JAVA J2SE

ANDRÉS CHILLÓN SESMA

00. CONFIGURACIÓN DEL SISTEMA.

1. Instalar el SDK. Configurar las variables de entorno:

Botón derecho sobre MiPC - Propiedades. Pestaña opciones avanzadas. Variables de entorno. Crear las variables 1 a 1 como de usuario.

► LA CASA DE JAVA (necesario para NetBeans ... si no, no instalará al no detectar un JDK):

JAVA_HOME = C:\JAVA\JDK 7

► LOCALIZACIÓN DE LA MÁQUINA VIRTUAL (JAVA.EXE) Y EL COMPILADOR DE CONSOLA (JAVAC.EXE):

PATH = C:\JAVA\JDK 7\bin

► LOCALIZACIÓN DE LAS CLASES: DIRECTORIO ACTUAL(.), rt.jar(J2SE) ... para web habría que localizar la ruta de j2ee.jar:

CLASSPATH = .;C:\JAVA\JDK 7\JRE 7\lib

Probar que todo funciona (línea de comandos – cmd): java –version (los modificadores siempre en minúsculas)
javac

2. Instalar NetBeans y configurar: → INSTALAREMOS INTELLIJ

* Show line numbers: Botón derecho sobre columna del medio (necesario tener 1 fichero abierto)

* Synchronize Editor with views: En la pestaña View.

* Templates: En la pestaña Tools, Templates, Java, Java Class – Open in editor.

```
<#if package?? && package != "">
package ${package};
</#if>

public class ${name} {

}
```

* Tamaño letra para proyector: Tools – Options – Fonts and Colors – Elegir Font – Default ... Tamaño de letra y negrita.

Accesos rápidos:

psvm	public static void main (String[] args) {}
sout	System.out.println(“”);
[CTRL] + [SPACE]	Obtener la ayuda del NetBeans para métodos y constructores.
[CTRL] + [MAYÚSC]+[C]	Comentar y descomentar código en base a 1 línea //.
[CTRL] + BotónIzqRatón	Saltar a la definición del método o constructor indicado en pantalla. También con super(xxx) y this(xxx).
[TABULAD] + - [MAYÚSC]	Mover el texto seleccionado hacia derecha o izquierda.

3. Bibliografía recomendada SCJP:

Mc Graw Hill SCJP 6.0.
Apress SCJP 5.0.
Thinking in Java v4.0.
Core Java 2.
Colección completa Head First.

4. Comentarios sobre la certificación:

Java estandar ..... SCJP	(Sun Certified Java Programmer)
Java web. Servlets y Jsp´s ..... SCWCD	(Sun Certified Web Component Developer)
Java enterprise. Ejb´s ..... SCBCD	(Sun Certified Business Component Developer)

exámenes de prueba en la red: Pass4Sure - ExamCollection - TestKing - Examworx.

5. Nomenclatura a utilizar:

Nombre de los proyectos:	*JAVAxx_NOMBRE_PROYECTO*	*xx = 00, 01, 02 ... etc.*
Carpetas de proyectos:	*JAVA/PROYECTOS/JAVA/ CONSOLA* *JAVA/PROYECTOS/JAVA/ NETBEANS*	*ordenado por directorios /WEB, /EJB ... etc.*
Paquetes y clases	*p0, p1, p2 ... A, B, C, Intruso, Animal, Gato*	*utilizar nombres sencillos y descriptivos.*

01. INTRODUCCIÓN A JAVA.

Lenguaje de programación orientado a objetos (POO). No hay un tallo central sino objetos colaborando entre si. El inicio está en un método main.

Independiente de la plataforma “writes once run anywhere”. Lenguaje interpretado ... éste es el éxito de Java.

Ciclo de creación: .java(block de notas).class(compilación/1 por clase)JVM(ejecución sobre el S.O).

*** con [MAYÚSC] + [F6] ejecuta el fichero en pantalla (no usa los parámetros). Con [F6] se ejecuta el proyecto principal (usa los parámetros).

02A. DECLARACIÓN DE VARIABLES.

Una variable es un elemento en el que podemos almacenar valores que pueden cambiar a lo largo del tiempo. Son primitivas y referenciadas.

Nombres válidos de variables: no empezar NUNCA con número ni ser una palabra reservada. Símbolos permitidos $ _.

Variables globales, también llamadas atributos, campos de la clase, variables miembro de la clase o variables de instancia. Almacenar en el Heap.

Variables finales (constantes).

Variables locales, aquellas definidas dentro de un método, constructor o bloque. Almacenar en el Stack (el objeto en sí siempre está en el Heap).

Inicialización por defecto de las globales. Necesidad de inicializar las locales antes de usarlas.

Ámbito de existencia globales y locales. Globales = toda la clase. Locales = sólo dentro del método.

Ambigüedad cuando una variable local y global tiene el mismo nombre. Solución mediante el operador this. Se admiten tipos distintos para cada una.

Variables estáticas (static) o de tipo compartido (la misma para todos). No requieren instanciación. Viven en la clase, no en la instancia.

Convenio de nombres para variables finales y no finales. El nombre de las finales se pone todo en mayúsculas.

Se puede usar *abstract* para declarar una variable local, pero NUNCA GLOBAL (aunque no tiene ninguna utilidad, al menos en apariencia).

02B. VARIABLES DE TIPO PRIMITIVO.

boolean(1 bit ... aunque realmente depende de la arquitectura de la máquina física)

byte(8), short(16), char(16), int(32), long(64 .. l, L) … los char no tienen signo (0 … 65535).

float(32 .. f, F), double(64 .. d, D) ... En cuanto pongamos el pto decimal, ya es double. Usar f o F para indicar Float
float f = 3.0F;

Inicialización (cuando es variable global):

boolean	-->	false. No es un número, no puede mezclarse con el resto de primitivos.
byte	-->	0
short	-->	0
char	-->	\u0000 (unicode 16 bits sin signo). También es un número.
int	-->	0
long	-->	0
float	-->	0.0
double	-->	0.0

Autopromoción a int (Upgrading --- pasar de un tipo inferior a uno superior ... byte, short, charint).

Los float no promocionan a double al sumarlos entre sí, pero algo como 3.0 es double y no float. Usaremos los sufícios f y F para indicar float (3.0F)

Mezcla de tipos numéricos. Obligatoriedad de almacenar el resultado en un tipo con tamaño suficientemente grande --> int o el mayor de todos ellos.

Notación hexadecimal con **0x** valor(0 ... F). ... 0x1, 0XA, 0xFFF Utilizar x ó X.

Notación octal con **0**valor(0 ... 7). ... 01, 07, 010, 0123

Notación binaria con **0b**valor(0 ... 1). ... 0b1, 0B10, 0b111 Utilizar b ó B.

*** Los tipos referenciados, se inicializan a null. Una “cadena de texto” es la clase java.lang.String.

03A. OPERADORES. (hasta aquí Incluido echar ojo)

Operador Unario:
Autoincrementales ++x x++ --x x--. Asignar y luego incrementar (y = x++) o incrementar y luego asignar (y = ++x).

Operador Binario:
Relacionales < > == <= >= != ... resulta un booleano.
Aritméticos + - / * % ... resulta un número.
Lógicos &&(AND inteligente) &(AND torpe) ||(OR inteligente) |(OR torpe) !(NOT) ... resulta un booleano.

Bit	& (AND) | (OR) ^ (XOR) ~ (NOT)	... resulta un número.
Desplazamiento	>> (con signo) >>> (sin signo) << (sin signo) ..... dato -operador- veces	... resulta un número.
Auto - asignación	+= -= /= *= %= &= |= ^= >>= >>>= <<=	... casting automático.

Operador Ternario: (condición) ? valor_si_true : valor_si_false; ... String aprobado = (nota>=5) **?** "SÍ" **:** "NO";
... Es equivalente a un condicional if – else y se puede anidar.

Casting (tipo) Para convertir unos tipos a otros ... cuidado con la posible pérdida de precisión y los errores ClassCastException.
Válido para primitivos(numéricos) y referenciados. Para estos últimos, las clases deben pertenecer a la misma línea de herencia, o de lo contrario el compilador dará un error.

```
int x = 2;
byte b = (byte) x;
Gato g = (Gato) animal; ---> ok        Gato g = (Gato) perro; ---> error compilador.
```

instanceof Comprobar si el objeto apuntado por la referencia pertenece a un determinado tipo o subtipo (animal instanceof Gato), devolviendo true o false. Al igual que con el casting, el compilador dará un error si detecta que no hay relación de herencia entre dicho objeto y la clase con la que se compara. Muy útil comprobar tipos en aquellas referencias que usen polimorfismo.

```
boolean cierto = animal instanceof Gato; ---> ok    boolean cierto = perro instanceof Gato; ---> error compilador.
```

03B. OPERACIONES SOBRE LOS DATOS.

AutoPromoción Conversiones implícitas de tipos primitivos de datos, que efectúa el compilador (conversión de un tipo a otro superior):

```
char c = (char) ('a' + 5);      ... convierte implícitamente el char 'a' a su equivalente int, luego
                                    forzamos casting para verlo como un char.

byte b1=1, b2=2;
byte c = (byte) (b1 + b2);      ... b1 y b2 son byte pero se autopromocionan a int para sumarse.

b1++;                           ... no se autopromociona en solitario.
```

Los tipos byte, short, char --> autopromocionan a int, salvo que sea un autoincrementador (++ o --).

Al mezclar diversos tipos de datos (compatibles entre si), el tipo del resultado será el correspondiente al mayor de ellos:
int --> long --> float --> double

No hay autopromoción entre tipos referenciados. Las clases de envoltorio NO HEREDAN UNAS DE OTRAS, pero sí de Object.

La autopromoción sólo se podrá dar si empleamos variables. Si utilizamos directamente numéros (1+1) se toma como int.

Con operaciones del tipo b += 1; donde b es de tipo byte, hay autoCasting al tipo de la variable utilizada, compensándose así la autopromoción generada de sumar dicha variable byte con otro valor. En operaciones del tipo b = b +1, siendo b de tipo byte, hay error de compilación causado por la autopromoción de b al tipo int (aquí hemos de forzar manualmente el casting).

AutoCasting Con operaciones de auto-asignación (+=, -=, *=, /=, %=) se produce un casting automático al tipo de variable primitivo utilizado.

```
byte b = 0;

byte b += 1;                    ... OK.
byte b += (int) 2;              ... OK.

byte b = b + 1;                 ... ERROR DE COMPILACIÓN, NECESARIO CASTING.
```

AutoBoxing Empaquetamiento automático de tipos primitivos a su correspondiente Wrapper (también llamada clase de envoltorio).

