, debe ser despertado (continúe con su ejecución). El método *notifyAll()* es similar a *notify()* excepto que se aplica a todos los hilos en espera. Estos métodos se suelen utilizar para codificar *bloques de sincronización* o *secciones críticas* para sincronizar hilos que necesitan acceder a un mismo recurso: dato, archivo, base de datos... ejemplo: productor-consumidor y por tanto, deben utilizarse siempre dentro de un método *synchronized*.

3. Cuando el hilo está esperando operaciones de E/S.

4. La función miembro *suspend()* de la clase *Thread* permite tener un control sobre el hilo de modo que podamos desactivarlo, detener su actividad durante un intervalo de tiempo indeterminado. Este método no detiene la ejecución permanentemente. El hilo es suspendido

indefinidamente y para volver a activarlo de nuevo necesitamos realizar una invocación a la función miembro *resume()*.

Un ***thread*** pasa automáticamente del estado ***Not Runnable*** a ***Runnable*** cuando cesa alguna de las condiciones anteriores o cuando se llama a ***notify()*** o ***notifyAll()***.

El método ***sleep()*** de la clase ***Thread*** recibe como argumento el tiempo en *milisegundos* que ha de permanecer detenido. Adicionalmente, se puede incluir un número entero con un tiempo adicional en *nanosegundos*. Las declaraciones de estos métodos son las siguientes:

```
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanosecons) throws InterruptedException
```

Considérese el siguiente ejemplo:

```
System.out.println ("Contador de segundos");
int count=0;
public void run () {
        try {
                sleep(1000);
                System.out.println(count++);
        } catch (InterruptedException e){}
}
```

Se observa que el método ***sleep()*** puede lanzar una ***InterruptedException*** que ha de ser capturada. Así se ha hecho en este ejemplo, aunque luego no se gestiona esa excepción.

La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***. La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos. Hay dos formas de llamar a ***wait()***:

1. Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado. Si en ese plazo se ejecutan los métodos ***notify()*** o ***notifyAll()*** que indican la liberación de los objetos bloqueados, el ***thread*** continuará sin esperar a concluir el tiempo indicado. Las dos declaraciones del método ***wait()*** son como siguen:

   ```
   public final void wait(long timeout) throws InterruptedException
   public final void wait(long timeout, int nanos) throws InterruptedException
   ```

2. Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

   ```
   public final void wait() throws InterruptedException
   ```

Los métodos ***wait()*** y ***notify()*** han de estar incluidas en un método ***synchronized,*** ya que de otra forma se obtendrá una excepción del tipo ***IllegalMonitorStateException*** en tiempo de ejecución. El uso típico de ***wait()*** es el de esperar a que se cumpla alguna determinada condición, ajena al propio ***thread***. Cuando ésta se cumpla, se utilizará el método ***notifyAll()*** para avisar a los distintos ***threads*** que pueden utilizar el objeto.

9.3.3.-Finalizar un Thread. Un ***thread*** finaliza cuando el método ***run()*** devuelve el control, por haber terminado lo que tenía que hacer (por ejemplo, un bucle ***for*** que se ejecuta un número determinado de veces) o por haberse dejado de cumplir una condición (por ejemplo, por un bucle ***while*** en el método ***run()***). Es habitual poner las siguientes sentencias en el caso de ***Applets Runnables***:

```
public class MyApplet extends Applet implements Runnable {
    // se crea una referencia tipo Thread
    private Thread AppletThread;
    ...
    // método start() del Applet
    public void start() {
        if(AppletThread == null){ // si no tiene un objeto Thread asociado
            AppletThread = new Thread(this, "El propio Applet");
            AppletThread.start(); // se arranca el thread y llama a run()
        }
    }
    // método stop() del Applet
    public void stop() {
        AppletThread = null; // iguala la referencia a null
    }
    // método run() por implementar Runnable
    public void run() {
        Thread myThread = Thread.currentThread();
        while (myThread == AppletThread) { // hasta que se ejecute stop()
            ...                                  // código a ejecutar
        }
    }
} // fin de la clase MyApplet
```

donde ***AppletThread*** es el ***thread*** que ejecuta el método ***run()*** MyApplet. Para finalizar el thread basta poner la referencia ***AppletThread*** a ***null***. Esto se consigue en el ejemplo con el método ***stop()*** del ***applet*** (distinto del método ***stop()*** de la clase ***Thread***, que no conviene utilizar).

Para saber si un ***thread*** está "vivo" o no, es útil el método ***isAlive()*** de la clase ***Thread***, que devuelve ***true*** si el ***thread*** ha sido inicializado y no parado, y ***false*** si el ***thread*** es todavía nuevo (no ha sido inicializado) o ha finalizado. Por ejemplo:

```
public static void main(String args[]) {
        MiThread t = new MiThread();
        System.out.println("isAlive() antes de iniciar: "+t.isAlive());
        t.start();
        System.out.println("isAlive() en ejecución: "+t.isAlive());}
```

La salida correspondiente al programa es:

```
isAlive()  antes de iniciar: false
isAlive()  en ejecución: true
```

Devolverá true en caso de que el hilo *t* esté vivo, es decir, ya se haya llamado a su método *run()* y no haya terminado su ejecución. En otro caso, devolverá false. ¿Qué pasaría si el hilo hubiese sido bloqueado o dormido?.

A veces necesitamos esperar a que finalice un hilo o grupo de ellos para seguir adelante con otras tareas, pero como los hilos se ejecutan concurrentemente con el programa que los lanzó, esto puede provocar salidas de programa no deseadas relacionadas con el orden de ejecución de las acciones programadas. Si queremos que una acción se realice después de finalizar un hilo hay que esperar a que éste acabe, utilizando el método *join()***.** Este método puede incluir como parámetro el nº de ms que queremos esperar a que acabe el hilo. En el ejemplo que viene a continuación, para que el mensaje del método *run()* se visualice en el orden correcto, será necesario llamar al método *join()*:

Ejemplo:

```
public class Hilojoin extends Thread{
   private String nombre;
   public Hilojoin(String nombre)
   {
      this.nombre=nombre;
   }
   public void run(){
      try
     {
                int x=(int)(Math.random()*5000);
                Thread.sleep(x);
                System.out.println("Soy "+nombre+ " ("+x+")");
      }
      catch (Exception ex){ }
   }
   public static void main(String[] args) throws InterruptedException  {
      Hilojoin t1 = new Hilojoin("Pedro");
      Hilojoin t2 = new Hilojoin("Pablo");
      Hilojoin t3 = new Hilojoin("Juan");
      t1.start();
```

```
        t2.start();
        t3.start();
        t1.join();
       t2.join();
       t3.join();
        System.out.println("fin del proceso");
    }
}
```

¿Qué pasaría si no utilizásemos el método *join()*?

9.4.-Sincronización. El problema de la sincronización de hilos tiene lugar cuando varios hilos intentan acceder a los mismos recursos o datos. Por ejemplo: un marido y una mujer intentan sacar dinero a la vez de su cuenta en distintos cajeros, ¿Qué pasaría si quisieran sacar 100€ cada uno y su saldo fuese de 150?. El acceso a los recursos compartidos (o *recursos críticos*), debe ser monitorizado. El fragmento de código que manipula esos recursos se llama **sección crítica**. La sección crítica debe ser mutuamente excluyente, es decir, que si un hilo está ejecutando su sección crítica, hay que asegurarse de que ningún otro hilo la esté ejecutando.

Cada **objeto** derivado de la clase *Object* tiene asociado un elemento de sincronización o ***lock*** intrínseco, que afecta a la ejecución de los métodos definidos como *synchronized* en el objeto:

- Cuando un objeto ejecuta un método *synchronized*, toma el *lock*, y cuando termina de ejecutarlo lo libera.
- Cuando un *thread* tiene tomado el *lock*, ningún otro *thread* puede ejecutar ningún otro método *synchronized* del mismo objeto.
- El *thread* que ejecuta un método *synchronized* de un objeto cuyo *lock* se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.

Cada **clase Java** derivada de *Object*, tiene también un mecanismo *lock* asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados *synchronized*.

Java dispone del modificador *synchronized* que aplicado a un método, garantiza que éste se ejecuta de forma excluyente. Una clase con un método *synchronized* se la llama **monitor**, (tubería, buffer, CubbyHole) porque dentro de éste se estará monitorizando algún recurso crítico.
Por ejemplo, si queremos que la variable c de este programa sea manipulada correctamente, en Java habría que codificarlo así:

```
public class SynchronizedCounter {
   private int c = 0;
   public synchronized void increment() {
```

```
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Java permite sincronizar una parte del código de un método. Para ello se utiliza la palabra clave ***syncronized*** indicando entre paréntesis el objeto que se desea sincronizar. Por ejemplo si se desea sincronizar el propio ***thread*** en una parte del método ***run()***, el código podría ser:

```
public void run() {
        while(true) {
                ...
                syncronized(this) {  // El objeto a sincronizar es el propio thread
                ... // Código sincronizado
                }
                ...
                Resto de instrucciones accesibles, porque no están sincronizadas
        }
}
```

Por otro lado este mecanismo nos permite la comunicación entre tareas, es decir se pueden diseñar tareas para utilizar objetos comunes, que cada tarea utiliza independientemente de la otra. Este es el caso del clásico del productor-consumidor, en el que contaremos con 2 hilos: el productor que genera, crea o produce elementos y los guarda y el consumidor que los extrae, saca o consume. Pueden darse muchas situaciones:

- los 2 quieren acceder a la vez al almacén,
- el productor genera a distinta velocidad que el consumidor,
- el productor ha llenado su reserva y no puede producir más hasta que el consumidor no saque algo,
- el consumidor no puede acceder porque el almacén está vacío...

Debe establecerse un mecanismo que controle el acceso de ambos al almacén (sincronización de tareas).

9.4.1.- Ejemplo de productor-consumidor sin monitor. En este ejemplo el productor genera un entero entre 0 y 9 (inclusive), lo almacena en un objeto "*CubbyHole*", e imprime el número generado, respetando el orden de introducción. Además, el productor duerme durante un tiempo antes de repetir el ciclo de generación de números.

```
class Productor extends Thread {
        private CubbyHole cubbyhole;
        private int numero;
        public Productor(CubbyHole c, int numero) {
                cubbyhole = c;
                this.numero = numero;  // retardo aplicado al productor
        }
        public void run() {
              for (int i = 0; i < 10; i++) {
                 cubbyhole.put(i);
                 System.out.println("Productor  pone: " +i);
                 try {
                    sleep(numero);
                 } catch (InterruptedException e) {}
              }
        }
}
```

El consumidor, por su parte, está "hambriento", consume los enteros de *CubbyHole* una vez y en el mismo orden que fueron introducidos, tan pronto como estén disponibles.

```
class Consumidor extends Thread {
        private CubbyHole cubbyhole;
        private int numero;
        public Consumidor(CubbyHole c, int numero) {
                cubbyhole = c;
                this.numero = numero;
        }
        public void run() {
              int value = 0;
              for (int i = 0; i < 10; i++) {
                 value = cubbyhole.get();
                 System.out.println("Consumidor saca:"+value);
                 try {
                    sleep(numero);
                 } catch (InterruptedException e) {}
              }
        }
}
```

CubbyHole se encarga de poner y extraer los datos del productor y consumidor

```
class CubbyHole {
   private int contents;
```

```
    public int get() {
       return contents;
    }

    public  void put(int value) {
      contents = value;
    }
}
```

La clase que utiliza los objetos de las clases creadas sería la siguiente:

```
public class TestMonitor {
    public static void main(String[] args) {
            CubbyHole b=new CubbyHole();
            Productor p=new Productor(b,1000);
            Consumidor c=new Consumidor(b,2000);
            p.start();
            c.start();
    }
}
```

En este ejemplo, el Productor y el Consumidor comparten datos a través de un objeto *CubbyHole* común. Ninguno de los dos controla que el consumidor obtiene cada valor producido una vez. La sincronización entre estos dos hilos ocurre a un nivel inferior, dentro de los métodos *get()* y *put()* del objeto *CubbyHole*. Sin embargo, si estos dos hilos no están sincronizados los problemas que podría provocar esta situación son:

1. El Productor es más rápido que el Consumidor y generara dos números antes de que el Consumidor tuviera una posibilidad de consumir el primer número. Así el Consumidor se saltaría un número. Parte de la salida se podría parecer a esto:

   Consumidor #1 obtiene: 3
   Productor #1 pone: 4
   Productor #1 pone: 5
   Consumidor #1 obtiene: 5

2. El consumidor es más rápido que el productor y consumiera el mismo valor dos o más veces. En esta situación el Consumidor imprimirá el mismo valor dos veces y podría producir una salida como esta.

   Productor #1 pone: 4
   Consumidor #1 obtiene: 4
   Consumidor #1 obtiene: 4
   Productor #1 pone: 5

De cualquier forma, el resultado es erróneo. Se quiere que el consumidor obtenga cada entero producido por el productor y sólo una vez. Los problemas anteriores, se llaman "condiciones de carrera". Suceden cuando varios hilos ejecutados asíncronamente intentan acceder a un mismo objeto al mismo tiempo y obtienen resultados erróneos. Para prevenir estas condiciones en nuestro ejemplo Productor/Consumidor, el almacenamiento de un nuevo entero en *CubbyHole* por el Productor debe estar sincronizado con la recuperación del entero por parte del Consumidor. El Consumidor debe consumir cada entero exactamente una vez. Esto se resuelve a través de los monitores.

9.4.2.- <u>Ejemplo de productor-consumidor con monitor (buffer, tubería, CubbyHole).</u> A los objetos como *CubbyHole*, a los que acceden varios hilos, son llamados "*condiciones variables*" o ***recursos críticos***. Una de las formas de controlar el acceso a estas condiciones variables o *recursos críticos*, y por tanto de sincronizar los hilos, son los **monitores** (o clases que cuentan con secciones críticas). Las ***secciones críticas*** son los segmentos del código donde los hilos concurrentes acceden a las *condiciones variables*. Estas secciones, en Java, se marcan normalmente con la palabra reservada **synchronized:**

synchronized int miMetodo();

Generalmente, las *secciones críticas* en los programas de Java son métodos, o fragmentos de código de un método. Java asocia un solo monitor a cada objeto que tiene un método sincronizado. El ejemplo del productor-consumidor tiene dos métodos de sincronización: *put()*, que cambia el valor de *CubbyHole*, y *get()*, para recuperar el valor actual. Este sería el código fuente del objeto *CubbyHole* utilizando técnicas de sincronización.