WRAPPERS : Boolean, Byte, Short, Character, Integer, Long, Float, Double.

```
Integer i = 5;
```

AutoUnBoxing Desempaquetamiento automático de un Wrapper a un tipo primitivo. Fábricas de conversión de tipos (Boolean y Character NO).

```
int x = new Integer(5);
```

Métodos genéricos de los Wrappers:

```
Crear el Wrapper      --->   WRAPPER.valueOf(primitivo);
                             new WRAPPER(primitivo);

Obtener el primitivo  --->   WRAPPER.xxxValue();          ... xxx son los tipos primitivos compatibles.
                             WRAPPER.parseXXX(String);    ... cada Wrapper sólo tiene el suyo (Character nada *).

Excepción conversión -->     NumberFormatException.
```

* Character no dispone del método parseChar(String), pero Boolean sí dispone del método parseBoolean(String) para obtener su primitivo.

Método elegido Al operar con valores primitivos y siempre que coincida el número de argumentos con el de la invocación al método, puede incurrirse en autopromoción o autoboxing (NÚNCA LOS DOS A LA VEZ), eligiéndose en primer lugar aquél método que tenga como argumento de entrada:

1. El tipo primitivo más pequeño que sirva para almacenar el dato envíado (tamaño >= necesario).
2. La clase Wrapper que coincida exactamente con el tipo de dato de la invocación (esto puede implicar autoboxing).
3. La clase Number o la clase Object. Se va subiendo en el árbol de jerarquía de clases.
4. Un argumento de entrada de longitud variable (int ... x). Esto puede requerir autopromoción o autoboxing.

01 - Tablas de lógica booleana: AND, OR, XOR, NOT.

02 - Tabla de prioridad de operadores.

*** Distinguir entre el operador de asignación (=) y el de igualdad (==).

04. PALABRAS RESERVADAS / CARACTERES DE ESCAPE / COMENTARIOS.

Comentarios de una sóla línea con //
Comentarios multi-línea con /* ... */
Comentarios tipo JavaDoc /** ... */

03 -Tabla con palabras reservadas en Java.

04 - Tabla con caracteres de escape.

05. CONTROL DE FLUJO – SENTENCIAS Y BLOQUES CONDICIONALES. BUCLES(Loops).

CONDICIONALES:

if	condicional simple.
if else	condicional doble
if else if else	condicional múltiple

No podemos declarar variables en los paréntesis del if, pero sí usar las externas a éste, asignándolas un valor directamente o mediante la invocación a un método. Dichas variables no pueden volverse a declarar dentro de las llaves del if.

Se pueden declarar variables dentro de las llaves del if, pero su visibilidad quedará limitada a dichas llaves (no se verán fuera).

En último caso, el valor de lo encerrado entre los paréntesis, siempre debe poder ser evaluado a un booleano.

Una vez encontrado un condicional válido (que devuelva true), no se evalúa el resto del grupo (if-else if-else).

La declaración de variables locales a un método, constructor o bloque, es siempre en orden de arriba a abajo, a diferencia de las globales que pueden declararse al final de la clase, a continuación de los métodos, sin que ésto afecte a su visibilidad.

bloque switch:

```
switch (valor_a_evaluar) { // es necesario utilizar (paréntesis) para poder encerrar la condición.

        case valor1:
                líneas a ejecutar;
                break;
        default:    líneas a ejecutar;

}
```

Funcionamiento de un bloque SWITCH:

Válido sólo para los tipos primitivos byte, short, char, int y los Wrappers de éstos gracias al autounboxing ... y ahora los String en Java 7.
Válido para tipos enumerados.
Vigilar el FALL-THROUGH o caída libre y dónde ponemos el default (en algunas ocasiones es beneficioso de cara a no repetir código).
Claúsulas case y default son opcionales.
No podemos repetir 2 o más condiciones case.
El único operador que se permite es el de igualdad (==), por lo que no podemos utilizar los operadores !, <, >, <=, >=. Así, no podremos comprobar rangos.

Entra por el primer case válido y sigue hacia abajo hasta el primer break. Por el default entraremos **si no hay ningún case válido** (no importa posición) y a partir de ahí, hacia abajo hasta el primer break (o el final del bloque), atravesándo todos aquellos case que pudiera encontrar en su camino (no evalúa su condición). Es por ésto que lo habitual es finalizar cada case con un break y es ubicar el default el último (aunque también sería válido si le añadimos a continuación un break a ese default colocado en una posición que no es la final.

BUCLES (Loops):

bucle for tradicional:

```
for (incialización; condición; operaciones)  {}
```

... vigilar los bucles infinitos (desbordamiento de pila).

... son opcionales: for(;;).
... inicialización válida: for(**int x=3, y=7**; ;) . Sólo se permite 1 tipo y
... sólo se realiza la 1ª vez en la ejecución del bucle.
... se permiten múltiples operaciones separadas por comas (,).
... da igual usar ++contador que contador++ en el apartado operaciones.
... inicializa -> comprueba -> { ejecuta código | sal } -> comprueba ... etc.

bucle for-each: ... no es válido ni para Enumeration ni para Iterator.
for (tipo variable : colección/array) {} ... obligatorio declarar la variable temporal en la cabecera del for-each.

while (condición) {código} ... 0 o infinitas veces. No podemos crear la variable entre los (paréntesis).

do {código} while (condición); ... 1 o infinitas veces … pero por lo menos 1. No olvidar el ; del final.
... Las variables de {código} no se verán en (condición)

SALTOS: break; ... rompe el bucle por completo.
continue; ... se salta un determinado paso del bucle, no todo.

SALTOS CON ETIQUETA (útil cuando tenemos anidamiento de bloques y queremos poder elegir el romper los internos o los externos):

```
salto:
    break salto;
    continue salto;
```

06. ARRAYS / MATRICES.

Es un tipo de objeto Java. Tamaño fijo, no crece dinámicamente y no puede contener distintos tipos, pero sí subtipos de uno dado (regla IS-A).

Elementos empiezan en posición 0 y se inicializan al valor por defecto del tipo correspondiente (porque un array es un objeto de una clase).

Propiedad .length (no es método como en los String).

Excepción no comprobada **ArrayIndexOutOfBoundsException** cuando superamos los límites del array [0 ... length-1].

Los que contienen objetos, en realidad sólo almacenan la referencia, no el objeto en sí (todos, incluído el propio array, están en Heap de memoria).

Unidimensionales (un array de elementos primitivos-referenciados). Formas de declararlo:

```
int[] datos = new int[3];          … relleno, por ejemplo, con un bucle for. Los elementos siempre se inicializan a su valor por defecto.
int[| datos = {1,2,3};             … en la parte de la declaración (izquierda), nunca se da la dimensión.
int[] datos = new int[] {1,2,3};   … si se indican los elementos, no se puede indicar la dimensión (cálculo automático del sistema).
```

Multidimensionales (un array de arrays, así, la segunda dimensión es un xxx[]):

```
int[][] matriz = new int[2][];          ... el valor de la segunda dimensión es opcional. Hay 2 elementos inicializados a null.
int[][] matriz = new int[2][3];         ... hay 2 sub-elementos array que contienen 3 int cada uno inicializados todos ellos a 0.

int[|[| matriz = { {1,1}, {2,2}, {3,3} };   ... declaración e incialización de un array 2D. Es la forma más cómoda.
int[][] matriz = new int[][] { {1,1}, {2,2}, {3,3} };
int[][] matriz = { new int[]{1,1}, new int[]{2,2}, new int[]{3,3,} };
```

Longitud – propiedad: .lenght ... en el caso de multidimensión, nos la la longitud de la primera dimensión (número filas).
Método especial de copia: .clone()
Otro forma de copiar: System.arraycopy(origen, pos_ini_origen, destino, pos_ini_destino, cuantos_elementos)

No es habitual el uso de arrays de más de 2 dimensiones. En cualquier caso, cada dimensión esta compuesta por 1 array, de forma que en 2D, tenemos un array principal que contiene en cada una de sus posiciones a otros arrays ... y cada uno de ellos a su vez a otros arrays (3D – 4D ... etc).

Para poder utilizar la referencia de un array con el proposito de manejar otro, AMBOS DEBEN TENER EXACTAMENTE LA MISMA DIMENSIÓN.
No podemos utilizar una referencia de tipo 2D para manejar uno de dimensión 1D ya que no son compatibles entre si.

2D --> **ARRAY_CONTENEDOR [** SUB_ARRAY_1, SUB_ARRAY2, SUB_ARRAY_3**] --> una matriz de arrays == filas de arrays (válido irregular).**

1er elemento = SUB_ARRAY_1 2do elemento = SUB_ARRAY_2 3er elemento = SUB_ARRAY_3

Así, para poder obtener un elemento en concreto con 2D, deberemos dar las coordenadas de cada dimensión [][] (fila y columna).
Si sólo damos la primera, obtendremos el array correspondiente a la primera coordenada (fila).

CASOS ESPECIALES EN LA DECLARACIÓN Y ASIGNACIÓN DE LOS ARRAYS:

Se puede manejar un array de subtipos con un array de tipo o supertipo de este:

```
Animal[] animales = new Gato[3];                    // CORRECTO, pero los objetos se ven como de tipo Animal.
```

Una colección con género no admite polimorfismo en la declaración del tipo (salvo que se use ? extends o ? super):

```
ArrayList<Animal> misAnimales                = new ArrayList<Gato>();    // ERROR COMPILADOR.
ArrayList<?> misAnimales0                    = new ArrayList<Gato>();    // OK ... no añadir nuevos elementos.
ArrayList<? extends Animal> misAnimales1     = new ArrayList<Gato>();    // OK ... no añadir nuevos elementos.
ArrayList<? super Animal> misAnimales2       = new ArrayList<Object>();  // OK ... sí añadir pero sólo tipo/subtipo de Animal.
```

Un array de tipos primitivos no puede manejar un array de otro tipo primitivo que no sea el suyo (ni siquiera aunque sea un tipo inferior en rango):

```
int[] numeros = new byte[3];                        // ERROR COMPILADOR.
```

Un array de tipos Wrapper no puede manejar un array de primitivos, ni siquiera del tipo primitivo que le corresponde:

```
Integer[] wrapper = new int[3];                     // ERROR COMPILADOR.
```

Podemos ordenar los elementos de un array (si estos son ordenables) mediante Arrays.sort(el_array) ... los cambios quedan sobre el array original.

Podemos utilizar for(int datos : new int[]{1,2,3}) ... el objeto se maneja por el bucle for-each y al finalizar, se marca como candidato a recolección.

07. LA CLASE Math – java.lang.Math.

Cuando queremos obtener números aleatorios, podemos hacer uso de la clase Math (pertenece a java.lang, al igual que String).

La forma de obtenerlos podría ser la siguiente:

int valores_1 = (int)(Math.random() * N);

... donde **N** es un valor de tipo int, que nos va a permitir determinar el rango de valores a obtener.

Math.random() ... generará valores entre 0 y 0.999999... en formato double.
* N ... genera un nuevo número resultado de multiplicar el anterior por dicho N. Se utiliza para desplazar la coma decimal.
(int) ... castea a int el resultado . Si no ponemos los () dará 0. Se elimina la parte decimal (no redondea).