```
class CubbyHole {
        private int contents;
        private boolean available = false;
        public synchronized int get() {
                while (available == false) {
                        try {
                                wait();
                        } catch (InterruptedException e) {}
                }
                available = false;
                notify();
                return contents;
        }
        public synchronized void put(int value) {
                while (available == true) {
                        try {
                                wait();
                        } catch (InterruptedException e) {}
                }
                contents = value;
                available = true;
                notify();
```

```
        }
    }
```

¿Qué sucedería si estos métodos no estuviesen señalados como ***sección crítica***?. Probad el ejemplo con el monitor sin sincronizar, con el productor durmiendo, con el consumidor durmiendo, estando los 2 durmiendo, estando los dos despiertos, observar lo que pasa con *wait()* y notify() cuando el bloque no está sincronizado.

Cuando el Productor invoca el método *put()* de *CubbyHole*, adquiere el *lock* del objeto *CubbyHole* y por lo tanto el Consumidor no podrá llamar a *get()* de *CubbyHole* y se quedará bloqueado (el método, *wait()* que libera temporalmente el monitor). Lo mismo sucede cuando el Consumidor invoca *get()*.

La variable *contents* tiene el valor actual de *CubbyHole* y *available* indica si se puede recuperar o no el valor. Cuando *available* es verdadero, el productor aún no ha acabado de producir. *CubbyHole* tiene dos métodos de sincronización, y Java proporciona un solo *lock* para cada ejemplar de *CubbyHole* (incluyendo el compartido por el Productor y el Consumidor). Siempre que el control entra en un método sincronizado, el hilo que ha llamado al método adquiere el *lock* del objeto al cual pertenece el método. Otros hilos no pueden llamar a un método sincronizado del mismo objeto mientras el *lock* no sea liberado.

Hay más formas de codificar monitores: con ***semáforos***, ***cerrojos***, ***colas***...

Hay otros problemas clásicos de concurrencia: cena de filósofos, donde hay más filósofos que tenedores para comer, (puede provocarse un *deadlock* o bloqueo porque todos se queden esperando a que alguno suelte un tenedor), lectores y escritores donde varios escritores pueden leer y solo uno escribir.

La sincronización es un proceso que lleva bastante tiempo a la CPU, luego se debe minimizar su uso, ya que el programa será más lento cuanta más sincronización incorpore.

9.5.-Prioridades.

Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***threads***, de forma que se produzca un reparto más eficiente de los recursos disponibles. Así, en un determinado momento, interesará que un determinado proceso acabe lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos. La forma de llevar a cabo esto es gracias a las prioridades.

Cuando se crea un nuevo ***thread***, éste hereda la prioridad del ***thread*** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase ***Thread***, que toman valores enteros que oscilan entre la máxima prioridad ***MAX_PRIORITY*** (normalmente tiene el valor 10) y la mínima prioridad ***MIN_PRIORITY*** (valor 1), siendo la prioridad por defecto ***NORM_PRIORITY*** (valor 5). Para modificar la prioridad de un ***thread*** se utiliza el método ***setPriority()***. Se obtiene su valor con ***getPriority()***.

El algoritmo de distribución de recursos en ***Java*** escoge por norma general aquel ***thread*** que tiene una prioridad mayor, aunque no siempre ocurra así, para evitar que algunos procesos queden "dormidos". Cuando hay dos o más ***threads*** de la misma prioridad (y además, dicha prioridad es la más elevada), el sistema no establecerá prioridades entre los mismos, y los ejecutará alternativamente dependiendo del sistema operativo en el que esté siendo ejecutado. Si dicho SO soporta el "time-slicing" (reparto del tiempo de CPU), como por ejemplo lo hace Windows 95/98/NT, los ***threads*** serán ejecutados alternativamente.

Un ***thread*** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro ***thread*** de la misma prioridad, mediante el método ***yield()***, aunque en ningún caso a un ***thread*** de prioridad inferior.

9.6.-Grupos de THREADS. Todo hilo de ***Java*** debe formar parte de un grupo de hilos (***ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgroup***, no puede cambiar de grupo.

Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***main***. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current thread***, respectivamente). Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***).

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***thread***, según uno de los siguientes constructores:

public Thread (ThreadGroup grupo, Runnable destino)
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)

A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGroup***. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***main***). La clase ***ThreadGroup*** tiene dos posibles constructores:

ThreadGroup(ThreadGroup parent, String nombre);
ThreadGroup(String name);

el segundo de los cuales toma como ***parent*** el ***threadgroup*** al cual pertenezca el ***thread*** desde el que se crea (***Thread.currentThread()***). Para más información acerca de estos constructores, dirigirse a la documentación del API de ***Java*** donde aparecen numerosos métodos para trabajar con ***grupos de threads*** a disposición del usuario (***getMaxPriority()***, ***setMaxPriority()***, ***getName()***, ***getParent()***, ***parentOf()***).

En la práctica los ***ThreadGroups*** no se suelen utilizar demasiado. Su uso práctico se limita a efectuar determinadas operaciones de forma más simple que de forma individual. En cualquier caso, véase el siguiente ejemplo:

```
ThreadGroup miThreadGroup = new ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup, ”un thread para mi grupo");
```

donde se crea un grupo de ***threads*** (***miThreadGroup***) y un ***thread*** que pertenece a dicho ***grupo*** (***miThread)***.

TEMA 10. EXCEPCIONES.

10.1.-Introducción. A diferencia de otros lenguajes de programación orientados a objetos como C/C++, ***Java*** incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje ***Java***, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas ***excepciones*** son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos "estilos" de hacer esto:

1. ***A la "antigua usanza"***: los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de ***if elseif*** …, gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.

2. ***Con soporte en el propio lenguaje***: En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o ***Exceptions***. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y ***Java***.

En los siguientes apartados se examina cómo se trabaja con los bloques y expresiones ***try***, ***catch***, ***throw***, ***throws*** y ***finally***, cuándo se deben lanzar excepciones, cuándo se deben capturar y cómo se crean las clases propias de tipo ***Exception***.

10.2.-Excepciones estándar en JAVA. Los errores se representan mediante dos tipos de clases derivadas de la clase ***Throwable***: ***Error*** y ***Exception***.

La clase ***Error*** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son ***irrecuperables*** y no dependen del programador ni debe preocuparse de capturarlos y tratarlos.

La clase ***Exception*** tiene más interés. Dentro de ella se puede distinguir:

1. ***RuntimeException***: Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar ***excepciones implícitas***.

2. Las demás clases derivadas de ***Exception*** son ***excepciones explícitas***. ***Java*** obliga a tenerlas en cuenta y chequear si se producen.

El caso de ***RuntimeException*** es un poco especial. El propio ***Java*** durante la ejecución de un programa chequea y lanza automáticamente las excepciones que derivan de ***RuntimeException***. El programador no necesita establecer los bloques ***try***/***catch*** para controlar este tipo de excepciones. Representan dos casos de errores de programación:

1. Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia ***null*** en un método.

2. Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un *ArrayIndexOutOfBoundsException* automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequear continuamente todo tipo de errores (que las ***referencias*** son distintas de ***null***, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de ***Exception*** pueden pertenecer a distintos packages de ***Java***. Algunas perenecen a ***java.lang*** (*Throwable*, *Exception*, *RuntimeException*, …); otras a ***java.io*** (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de ***Throwable*** todos los tipos de excepciones pueden usar los métodos siguientes:

1. String ***getMessage()*** Extrae el mensaje asociado con la excepción.

2. String ***toString()*** Devuelve un String que describe la excepción.

3. void ***printStackTrace()*** Indica el método donde se lanzó la excepción.

10.3.-Lanzar una Excepción.

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzamiento de una excepción es el siguiente:

1. Se crea un objeto ***Exception*** de la clase adecuada.

2. Se lanza la excepción con la sentencia ***throw*** seguida del objeto ***Exception*** creado.

```
/* Código que lanza la excepción MyException una vez detectado el error */
MiExcepcion me = new MiExcepcion("mensage");
throw me;
```

Esta excepción deberá ser capturada (***catch***) y gestionada en el propio método o en algún otro lugar del programa (en otro método anterior en la ***pila*** o ***stack*** de llamadas), según se explica en el Apartado siguiente.

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques ***try/catch/finally*** se ejecutará el bloque ***catch*** que la captura o el bloque ***finally*** (si existe).

Todo método en el que se puede producir uno o más tipos de excepciones (y que no utiliza directamente los bloques ***try/catch/finally*** para tratarlos) debe ***declararlas*** en el encabezamiento de la función por medio de la palabra ***throws***. Si un método puede lanzar varias excepciones, se ponen detrás de ***throws*** separadas por comas, como por ejemplo:

public void leerFichero(String fich) throws EOFException, FileNotFoundException {…}

Se puede poner únicamente una ***superclase de excepciones*** para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. El caso anterior sería equivalente a:

public void leerFichero(String fich) throws IOException {…}

Las excepciones pueden ser lanzadas directamente por ***leerFichero()*** o por alguno de los métodos llamados por ***leerFichero()***, ya que las clases ***EOFException*** y ***FileNotFoundException*** derivan de ***IOException***.

Se recuerda que no hace falta avisar de que se pueden lanzar objetos de la clases ***Error*** o ***RuntimeException*** (excepciones implícitas).

10.4.-Captura de una Excepción.

Como ya se ha visto, ciertos métodos de los packages de ***Java*** y algunos métodos creados por cualquier programador producen ("lanzan") excepciones. Si el usuario llama a estos métodos sin tenerlo en cuenta se produce un error de compilación con un mensaje del tipo: "*… Exception java.io.IOException must be caugth or it must be declared in the throws clause of this method*". El programa no compilará mientras el usuario no haga una de estas dos cosas:

1. ***Gestionar la excepción*** con una construcción del tipo ***try {…} catch {…}***.

2. ***Re-lanzar la excepción*** hacia un método anterior en el ***stack***, declarando que su método también lanza dicha excepción, utilizando para ello la construcción ***throws*** en el header del método.

El compilador obliga a capturar las llamadas ***excepciones explícitas***, pero no protesta si se captura y luego no se hace nada con ella. En general, es conveniente por lo menos imprimir un mensaje indicando qué tipo de excepción se ha producido.

10.4.1.-Bloques try y catch.

En el caso de las excepciones que no pertenecen a las ***RuntimeExceptio***n y que por lo tanto ***Java*** obliga a tenerlas en cuenta habrá que utilizar los bloques ***try***, ***catch*** y ***finally***. El código dentro del bloque ***try*** está "vigilado": Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque ***try*** y pasa al bloque ***catch***, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques ***catch*** como sean necesarios, cada uno de los cuales tratará un tipo de excepción.

Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas.

El bloque ***finally*** es opcional. Si se incluye sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna. El bloque ***finally*** se ejecuta aunque en el bloque ***try*** haya un ***return***.

En el siguiente ejemplo se presenta un método que debe "controlar" una ***IOException*** relacionada con la lectura ficheros y una ***MyException*** propia:

```
void metodo1(){
    ...
    try {
        /* Código que puede lanzar las excepciones IOException y MyException */
    } catch (IOException e1) {/* Se ocupa de IOException simplemente dando aviso */
        System.out.println(e1.getMessage());
    } catch (MyException e2) {
        /* Se ocupa de MyException dando un aviso y finalizando la función */
        System.out.println(e2.getMessage()); return;
    } finally { /* Sentencias que se ejecutarán en cualquier caso */
        ...
    }
} // Fin del metodo1
```

10.4.2.-Relanzar una Exception. Existen algunos casos en los cuales el código de un método puede generar una ***Exception*** y no se desea incluir en dicho método la gestión del error. ***Java*** permite que este método pase o relance (***throws***) la ***Exception*** al método desde el que ha sido llamado, sin incluir en el método los bucles ***try/catch*** correspondientes. Esto se consigue mediante la adición de ***throws*** más el nombre de la ***Exception*** concreta después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques ***try/catch*** o volver a pasar la ***Exception***. De esta forma se puede ir pasando la ***Exception*** de un método a otro hasta llegar al último método del programa, el método ***main()***.

El ejemplo anterior (***metodo1***) realizaba la gestión de las excepciones dentro del propio método. Ahora se presenta un nuevo ejemplo (***metodo2***) que relanza las excepciones al siguiente método:

```
void metodo2() throws IOException, MyException {
    ...
    /* Código que puede lanzar las excepciones IOException y MyException */
    ...
} // Fin del metodo2
```

Según lo anterior, si un método llama a otros métodos que pueden lanzar excepciones (por ejemplo de un package de ***Java***), tiene 2 posibilidades:

1. ***Capturar*** las posibles excepciones y gestionarlas.

2. Desentenderse de las excepciones y ***remitirlas*** hacia otro método anterior en el ***stack*** para éste se encargue de gestionarlas.

Si no hace ninguna de las dos cosas anteriores el compilador da un error, salvo que se trate de una ***RuntimeException***.

10.4.3.-Método finally. El bloque ***finally {...}*** debe ir detrás de todos los bloques ***catch*** considerados. Si se incluye (ya que es opcional) sus sentencias se ejecutan siempre, sea cual sea el tipo de excepción que se produzca, o ***incluso si no se produce ninguna***. El bloque ***finally*** se ejecuta incluso si dentro de los bloques ***try/catch*** hay una sentencia ***continue***, ***break*** o ***return***. La forma general de una sección donde se controlan las excepciones es por lo tanto:

```
try {
        /* Código "vigilado" que puede lanzar una excepción de tipo A, B o C */
} catch (A a1) {
        // Se ocupa de la excepción A
} catch (B b1) {
        // Se ocupa de la excepción B
} catch (C c1) {
        // Se ocupa de la excepción C
} finally {
        // Sentencias que se ejecutarán en cualquier caso
}
```

El bloque ***finally*** es necesario en los casos en que se necesite recuperar o devolver a su situación original algunos elementos. No se trata de liberar la memoria reservada con ***new*** ya que de ello se ocupará automáticamente el *garbage collector*.

Como ejemplo se podría pensar en un bloque ***try*** dentro del cual se abre un fichero para lectura y escritura de datos y se desea cerrar el fichero abierto. El fichero abierto se debe cerrar tanto si produce una excepción como si no se produce, ya que dejar un fichero abierto puede provocar problemas posteriores. Para conseguir esto se deberá incluir las sentencias correspondientes a cerrar el fichero dentro del bloque ***finally***.

10.5.-Crear nuevas Excepciones. El programador puede crear sus propias excepciones sólo con heredar de la clase ***Exception*** o de una de sus clases derivadas. Lo lógico es heredar de la clase de la jerarquía de ***Java*** que mejor se adapte al tipo de excepción. Las clases ***Exception*** suelen tener dos constructores:

1. Un ***constructor*** sin argumentos.

2. Un ***constructor*** que recibe un ***String*** como argumento. En este ***String*** se suele definir un mensaje que explica el tipo de excepción generada. Conviene que este constructor llame al constructor de la clase de la que deriva ***super(String)***.

Al ser clases como cualquier otra se podrían incluir variables y métodos nuevos. Por ejemplo:

```
class MiExcepcion extends Exception {
        public MiExcepcion() {
                super();
        }
        public MiExcepción(String s) {
                super(s);
        }
}
```

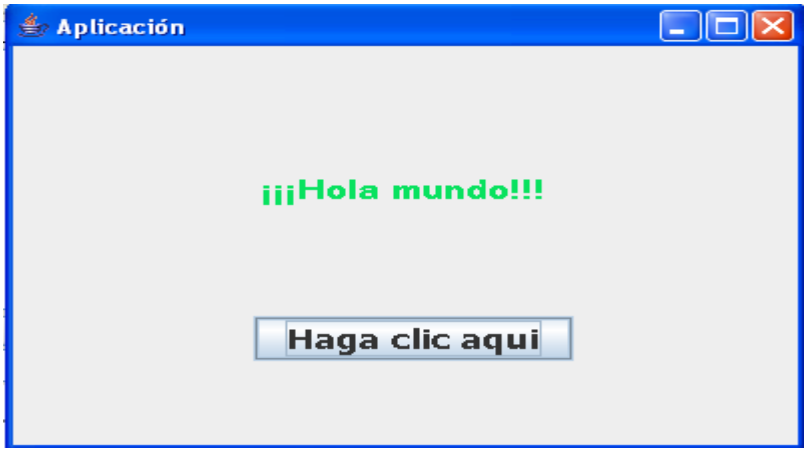
10.6.-Herencia de clases y tratamiento de Excepciones. Si un método redefine otro método de una super-clase que utiliza ***throws***, el método de la clase derivada no tiene obligatoriamente que poder lanzar todas las mismas excepciones de la clase base. Es posible en el método de la subclase lanzar ***las mismas excepciones o menos***, pero no se pueden lanzar más excepciones. No puede tampoco lanzar nuevas excepciones ni excepciones de una clase más general.

Se trata de una restricción muy útil ya que como consecuencia de ello el código que funciona con la clase base podrá trabajar automáticamente con referencias de clases derivadas, incluyendo el tratamiento de excepciones, concepto fundamental en la *Programación Orientada a Objetos* (***polimorfismo***).

TEMA 11. Entorno de Desarrollo Integrado para JAVA: NetBeans.

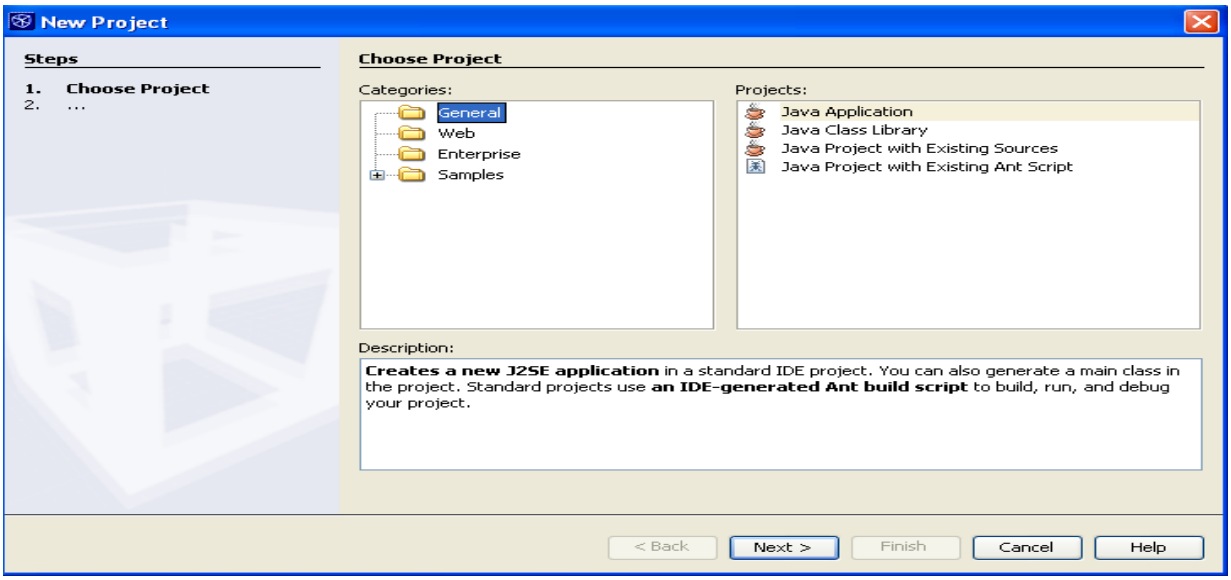
11.1. Introducción. Para escribir programas en Java podemos utilizar un entorno de desarrollo integrado (EDI ó IDE) que nos facilite las tareas de creación de la interfaz gráfica de usuario, edición del código, compilación, ejecución y depuración. Existem muchos IDEs para desarrollo de código Java. Nosotros vamos a utilizar el de Sun Microsystems, a saber, el **NetBeans**.

11.2. Diseño de una aplicación con interfaz gráfica. Para ello vamos a plantearnos un ejemplo sencillo:



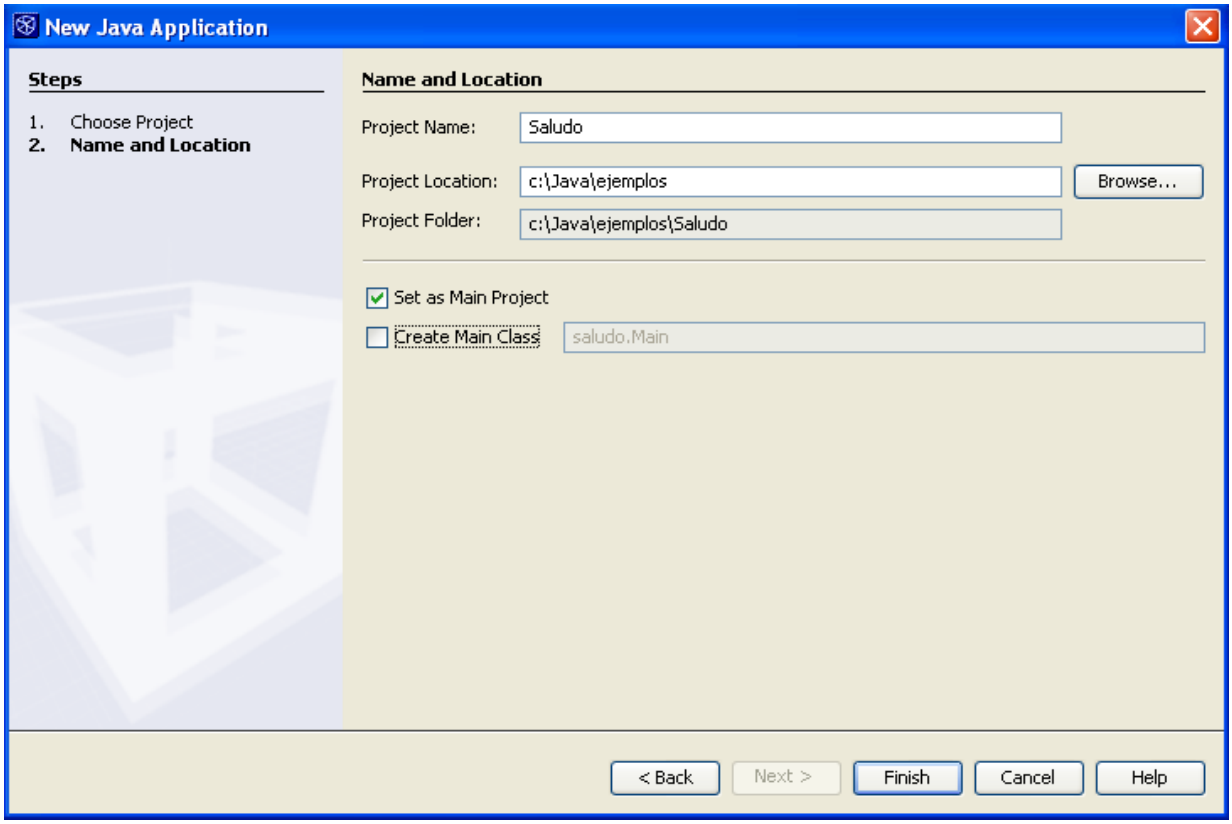
Los pasos a seguir son los siguientes:

1. Suponiendo que ya estás visualizando el entorno de desarrollo, ejecuta la orden *File > New Project*, o bien haz clic en el botón *Project*. Se muestra la siguiente ventana:



2. Selecciona *General* en la lista *Categories* y en la lista *Projects* selecciona *Java Application*.

Después haz clic en el botón *Next*. Se muestra la ventana *New Java Application*.



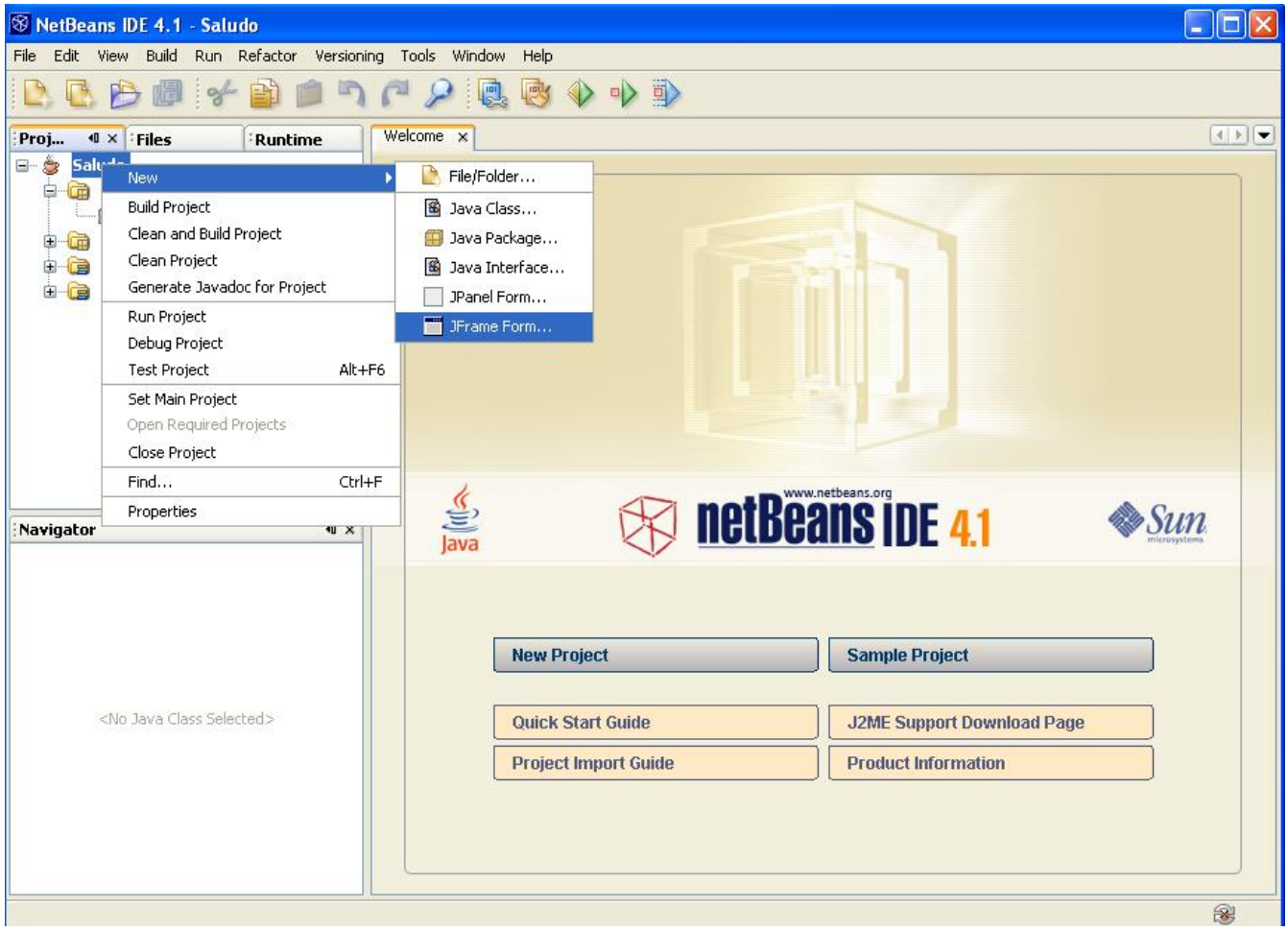
3. Escribe el nombre del proyecto (*Project Name*) y a, continuación selecciona la carpeta donde quieras guardarlo.

4. No marques la opción *Create Main Class*. De esta forma se creará un proyecto vacío.

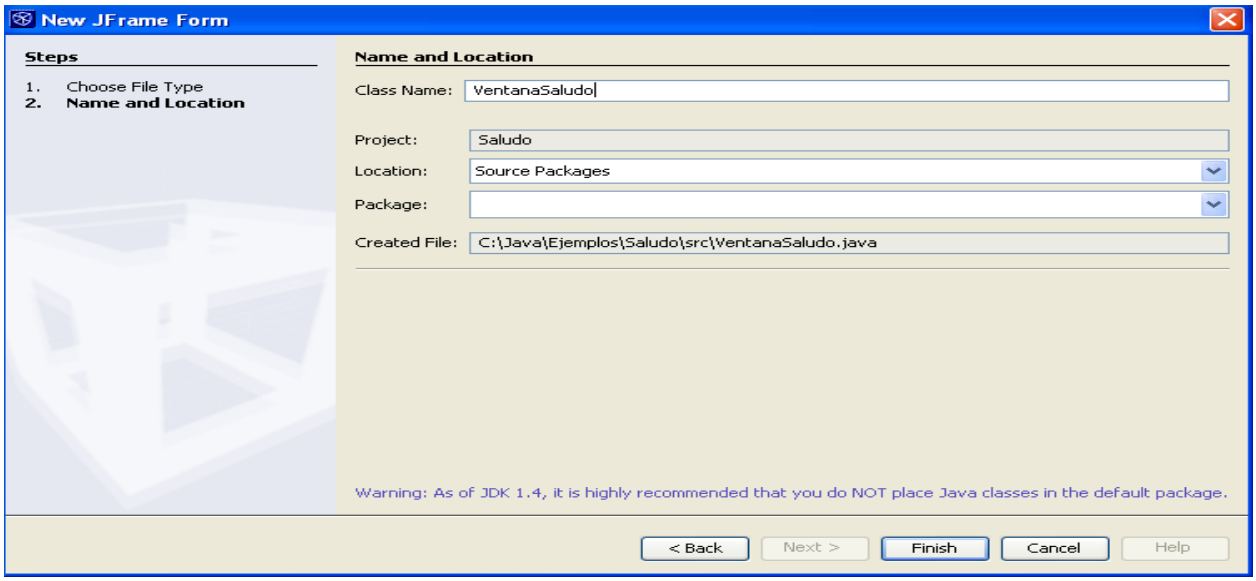
5. Para finalizar, haz clic en el botón *Finish*. El EDI crea la carpeta Ejemplos\Saludo en la que guardará el proyecto.

Una vez creado un proyecto vacío, el paso siguiente consistirá en añadir una nueva clase derivada de **JFrame** que nos sirva como contenedor de la interfaz gráfica.

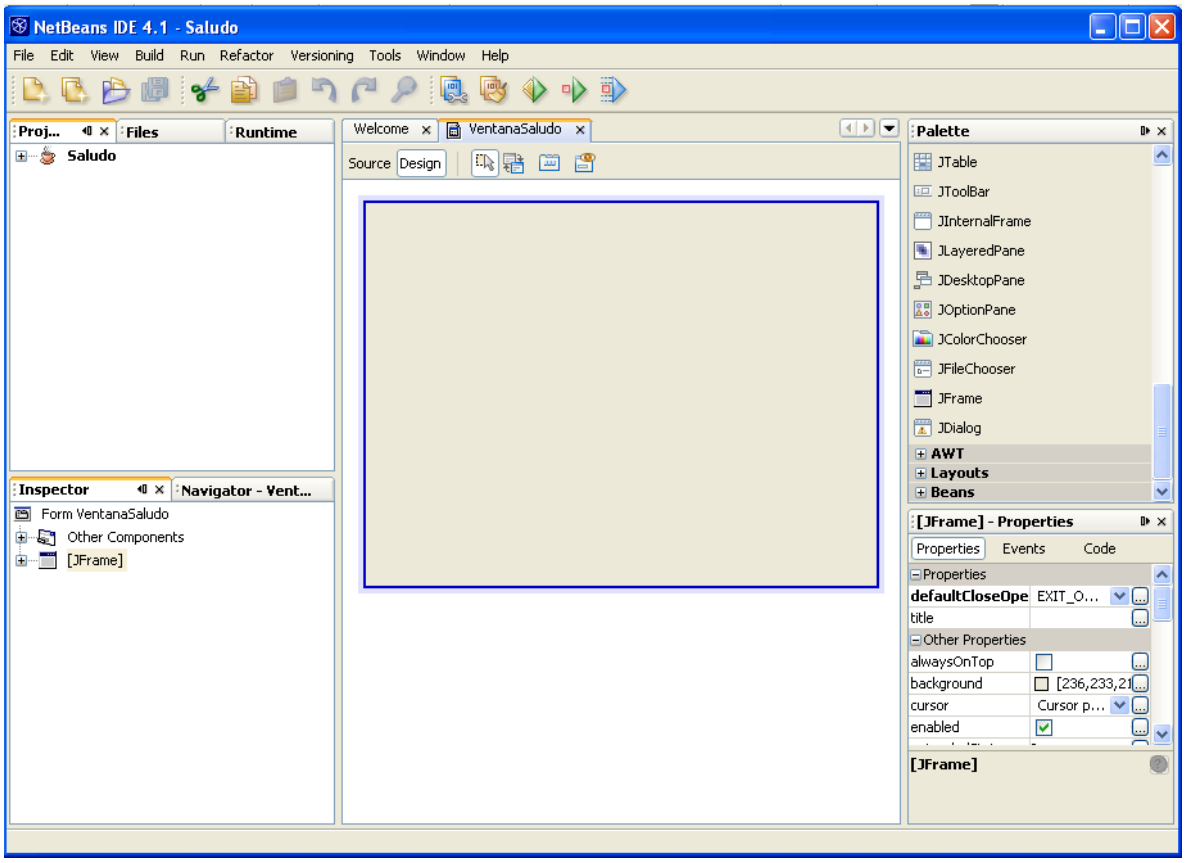
Para añadir una ventana marco (objeto **JFrame**) al proyecto, haz clic sobre el nombre del mismo, utilizando el botón derecho del ratón, y selecciona del menú contextual que se visualiza la orden New > *JFrame Form...*



Se visualizará una ventana en la que se puede elegir el nombre para la nueva clase de objetos que vamos a añadir. En nuestro caso elegiremos como nombre, por ejemplo, VentanaSaludo.



Para continuar haz clic en el botón *Finish*. Se visualizará la ventana mostrada a continuación, en la que observamos que la clase VentanaSaludo almacenada en el fichero VentanaSaludo.java, es la que da lugar a la ventana marco, tambien llamada formulario, que se visualiza en el centro de la pantalla (objeto **JFrame**).



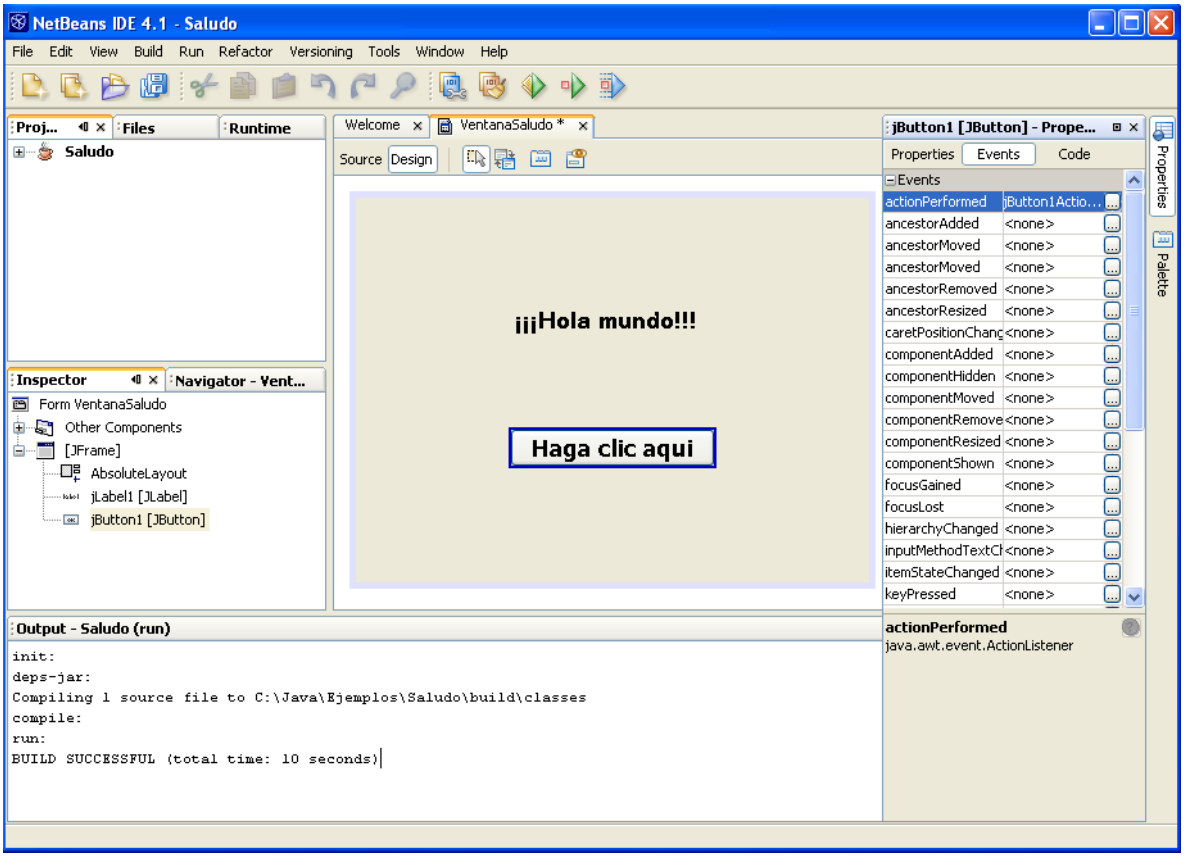
Observa que encima del formulario hay una barra de herramientas con una serie de botones. Los dos primeros (*Source* y *Design*) nos permitirán alternar entre el panel de edición (el que muestra el código fuente de la clase *VentanaSaludo* en nuestro ejemplo) y el panel de diseño (el que se está mostrando).

A la derecha del formulario (**JFrame**), se observan otras dos ventanas: una muestra varias paletas de herramientas (la que se está mostrando es la paleta de componentes *swing*) y la otra está mostrando el panel de propiedades con las propiedades del formulario. La primera nos muestra los controles o componentes que podemos seleccionar y colocar sobre el formulario: observa que además de la paleta de componentes *swing* hay otras tres: *AWT*, *Layouts* y *Beans* (componentes reutilizables). La segunda nos permite mostrar y editar las propiedades del componente seleccionado (tamaño, color, fuente ...), los manejadores de eventos, etc.

A la izquierda del formulario, debajo del panel del proyecto, hay otra ventana con dos paneles: el supervisor (*Inspector*) y el navegador (*Navigator*). El supervisor permite ver y seleccionar los componentes que forman la interfaz gráfica y el navegador permite navegar por los componentes software de la aplicación (clases, métodos, etc).

11.2.1. Añadir componentes al contenedor. Antes de nada vamos a añadir un título a la ventana. Para ello seleccionamos el componente **JFrame** en el *Inspector* y editamos su propiedad *title.* Para añadir componentes al formulario, debemos previamente seleccionar un *layout manager* para el mismo. Para ello seleccionamos el componente **JFrame** en el *Inspector* y pulsamos el botón derecho del ratón. Nos aparecerá un menú emergente, donde seleccionamos la opción *Set Layout->AbsoluteLayout*. o bien *Set Layout->Null Layout*. A continuación le añadimos una etiqueta (*JLabel*) con el texto "*¡¡¡Hola mundo!!!",* centrada y con fuente (*Font*) *Micrsoft San Serif* , estilo(*Font Style*) *Bold* y tamaño de la fuente 18 y un botón (*jButtom*) con la etiqueta "*Haga clic aquí",* centrada e igual fuente que el *jLabel*. Modificamos, tambien, su propiedad *toolTipText* para que muestre el mensaje "botón de pulsación".

11.2.2. Asignar manejadores de eventos a un componente. Cada vez que el usuario haga clic sobre el botón que acabamos de añadir, se generará un evento del tipo *actionEvent* que podrá ser recogido por un manejador de este tipo de eventos, si es que tiene uno asociado. La respuesta será mostrar en la etiqueta (*jLabel*) el mensaje "¡¡¡Hola mundo!!! en un determinado color generado aleatoriamente. Para asignar un manejador de eventos a nuestro botón seleccionamos en la ventana de propiedades, el panel *Events* (eventos), el método *actionPerformed, y* clicamos sobre él.



NetBeans generará automáticamente el código necesario para asociar un manejador de eventos al componente *JButton*. El resultado será que a *VentanaSaludo* se añade el siguiente código:

```
jButton1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            jButton1ActionPerformed(evt);
        }
});
 .
 .
 .
 private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
}
```

Como podemos observar *NetBeans* genera un manejador vacío. Para finalizar deberemos modificar, en nuestro caso añadir el siguiente código a nuestro manejador de eventos:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        float rojo = (float) Math.random();
        float verde = (float) Math.random();
        float azul = (float) Math.random();
        jLabel.setForeground(new java.awt.Color(rojo, verde, azul));
        jLabel.setText("¡¡¡Hola mundo!!!);
}
```

11.2.3. Compilar y ejecutar la aplicación. Ua vez finalizada la aplicación, puedes compilarla y ejecutarla. Para compilar la aplicación puedes ejecutar la orden *Build Main Project* del menú *Build*; pulsar el botón *Build Main Project* de la barra de herramientas; o bien pulsar la tecla de función F11.

Una vez que esté depurada la aplicación, podemos ejecutarla seleccionando la orden *Run Main Project* del menú *Run;* pulsando el botón *Run Main Project* de la barra de herramientas; o bien pulsando la tecla de función F6.

UNIDAD DIDACTICA 4ª. Trabajando con Bases de Datos.

TEMA 12. ACCESO A BASE DE DATOS (JDBC).

12.1. Introducción. JDBC son las siglas de *Java Database Connectivity*. Consta de un conjunto de clases e interfaces Java que nos permiten acceder de una forma genérica a las Bases de Datos independientemente del proveedor del SGBD. Cada proveedor dispondrá de una implementación de dichos interfaces. Dicha implementación se sabe comunicar con el SGBD de ese proveedor. Dichas clases forman parte del package ***java.sql.***

Básicamente, una aplicación que usa JDBC debe realizar los siguientes pasos:

- Carga del driver JDBC.
- Establecer una conexión con una Base de Datos.
- Crear y envíar una senetencia SQL a la Base de datos.
- Procesar el resultado.

SQL son las siglas de Structured Query Language. SQL es el lenguaje estándar de consulta de Bases de Datos.

12.2. Drivers JDBC. Los drivers JDBC son la implementación que cada proveedor ha realizado del API JDBC.

Existen cuatro tipos:

- JDBC-ODBC Bridge (puente JDBC-ODBC).
- Native-API partly-Java.
- JDBC-Net pure Java.
- Native protocol pure Java.

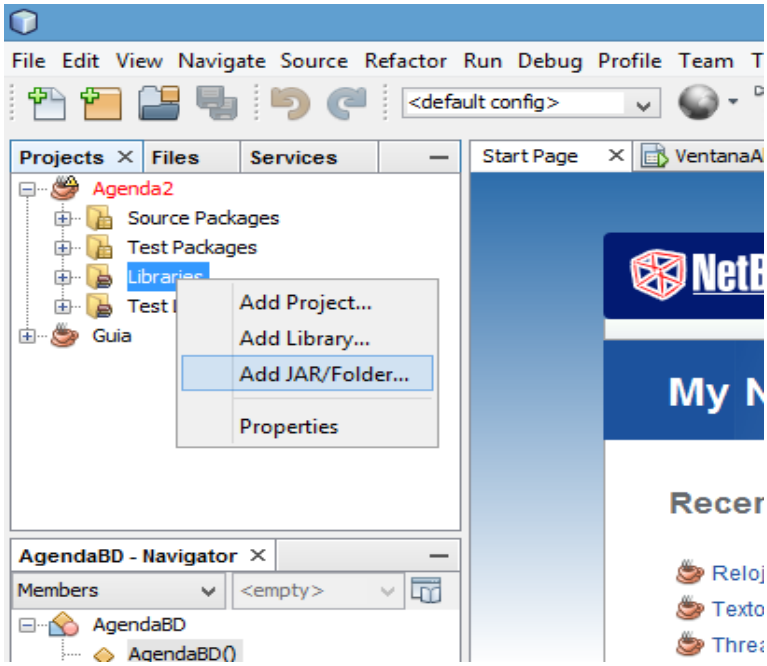
Los SGBD tendrán un fichero JAR o ZIP con las clases del Driver JDBC que habrá que añadir a la variable CLASSPATH del sistema.

Nosotros en este curso utilizaremos SQLite como Base se Datos y el driver *sqlite-jdbc*, que es un Driver JDBC-Net pure, para acceder a nuestra B.D. SQLite. Deberemos copiar en la carpeta *lib* de nuestro JDK el siguiente archivo:

sqlite-jdbc-3.8.11.2.jar

e incluirlo en el CLASSPATH.

Si estamos realizando un proyecto con Netbeans deberemos añadir este archivo *jar* a la librería de nuestro proyecto:



Por último, deberemos copiar el archivo *sqlite3.exe* en la carpeta donde se encuentre nuestra B.D. o bien añadir el path de dicho archivo a la variable de entorno del sistema *path*.

12.3. Componentes del JDBC.

Los componentes del API JDBC son:

- El gestor de los drivers: *java.sql.DriverManager*.
- La conexión con la Base de Datos: *java.sql.Connection*.
- La sentencia a ejecutar: *java.sql.Statement*.
- El resultado: *java.sql.ResultSet*.
- Otros: *java.sql.PreparedStatement* y *java.sql.CallableStatement* (procedimientos almacenados).

12.3.1. El gestor de los drivers.

El Driver Manager lleva el control de los drivers JDBC cargados en memoria. Además es el encargado de realizar la conexión con la Base de Datos. Hay que cargar en memoria los drivers JDBC para registrarlos al Driver Manager.

12.3.2. La conexión con la Base de Datos.

La conexión con la Base de Datos la abriremos creando un objeto de la clase *Connection* del paquete *java.sql*. El encargado de abrir una conexión es el Driver Manager mediante el método estático:

public static Connection getConnection(driver, url) throws SQLException;

donde:

url: es el identificador de la Base de Datos.