Si por ejemplo queremos obtener números entre 0 y 9 (ambos incluídos), colocaremos:

*int valores_1 = (int)(Math.random()*10);*
0.0000.. * 10 = 00.000.. -> queda 0.
0.1000.. * 10 = 01.000.. -> queda 1.
..
0.9999.. * 10 = 09.999.. -> queda 9.

De lo que se deduce que la formula anterior nos devolverá el rango dinámico de valores (incluyendo los extremos) **[0 .. N-1]**.

Si queremos un rango de números **[1 .. N]** debemos modificarla simplemente añadiendo +1 (observar que ahora el límite inferior no es 0 sino 1):

int valores_2 = (int)(Math.random() * N) + 1;

*int valores_2 = (int)(Math.random()*10) + 1;*
0.0000.. * 10 = 00.000.. +1 -> queda 1.
0.1000.. * 10 = 01.000.. +1 -> queda 2.
..
0.9999.. * 10 = 09.999.. +1 -> queda 10 .

Para simular el clásico dado (1 .. 6), utilizaremos N=6 y sumaremos +1 ---> int valores_3 = (int)(Math.random() * 6) + 1;

Si queremos valores negativos, multiplicar todo por –1. ---> int valores_4 = -1 * ((int)(Math.random() * 6) + 1);

1-00 LOS 4 PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS.

ABSTRACCIÓN:

Se trata de ocultar aquellos detalles que no es necesario conocer, para poder utilizar algo. Utilizando la encapsulación, podemos limitar o prohibir el acceso DIRECTO a ciertas partes del código que no queremos que otros elementos vean, sin que por ello pierdan funcionalidad. Así, es habitual disponer de elementos públicos que internamente acceden a otros privados, es decir, aquellos a los que directamente no tendríamos acceso.

Podemos por tanto, abstraernos de LOS DETALLES DE LA IMPLEMENTACIÓN (el código de la clase en sí), a cambio de disponer de un conjunto de métodos públicos (fachada o interfaz de negocio) que nos permitan utilizar el objeto en cuestión.

También podemos entender la abstracción como el obtener las características comúnes a una determinada familia de objetos, de cara a declarar dichas características en una clase madre y reutilizarlas en las clases hijas mediante la herencia (usar protected #).

Ejemplos:

► Un iterador me permite recorrer los elementos de una colección sin conocer realmente como está construida internamente.
► Un método comprueba la valided del PIN de la tarjeta de crédito devolviendo true ó false (el proceso real de verificación queda oculto).
► Todos los trabajadores tienen un NIF, Seguridad Social ... se crea una clase persona con los detalles comunes a todos ellos.

ENCAPSULACIÓN:

Ocultar o limitar el acceso a algo según quién lo solicite y desde dónde. Se facilita el desacoplamiento de clases (private) la seguridad ... etc.
Se definen 4 niveles en Java. Desde el más restrictivo al más permisivo:

private --> "default" --> protected --> public

HERENCIA:

La capacidad de adquirir ciertas propiedades y funcionalidad de la clase desde la que estamos heredando. [MADRE] <-- [HIJA].
En Java NO HAY MULTIHERENCIA, pero puede simularse mediante el uso de interfaces (no hay colisión de métodos).
Toda clase en Java siempre hereda, ya sea de forma directa o indirecta, de la super clase Object (salvo ella de si misma), luego al hablar de clases en Java, diremos que SIEMPRE HAY HERENCIA (el decir que todo se hereda no implica obligatoriamente que sea directamente accesible) y que ésta SIEMPRE ES SIMPLE.

POLIMORFISMO:

El poder manejar distintos tipos de objetos mediante una misma referencia (nos lleva a distintos comportamientos). Para ello, se utiliza lo que se conoce como LAZY-BINDING (enlace tardío o enlace en tiempo de ejecución, con métodos no estáticos) .
El polimorfismo suele requerir el uso de un (CASTING) de tipos, salvo que se use programación <GENÉRICA> o "de tipo".

*** LA P.O.O NOS PERMITE DISEÑAR PROGRAMAS ROBUSTOS, ESCALABLES, REUTILIZABLES, SEGUROS ... etc ***

1-01 CLASES EN JAVA.

Una clase se puede entender como una estructura que sirve para representar un determinado concepto o idea abstracta (ej: un gato).

La definición de una clase sigue la forma básica:

public / "default" **class** Nombre **extends** ClaseMadre **implements** Interface1, Interface2... {}

Observar como sólo se definen 2 niveles posibles de encapsulación: public y "default". Estos constituyen la primera llave de acceso a los miembros de la clase. Si no se supera dicho primer nivel, no podremos acceder a la clase ni a ninguno de sus miembros, por muy públicos que se quieran definir. Tampoco se podrá realizar la importación de la clase si la encapsulación no lo autoriza.

Podemos añadir otros modificadores como:

abstract	...	para indicar que es una clase ABSTRACTA. Punto inicial de jerarquía de clases (supertipo). No permite instanciación.
final	...	para indicar que no puede heredarse de dicha clase. Es el punto final de la jerarquía de clases (String es final).

Podemos definir 2 o más clases en el mismo fichero .java, pero sólo puede haber una clase PÚBLICA, que será la que dé nombre al fichero. Sin embargo, por cada clase encontrada se generará de forma independiente un fichero que represente su .class (ésto es lo que realmente necesita la JVM).

Una clase está formada por lo que se denomina como **MIEMBROS DE LA CLASE**:

ATRIBUTOS	**...**	**define características de la clase: color del pelo, raza, edad ... etc**
CONSTRUCTORES	**...**	**las formas de poder crear las instancias de la clase en base a argumentos de entrada.**
MÉTODOS	**...**	**para definir el comportamiento, lo que puede hacer dicha clase.**
		Si devuelven algo, se utiliza return.

... y tipos enumerados, clases internas, bloques de inicialización estáticos / no estáticos ... etc.

Toda clase viene determinada por lo que se denomina como FULLY-QUALIFED-NAME (nombre totalmente cualificado de la clase), que consiste en indicar la ruta de paquetes (separados por punto), acabando con el nombre de la clase. Esto genera los espacios de nombres, es decir, que dos o más clases que se llamen igual, serán distintas si pertenecen a rutas de paquetes distintas ... ejemplo java.lang.String --- misclases.String.

El CLASS-LOADER se ocupa de cargar en la memoria física de la máquina, el código de la clase que queremos utilizar. Se hace 1 vez y a partir de ahí, generamos los objetos. Se lee el .class del disco duro (habitualmente) y se carga en memoria. En este punto se declaran e inicializan las variables de tipo estático (aquellas que no pertenecen a ningún objeto).

RECORDAR:

TODOS LOS MIEMBROS DE LA CLASE ADMITEN LOS 4 MODIFICADORES DE ENCAPSULACIÓN.

NOCIONES UML (Unified Modeling Language).

► Descripción aislada de la clase. Es lo que se conoce como modelado estático de clases. Muestra atributos y métodos, encapsulación ... etc.

► Cardinalidad : **[0..1] - [1] - [1..*] - [*] - [N..M] - [N]**. Cuántos objetos de una clase se relacionan con cuántos de otra.

► Composición (composición fuerte) Rombo relleno desde la clase contenedora hacia la clase contenida (si la cardinalidad es 1, no poner nada)
Elementos contenidos no compartidos, al morir el contenedor, mueren también los elementos contenidos.
Mano – (5)Dedo. En el código de la clase aparecerán como un array de 5 objetos dedo.

► Agregación (composición débil) Rombo vacío desde la clase contenedora hacia la clase contenida.
Elementos contenidos se pueden compartir y no mueren al desaparecer el contenedor.
Mano – Guante. En el código de la clase aparecerá como una variable de tipo Guante.

1-02 MODIFICADORES DE ENCAPSULACIÓN.

Se emplean para ocultar o prohibir el acceso a los miembros de la clase. Se establece un nivel de privacidad con las palabras reservadas:

DE MÁS OCULTO ---> A MÁS VISIBLE:

private ---> "default" ---> protected ---> public .

private = Sólo visible por la propia clase.

"default" = Visible por la propia clase y por clases que pertenezcan a su mismo paquete.

protected = "Un private que se hereda". Se trata de que la hija use algo que le da la madre pero a través de una instancia de la hija, no de la madre.

Si mismo paquete que la madre:
Visible por la propia clase madre y el resto de clases del paquete, sean estas o no hijas (se entra por encapsulación default, que es más restrictiva que protected).

Si distinto paquete que la madre:
Visible por la propia clase madre y las clases hijas de ésta, pero a través de una instancia de la clase hija, nunca de la clase madre. El resto de clases del paquete no tendrán acceso a dicho atributo o método. **Se mira EN QUÉ PAQUETE se definió el atributo que ahora se está utilizando por herencia gracias a protected**. Si el paquete actual no es el mismo que aquel en el que se definió el atributo o método y la clase no es heredera, no se permite el acceso. Si es heredera, sólo se permite el acceso a su propia línea de herencia (no hermanas) y siempre a través de un objeto de la clase hija (en este caso, una hija de la hija).

Así, si es distinto paquete, default ya no es válido puesto que:
p0 (paquete de definición) != p1 (paquete de uso).

public = Visible por todas las clases de cualquier paquete.

Siempre habremos de responder a un par de preguntas, antes de determinar si algo es o no visible (accesible) por o desde otro elemento:

¿quién? ... De qué tipo es la clase desde la que tratamos de utilizar un determinado elemento de otra clase (puede ser la misma clase).

¿dónde? ... Cuáles son los paquetes, tanto para la clase que declaró originalmente el elemento al que se intenta acceder, como para la clase que trata de acceder a dicho elemento.

Prestar especial atención al nivel protegido (es el más complejo):

El nivel de encapsulación PROTEGIDO (protected) hace referencia a algo que se va a poder utilizar en un clase heredera y subclases de ésta, gracias precisamente a que se obtiene por herencia, sin importar el paquete en el que esté declarada dicha clase.

Las clases herederas hacen uso de un atributo o método protected a través de una instancia de las mismas, Y NUNCA A TRAVÉS DE UNA INSTANCIA DE LA MADRE.

Los constructores NO SE HEREDAN, luego no van a formar parte de la clase hija como podrían ser los métodos y atributos heredados. Podremos sin embargo, invocar a un constructor protegido desde un constructor de la clase hija mediante super(xxx), para completar la cadena de jerarquía y que dicha clase hija compile sin problemas, pero no para instanciar objetos de la madre desde la hija si ambas estan en paquetes distintos (no tenemos en este caso los privilegios de un nivel de encapsulación "default").

En este punto conviene aclarar que cuando tanto madre como hija están en el mismo paquete, la hija es capaz de acceder a los atributos y métodos (además de los propios constructores) de tipo protegido en la madre. Esto es así porque el nivel que se está utilizando para conseguir el permiso de acceso es DE PAQUETE, también llamado "default" y no como cabría esperar, protected. De hecho, en la situación contraria, es decir, cuando madre e hija están en distintos paquetes, NO PUEDE ACCEDERSE a los atributos o métodos (ni a los constructores) protegidos de la clase madre a través de una instancia de dicha clase madre, sólo se podrá utilizando la propia clase hija (no podremos utilizar los constructores protegidos de la madre para crear instancias de esta, pero sí invocarlos mediante el super(xxx) correspondiente para conseguir inicialización de atributos que se van a heredar y el encadenamiento, ya mencionado, hacia Object).

En el caso de que una clase hija, definida en un paquete distinto al de la madre, reciba un atributo protected creado por su clase madre, el resto de clases que pertenezcan al mismo paquete en el que se encuentre dicha clase hija NO TENDRÁN ACCESO al atributo heredado por ésta, ni tan siquiera siendo dichas clases también hijas de la misma clase madre (las que podríamos llamar hermanas).

Así, para ver si el resto de clases del paquete al que pertenece la clase heredera tienen o no acceso al atributo protected heredado por ésta, hay que comprobar:

¿PAQUETE DE DEFINICIÓN DEL ATRIBUTO == PAQUETE ACTUAL INTENTANDO USAR ATRIBUTO PROTEGIDO DE OTRA CLASE?

► Si la comprobación es correcta, por "default" nos permitirá el acceso a dicho atributo, método o constructor aunque no seamos clase heredera.

► Si no es el mismo paquete, no podremos acceder al atributo, método o constructor a menos que seamos clase hija y a través de una instancia de ésta.

También es posible utilizar static conjuntamente con protected, en cuyo caso, sólo se tendría acceso al atributo protegido y estático (el mismo para todos) a través de una instancia de la hija (no es adecuado hacerlo así) o indicando el nombre de la clase, punto y el nombre del elemento estático. Aquellas clases no herederas, que estén fuera del paquete de definición del elemento, no podrán invocar el elemento estático ni a través de un objeto hija ni indicando directamente el nombre de la clase (salvo que dicha clase intrusa estuviese en el paquete de definición original de dicho atributo) ... en definitiva: sólo dentro del paquete original.

Si es distinto paquete, lo primero que debemos hacer es realizar la importación de clases, siempre y cuando la encapsulación de la clase a importar lo permita.

05 - Tabla Modificadores de encapsulación en Java

1-03 MODIFICADOR static.

Cuando se quiere declarar que algo pertenece, que "vive" en la clase, se emplea el modificador static (estático).

Algo de tipo estático **no requiere la instanciación de un objeto para poder ser usado**. Simplemente se indica el nombre de la clase seguido de un punto y el nombre de la variable o método estático que se quiere emplear. Aunque también se puede acceder mediante una instancia de la clase, no es la forma más adecuada (de hecho, NetBeans nos dará un warning y el .class contendrá la referencia a la clase, no a la instancia).

Un elemento estático no ve, no accede a aquellos elementos que no sean estáticos (no puede saber sobre cuál de los 0 ó infinitos objetos que se puedan crear de la clase, quiere invocarse el método o accederse al atributo en cuestión). Sin embargo, algo no estático sí puede ver a lo estático siempre y cuando el nivel de encapsulación empleado lo permita, ya que es único y el mismo para todos.

El primer momento en el que se ejecuta el programa, ningún objeto existe, luego no hay realmente un punto de entrada. Se define el main como estático para salvaguardar esta circunstancia y poder empezar a crear los objetos que necesita la aplicación para funcionar.

En el caso de un atributo estático, las modificaciones que se realicen sobre dicho atributo afectarán por igual a todos aquellos elementos que lo utilicen.

En el caso de un método, no se permite que cada objeto redefina su comportamiento (ha de ser el mismo para todos), luego NO SE PUEDE SOBRESCRIBIR. Intentar usar @Override con un método estático genera un error de compilación. Sin embargo, sí se permite que cada clase pueda definir sus propios métodos estáticos sin importar los que pueda tener su clase madre o el resto declases (la ambigüedad se resuelve utlizando la forma ya conocida de nombreClase.miembroEstático).

Trabajemos con atributos o métodos, estáticos o no, recordar que el nivel de encapsulación con el que se haya declarado el elemento, va a ser siempre el que permita o no su utilización.

Un constructor nunca jamás puede definirse como estático.

Los modificadores this y super no son válidos dentro de un contexto estático, ya que ambos hacen referencia a la instancia (al futuro objeto/s que se creará del molde que es la clase), y no a la propia clase (el concepto a partir del cuál escribimos su código). Recordar que un atributo o método estático es único y el mismo para todos, mientras que objetos de una clase pueden crearse todos los que se quiera.

Un atributo marcado como estático no forma parte del estado de un determinado objeto (simplemente se cumple que el objeto puede verlo). Así, dicho atributo estático no se serializará, algo que tampoco podremos hacer con aquellos declarados como transient (se evita la serialización por motivos de seguridad).

A pesar de que no se admite sobrescritura de un método estático (@Override) y de que la hija puede declarar su propio método estático, (sin el @Override) este ha de cumplir en su declaración con las reglas de la sobrescritura.

Por último, debemos vigilar la encapsulación del método estático (como miembro de la clase que es admite los 4 niveles).

1-04 ATRIBUTOS DE LA CLASE. AFIRMACIÓN IS–A vs HAS–A.

Reciben diversos nombres:

+ atributos
+ variables de instancia (sólo si no son estáticos).
+ variables miembro
+ campos de la clase (sólo si son estáticos).
+ variables globales ... etc

Todos ellos utilizados para definir las características o propiedades de la clase. Ej: Gato ... colorPelo, raza, edad, numeroPatas ... etc.

Cada objeto o instancia creada a partir de dicha clase va a tener su propio conjunto (o copia) de los atributos de la clase, siendo totalmente independientes unos objetos de otros.

Los atributos pueden declararse, además de con los 4 niveles de encapsulación (como miembros de la clase que son), con:

+ final --> no cambiar valor (tipo primitivo) o no desvincular objeto, pero si actuar sobre él (tipo referenciado)
+ static --> es el mismo para todos, luego los cambios que haga uno los verán los demás.
+ transient --> no formará parte de la serialización (a voluntad del programador y como medida de seguridad).
+ volatile --> elemento compartido que debe actualizar frecuentemente su valor.

Las clases se pueden componer a partir de objetos de otras clases: Persona -->Mano -->Dedo ...Guante.

Así, una afirmación **HAS – A** se da cuando tenemos una relación de COMPOSICIÓN (fuerte o débil) entre dos elementos, de forma que uno de ellos, el elemento conocido como "contenedor", dispone en su interior, en forma de atributos de su clase, de objetos del otro elemento "los contenidos".

A modo de ejemplo, disponemos de la clase Botella y las clases Tapon, Etiqueta y Liquido, pudiendose afirmar que:

Botella	tiene un	Tapon.
Botella	tiene una	Etiqueta.
Botella	tiene/contiene un	Liquido.

Una afirmación **IS – A** se da cuando tenemos una relación de HERENCIA o de IMPLEMENTACIÓN entre dos elementos, de forma que aquel de los dos que herede o implemente al otro, de una forma directa o indirecta, se puede afirmar que además de ser el mismo, es también el otro elemento. La relación es siempre de caracter unidireccional, desde la clase hija hacia la clase madre (o la interfaz).

Por ejemplo, dada la clase Gato que hereda de la clase Animal y esta a su vez, implementa la interfaz SerVivo, podemos afirmar que:

Gato	...	es un Gato, es un Animal y es un SerVivo. Es además Object.
Animal	...	es un Animal y es un SerVivo. Es además Object.
SerVivo	...	es un SerVivo. NO ES Object.

Así, para el primer ejemplo, sería erróneo decir que Botella es un Tapon, una Etiqueta o un Liquido, algo que indicaría una relación IS – A. Sí podríamos afirmar, en el caso de que la clase Botella heredase de la clase Envase, que Botella es un Envase.

05A CONSTRUCTORES PARA INSTANCIAR LA CLASE. OBTENCIÓN DE OBJETOS.

Para poder instanciar, crear un objeto de la clase, hemos de utilizar (de forma directa o indirecta .. a través de un método), un constructor:

El constructor es similar a un método, pero a diferencia de éste, NUNCA VA A DEVOLVER NADA, NI SIQUIERA void.
Ha de llamarse como la clase, puede tener cualquiera de los 4 niveles de encapsulación y no es accesible por herencia (sí mediante encadenamiento).

La misión de un constructor es la de:

1. Reservar memoria de la máquina física para almacenar el objeto a crear.
2. Inicializar los atributos de dicho objeto, bien a su valor por defecto, o al indicado por su línea de argumentos de entrada.
3. Devolver una referencia (manejador o "handler") para poder localizar y usar dicho objeto en el HEAP de memoria.

Podemos tener distintos constructores que se diferencien entre sí a través de la línea de parámetros de entrada, lo que da lugar a lo que se denomina como SOBRECARGA DE CONSTRUCTORES (diversas formas de crear objetos de una clase).

Desde un constructor de la clase, podemos llamar a otros constructores de dicha clase o bien a constructores de la clase madre. Para ello se utiliza:

super(firma_parámetros)	---> llama al constructor de la clase madre que tenga dicha firma de parámetros.
this(firma_parámetros)	---> llama al constructor de la propia clase que tenga dicha firma de parámetros.

SÓLO PUEDE APARECER UNA DE LAS DOS (super(xxx) o this(xxx)) Y HA DE ESTAR EN LA PRIMERA LÍNEA DEL CÓDIGO DEL CONSTRUCTOR. NO CONFUNDIRLO CON super.atributo/método o this.atributo/método, que llevan **.** (punto), no **(**paréntesis**)**.

El proceso de instanciación es el siguiente:

!!! hay que ver que el constructor EXISTA y que su NIVEL DE ENCAPSULACIÓN nos permita utilizarle !!!

1 Declarar la varible de referencia o manejador a la clase.

Gato g;

2 Llamar al constructor (el que nos interese de los disponibles) y asignar el valor devuelto por este a nuestra referencia.

g = new Gato();

Todo en una línea (es la forma más habitual):

Gato g = new Gato();

OBSERVAR CÓMO PARA LLAMAR A UN CONSTRUCTOR DESDE UN MÉTODO SE UTILIZA LA PALABRA RESERVADA **new**.
NO PODEMOS UTILIZAR this(xxx) o super(xxx) DENTRO DE UN MÉTODO.

SIN EMBARGO, PARA LLAMAR A UN CONSTRUCTOR (PROPIO O DE LA MADRE), DESDE EL INTERIOR DE OTRO CONSTRUCTOR PODEMOS UTILIZAR TANTO new COMO this(xxx) o super(xxx).

NUNCA SE PUEDE INVOCAR UN CONSTRUCTOR POR SU NOMBRE SI NO UTILIZAMOS new. ASÍ, PARA INVOCARLE, O BIEN USAMOS new O BIEN this(xxx) o super(xxx) PERO SÓLO DESDE DENTRO DE OTRO CONSTRUCTOR.
Ej: en el interior de un constructor podemos hacer: new Gato(), this(), super("Garfield) ... pero no así Gato(), ya que genera error.