Para establecer la conexión con la Base de Datos de nombre *bibio.s3db*, localizada en la carpeta Java de la unidad C:, utilizando el driver *sqlite-jdbc*, el código a ejecutar sería:

```
import java.sql.*;
Connection conexion;
String url= "C:\\Java\\biblio.s3db ";
try {
    // Carga el driver sqlite-jdbc y establece conexión con la B.D. biblio.s3db
    conexion=DriverManager.getConnection("jdbc:sqlite:" + url);
    if (conexion!=null) {
            System.out.println("Conectado");
            return conexion;
    }
} catch (SQLException e) {
    System.out.println("Se ha producido un error al establecer la conexión con la Base de Datos biblio");
    return null;
}
return conexion;
```

Una vez que hayamos terminado de trabajar con una conexión debemos liberarla. Una conexión abierta significa recursos consumiéndose en el SGBD. Las conexiones se cierran mediante el método:

```
public void close() throws java.sql.SQLException;
```

12.3.3. La sentencia a ejecutar. A través de la conexión nos comunicaremos con la Base de Datos, enviándole sentencias SQL. Las sentencias SQL se envían a través de objetos de la clase *java.sql.Statements.* Para generar dichos objetos debemos ejecutar el método:

```
public java.sql.Statement createStatement( ) throws SQLException;
```

de la clase *java.sql.Connection*. También podemos utilizar objetos de la clases *PreparedStatement* y *CallableStatement*, en cuyo caso deberemos ejecutar los métodos:

```
public java.sql.PreparedStatement prepareStatement();
public java.sql.CallableStatement prepareCall();
```

respectivamente. Ambos métodos son de la clase *Connection*. Nosotros utilizaremos el primer método expuesto.

Las sentencias se cierran mediante el método:

```
public void close() throws java.sql.SQLException;
```

de la clase *java.sql.Statement*.

El método para ejecutar una sentencia SQL depende del tipo de sentencia SQL que contenga:

- Sentencias SELECT: se usa el método

  executeQuery(String sql);

  de la clase *java.sql.Statement* y devuelve una instancia (un objeto) de la clase *java.sql.ResultSet*.

  Por ejemplo:

```
import java.sql.*;
.
.
Connection conexion;
Statement sentencia;
ResultSet resultado;
try {
     // Carga del driver sqlite-jdbc y establecimiento de la conexión con la Base de Datos
     conexion = DriverManager.getConnection("jdbc:sqlite://C:/Java/biblio.s3db");

    // Creación de la sentencia SQL
    sentencia = conexión.createStatement();
    resultado = sentencia.executeQuery("SELECT * FROM libros");
    while (resultado.next()) {
        String nregistro = resultado.getString("NREGISTRO");
        String titulo = reultado.getString("TITULO");
        .
        .
    }

    // Cerrar sentencia
    sentencia.close();

     // Cerrar conexión
    conexion.close();

} catch (SQLException e) {

   e.printStackTrace();
}
```

- Sentencias INSERT, UPDATE y DELETE: se usa el método

  executeUpdate(String sql);

  de la clase *java.sql.Statement* y devuelve un *int* con el número de filas afectadas. Por ejemplo:

```
import java.sql.*
.
.
Connection conexion;
```

```
Statement sentencia;
ResultSet resultado;
try {
   // Carga del driver sqlite-jdbc y establecimiento de la conexión con la Base de Datos
    conexion = DriverManager.getConnection("jdbc:sqlite://C:/Java/biblios3mdb");

   // Creación de la sentencia SQL
    sentencia = conexión.createStatement();
    sentencia.executeUpdate("INSERT INTO LIBROS(NREGISTRO,TITULO) " +
                                  " VALUES('uno', 'Pascal')");

    // Cerrar sentencia
    sentencia.close();

    // Cerrar conexión
    conexion.close();

} catch (SQLException e) {
   e.printStackTrace();
}
```

- Sentencias CREATE TABLE y DROP TABLE: Se usa el método

```
execute(String sql);
```

de la clase *java.sql.Statement* y devuelve un *int* que siempre vale 0. Por ejemplo:

```
import java.sql.*
.
.
Connection conexion;
Statement sentencia;
ResultSet resultado;
try {
     // Carga del driver sqlite-jdbc ye stablecimiento de la conexión con la Base de Datos
     conexion = DriverManager.getConnection("jdbc:sqlite://C:/Java/biblio.s3db");

    // Creación de la sentencia SQL
    sentencia = conexión.createStatement();

    // Eliminar una tabla
    sentencia.execute("DROP TABLE NOTAS");

    // Crear una tabla
    sentencia.execute("CREATE TABLE NOTAS ("NOMBRE VARCHAR(15), " +
                          "CALIFICACION VARCHAR(30), " +
                          "CONSTRAINT CLAVE PRIMARY KEY (NOMBRE))");
```

```
// Cerrar sentencia
sentencia.close();

// Cerrar conexión
conexion.close();

} catch (SQLException e) { e.printStackTrace();}
```

12.3.4. El resultado: *java.sql.ResultSet*. Representa el resultado de una sentencia SQL, es decir, lleva asociadas todas las filas que cumplían las condiciones de la sentencia SQL. Implementa métodos para:

- Acceder a las filas que componen el resultado.
- Acceder al valor de cada columna de la fila seleccionada.

El método para acceder a la fila siguiente del resultado es :

public bolean next() throws SQLException;

El método para acceder a la fila anterior del resultado es:

public bolean previous() throws SQLException;

El método a usar para acceder al valor de una columna, dependerá del tipo de dato:

public xxx getXXX(int column)throws SQLException;

ó

public xxx getXXX(<nombre column>)throws SQLException;

La siguiente tabla muestra la relación entre el método y el tipo de dato:

getObject	getCharacterStream	getRef	getArray	getBlob	getClob	getBinaryStream	getUnicodeStream	getAsciiStream	getTimestamp	getTime	getDate	getBytes	getString	getBoolean	getBigDecimal	getDouble	getFloat	getLong	getInt	getShort	getByte	
x													x	x	x	x	x	x	x	x	X	TINYINT
x													x	x	x	x	x	x	x	X	x	SMALLINT
x													x	x	x	x	x	x	X	x	x	INTEGER
x													x	x	x	x	x	X	x	x	x	BIGINT
x													x	x	x	x	X	x	x	x	x	REAL
x													x	x	x	x	X	x	x	x	x	FLOAT
x													x	x	x	x	X	x	x	x	x	DOUBLE
x													x	x	X	x	x	x	x	x	x	DECIMAL
x													x	x	X	x	x	x	x	x	x	NUMERIC
x													x	X	x	x	x	x	x	x	x	BIT
x	x						x	x	x	x	x		X	x	x	x	x	x	x	x	x	CHAR
x	x						x	x	x	x	x		X	x	x	x	x	x	x	x	x	VARCHAR
x	X						X	X	x	x	x		x	x	x	x	x	x	x	x	x	LONGVARCHAR
x	x					x	x	x				X	x									BINARY
x	x					x	x	x				X	x									VARBINARY
x	x					X	x	x				x	x									LONGVARBINARY
x									x		X		x									DATE
x									x	X			x									TIME
x									X	x	x		x									TIMESTAMP
x					X																	CLOB
x				X																		BLOB
x			X																			ARRAY
x		X																				REF
X																						STRUCT
X																						JAVA_OBJECT

donde una x indica una solución legal mientras que una **X** indica la solución recomendada.

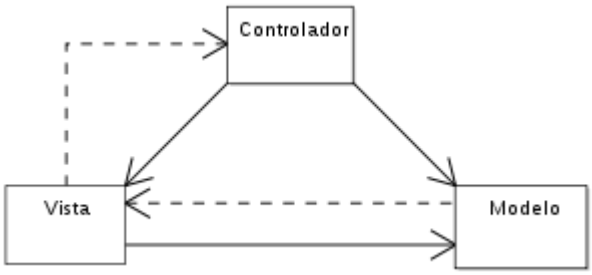
Se suele implementar un bucle de tipo while para recorrer y tratar todo el ResultSet.

Los ResultSet se cierran mediante el método:

public close() throws SQLException;

12.4. Modelo vista-controlador.

El **modelo vista-controlador (MVC)** es un patrón de ***arquitectura de software*** que separa los ***datos*** y la ***lógica de negocio*** de una aplicación de la ***interfaz de usuario*** y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de ***reutilización de código*** y la ***separación de conceptos***, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.



Un diagrama sencillo que muestra la relación entre el modelo, la vista y el controlador. Nota: las líneas sólidas indican una asociación directa, y las punteadas una indirecta

APENDICE A: Introducción a la metodología de la programación.

Tema 13: Concepto de algoritmo.

13.1.- Introducción. Un algoritmo es el conjunto ordenado de los pasos a seguir para resolver un problema concreto, sin ambigüedad alguna, en un tiempo finito. Todo algoritmo ha de verificar necesariamente las siguientes características:

- **Precisión**: el algoritmo deberá indicar claramente y sin ambigüedad, el orden de realización de cada paso. Además, debe ser concreto.
- **Repetitividad**: el algoritmo debe poder repetirse tantas veces como sea necesario, dando siempre los mismos resultados para las mismas entradas, independientemente del momento de su ejecución.
- **Finitud**: todo algoritmo debe alcanzar la solución esperada en algún momento, es decir, en un tiempo finito y, además, razonable.

Otra serie de características deseables son:

- La **validez** del algoritmo: que el algoritmo haga exactamente lo que se pretende que haga.
- La **eficiencia**: si el algoritmo diseñado da una solución en un tiempo razonable.
- La **optimización**: si el algoritmo construido es el mejor algoritmo para resolver el problema concreto que tenemos entre manos.

Cada uno de los pasos especificados en un algoritmo se denomina **sentencia** o **instrucción**.

Llamamos **primitivas** a aquellas acciones que el ordenador es capaz de entender y realizar directamente de forma automática. Es decir, una primitiva es la unidad mínima de ejecución.

13.2.- La memoria del ordenador. Toda la información que maneja un algoritmo ha de almacenarse necesariamente en algún sitio accesible, que en nuestro caso es la memoria principal. La memoria de un ordenador se divide en casillas o celdas de igual tamaño. Cada casilla es accesible mediante una dirección, a la que se denomina **dirección de memoria** o posición de memoria, que identifica unívocamente a cada casilla.

La descripción de un algoritmo mediante un lenguaje que entienda el ordenador, junto con la correcta representación en memoria de la información que maneja, se denomina **programa**. Tanto los programas como la información que éstos manejan se alojan en la memoria.

Los lenguajes de programación de alto nivel manejan los datos asociándoles un nombre y no mediante su dirección. La gestión de las localizaciones y asignación de memoria a los datos que manejan los programas, es tarea del compilador.

13.3.- Datos, tipos de datos y expresiones.

13.3.1.- Datos. Un dato es un conjunto de celdas o posiciones de memoria que tiene asociado un *nombre* (identificador) y un *valor* (contenido).

No todos los datos ocupan el mismo número de celdas de memoria. En los datos hay que distinguir pues, dos partes: nombre y contenido.

Los datos que manejan un programa puede clasificarse de forma muy general en:

- *Constantes.* Aquellos datos cuyo contenido no varía a lo largo de la ejecución de un programa.
- *Variables.* Aquellos datos cuyo contenido puede modificarse a lo largo de la ejecución de un programa.

13.3.2.- Tipos de datos. La mayoría de los ordenadores pueden trabajar con distintos tipos de datos. Cuando en un algoritmo o programa se especifica que un dato es de un determinado tipo, se está dando al ordenador, de manera implícita, la siguiente información:

- El **rango** de los valores permitido para ese dato.
- El conjunto de **operaciones** (primitivas) que pueden aplicarse a los datos de ese tipo.

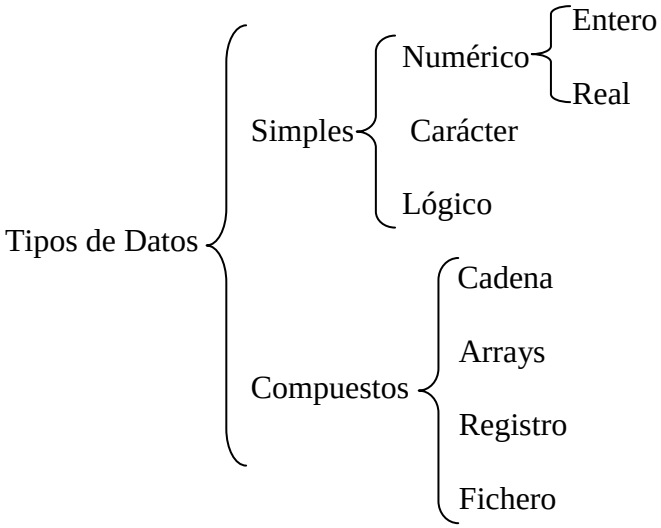
Cada dato usado en un programa debe, estar asociado a un único tipo de entre todos los permitidos por el lenguaje que se esté utilizando.

Operadores. Cualquier operador (función) puede clasificarse, según el número de datos sobre los que opera en:

- *Unarios*: Se aplican sobre un solo dato o argumento.
- *Binarios*: Se aplican sobre dos datos o argumentos.
- *Enearios*: Se aplican sobre N argumentos.

Denominamos **operadores** a las funciones y **operandos** a sus argumentos.

13.3.3.- Clasificación de los tipos de datos El siguiente esquema muestra una clasificación de los tipos de datos más comunes de los admitidos por un lenguaje de programación:



El tipo de dato ENTERO.

- *Rango*: es finito y será un subconjunto matemático Z.

- Operadores: +, -, *, /, abs, sqr, sqrt, ln, exp, sin, cos, tan, ^, div, mod.

El tipo de dato REAL.

- *Rango:* es un subconjunto finito de R.

- *Operadores*: los mismos que el tipo de dato entero y además: trunc ,round ,ent.

Los operadores relacionales clásicos, a saber, =, <, >, <>, >=, <=; pueden aplicarse a operadores tanto enteros como reales y tienen el mismo sentido que en matemáticas.

El tipo de dato CARACTER.

- *Rango:* es un conjunto finito y ordenado de caracteres. Este conjunto suele ser, normalmente, el llamado código ASCII. Este código, en su forma original, tiene 128 elementos numerados desde 0 al 127, entre los que se incluyen las letras minúsculas y mayúsculas, los dígitos del 0 al 9, y otros caracteres especiales. Posteriormente, este código fue extendido hasta completar 256 elementos llamándose entonces ASCII extendido.

- *Operadores*:
  - **ord** : Devuelve el código ASCII del carácter.

- **chr**: Inverso del operador ord.

El tipo de dato LOGICO o BOOLEANO.

- *Rango:* Un dato lógico, sólo puede tomar dos valores: Verdadero o Falso.

- *Operadores:* NO, Y, O.

Tipos ordinales. Sus valores pertenecen a un conjunto finito y ordenado. Cada posible valor a tomar por un dato de un tipo ordinal, tiene un número de orden, dentro del conjunto al que pertenece.

Los tipos de datos entero, lógico y carácter son **ordinales**, porque cada valor tiene un antecesor y un predecesor, salvo el primero y el último, respectivamente.

A las constantes y variables de tipo ordinal, se les pueden aplicar, además de las funciones propias de cada tipo, tres funciones comunes que son: pred, succ, ord.

13.3.4.- Expresiones.

Un **literal** es la especificación de un valor de un tipo concreto.

Una **expresión** se define de la siguiente forma:

- Una constante, una variable o un literal es una expresión.
- <operador unario>(< expresión>) es una expresión.
- <expresión><operador binario><expresion> es una expresión.
- (<expresión>) es una expresión.

Las **expresiones** son combinaciones de constantes, variables, literales, símbolos de operación, paréntesis y funciones especiales, que se rigen por una sintaxis concreta.

Cada expresión toma un valor. Las expresiones se clasifican, dependiendo del resultado obtenido en:

- Aritméticas.
- Lógicas.
- De caracteres.

13.3.4.1.- Expresiones Aritméticas. Las constantes y variables que intervienen en la expresión suelen ser enteras o reales y las operaciones que intervienen son las que devuelven un valor numérico.

El orden en el que, por defecto, se aplican los operadores es el siguiente:

- Se resuelven primero los paréntesis.
- Los operadores unarios tales como: exp, ln, sin, cos, sqr, sqrt, etc.
- La potencia.
- *, /, div, mod.
- +, -.

En cualquier expresión se evalúan en primer lugar, los operadores de mayor precedencia y, en caso de conflicto, se evalúan de izquierda a derecha.

13.3.4.2.- Expresiones Lógicas. Son expresiones cuyo resultado es uno de los valores del conjunto {Verdadero, Falso}. Se les denomina también expresiones booleanas.

Para las expresiones lógicas, el orden de precedencia es el siguiente:

- Los paréntesis.
- NO.
- Y.
- O.
- Los operadores relacionales: =, >, <, >=, <=, <>.

13.4.- Notación para describir algoritmos.

El pseudocódigo se define como un lenguaje mixto que utiliza el vocabulario de un lenguaje natural (en nuestro caso el español) y la sintaxis de un lenguaje estructurado.

El formato general de un algoritmo en pseudocódigo es el siguiente:

Algoritmo <Nombre-Algoritmo>
[Breve descripción en lenguaje natural de lo que hace el algoritmo]
[**Constantes**]
    <Definición de constantes>

[**Variables**]
    <Declaración de variables>
Inicio
    .....
    <Secuencia de primitivas que conforman el algoritmo>

.....

Fin.

13.4.1.- Definición de constantes. La sección en la que se definen las constantes que va a utilizar el algoritmo, se llama **Constantes** y el formato de la definición es el siguiente:

<Nombre-constante> = <valor>

donde <Nombre-constante> será el nombre asociado a la constante y <valor> será el valor asociado a dicho nombre.

13.4.2.- Declaración de variables. El formato de la declaración de una variable es el siguiente:

<tipo> <Nombre-variable>

donde <Nombre-variable> es el identificador de la variable y <tipo> es el tipo de dicha variable, que podrá ser: entero, real, lógico, carácter,.....

Cuando vayamos a declarar varias variables del mismo tipo, éstas las pondremos en la misma línea de declaración, separadas por comas.

Tras realizarse la reserva de memoria de una variable, ésta tendrá un contenido desconocido, hasta que no se le asigne un valor significativo dentro del algoritmo.

13.5.- Operaciones primitivas. Una operación primitiva es aquella que el ordenador es capaz de ejecutar directamente. Cualquier otra acción a llevar a cabo por el ordenador habrá de ser especificado en términos de primitivas.

13.5.1.- Operación de Asignación. Mediante esta primitiva, se podrá dar valores a las variables. El formato será el siguiente:

<variable>=<expresión>

donde, <expresión> podrá ser una expresión cualquiera, siempre y cuando dicha expresión sea del mismo tipo que la variable a la cual se le va asignar el resultado. Tan sólo existe una excepción a esta norma: se podrá asignar el resultado de una expresión de tipo entero a una variable de tipo real.

13.5.2.- Operación de Entrada. A la operación de dar valor a una variable desde el teclado se le llama **lectura**. La primitiva que nos permite realizar dicha asignación desde el teclado es la siguiente:

lee (<lista-variables>)

donde <Lista-variables> es la lista de variables cuyo valor se va a obtener por el teclado. Las variables deberán ir separadas por comas.

```
Ejemplo: Variables
            entero x,y
         Inicio
                  .
                  .
                  .
            lee (x,y)
         Fin.
```

13.5.3.- Operación de Salida. A la operación que permite mostrar el contenido de una o varias variables en la pantalla se le denomina **escritura**, y el formato es el siguiente:

escribe (<Lista-expresiones>)

donde <Lista-expresiones> es una lista de expresiones separadas por comas.

```
Ejemplo: Variables
            entero x,y
         Inicio
            .....
            escribe (x,y)
            .....
         Fin.
```

Existe un segundo formato para la sentencia de escritura,que permite escribir mensajes en la pantalla. Tales mensajes habrán de ir, necesariamente, entre comillas simples. El formato es:

escribe (<'Mensaje'>)

La mayor parte de los lenguajes de programación permiten agrupar mensajes y expresiones en la misma sentencia de escritura, separándolos entre sí por comas.

```
Ejemplo: Variables
            real x,y
         Inicio

            escribe ('El valor de x es: ', x)
            escribe ('Hola')
         Fin.
```

13.6. Ejercicios propuestos.

13.1. Realizar un algoritmo para calcular el valor de la hipotenusa de un triángulo rectángulo conocidos los 2 catetos aplicando el teorema de Pitágoras.

13.2. Para sacar al mercado un determinado producto intervienen 4 personas, una que lo diseña y 3 que lo fabrican. Diseñar un algoritmo que calcule cuanto cobra cada uno de ellos, sabiendo que el diseñador cobra el doble que los fabricantes y conocidos el nº de unidades vendidas y el precio de venta de cada unidad.

13.3. Diseñar un algoritmo que a partir de las longitudes de los lados de un triángulo, calcule el área del mismo, de acuerdo con la siguiente fórmula:

· $\text{Area} = \sqrt{T*(T-S1)*(T-S2)*(T-S3)}$

donde T = (S1+S2+S3) / 2 y S1, S2, S3 son las longitudes de los lados del triángulo.

Tema 14: Estructuras de control.

14.1.- Introducción. Cuando una máquina empieza la ejecución de un algoritmo, empieza leyendo la primera línea de código y la ejecuta, pasa a la siguiente y la ejecuta y así sucesivamente hasta llegar al final del algoritmo. La trayectoria que sigue al ir realizando todos los pasos presentes en el algoritmo constituye lo que se denomina **flujo de control del programa**.

Cualquier herramienta que controle el flujo de control de un algoritmo constituye lo que denominaremos una **estructura de control**.

Existen distintos tipos de estructuras de control que pueden clasificarse en tres tipos básicos:

1. *Estructura secuencial*. Es la vista hasta ahora.

2. *Estructura selectiva*. Dirigen el flujo de ejecución del programa a unas instrucciones u otras dependiendo del cumplimiento de una ejecución.

3. *Estructura repetitiva*. Permiten la ejecución repetida de un conjunto de instrucciones, un número determinado o indeterminado de veces.

14.2.- Estructura secuencial.- Es la que sigue la máquina si no se dice lo contrario. Asegura que las sentencias se ejecutan siguiendo el orden en que están escritas.

Ejemplo: Realizar un algoritmo para la resolución de una ecuación de segundo grado.

```
Algoritmo Raices_Cuadradas_1
Variables
        real a,b,c,x1,x2
Inicio
        escribe ('Introduce valor del coeficiente x²: ')
        lee (a)
        escribe ('Introduce valor del coeficiente x: ')
        lee (b)
        escribe ('Introduce valor del término independiente: ')
        lee (c)
        x1 = (-b + sqrt(b^2 - (4*a*c))) /( 2*a)
        x2 = (-b – sqrt(b^2 - (4*a*c))) /( 2*a)
        escribe ('Las raices valen: ', x1, x2)
Fin.
```

14.3.- Estructura condicional. Una **ejecución condicional** consiste en la ejecución de una o varias sentencias dependiendo de la evaluación de una condición.

Una **estructura condicional** es una estructura que permite una ejecución condicional.

14.3.1.- Estructura condicional simple. Consiste en realizar una pregunta y utilizar la respuesta para la ejecución o no de una serie de instrucciones. La pregunta vendrá formulada a través de una *expresión lógica.* El formato es el siguiente:

```
si <condición> entonces
    <bloque- si>
fin- si
```

donde <condición> es una expresión lógica.

Todo el bloque englobado por la estructura de control constituye para nosotros una única sentencia, por lo que a partir de ahora usaremos el término **sentencia condicional.**

Ejemplo: Mejora del ejercicio anterior.

```
Algoritmo Raices_Cuadradas_2
Variables
        real a,b,c,x1,x2
Inicio
        escribe ('Introduzca valor del término x²: ')
        lee (a)
        escribe ('Introduzca valor del término x: ')
        lee (b)
        escribe ('Introduzca valor del término independiente: ')
        lee (c)
        si (-b^2 –4*a*c) > 0 entonces
            x1 = (-b + sqrt(b^2 – 4*a*c))) /( 2*a)
            x2 = (-b – sqrt(b^2– 4*a*c))) /( 2*a)
            escribe ('Las raices valen: ', x1, x2)
        fin- si
Fin.
```

14.3.2.- Estructura condicional doble. Una estructura condicional doble consiste, en realizar una pregunta y ejecutar una serie de instrucciones si la respuesta es verdad o bien, ejecutar otra serie de instrucciones si la respuesta es falsa. El formato es el siguiente:

```
si <condición> entonces
    <bloque si>
si-no
    <bloque si-no>
fin-si
```

En cualquiera de las estructuras condicionales simples estudiadas en los apartados anteriores, en cada una de las listas de instrucciones pueden aparecer de nuevo otras estructuras condicionales. Cuando esto sucede, la estructura así formada se llama **estructura condicional anidada**.

Ejemplo: Mejora del ejercicio anterior:

```
Algoritmo Raíces_Cuadradas_3
Variables
        real x1,x2,a,b,c
Inicio
        escribe ('Introduzca valor del término x²: ')
        lee (a)
        escribe ('Introduzca valor del término x: ')
        lee (b)
        escribe ('Introduzca valor del término independiente: ')
        lee (c)
        si (b^2 – 4*a*c) = 0 entonces
            x1 = -b / (2*a)
            Escribe ('El resultado es una raíz real doble: ', x1)
        si-no
            si (b^2 – 4*a*c) > 0 entonces
                x1 = (-b +sqrt(b^2 – 4*a*c)) /( 2*a)
                x2 = (-b –sqrt(b^2 – 4*a*c)) / (2*a)
                Escribe ('Las raices valen: ', x1, x2)
            si-no
                Escribe ('Raices complejas')
            fin-si
        fin-si
Fin.
```

14.3.3.- Estructura condicional múltiple. Cuando el número de alternativas es elevado, es conveniente utilizar la estructura condicional múltiple. El formato es el siguiente:

```
según-sea <expresión>
    <lista-1> : <bloque-1>
    <lista-2> : <bloque-2>
                .
                .
                .
    [si-no <bloque si-no>]
fin-según-sea
```

donde: <expresión> es una expresión ordinal.

<lista-i>: son valores del mismo tipo de datos que <expresión>. Pueden ser literales o constantes, expresiones ordinales, varios valores separados por comas, o bien, rangos de valores que vienen dados por el extremo izquierdo y el derecho, separados por puntos.

<bloque-1>: es una lista de sentencias.

Ejemplo: Algoritmo para sumar, restar, multiplicar o dividir dos números introducidos desde teclado, utilizando un menú de opciones como el siguiente:

1.- Sumar
2.- Restar
3.- Multiplicar
4.- Dividir
Elija una opción (1/4):

```
Algoritmo Menú
Variables
        real Num1, Num2
        entero opción
Inicio
        escribe ('Introduzca un primer número: ')
        lee (Num1)
        escribe ('Introduzca un segundo número: ')
        lee (Num2)
        escribe ('Menú')
        escribe ('1.- Sumar')
        escribe ('2.- Restar')
        escribe ('3.- Multiplicar')
        escribe ('4.- Dividir')
        escribe ('Elija una opción (1/4): ')
        lee (opción)
        según-sea opcion
            1: escribe (Num1,' + ', Num2, ' = ', Num1+Num2)
            2: escribe (Num1, ' - ', Num2, ' = ', Num1-Num2)
            3: escribe (Num1, ' * ', Num2, ' = ',  Num1*Num2)
            4: escribe (Num1, ' /  ', Num2, ' = ', Num1/Num2)
            si-no escribe ('Error al introducir la opción')
        fin-según-sea
Fin.
```

14.4.- Estructuras repetitivas.

Llamaremos **bucle** o **ciclo** a toda estructura de control que nos permita ejecutar una serie de acciones, varias veces, en el mismo orden en que aparecen. Al conjunto de dichas acciones lo llamaremos **cuerpo del bucle** y cada vez que se ejecutan las acciones presentes en el cuerpo del bucle, se dice, que se produce una **iteración**.

14.4.1.- Bucles controlados por Contador.

Se utilizan cuando el número de iteraciones a ejecutar de un bloque, se conoce antes de entrar en el bucle. Como el número de iteraciones es conocido de antemano, podemos utilizar una variable que cuente cada una de las iteraciones que se van

produciendo. A dicha variable la llamaremos **contador** o **variable contadora,** y a la estructura que implementa el bucle la denominaremos **Bucle Contador**. El formato es el siguiente:

```
repite para <variable-control> = <valor-inicial>, <valor-final>, [<incremento>]
    <lista de instrucciones>
fin-repite
```

donde <variable-control> es nuestra variable contadora, que debe ser de tipo ordinal.

Ejemplo: Hacer un algoritmo que nos sume los números naturales de 1 al 100, y nos visualice el resultado.

```
Algoritmo Número_naturales
Variables
        entero suma, cont
Inicio
        suma = 0
        repite para cont = 1,100
            suma = suma + cont
        fin-repite
        escribe ('La suma total es: ', suma)
Fin.
```

14.4.2.- Bucles controlados por Centinela: Bucles "Mientras" y "Hasta". A la estructura de control que nos permita ir ejecutando una serie de acciones mientras que se verifique cierta condición lo llamaremos **bucle controlado por centinela**. Existen dos tipos diferentes de bucles centinela:

1. *Bucles Mientras*. El formato es el siguiente:

```
repite mientras <condición>
    <lista-instrucciones>
fin-repite
```

2. *Bucles Hasta*. El formato es el siguiente:

```
repite
    <lista-instrucciones>
hasta <condición>
```

Ejemplo: Realizar un algoritmo para sumar valores desde teclado hasta que se introduzca el número 0, en cuyo caso finalizará el programa.

```
Algoritmo Suma_Valores_Bucle_Mientras
Variables
        entero suma, num
Inicio
        num = 10
        suma = 0
        repite mientras num <> 0
            escribe ('Introduce un nº: ')
            lee (num)
            suma = suma + num
        fin-repite
        escribe ('La suma de los números introducidos es: ', suma)
Fin

Algoritmo Suma_Valores_Bucle_Hasta
Variables
        entero suma, num
Inicio
        num = 10
        sum = 0
        repite
            escribe ('Introduce un nº: ')
            lee (num)
            suma = suma + num
        hasta num = 0
        escribe ('La suma de los números introducidos es: ', suma)
Fin.
```

Los bucles centinela son imprescindibles: su funcionamiento no puede simularse con un bucle contador.

Los bucles contador no son imprescindibles: su funcionamiento puede simularse con un bucle centinela.

Los bucles centinela son equivalentes.

14.5. Ejercicios propuestos.

14.1. Hacer un algoritmo para que leído desde el teclado un número nos diga si dicho número es positivo o negativo.

14.2. Hacer un algoritmo para ver si un número introducido por teclado es par o impar.

14.3. Realizar un algoritmo que lea dos valores desde teclado y diga si cualquiera de ellos divide de forma entera al otro.