UN CONSTRUCTOR PUEDE LLAMARSE A SÍ MISMO (RECURSIVIDAD), PERO SI ESTA ES INFINITA, OBTENDREMOS UNA EXCEPCIÓN (ERROR EN TIEMPO DE EJECUCIÓN).

COMO REGLA A SEGUIR, LOS ATRIBUTOS (campos de la clase, variables miembro, variables de instancia ... como queramos llamarlos), SE DEFINEN CON UN NIVEL DE ENCAPSULACIÓN **private**, Y SIN INICIALIZAR. ESTAS VARIABLES SE CONOCEN CON EL NOMBRE DE **BLANKED** (EN BLANCO). ES EN EL CONSTRUCTOR DONDE SE INICIALIZAN LAS MISMAS CON LOS VALORES RECIBIDOS EN LA INVOCACIÓN A DICHO CONSTRUCTOR.

SI LA VARIABLE ES static final, HABREMOS DE DARLE VALOR EN LA PROPIA DECLARACIÓN O EN UN BLOQUE DE INICIALIZACIÓN ESTÁTICO ... static{}

```
private String nombre;

private int edad;

public Gato(String nombre, int edad) {
        this.nombre = nombre;
        this.edad   = edad;
}
```

UNA VEZ OBTENIDA LA REFERENCIA AL OBJETO CREADO, PODEMOS ASIGNAR DICHA REFERENCIA A OTRAS REFERENCIAS DEL MISMO TIPO O SUPERTIPO, DE FORMA, QUE TENDREMOS DISTINTOS MANEJADORES ACTUANDO SOBRE EL MISMO OBJETO ... ALGO QUE SUCEDE POR EJEMPLO, EN EL PASO DE OBJETOS COMO PARÁMETROS DE UN MÉTODO.

ES EL MANEJADOR, LO QUE CONOZCA DE LA CLASE, EL QUE NOS VA A PERMITIR USAR UNA SERIE DE ATRIBUTOS O MÉTODOS ... aquí entra en juego el polimorfismo y cómo manejar objetos de una determinada clase con referencias de otras, de jerarquía superior.

Animal a = new Gato();

05B EL CONSTRUCTOR IMPLÍCITO Y EL super() IMPLÍCITO.

Toda clase, para poderse instanciar, NECESITA POR LO MENOS 1 CONSTRUCTOR.

El hecho de no tener que escribir ningún constructor a la hora de crear nuestra clase, NO SIGNIFICA, que el constructor no esté. De hecho, el compilador, a la hora de generar el .class a partir del .java, COLOCA AUTOMÁTICAMENTE, las líneas de código correspondientes al denominado CONSTRUCTOR IMPLÍCITO, entre otras cosas.

El constructor implícito tiene el nivel de encapsulación de la clase a la que pertenece (public o "default" para clases no internas) Y NO TIENE PARÁMETROS DE ENTRADA. Se le denomina implícito porque lo escribe el sistema por nosotros. También podríamos optar por escribirlo, pero en dicho caso YA NO ES EL IMPLÍCITO, porque lo estamos definiendo nosotros a mano, explícitamente. Así, sería mas correcto llamarlo constructor por defecto. Ej: public Gato() {}

En el momento en el que escribamos un constructor, el compilador YA NO NOS PONDRÁ EL IMPLÍCITO, luego hay que vigilar que no se de el caso de que algún otro elemento o parte del programa haga uso de dicho constructor que ya no tenemos de forma automática. De ser así, tendremos que escribirlo a mano o cambiar la forma en la que los otros constructores llaman a esta clase (herencia – invocación super(xxx)).

Dentro de todo constructor, sea implícito o no y siempre y cuando no lo pongamos explícitamente, tendremos una llamada super(); implícita que tratará de invocar al constructor de la clase madre. Puede darse el caso de que dicho constructor en la madre no exista, con lo que obtendremos un error de compilación al quedar rota la cadena de constructores.

En el caso de realizarse un llamamiento explícito a un constructor de la madre o a uno de la propia clase mediante super(xxx) o this(xxx) respectivamente, éste deberá ir colocado obligatoriamente en primera línea y deberá ser único (no podemos combinar super con this ya que sólo lo podremos poner 1 vez en el cuerpo de cada constructor).

implícito:

```
public ó "default"  Gato() {
                        super();
}
```

05C ENCADENAMIENTO DE CONSTRUCTORES. ELEMENTOS DE LA CLASE MADRE.

Como ya sabemos, toda clase Java hereda de Object, directa o indirectamente, luego SIEMPRE HAY HERENCIA (salvo con la propia Object).

Así, SIEMPRE HAY UNA LLAMADA A UN CONSTRUCTOR DE LA CLASE MADRE, que a su vez llamará a su propia clase madre y así hasta llegar al final de la cadena de jerarquía que es Object.

El motivo por el que sucede ésto es porque la clase heredera o HIJA, necesita de aquellos elementos presentes en la clase madre, SEAN ESTOS O NO ACCESIBLES DIRECTAMENTE DESDE DICHA CLASE HIJA. Es necesario hacer la siguiente aclaración:

¿HEREDADOS Y NO HEREDADOS (ANEXADOS O DIRECTAMENTE NO ACCESIBLES DESDE LA HIJA)?

Se dice que los atributos privados y constructores NO SE HEREDAN en Java. Esta afirmación es CORRECTA, pero hay que entenderla como que no pueden ser utilizados directamente por la clase heredera, es decir, QUE NO SON DIRECTAMENTE ACCESIBLES.
Sin embargo, SÍ ESTÁN PRESENTES O ANEXADOS A LA CLASE HIJA PARA QUE ÉSTA, DE LA FORMA ADECUADA, PUEDA UTILIZARLOS.

¿CÓMO?:

atributos privados de la madre ---> mediante métodos públicos/protegidos/"de paquete" de la madre que pueda usar la hija
(Un miembro público puede acceder a otro miembro protegido de su misma clase).

constructores ---> mediante super(xxx) (salvo que sean private) o bien mediante métodos publicos/protegidos/"de paquete" a usar por la hija Un constructor protected de la madre, puede ser invocado desde la hija, sin importar el paquete al que esta pertenezca.
Un método puede invocar a un constructor usando new().

---> podemos igualmente utilizar otros constructores de la madre a los que sí tengamos acceso para a continuación, emplear this(xxx) desde éstos y llamar a los privados de su propia clase.

ASÍ, EL ORDEN DE LLAMAMIENTO A LOS CONSTRUCTORES ES EL QUE SE ESTABLEZCA EN LA JERARQUÍA DE CLASES, EMPEZANDO SIEMPRE POR LA QUE ESTÉ MÁS ARRIBA, ES DECIR, Object Y DE AHÍ HACIA ABAJO

HASTA LLEGAR A CLASE QUE ESTAMOS DEFINIENDO. En realidad entramos por el invocado en la hija, pero una vez dentro de éste, LA PRIMERA LÍNEA QUE SE EJECUTA ES ESE super(xxx) o this(xxx), LO QUE NOS ACABA LLEVANDO SIEMPRE AL CONSTRUCTOR DE Object, ejecutándose en primer las líneas de código que pertenezcan a él.

Podemos utilizar el constructor de la clase madre, desde la clase hija, para decidir los valores de los atributos que van a heredar las clases hijas, gracias al uso de éste por parte de dichas clases hijas. Es la forma de inicializar valores en una clase abstracta.

05D SOBRECARGA DE CONSTRUCTORES.

En una clase podemos tener 0 (realmente es 1 a causa del constructor implícito) o infinitos constructores, para poder instanciar los objetos de diversas formas en base a LA FIRMA DE ARGUMENTOS DE ENTRADA.

El declarar un constructor requiere, como ya sabemos, que dicho constructor se llame como la clase y que no devuelva absolutamente nada.

Para hablar de sobrecarga, utilizaremos por tanto el nombre de la clase y DISTINTAS FIRMAS DE ARGUMENTOS DE ENTRADA. El hecho de usar uno u otro constructor se decidirá en base a la firma que en cada caso estemos utilizando.

Se dice que dos firmas son distintas entre si, en base a (el nombre los parámetros no es relevante):

Número de parámetros.
Tipo de dato de cada parámetro.
Posición de cada uno de ellos en la firma.

Como sobrecarga que es, cada constructor podrá definir su nivel de encapsulación de entre los 4 posibles e igualmente el número y tipo de excepciones que pueda lanzar, independientemente del resto. No hay ninguna regla a respetar (como ocurre en la sobrescritura de métodos), salvo la de no repetir la línea de argumentos entre constructores sobrecargados.

Un constructor puede llamar a otro de la propia clase mediante el uso de this(xxx). Recordar que salvo que utilizemos new, nunca deberemos usar el nombre de la clase para invocar al constructor.

No tiene sentido hablar de sobrescritura de constructores puesto que los constructores no se heredan.

06 EL HEAP Y EL STACK – ALMACENAMIENTO EN MEMORIA.

¿Dónde se guardan los objetos y las variables con las que se llaman a los métodos?

Son dos zonas de memoria gestionada por la máquina virtual (JVM). En el **HEAP** se almacenan todos los objetos, mientras que en el **STACK**, que tiene funcionamiento LIFO (Last In Firt Out) se van a almacenar las variables que intervienen en el llamamiento a métodos, siendo estas destruidas al finalizar dicho método y las referencias a objetos declaradas localmente (no importa si el objeto está creado localmente, éste siempre se almacena en el Heap).

Los objetos siempre se crean en el Heap. Los variables de instancia se guardan también en el Heap.

Dado que Java NO OFRECE AL PROGRAMADOR EL PODER TRABAJAR CON ARITMÉTICA DE PUNTEROS (manejar la memoria física de la máquina directamente, aunque internamente la arquitectura sí los usa), DEBEMOS EMPLEAR LAS REFERENCIAS (MANEJADORES O HANDLERS), PARA PODER UTILIZAR LOS OBJETOS ALMACENADOS EN EL HEAP.

El hecho de anular todas y cada una de las referencias a un objeto hace que éste quede elegido como candidato a recolección (anulación referencia, cambio de la referencia a otro objeto o islas de aislamiento).

Una colección o un array almacena una referencia (handler) a un objeto del Heap. Mientras dicho objeto colección o array exista, el objeto apuntado núnca será candidato a recolección.

07 EL GARBAGE COLLECTOR – HILO PARA LIMPIEZA DE LA MEMORIA.

De la misma forma que no utilizamos punteros, cuya gran ventaja es el no tenernos que preocupar por reservar memoria física de la máquina (como por ejemplo en C), tampoco nos tenemos que preocupar de liberar los recursos asociados a un objeto una vez que éste ya no nos sea útil.

Para ello, Java proporciona su Colector de Basura (GC) cuyo funcionamiento particular depende de la versión que estemos utilizando de la máquina virtual y que se va a encargar automáticamente de liberar los recursos asociados a aquellos objetos que ya no resulten necesarios.