14.4. Realizar un algoritmo para deducir el mayor de tres valores introducidos por teclado.

14.5. Diseñar un algoritmo para leer las longitudes de los lados de un triángulo y determine que tipo de triángulo es, de acuerdo a los siguientes casos: suponiendo que A es el mayor de los lados y que B y C corresponden a los otros 2 lados:

Si A >= B + C No es un triángulo
Si $A^2 = B^2 + C^2$ Triángulo rectángulo
Si $A^2 > B^2 + C^2$ Triángulo obtusángulo
Si $A^2 < B^2 + C^2$ Triángulo acutángulo

14.6. Hacer un algoritmo que nos sume los números naturales, comprendidos entre dos números introducidos por teclado.

14.7. Hacer un algoritmo que nos sume los números naturales que sean pares y sean menores que un número introducido por teclado.

14.8. Hacer un algoritmo que imprima, sume y cuente los números pares que hay entre dos números determinados.

14.9. Hacer un algoritmo para calcular el factorial de un número natural positivo.

14.10. Hacer un algoritmo para calcular el factorial de cualquier número.

14.11. Hacer un algoritmo para mostrar por pantalla 1000 veces de una forma alternativa: Hola, Adiós, utilizando un switch.

14.12. Hacer un algoritmo para mostrar por pantalla los números múltiplos de 3 que hay entre dos números determinados, de forma alternativa.

14.13. Realizar un algoritmo para calcular el valor máximo y el mínimo de una lista de n números que se introducen por teclado.

14.14. Hacer un algoritmo para calcular el valor máximo y el valor mínimo de una lista de números que se introducen por teclado, se terminará la operación cuando se introduzca el número 0.

14.15. Hacer un algoritmo para calcular la potencia de un número A. Se introducen por teclado los números n y A.

14.16. Hacer un algoritmo que calcule el valor de un número combinatorio a partir de dos valores, A y B, que se introducen por teclado, aplicando la siguiente fórmula:

$$\binom{a}{b} = \frac{a!}{b! * (a-b)!}$$

14.17. Realizar un algoritmo que cuente los números positivos y negativos que aparezcan en una lista de números que se introducen por teclado. El proceso finalizará introduciendo el número 0.

14.18. Realizar un algoritmo que calcule la media aritmética de una lista de números que se introducen por teclado. El proceso finalizará con la introducción del número 0.

14.19. Realizar un algoritmo con un menú de 4 opciones. La selección de cada opción se realizará usando una variable de tipo carácter. Cada una de las opciones realizará las siguientes tareas:

F: Calculará el factorial.
R: Calculará la raíz cuadrada de un número si es positivo y si es negativo, dará un mensaje de error.
C: Calculará el cuadrado de un número.
T: Finalizará el algoritmo.

14.20. Realizar un algoritmo para imprimir las tablas de multiplicar del uno al diez.

14.21. Hacer un algoritmo para calcular si un número es primo o no.

14.22. Hacer un algoritmo para que introducido el número de mes lo visualice en letra. Repetir el proceso cuantas veces se quiera.

14.23. Introducir la nota de una asignatura por teclado, que esté comprendida entre 0 y 10 y escribir la nota en letra, atendiendo a:

0 <= Nota < 3	Muy deficiente
3 <= Nota < 5	Insuficiente
5 <= Nota < 6	Suficiente
6 <= Nota < 7	Bien
7 <= Nota < 9	Notable
9 <= Nota <= 10	Sobresaliente

14.24. Hacer un algoritmo para que nos calcule la estadística de una serie de notas introducidas por teclado. La serie finalizará con la introducción del 0, sabiendo que :

1 >= Nota < 5	Insuficiente
5 >= Nota < 6	Suficiente
6 >= Nota < 7	Bien
7 >= Nota < 9	Notable
9 >= Nota =< 10	Sobresaliente

14.25. Sabiendo que la función rnd() nos devuelve un nº real entre 0 y 1: [0,1[, simular el lanzamiento de una moneda al aire y visualizar por pantalla si ha salido cara o cruz. Repetir el proceso tantas veces como se quiera.

14.26. Simular 100 tiradas de un dado y contar las veces que aparece el nº 6.

14.27. Simular 100 tiradas de 2 dados y contar las veces que entre los dos suman 10.

14.28. Generar aleatoriamente una quiniela de una columna, si la probabilidad de que salga 1 es del 50%, la x es del 30% y la del 2 es del 20%.

Tema 15: Funciones.

15.1.- Programación modular.

Es una filosofía de programación que consiste básicamente en dividir un algoritmo en unidades de menor tamaño en donde cada fragmento realiza una tarea explícita y única. Cada uno de esos fragmentos recibe el nombre de **subprograma** o **módulo.** En algunos lenguajes de programación, los módulos reciben el nombre de **subrutina**.

Un **módulo** es un algoritmo que puede diseñarse de forma independiente del contexto en el cual va a utilizarse.

Se dice que un módulo que utiliza otro modulo, **activa**, **llama** o **invoca** a dicho módulo. Al primer módulo lo llamaremos **módulo llamador** o **módulo principal** y al módulo invocado lo llamaremos **módulo llamado** o **subordinado**.

15.2.- Declaración de funciones.

Matemáticamente, una función es una operación que toma uno o más valores, llamados argumentos, y produce uno o más valores llamados resultados.

A nosotros nos interesará implementar aquellas funciones que devuelvan un único valor o no devuelvan nada.

Todos los lenguajes tienen funciones incorporadas o intrínsecas, es decir, predefinidas. Sin embargo, los lenguajes no pueden aportar todas las funciones necesarias por lo que permiten que sea el propio programador el que defina sus funciones para, posteriormente, utilizarlas.

El formato de la declaración de una función es el siguiente:

```
[tipo-de-dato] Función <nombre> (<lista de parámetros formales>)
[{<descripción>}]
[Constantes Locales
        <Definiciones>]
[Variables Locales
        <Declaraciones>]
Inicio
        <instrucciones>
        [retorna <valor>]
Fin
```

Ejemplo:

```
real Función cuadrado (entero x)
 Inicio
    retorna (x ^ 2)
 Fin
```

Un **parámetro formal** es todo aquel identificador que aparece en la cabecera de la declaración de un módulo.

Ejemplos:

```
real Función superficieTriangulo (entero b, a)
Inicio
        retorna ((b * a) / 2)
Fin

real Función factorial (entero num)
Variables
        real fact
        entero cont
Inicio
        fact = 1
        repite para cont = 1, num
            fact = fact * cont
        fin-repite
        retorna fact
Fin
```

Usualmente la declaración de la función se debe realizar antes del inicio del algoritmo principal. Una vez declarada una función, puede utilizarse en cualquier sitio del programa principal en el cual está definida. Una función, a su vez, puede llamar a otra u otras funciones.

En la lista de parámetros formales se especifica el nombre de cada parámetro junto con su tipo de dato, por lo que ya quedan declarados para el compilador y no hace falta declararlos al principio del algoritmo.

Los parámetros formales se podrán utilizar en cualquier sentencia de la lista de acciones de la función. Los parámetros van separados entre sí por puntos y comas. Cuando varios parámetros tengan el mismo tipo, los separaremos por comas.

Los nombres de los parámetros formales pueden coincidir con identificadores ya utilizados en la sección de declaración de variables del algoritmo principal, pero no pueden coincidir con los utilizados en la declaración de constantes y variables locales.

Ejemplo:

```
Algoritmo Factorial
Variable
    entero n

real Función factorial (entero n)
Variables
    real fact
```

```
        entero cont
    Inicio
        fact = 1
        repite para cont = 1, n
            fact = fact * cont
        fin-repite
        retorna fact
    Fin
    Inicio
        escribe ('Introduce un nº: ')
        lee (n)
        escribe ('El factorial de ',n,' es: ', factorial(n))
    Fin.
```

Las constantes y variables locales, son constantes y variables que sólo existen dentro de la función y no en el algoritmo principal.

Al terminar de ejecutarse el cuerpo de la función, automáticamente se retorna al módulo principal, o bien, al módulo llamador, para terminar la ejecución de la sentencia que provocó la llamada a la función, y, si procede, se devuelve como resultado de la evaluación de la función el valor asociado a la sentencia retorna.

15.3.- Invocación de funciones. Una función es llamada desde el algoritmo principal, o en general, desde otro módulo, de la siguiente forma:

<nombre_función> (<lista parámetros actuales>)

Un **parámetro actual** es cada uno de los valores que aparece en la llamada a un módulo.

Cuando la función tiene que retornar un valor, la llamada a la función no constituye una sentencia válida por sí sola. En este caso, las dos formas más típicas de utilizar la función son:

<variable-tipo-función> = <expresión con función (argumentos)>
escribe (<expresión con función (argumentos)>)

Al terminar de ejecutarse una función se retorna a la misma posición del módulo principal en la que se llamó.

15.3.1.- Proceso de invocación de un módulo: la Pila. Una llamada a un módulo implica los siguientes pasos:

1. Cuando se está ejecutando el algoritmo principal, se está trabajando en una zona de memoria reservada de la memoria principal, en la que se almacenan las variables y constantes declaradas en él. Por otra parte, hay una zona separada en la memoria del ordenador, llamada **pila** y que se reserva para trabajar con los módulos. Cuando se llega a una sentencia del algoritmo principal en

la que hay una llamada a un módulo M, se crea en la pila lo que llamaremos **un registro de activación** o **entorno** E, que es un bloque de memoria reservada para trabajar con dicho móduloM. En el entorno asociado a un módulo se almacena, entre otra información, las constantes y variables locales, así como los parámetros formales.

2. Una vez creado el entorno E, se procede a ejecutar el cuerpo del módulo. Si dentro de M se produce una llamada a otro módulo M', entonces se crea en la pila, justo encima de E, otro entorno E' reservado para trabajar con M'. El entorno activo en cada momento es el último llamado y ocupa el lugar más "alto" de la pila.

3. Terminada la ejecución del módulo subordinado M' se produce un **retorno** al lugar del módulo M en el que se llamó a M' y se borra de la pila el entorno E' correspondiente a M'. Entonces, el entorno E correspondiente a M pasa a ser el entorno que está en la cabecera de la pila, y por tanto, el actual entorno activo. Cuando se haya completado la ejecución de M, la pila se habrá quedado vacía, y se produce un retorno al lugar del algoritmo principal en el que se produjo la llamada, dónde se continuará ejecutando.

15.4.- Constantes y variables locales.

Las constantes y variables locales desaparecen al retornar al entorno principal.

Las **constantes** y **variables locales** a un módulo son aquellas que han sido declaradas en dicho módulo y se almacenan en el entorno reservado para él en la pila.

15.5.- Paso de parámetros en funciones.

La correspondencia entre los parámetros formales y los actuales se efectúa como sigue:

1. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales y la correspondencia se establece por el orden de aparición y no por los nombres de los distintos parámetros.

2. Cada parámetro actual y su correspondiente parámetro formal han de ser del mismo tipo.

15.5.1.- Paso de parámetros por Valor.

Si en la cabecera de la declaración de la función no se dice nada sobre un parámetro, éste se dice que se pasa por **valor**.

Las variables y constantes locales, al principio tienen un valor indeterminado.

Los parámetros formales pasados por valor se consideran **variables locales,** por lo que reciben el mismo tratamiento.

Al principio, el valor del parámetro formal es el que tuviese el correspondiente parámetro actual en el momento de la llamada a la función. De hecho, lo que se produce es una copia del contenido del parámetro actual y el ordenador lo asigna al correspondiente parámetro formal en el entorno de la función.

El parámetro formal queda, con ese valor, como una variable local al entorno del módulo por lo que cualquier operación que hagamos sobre él no afectará al valor que tenía el parámetro actual en el algoritmo principal.

Como regla general, a una función le pasaremos por valor todo aquello que dicho módulo necesite utilizar pero no modificar en sus cálculos.

15.5.2. Paso de parámetros por referencia. En este caso lo que se le pasa a la función no es el valor del parámetro sino la dirección de memoria donde se encuentra dicho parámetro.

15.6.- Recursividad.

Un subprograma (función) puede llamar a cualquier otro subprograma previamente definido e incluso alguno definido posteriormente.

Cuando un subprograma se llama a sí mismo, se habla entonces de **recursividad.** En general, la recursividad conlleva una mayor ocupación de memoria y suele ser lenta, por lo que no se recomienda su uso salvo que el problema lo exija.

Por otro lado, un error frecuente es la realización de programas recursivos sin salida con lo cual el programa entra en un bucle sin salida, es decir, todo algoritmo recursivo debe terminar alguna vez.

Ejemplo: Función potencia para enteros positivos utilizando recursividad:

```
real Función potencia (entero b, e)
Inicio
        si e = 0 entonces
              retorna 1
        si-no
              retorna  (b * potencia (b, e-1))
        fin-si
Fin.
```

15.7. Ejercicios propuestos.

15.1. Hacer una función para que, pasándole un nº positivo, nos devuelva el valor verdadero, si el nº es primo y valor falso, si el nº no es primo.

15.2. Optimizar la función anterior.

15.3. Averiguar si un número es primo o no utilizando la función anterior.

15.4. Hacer un algoritmo que nos visualice por pantalla los números primos entre dos números dados, introducidos por teclado.

15.5. Construir la función factorial. Utilizando dicha función, hacer un algoritmo que calcule el número combinatorio A sobre B.

15.6. Hacer un algoritmo que compruebe si dos números son amigos, utilizando subprogramación. Dos números son amigos si la suma de los divisores del primero, excepto él mismo, es igual al segundo número y viceversa.

15.7. Hacer un algoritmo para comprobar si un número es perfecto. Un número es perfecto cuando la suma de sus divisores excepto él mismo es igual al propio número.

15.8. Visualizar por pantalla las parejas de números amigos que hay entre 2 números introducidos por teclado.

15.9. Visualizar los números perfectos que hay entre 2 números introducidos por teclado.

15.10. Hacer un algoritmo para adivinar un número entre 1 y 100 generado aleatoriamente por el ordenador, indicando en cada momento el intervalo en el que se encuentra el número y sabiendo que se cuenta con un máximo de 5 intentos para adivinarlo. Repetir el proceso tantas veces como se quiera.

15.11. Hacer una función para determinar cuantas cifras posee un número entero positivo. Utilizando dicha función hacer un algoritmo para calcular el número de cifras que posee un número introducido por teclado. Repetir el proceso cuantas veces se quiera.

15.12. Hacer una función para comprobar si una determinada fecha es correcta o no.

15.13. Calcular la última cifra del cuadrado de un número y el número de cifras.

15.14. Hacer una función a la que se le pasen dos números enteros y nos devuelva el valor verdadero, si ambos números son primos entre sí, y el valor falso en caso contrario. Dos números son primos entre si cuando el único divisor común de los dos es la unidad.

15.15. Construir una función llamada **euler** a la cual se le pasa como argumento un número entero y debe devolver el valor de dicha función para este número. La función de euler es el número de enteros positivos inferiores a dicho número y que son primos con él.

15.16. Construir una función que nos calcule cuál es el valor de la suma de los n primeros términos de la serie: 1 + 1/2 + 1/3 + 1/4+ . . . + 1/n.