El GC es al fin y al cabo un HILO, lanzado por la JVM que limpia la memoria física del sistema. Podemos invocar al GC, pero no se asegura el momento exacto en el que este va a intervenir (método **System.gc()**) ya que el hecho de realizar limpieza puede consumir por si mismo una gran cantidad de recursos, ralentizar el sistema ... en definitiva, puede no merecer la pena. Lo que si nos aseguran es que “el basurero pasará”, pero sin especificar exactamente el momento.

Otra forma de invocar al colector es mediante: **Runtime.getRuntime().gc()**

Todo objeto tiene el método heredado finalize(), el cual será invocado justo antes de eliminar el objeto (como si fuese ”su última voluntad”).

Se dice que un objeto es candidato a recolección (eliminación) cuando no hay ninguna referencia que apunte hacia él ... es decir, no tiene ningún manejador asociado (“está huerfano”).

Recordar como una variable definida dentro de un bloque de código (método o constructor), existe únicamente dentro de dicho bloque ... salvo que sea envíada a otro lugar, como sucede al utilizar la sentencia return.

SI UNA VARIABLE ES UTILIZADA ÚNICAMENTE EN EL INTERIOR DEL BLOQUE EN EL QUE SE HA CREADO, DICHA VARIABLE SERÁ DESTRUIDA (O ETIQUETADA COMO CANDIDATA A RECOLECCIÓN) AL FINALIZAR EL CITADO BLOQUE.

08 MÉTODOS / FUNCIONALIDAD DE LAS CLASES.

Como tercer grupo principal de elementos que podemos encontrar dentro de una clase (podemos ver a esta como una bolsa o contenedor de elementos), tenemos **los métodos**. Los métodos aportan funcionalidad a la clase, es decir, establecen aquello que la clase es capaz de hacer.

La declaración de un método sigue la forma:

encapsulación tipo_devuelto nombreMetodo (parámetros_entrada) throws Excepcion1,Excepcion2 ... { código }

public void comer(String tipoComida, float cantidad) { ... }

Los métodos han de indicar siempre qué devuelven o un tipo de dato (primitivo o referenciado) o nada, en cuyo caso se pone void.

Así, respecto a la declaración de qué devuelve un método, podemos tener:

+ void.
+ un tipo primitivo.
+ un tipo referenciado (más tarde, el return del método podrá devolver **null**).

La devolución se realiza mediante la palabra reservada **return** y ésta ha de colocarse al final del código del método. En el caso de colocarla dentro de bloques de código (ejemplo if – else), cada bloque deberá tener su propio return u obtendremos un error de compilación.

Podemos utilizar el nombre de la clase para el método, PERO NO ES LO ACONSEJABLE. La forma de ver que no se trata de un constructor es fijándonos en si hay un tipo devuelto, en cuyo caso tendremos un método (no hay error de compilación, la circunstancia es válida).

El hecho de que el método nos devuelva algo, no nos obliga a tener que capturarlo en el código que lo llamó.

Los modificadores a emplear, a parte de los 4 niveles de encapsulación como miembros de la clase que son:

+ final --> no se puede sobrescribir.
+ abstract --> es método abstracto (está declarado ; no definido {}).
+ static --> pertenece a la clase, no necesita de instanciación para usarlo. No se puede sobrescribir.

SOBRECARGA (ES NECESARIO TENER 2 O MÁS MÉTODOS CON EL MISMO NOMBRE ACCESIBLES EN LA CLASE**):**

Podemos utilizar el mismo nombre para crear distintos métodos, EN BASE A LA LÍNEA DE PARÁMETROS. Podría darse el caso de que en función de lo que le pasemos al método, esté deba actuar de una u otra forma ... como ocurría con los constructores.

* Los distintos métodos sobrecargados, entre sí, pueden tener:
  Distinto nivel de encapsulación, distinto tipo de dato devuelto, distinto número y tipo de excepciones que pueden lanzar, ya sean CHECKED o NON CHECKED.

SOBRESCRITURA (ES NECESARIO QUE HAYA HERENCIA DE POR MEDIO**):**

Cuando redefinimos el código o comportamiento del método que estamos heredando de la clase madre.

Podemos utilizar la notación (Java 5) **@Override** para que el compilador verifique que se da realmente la sobrescritura.

UN MÉTODO DECLARADO CON **final** NO SE PUEDE SOBRESCRIBIR.
UN MÉTODO ENCAPSULADO CON **private** NO SE HEREDA LUEGO NO CABE HABLAR DE SOBRESCRITURA.

Para poder elegir entre el método heredado y el sobrescrito, habremos de utilizar super (usa el heredado) o this (usa el sobrescrito).
Para llamar al sobrescrito, es decir el que se define en la clase hija, también es válido no indicar this.

* El método sobrescrito, con respecto al heredado de la madre ha de tener:
  Nivel de encapsulación igual o más visible, mismo tipo de dato devuelto o covariante (subtipo), igual o menor número de excepciones a lanzar DE TIPO CHECKED (vigilar la granularidad – jerarquía de las mismas). Respecto a las de tipo NON CHECKED, no hay ninguna regla a seguir.

* Un método de tipo static NO SE PUEDE SOBRESCRIBIR, ya que es compartido por todas las instancias, todas deben ver los mismo, luego no se admite que cada una redefina a su antojo.

Es factible el hecho de que se dé sobrecarga y sobrescritura al mismo tiempo, YA QUE PODEMOS SOBRECARGAR UN MÉTODO HEREDADO, CON UNO DEFINIDO EN LA PROPIA CLASE HIJA O BIEN, PODEMOS HEREDAR 2 O MÁS MÉTODOS DE LA MADRE QUE DE POR SI SE SOBRECARGUEN MUTUAMENTE, AL MARGEN DE LOS QUE PODAMOS CREAR EN LA CLASE HIJA.

¿Cómo distingue Java un método de otro?:
Distingue por: nombre y línea de parámetros
No distingue por: nivel encapsulación – tipo de dato devuelto – excepciones lanzadas (checked y non checked).

Tabla comparativa sobrecarga VS sobrescritura.

09 PASO DE PARÁMETROS A UN MÉTODO.

RECORDAR:

SIEMPRE ES PASO POR VALOR, NUNCA POR REFERENCIA:

En Java, los argumentos (la información enviada) a un método y que es recogida por éste a través de los parámetros de entrada, siempre son copias de los valores originales enviados, sean estos tipos primitivos o referenciados.

La referencia a un objeto, contiene un valor o que nos permite llegar a dicho objeto en el Heap. Al pasar dicha referencia a un método lo que estamos haciendo es pasar una copia de dicho valor, de manera y modo que SÍ vamos a poder manejar dicho objeto de igual forma que si lo hiciésemos con la referencia original. El hecho de anular la referencia (es una copia) en el método, NO DESVINCULA A LA ORIGINAL del objeto en cuestión. ESTO JUSTIFICA EL HECHO DE DECIR QUE **TODO SE PASA POR VALOR** (en .Net, a diferencia de Java hay palabras del sistema byRef // byVal que especifican el comportamiento a seguir).

Recordar que los String y los Wrappers son INMUTABLES, es decir, no se modifica el valor de el objeto sino que se crea uno nuevo y se hace apuntar la referencia a dicho nuevo valor. Así, si pasamos un String o un Wrapper a un método y tratamos de modificar su valor, al finalizarse el método, la referencia original seguirá apuntando a su objeto, que por ser inmutable, no reflejará los cambios que en un principio parecía que se habían producido dentro de dicho método.

Podemos tener diversos tipos de datos en la línea de parámetros de entrada, debiéndose respetar el orden y tipo en la invocación al método. De hecho, es dicha línea la que determina el método a llamar en el caso de tener sobrecarga. Si no existe ningún método que corresponda con el invocado obtendremos un error de compilación.

```
public int sumar (int a, int b, float c) { ... }
```

Parámetros finales:

Los parámetros pueden declararse como final dentro del método. Así, si es primitivo no se podrá cambiar su valor y si es referenciado no se podrá desvincular la referencia cargada en el parámetro del método, respecto del objeto que se le asocie (aunque sí cambiar el estado de éste). No obstante, en el momento en el que el método finaliza, dicha variable se destruye produciéndose la desvinculación automática (es por código cuándo no podremos desenganchar la variable del objeto, una vez asignado).

Parámetros variables:

Java 5 introdujo el parámetro de entrada variable, que nos permite no tener que especificar el número de argumentos de un determinado tipo con el que se puede llamar al método.

Es necesario que dicho parámetro (tipo ... nombre), aparezca el último en la línea de parámetros y que sólo exista 1 variable. Tendremos un ARRAY CON LOS DATOS DE ENTRADA, que tiene el nombre especificado en la declaración y cuyos elementos son del tipo (subtipo) indicado.

```
metodo (   ,   ,    , tipo ... nombre) { }

public int sumar(String operacion, int ... datos) {}
```

Puede darse el caso de que no se reciba ningún parámetro variable y que por tanto, dicho método coincida exactamente con otro de los que constituyan la sobrecarga. Así, ¿cuál es el método elegido por el sistema de entre los dos posibles?:
SIEMPRE SE ELIGE EL MÉTODO QUE NO TIENE PARÁMETROS VARIABLES.

```
m1(int) – m1(Integer) – m1(Number) – m1(Object) – m1(int...n).
```

10 INTERFACES – SUPERTIPO 1 – IMPLEMENTACIÓN.

Representan **el concepto más puro de abstracción**, ya que NO DEFINEN ningún comportamiento (código del cuerpo del método), sino que sencillamente, se limitan A DECLARARLO (su nivel de encapsulación, qué devuelve, cómo se llama, qué línea de parámetros tiene, qué excepciones puede lanzar o propagar).

En las interfaces todos los métodos son de tipo **public abstract**, aunque no se indique explícitamente. Esto quiere decir que el cuerpo del método, lo que va a realizar, no está definido y su encapsulación es pública.

De igual forma, no se definen atributos sino constantes, que obligatoriamente son de tipo **public static final**. Es habitual el uso de una interfaz para definir constantes que luego serán utilizadas en otros puntos del programa. Ej: Calculo.PI ---> 3.1415... .

La clase que implementa una determinada interfaz, tiene acceso directo (por clase o instancia de esta), a las posibles constantes definidas en dicha interfaz.

La interfaz establece un CONTRATO (algo que obliga), con aquella clase o clases que la implemente, de forma que estas han de definir, dar código a todos y cada uno de los métodos declarados dentro de la interfaz. En el caso de que la clase deje sin definir 1 o más de dichos métodos, se obliga a declarar dicha clase como abstracta. De no hacerse esto último, el compilador generará un error.

Una interfaz puede HEREDAR de otra/s interfaz, es decir adoptar la declaración de los métodos de aquella que pretende extender. Lo que no puede es IMPLEMENTAR a otra interfaz porque la implementación conlleva la definición (no confundir con declaración) de todos los métodos de dicha interfaz ... y una interfaz como ya sabemos, sólo declara , no define.

Cuando una interfaz hereda de múltiples interfaces, no existe ningún problema de **COLISIÓN DE MÉTODOS**, ya que ninguno de ellos está definido. En Java no disponemos del concepto de HERENCIA MÚLTIPLE (en C++ sí existe), pero sí puede simularse éste mediante el uso de una múltiple implementación.

De igual forma, una clase puede implementar 1 o infinitas interfaces, debiendo implementar (definir) cada uno de los métodos declarados en dichas interfaces (recordar que con 1 sólo que no estemos definiendo, o bien hacemos la clase abstracta o nos dará un error el compilador).

Si se diese el caso de que existiese el mismo método en 2 o más interfaces que vaya a implementar una determinada clase, ésta se limitará a definirlo una vez, ya que como hemos dicho, no va a ver colisión (ve el mismo método en ambos casos y con definirlo 1 vez, tiene suficiente). Puede darse el caso de SOBRECARGA, pero el conjunto de métodos sobrecargados son, como ya sabemos, distintos entre sí a pesar de compartir el mismo nombre.

A la hora de definir los métodos declarados en la interfaz y dado que estos SON SIEMPRE PÚBLICOS (**además de abstractos**), NO PODREMOS REDUCIR SU VISIBILIDAD en la clase. Tampoco podemos lanzar un número o tipo de excepciones COMPROBADAS(CHECKED) que no estén contempladas en la declaración del método. Por último, el tipo de dato ha de ser el mismo declarado en la interfaz o bien un COVARIANTE (subtipo de dicho tipo de datos ... con los primitivos esto no es posible).

Insistimos en que a diferencia de lo que sucede con las clases, una interfaz puede heredar de diversas interfaces. Así, es correcto decir:

```
interface I3 extends I2, I1, I0 {}
```

pero es incorrecto decir:

```
class C3 extends C2, C1, C0 {}
```

todos los métodos declarados en el resto de interfaces estarán declarados en la interfaz I3, de manera que toda aquella clase que implemente dicha interfaz I3, deberá también satisfacer los métodos declarados en I2, I1 e I0.

La interfaces NO PUEDEN DECLARAR CONSTRUCTORES.

Para que una clase pueda implementar una o más interfaces, se debe utilizar la palabra reservada **implements**. Implementar obliga, como ya hemos dicho, a definir código.

Si una clase hereda de otra y a la vez implementa una o más interfaces, primero se escribe extends y a continuación implements. Si cambiamos dicho orden, el compilador generará un error.

11 CLASES ABSTRACTAS – SUPERTIPO 2 – HERENCIA.

Conjuntamente con las interfaces, constituyen un SUPERTIPO. A diferencia de estas, sí pueden definir métodos (no sólo declarar) que dotan a la clase abstracta de una funcionalidad que va ser adquirida por todas aquellas subclases que hereden de esta (siempre y cuando el nivel de encapsulación así lo permita).

Una clase abstracta no pude realizar una instanciación directa, es decir, no puede generar objetos u instancias. Es necesario que otra clase no abstracta, herede de ella y a continuación instanciar a partir de la clase no abstracta..

El hecho de no permitir la creación de objetos, no implica el que la clase abstracta no pueda definir constructores. Los constructores se emplean en la clase abstracta para INICIALIZAR aquellos atributos de esta, que generalmente son heredados por las clases hijas.

Para definir una clase como abstracta, se ha de emplear la palabra reservada **abstract**. Una clase puede no declarar ningún método abstracto y ser definida como abstracta., pero si sucede, que tiene al menos 1, entonces el compilador obliga a declarar la misma como abstracta.

Una clase abstracta puede implementar interfaces y no definir los métodos declarados en las mismas, así como puede heredar de otra clase abstracta y tampoco definir los posibles métodos abstractos o sobrescribir métodos no abstractos, de ésta.

Al final de la cadena de herencia, la que SÍ debe definir todos y cada uno de los métodos abstractos será la clase (estándar o no abstracta) que herede de la ya mencionada clase abstracta.

Abstract no es compatible con private, final o static.

Para que una clase pueda heredar de otra (la herencia directa es siempre simple), ha de utilizar la palabra reservada **extends**.

Recordar que extends se coloca por delante de implements o tendremos un error de compilación.

12 HERENCIA.

Como ya se ha indicado, la clase madre (aquella situada inmediatamente por encima de la actual en el árbol de jerarquía) se puede emplear para definir características y funcionalidad común a las subclases (clases hija) de ésta. De esta forma, se consigue **reutilizar el código** (no tenemos que volver a definir los elementos en las clases hijas, que ya habíamos establecido en la clase madre) y se **centraliza** el mismo, de cara a efectuar futuros cambios sobre él (al estar en un sólo lugar, tendremos lógicamente que cambiarlo en un sólo sitio).

Como ya sabemos,:

LOS ATRIBUTOS Y MÉTODOS PRIVADOS NO SE HEREDAN.

LOS CONSTRUCTORES NO SE HEREDAN.

Lo cuál quiere decir que no vamos a poder hacer uso de ellos a través de la clase hija ni de una instancia de ésta. Así, no podremos sobrescribir un método, ni invocar a un constructor, QUE NO TENEMOS. No obstante, hay que aclarar que el hecho de que la clase hija NO PUEDA USARLOS DIRECTAMENTE, no quiere decir que no estén presentes en dicha clase hija.

De hecho, lo que realmente ocurre es que en la clase hija se tiene ABSOLUTAMENTE TODO lo que pueda tener su clase madre (y las clases por encima de ésta ... madre de madre), pero sólo podremos usar aquello que este DIRECTAMENTE ACCESIBLE en la hija (lo heredado es directamente accesible).

El poder usar el resto de elementos NO DIRECTAMENTE ACCESIBLES (lo privado por ejemplo) se debe a que la clase madre ha proporcionado por herencia o encapsulación métodos miembro lo suficientemente visibles y que son capaces de acceder a otros métodos miembro, no visibles en la hija. Cumplen la misión de métodos “puente o intermediarios”.

Por tanto, podríamos ver el siguiente tipo de elementos en una clase hija, que hereda de una clase madre:

VISIBLES O DIRECTAMENTE ACCESIBLES:

1. ATRIBUTOS Y MÉTODOS PROPIOS(sobrescritos y no sobrescritos) DEFINIDOS EN LA PROPIA CLASE HIJA.
2. ATRIBUTOS Y MÉTODOS HEREDADOS DE LA CLASE MADRE.
   ... uso de this. y super.

NO VISIBLES O NO DIRECTAMENTE ACCESIBLES (ANEXADOS):

3. ATRIBUTOS, MÉTODOS Y CONSTRUCTORES NO VISIBLES DE LA CLASE MADRE.

Esto se justifica en el hecho de que un objeto de la clase hija puede hacer uso de métodos heredados lo suficientemente visibles de la madre que hagan uso de un atributo privado definido en la misma, pudiéndose por ejemplo, consultar o modificar su valor. Comprobaremos como cada objeto instanciado de la clase hija tiene su propia copia ANEXADA (no visible / accesible directamente) de dicho atributo privado de la clase madre. (algo similar ocurre con los JavaBeans y los atributos privados acompañados de los getters y setters públicos).

Así, se podría decir que en Java "todo se hereda", pero de cara a ver si podemos o no utilizar sobrescritura de métodos privados y constructores de la clase madre, es más correcto decir que éstos no se heredan o que no se pueden utilizar de una forma directa, lo cuál no quiere decir que no estén presentes en la propia clase hija.

Cada objeto creado recibe una copia de los atributos heredados y de los declarados en su propia clase, siendo los unos independientes de otros.

13 POLIMORFÍSMO – LAZY BINDING (Enlazado tardío o dinámico).

Esta propiedad del lenguaje confiere al mismo una gran potencia a la hora de diseñar un programa, ya que gran parte de su funcionalidad puede diseñarse independientemente de las características propias de una determinada implementación con respecto a aquellos objetos que más tarde se van a utilizar. Es decir, hace que el código sea de carácter universal y por tanto reutilizable para diversas clases siempre y cuando estas cumplan una serie de requisitos de diseño (métodos que deben declararse).

Los objetos no son polimórficos, es decir, no pueden adoptar distintas formas o más correctamente, comportamientos más allá de aquellos con los que han sido definidos originalmente. Las referencias, también llamadas manejadores o HANDLERS, tampoco pueden ser polimórficas, ya que únicamente pueden manejar el tipo de dato con el que han sido definidas.

Entonces, ¿dónde radica el poder usarse el polimorfismo?. Pues precisamente en el hecho de que toda clase y por tanto la instancia creada a partir de él, puede verse de una forma polimórfica. Esto es así gracias a la afirmación **IS – A** y el hecho de que un objeto puede verse como distintas cosas en base a la jerarquía de clases a la que pertenece. Es por tanto que una relación de herencia o de implementación, por parte de una clase o interfaz, respecto de otra clase o interfaz (analizar en cada caso la situación correcta), hace que la primera sea también la segunda, además de si misma.

Así, dada la clase D que hereda de la clase C y ésta de la B y ésta de la A que a su vez implementa las interfaces X, Y, y que ésta última a su vez hereda de la interfaz Z, se cumple que dicha clase D es:
A, B, C, X, Y, Z, además de ser la propia D, pudiéndose cargar una referencia de cualquiera de ellas con un objeto de la clase D ... POLIMORFISMO.

Lo que no podremos hacer es cargar una referencia de tipo D con elementos de las anteriores clases o interfaces, SALVO QUE APLIQUEMOS CASTING, y de hacerse esto último, habrá que vigilar que no se pueda dar una excepción de tipo ClassCastException, que podría ser lanzada por la Máquina Virtual (JVM) en tiempo de ejecución.

De forma más sencilla, dada la clase Animal y la subclase de ésta, Gato, a la pregunta ¿Gato es Animal?, contestaremos SÍ, ya que Gato hereda de Animal. A la pregunta Animal es Gato, contestaremos NO. Así, hay que ver la dirección de la flecha (relación unidireccional) de herencia para ver qué es cada cosa.

Gracias al polimorfismo podemos diseñar buena parte del código de un programa en base a un supertipo, ya sea una interfaz o una clase abstracta (recordar que sólo podemos heredar una vez), para más tarde poder "rellenar" dicho supertipo con una instancia de una clase que constituya una implementación del mismo (examinar el árbol de jerarquía de clases e interfaces).

De hecho, se impone la máxima de "programar en base a un supertipo en lugar de respecto a una implementación (una clase en concreto)". De cara a reutilización y mantenimiento, la primera forma ofrece una enorme cantidad de ventajas.

Para poder conocer el tipo exacto contenido en un supertipo, hemos de utilizar el operador **instanceof**. Dicho operador nos devolverá un booleano indicándonos si el tipo de dato contenido es clase o subclase de aquel con el que estamos comparando.