15.17. Sabiendo que $A^n = A * A^{n-1}$, realizar una función para elevar un número a otro cualquiera.

15.18. Retocar el ejercicio de la potencia para que funcione con exponentes negativos.

15.19. Construir la función máximo común divisor (MCD) a la que se le pasa a y b de tipo entero y devuelva el MCD de dichos números, sabiendo que:

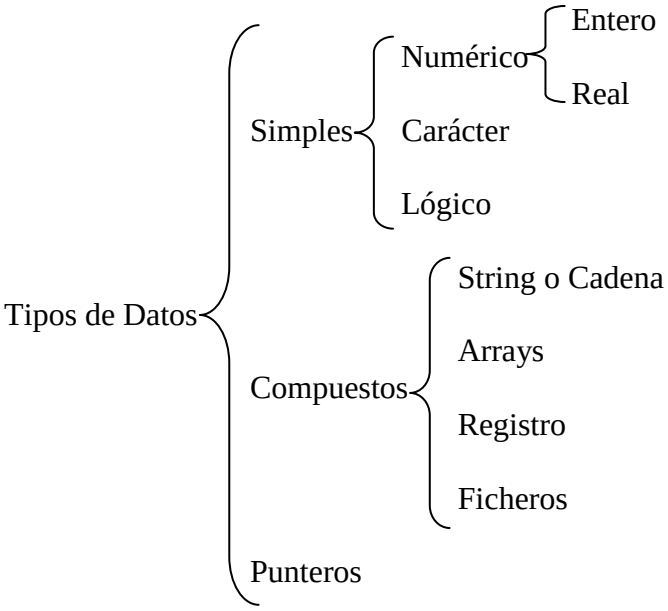
MCD (a, b) = a si a = b
MCD (a, b) = MCD (a-b, b) si a > b
MCD (a, b) = MCD (a, b-a) si b > a

Tema 16: Vectores y Matrices.

16.1.- Vectores.

Hasta ahora hemos tratado con tipos de datos simples, sin embargo, en algunos problemas, es necesario mantener una relación entre variables diferentes o almacenar y referenciar variables como un grupo.

Un **tipo de dato compuesto** es una composición de tipos simples o compuestos, caracterizado por la organización de sus datos y por las operaciones que se definen en él.



Una **matriz** o **array** es un tipo de dato compuesto de un número fijo de componentes del mismo tipo y dónde cada uno de ellos es directamente accesible mediante uno o varios índices. Se denomina **vector** a un *array* de una dimensión.

16.1.1.- Declaración de vectores.

Un vector se declara de la siguiente manera:

tipo nombreArray[tamaño]

donde *tamaño* indica el número de elementos que tiene el vector y delimita el rango del índice:

0 <= índice < tamaño

tipo hace referencia al tipo de dato común a todos los componentes del vector.

Ejemplos: entero nota[35]
carácter dato[10]

16.1.2.- Almacenamiento en memoria. Al declarar un vector o en general una matriz, se reserva una cantidad fija de memoria inalterable durante toda la ejecución del programa, ocupando cada componente un espacio igual al del tipo de dato que aparece en la declaración, y estando todas ellas en posiciones contiguas de memoria.

Cuando se declara una variable de tipo matriz no se asigna un valor por defecto, siempre el mismo a todas las componentes, sino que tendrá un valor desconocido por el programador. Dicho valor corresponde al estado en el que la memoria asignada a la variable, se encuentre en ese momento.

16.1.3.- Operaciones con vectores.

- Acceso. Cada elemento o componente del vector se nombra con el identificador del vector al que pertenece y, entre corchetes, el valor del índice:

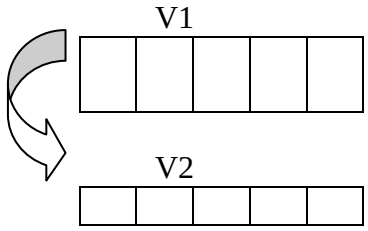
<identificador de vector>[<valor de índice>]

- Asignación. La asignación con vectores se podrá realizar de dos formas:

La primera forma consiste en dar valores a cada elemento del vector: se realizará con la instrucción de asignación usual.

Ejemplo:
```
repite para cont = 0, 9, 1
   nota(cont)=0
fin-repite
/* inicializa un vector de 10 elementos a 0 */
```

La otra forma de asignar valores a un vector es a través de otra variable vector a la que previamente le hayamos asignado valores componente a componente.



V2 = V1

- Lectura y escritura: La lectura y escritura se realiza componente a componente, en la forma habitual.

lee (<identificador vector> [índice])

escribe (<identificador vector> [índice])

Ejemplo: Crear un vector de 20 elementos y rellenarlo con números enteros introducidos desde teclado.

```
Algoritmo vector_1
Variables
        entero numero[20]
        entero cont
Inicio
        repite para cont = 0,19,1
            escribe ("Introduzca un nº: ")
            lee (numero[cont])
        fin-repite
Fin.
```

Ejemplo: Hacer un algoritmo para visualizar los 20 números del vector anterior.

```
Algoritmo vector_2
Variables
        entero numero[20]
        entero cont
Inicio
        repite para cont =0,19,1
            escribe ("El valor es: ", numero[cont])
        fin-repite
Fin.
```

16.1.4.- Algoritmos de búsqueda en vectores.

Un **algoritmo de búsqueda** tiene como finalidad averiguar si un elemento se encuentra en un conjunto de datos, en nuestro caso, un vector; en caso de que se encuentre, el algoritmo dará como resultado la posición en la que se encuentra dicho elemento. Existen dos formas de buscar un elemento de un vector: búsqueda lineal y búsqueda dicotómica o binaria.

16.1.4.1.- Búsqueda lineal. Consiste en hacer un recorrido secuencial por el vector, comparando cada componente del vector con el elemento que se quiere encontrar. Hay dos condiciones que interrumpen la búsqueda:

- se ha localizado el elemento que buscaba.
- se recorre todo el vector y no se ha encontrado el elemento.

Ejemplo: Crear un vector con 20 números comprendidos entre 1 y 100 y escribir un procedimiento que nos busque un elemento dentro de dicho vector, utilizando búsqueda lineal.

```
Algoritmo búsqueda_lineal
Variables
        entero vector[20]
        entero n, cont
```

```
Función busquedaLineal (entero vector[20], n )
Inicio
        cont = -1
        repite
            cont = cont + 1
        hasta (vector[cont] = n) o (cont =20)
        si vector[cont]= n entonces
            escribe ("El elemento ocupa la posición: ", cont)
        si-no
            escribe ("El elemento no existe")
        fin-si
Fin

Inicio
        repite para cont = 0, 19
            vector [cont] = int(rnd(100)+1)
        fin-repite
        escribe ("Introduce el nº a buscar: ")
        lee (n)
        busquedaLineal (vector, n )
Fin.
```

16.1.4.2.- Búsqueda dicotómica o binaria. La búsqueda puede hacerse más efectiva si los datos presentes en el vector ya están ordenados. Suponiendo que esto es así, una solución sería elegir un componente del vector al azar y compararlo con el elemento buscado. Si ambos son iguales, hemos acabado la búsqueda; si el elemento buscado es mayor, buscamos a la derecha y en caso contrario buscamos a la izquierda. La solución óptima se consigue tomando como partida la mediana del vector. La mediana es aquel valor del vector tal que la mitad de los elementos de dicho elemento son menores que él y la otra mitad son mayores.

Ejemplo: búsqueda dicotómica.

```
Función búsquedaDicotómica(entero vector[20], n)
Variables
        entero sw, mitad, primero, ultimo, cont
Inicio
        sw = 0
        primero = 0
        ultimo = 19
        repite mientras sw = 0 y primero <= ultimo
            mitad = (primero + ultimo) div 2
            si vector[mitad]= n entonces
                sw = 1
            si-no
                si vector[mitad] < n entonces
```

```
                    primero = mitad + 1
                si-no
                    ultimo = mitad – 1
                fin-si
            fin-si
        fin-repite
        si sw = 0 entonces
            escribe (“El nº no existe”)
        si-no
            escribe (“La posición es: “, mitad)
        fin-si
Fin.
```

16.1.5.- Algoritmos de ordenación. Ordenar es el proceso de reorganizar un conjunto de objetos en una secuencia especificada.

Los algoritmos que vamos a estudiar se aplican sobre vectores y al igual que los algoritmos de búsqueda, pueden generalizarse fácilmente para tratar con otras estructuras de datos. Nos centraremos en la ordenación ascendente.

16.1.5.1.- Ordenación por Intercambio (método de la burbuja). Este algoritmo se basa en el principio de comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados. Para ello, se hacen repetidas pasadas sobre el vector moviendo en cada una, el elemento de clave mayor hasta el extremo derecho del vector que resta por ordenar.

Ejemplo: Algoritmo de ordenación por el método de la burbuja.

```
Funcion burbuja (entero vector[101])
Variables
        entero cont, cont2, aux
        booleano ordenado
Inicio
        ordenado = Falso
        cont = 1
        repite mientras (cont <=100) y (ordenado = Falso)
            ordenado  = Verdadero
            repite para cont2 = 1, n-cont
                si vector [cont2] > vector [cont2 + 1]
                    aux = vector [cont2]
                    vector [cont2]= vector [cont2 + 1]
                    vector [cont2 + 1] = aux
                    ordenado = Falso
                fin-si
            fin-repite
            cont = cont + 1
        fin-repite
Fin.
```

16.1.5.2.- Ordenación por Selección. Este método se basa en ir seleccionando el elemento con clave mínima de todo el vector e intercambiarlo por el primero. A continuación se repiten estas operaciones con los elementos restantes hasta que quede un único valor, el mayor.

Ejemplo: Algoritmo para ordenar un vector de n elementos por el método de selección.

```
Funcion selección (entero vector[101], ult_elem)
Variables
        entero i, j, n, aux, pos_menor, menor
Inicio
        n = ult_elem
        repite para i = 1, n – 1
            pos_menor = i ; menor = vector[i]
            repite para j = i+1, n
                si vector [j] < menor entonces
                    menor = vector [j]
                    pos_menor = j
                fin_si
            fin-repite
            vector [pos_menor] = vector [i]
            vector [i] = menor
        fin_repite
Fin.
```

16.1.5.3.- Ordenación por Inserción. Considera que el primer elemento está ordenado y ordena el segundo respecto al primero, es decir, lo coloca delante o lo deja donde está según proceda. Luego ordena el tercer elemento respecto de los dos anteriores. Y así sucesivamente hasta el último elemento, que lo debe colocar en el lugar que le corresponda respecto a todos los demás.

Ejemplo: Algoritmo para ordenar una lista de n elementos por el método de inserción.

```
Funcion inserción (entero vector[101], ult_elem)
Variables
        entero i, j, n
Inicio
    n = ult_elem
        repite para i =2, n
          j = i
          repite mientras ((vector[j-1] > vector[i]) y (j > 1))
                vector[j] <-> vector[j-1]
                j = j-1
          fin-repite
      fin-repite
Fin.
```

16.1.5.4.- Método shell. Está basado en la ordenación por inserción. Se ordenan los elementos que están separados por una distancia de comparación que en un principio es la mitad de la longitud del array. En cada etapa esta distancia se va reduciendo a la mitad hasta que sea menor que la unidad, en cuyo caso la lista ya estará ordenada. Si el número de elementos es impar, calcularemos la parte entera de su mitad. La comparación de los elementos que se encuentran a una distancia determinada debe estar realizándose mientras que haya cambios, es decir, no están ordenados los elementos respecto a ese salto.

Ejemplo: Algoritmo para ordenar un vector de n elementos mediante el método shell.

```
Funcion SHELL (entero vector[101], n)
Variables
        entero sw, i, salto, aux
Inicio
        salto = n
        repite mientras salto <> 1
            sw = 1,  salto = int(salto/2)
            repite mientras sw<>0
              i = 1, sw = 0
              repite mientras i <= (n-salto)
                  si vector [i] > vector [i + salto] entonces
                      aux = vector [i + salto]
                      vector [i + salto] = vector [i]
                      vector [i]= aux
                      sw = 1
                  fin-si
                  i = i + 1
              fin-repite
            fin-repite
        fin-repite
Fin.
```

16.2.- Matrices de varias dimensiones.

Para declarar un *array* de varias dimensiones, el formato es el siguiente:

tipo nombre_array[tamaño_1][tamaño_2]....[tamaño_n]

donde *tamaño_1*, *tamaño_2*, ...*tamaño_n* <v11>..<v1n> son los valores que delimitan el rango de los valores que podrán tomar cada uno de los correspondientes índices, de tal forma que se debe cumplir:

0<= índice_i <tamaño_i

y *tipo* es el tipo de dato común a todas las componentes de la matriz.

No hay restricción alguna, salvo la capacidad de memoria del ordenador, para el número de índices que se pueden utilizar. En el caso de que se utilicen dos índices, diremos que la matriz es **bidimensional** y si se utilizan más de dos índices, que es **multidimensional**.

Para matrices bidimensionales utilizaremos los términos **fila** para referirnos al primer índice y **columna** para el segundo.

16.2.1.- Almacenamiento en memoria.

Al igual que los vectores, todas las componentes de una matriz se almacenan en posiciones contiguas en memoria, normalmente por filas consecutivas.

16.2.2.- Operaciones con matrices.

- Acceso. Para acceder o referenciar un elemento particular de una matriz necesitamos dos índices, el primero que hace referencia a la fila y el segundo a la columna.. Por ejemplo:

m[fil][col] es el elemento situado en la fila **fil** y columna **col**.

- Asignación. La asignación se realiza como en los vectores, bien asignando una variable matriz a otra del mismo tipo, o bien componente a componente. Por ejemplo:

```
entero M1[4][5]
entero M2[4][5]
1º) M1 = M2
2º) M1[1][4] = M2 [3][2]
```

- Lectura y escritura. Tanto la lectura como la escritura se realizan en la forma habitual, componente a componente. El formato es el siguiente:

lee (<identificador> [<fila>][<columna>])

escribe (<identificador> [<fila>][<columna>])

16.3. Ejercicios propuestos.

16.1. Hacer un algoritmo que lea del teclado 20 números enteros y los almacene en un vector. Utilizando este vector, visualizar y sumar los elementos que ocupan las posiciones pares.

16.2. Algoritmo para visualizar, contar y sumar los números pares que ocupan las posiciones impares del vector anterior. Visualizar las posiciones que ocupan dichos elementos en la lista.

16.3. Algoritmo que nos rellene un vector con 50 números aleatorios de 1 a 100. Visualizar los elementos, así como la posición que ocupan, que sean al mismo tiempo múltiplos de 2 y múltiplos de 3. Calcular su suma.

16.4. Generar y visualizar un vector con la tabla de multiplicar de un número introducido por teclado. Imprimir el elemento que ocupa la posición 3.

16.5. Crear e imprimir un array unidimensional de 50 elementos con números aleatorios entre 1 y 100 de tal forma que no se repita ninguno.

16.6. Crear una función para insertar números en un vector, por posición. Hacer un algoritmo que utilizando dicha función nos permita insertar un número en un vector numérico de 10 elementos.

16.7. Hacer una función para borrar un número de un vector . Si no lo encuentra deberá devolver un valor falso. Hacer un algoritmo para borrar un número introducido desde teclado en el vector anteriormente creado.

16.8. Hacer una función que seleccione el mayor de los números de un vector numérico y lo lleve a la última posición.