Dada la clase Animal, de la que heredan las clases Gato y Perro:

```
Animal relleno = new Gato();     // Utilizamos una referencia de la clase Animal para manejar una instancia (objeto) de la clase heredera Gato.

if (relleno instanceof Gato) {
        System.out.println("RELLENADO CON UN GATO");    <----------  NOS DARÁ ESTE MENSAJE.
}

if (relleno instanceof Perro) {
        System.out.println("RELLENADO CON UN PERRO");
}
```

El manejador no está interesado en el tipo de objeto que realmente estamos asociándole. Simplemente se encargará de utilizar el objeto allí hasta donde él lo conoce, es decir, todo aquello que haya sido definido en su propia clase. Los atributos o métodos de otra clase que no sea exactamente la del propio manejador no podrán ser utilizados a través de éste ya que le son desconocidos. Para poder hacer uso de toda la funcionalidad y características de un objeto, deberemos hacer CASTING al tipo del objeto. Es importante indicar que aunque el compilador no nos genere ningún error tras el casting, puede que la máquina virtual detecte que el tipo de objeto asociado al manejador no sea el indicado, en cuyo caso lanzará la ya conocida ClassCastException:

Animal a = new Perro(); // Rellenamos un supertipo Animal con una instancia de la clase Perro (Perro es un Animal ... herencia).

Gato g = (Gato) a; // Puede hacerse el casting porque hay una relación de herencia entre Gato y Animal (Gato es Animal). Así, relleno un
// Animal con un Gato, para más tarde, mediante el casting, volver a obtener dicho Gato.

g.maullar(); // Ahora llamo al método maullar que es propio de la clase Gato (un método común a ambos también producirá la excepción).

Al ejecutarse el programa, la JVM ve que lo que realmente tenemos es una referencia de tipo Gato apuntando a un objeto de tipo Perro, por lo que genera la excepción que nos indica que el tipo de clase no es el adecuado. No importa que la clase Perro dispusiese también del método maullar o que no se invocase ningún método, LO IMPORTANTE ES QUE LA CLASE A MANEJAR NO ES LA ADECUADA, motivo por el que se lanza la excepción.

LAZY BINDING :

¿QUÉ MÉTODO ES EL QUE REALMENTE SE LLAMARÁ.?

El lazy-binding, también llamado enlazado dinámico, tardío o perezoso, es la forma en la que la máquina virtual determina, en tiempo de ejecución, el objeto del que efectivamente debe invocarse el método solicitado (recordar el mismo método puede estar definido en diversos tipos de objetos a manejar por una determinada referencia).

Cuando una determinada clase, de la que se va a instanciar un objeto, dispone de métodos **no estáticos**, se aplica esta forma de resolución. Con aquellos métodos que son de carácter **estático** (viven en la clase), nunca se aplica el lazy-bindig, ya que se conoce perfectamente, en tiempo de compilación, cuál debe ser el método que se debe llamar a la hora de ejecutar el programa.

Así, hay que distinguir entre métodos estáticos y no estáticos.

+ Con métodos ESTÁTICOS (DIREMOS QUE MANDA LA REFERENCIA):

No se pueden sobrescribir, puesto que debe existir una única versión del mismo y esta ha de ser igual para todos aquellos que lo usen. Cuando una referencia, de un tipo determinado, invoca a un método estático, el método que efectivamente se llame será el apuntado por el tipo de dicha referencia, sin importar el tipo del objeto con el que realmente estemos cargando dicha referencia. Así, **con los estáticos, es el tipo de la referencia la que decide el método que se ha de llamar**:

Ej: Dadas las clases Animal y Gato, donde Gato es heredera de Animal y ambas tienen el método estático comer(), ¿cuál de los dos se llama en el siguiente caso?:

Animal a = new Gato();
a.comer();

El método estático comer() que realmente se llama es el de Animal, a pesar de que estamos cargando la referencia de tipo Animal con un objeto de tipo Gato. Vemos cómo en el caso de métodos estáticos, es la referencia y no el objeto al que apunta, la que decide el método qué se debe llamar.

Cada clase heredera puede definir sus propios métodos estáticos, sin importar que el mismo método esté ya definido en la clase madre (también como estático). La forma de diferenciarlos se basa en que para poder acceder a los de la madre, habremos de anteponer el nombre de la clase que ésta representa (si sólo existiese el de la madre, no sería necesario, ya que podríamos acceder a él a través de una instancia de la hija ... siempre y cuando el nivel de encapsulación lo permita).

+ Con métodos NO ESTÁTICOS (DIREMOS QUE MANDA EL OBJETO):

En el caso de tener sobrescritura:
Sí se heredan (siempre y cuando el nivel de encapsulación de éste así lo permita) . Cuando una referencia, de un tipo determinado, invoca a un método no estático, el método que efectivamente se llamé será ahora el apuntado por el tipo del objeto con el que cargue dicha referencia. Así, **con los no estáticos, es el tipo del objeto con el que se carga la referencia el que decide el método que se ha de llamar**:

Ej: Dadas las anteriores clases y los métodos no estáticos dormir(), ¿cuál de los se llama en el siguiente caso?:

Animal a = new Gato();
a.dormir();

El método dormir() que realmente se llama es el de Gato. Vemos cómo en el caso de métodos no estáticos, que es el objeto con el que se carga la referencia y no ésta, el que decide el método qué se debe llamar.

Recordar cómo podemos emplear super. en el interior de un método de la clase hija para invocar a la versión del método definida en la clase madre. Si quisiésemos hacer esto, no podemos escribir:

Gato g = new Gato();
g.super.dormir(); /// error de compilación.
Hay que crear un método en Gato que internamente invoque al dormir de Animal (dentro tendrá: super.dormir();) :
g.dormirAnimal();

En el caso de no tener sobrescritura:
Sólo tenemos 1 versión del método y será aquella con la que se ha definido esté en la clase madre. Así, no queda más que la posibilidad de llamar a dicho método, aunque estemos utilizando un objeto de la clase hija para cargar una referencia de la clase madre, pero no debemos olvidar que dicho método es parte de la propia clase hija.

¿QUÉ OCURRE CON LOS ATRIBUTOS, CUÁL DE ELLOS SE USARÁ EN FUNCIÓN DEL MÉTODO QUE SE EMPLEÉ?

Cuando en la clase madre se definen atributos que pueden ser heredados en la clase/s hija, se genera y se entrega a cada instancia de dicha clase/s hija una copia de los atributos heredados de la madre (además de los anexados, aquellos que no son directamente accesibles). El conjunto de atributos de un objeto es totalmente independiente de los atributos de otros objetos, aun cuando todos ellos sean instancias de una misma clase y siempre y cuando no sean de tipo estático.

A este respecto, hay varias posibilidades que es necesario estudiar y que se muestran a continuación:

Partimos de que en la clase madre se define tanto un atributo como un método, que dicho método hace uso del citado atributo y que ambos se heredan. Así, en la clase hija tendremos también dichos método y atributo (protected int x = 1; protected void imprimirX() { sout(x); }
Además, redefinimos en la clase hija el atributo heredado (int x = 2;), de forma de que tenemos 2 variables x sobre las que operar.

¿Qué resultado obtenemos si ... ?:

1 - En la clase hija se invoca el método heredado (no hay sobrescritura):
éste hará uso del atributo heredado x de la clase madre (x=1) y no del redefinido en la clase hija (x=2). También podemos modificar el atributo heredado de la madre (super.x = 3) y veríamos como se imprime el nuevo valor x=3.
---> x = 1;

2- En la clase hija se sobrescribe dicho método heredado para que haga uso del atributo x tal y como se estableció en la madre. Veremos como ahora se hace uso del propio atributo de la clase hija y no del heredado (aunque también podemos modificar su comportamiento para que use el heredado mediante el uso de super.x).
---> x = 2;

En conclusión, vemos como en principio, el método utilizado hace uso del atributo que tiene definido en su propia clase. Cuando dicho método es heredado por otra clase, en la que se puede definirse nuevamente el atributo sobre el que actúa, dicho método seguirá utilizando el atributo de la clase en la que fue definido (realmente la copia que recibe de ésta). Hemos de sobrescribir el método heredado para que pueda hacer uso (si así lo deseamos) del atributo definido en la clase heredera.

En el caso de sobrescribir el método heredado pero no volver a declarar el atributo sobre el que actúa, tanto el método heredado como el método sobrescrito actuarán sobre el atributo heredado, ya que no se dispone de ningún otro. Sólo tendremos una versión de la x, que es la de la madre.

Para finalizar y como ya se ha comentado, en el caso de disponer de 2 versiones de algo (atributo o método – heredado y propio), usaremos super.xxx para utilizar el heredado y this.xxx para llamar al propio. Siempre y cuando no se dé ambigüedad, lo mejor es NO poner el this a la hora de utilizar un determinado elemento.

¿ Y LOS MÉTODOS, SI HEREDO DOS MÉTODOS EN LOS QUE UNO LLAMA AL OTRO Y SOBRESCRIBO EL LLAMADO EN LA HIJA?

Supongamos que en la clase madre tenemos definidos los métodos m1() y m2(), de forma que m1() llame a m2(). Suponer además que creamos una clase hija que hereda dichos métodos de la madre, pero en la que hemos sobrescrito el método llamado por m1(), es decir, creamos una versión propia del método m2() en la clase hija. La pregunta es ¿que método m2() será llamado cuando cree una instancia de la clase hija y llame a su método m1() ... heredado porque no tengo otro?

La respuesta es que el método m2() llamado por el método m1() heredado de la madre, ES EL m2() DE LA PROPIA HIJA, no el heredado de la madre (ésto ocurre igualmente en J2EE a la hora de heredar el service(), junto al doGet() y doPost() desde la clase abstracta HttpServlet, para, habitualmente, sobrescribir en mi clase servlet sólo estos 2 últimos).

A diferencia de lo que ocurre con los atributos, el método heredado elige en primer lugar aquel método que haya sido definido en la clase hija. Si no se encuentra, se usará el método adicional heredado de la madre.

De la misma forma, si en la clase hija sobrescribo un método heredado de la madre, el método que realmente se llama al utilizar un objeto de la clase hija es el propio definido por ésta. Recordemos también que aquello definido como privado no se hereda (pero sí queda anexado), luego no cabe hablar de sobrescritura ni de sobrecarga con respecto a dicho método de la madre. Sólo debemos fijarnos en lo que tenemos en la clase hija.

14 TIPOS ENUMERADOS.

Se definieron los tipos enumerados en Java 5.0 para establecer lo que se conoce como un SAFE TYPE o tipo seguro, en el que sólo vamos a poder utilizar aquellos elementos que están definidos dentro de un grupo considerado como válido. En el caso de tratar de utilizar un valor no contemplado en dicho grupo, el compilador generará un error.

Un ejemplo de tipo enumerado podría estar formado por los palos de la baraja de poker :

```
public enum Poker {CORAZONES, DIAMANTES, PICAS, TREBOLES}

Poker cartas = Poker.DIAMANTES;  ... con los enumerados no se puede invocar a un constructor como con las clases
```

Para poder definir un tipo enumerado se utiliza la palabra **enum**. Dichos tipos pueden ser definidos como un miembro de la propia clase o constituir algo parecido a una clase en si mismo.

Lo que nunca se puede hacer es tratar de definirlos dentro del cuerpo de un método.

La potencia de un tipo enumerado no se limita simplemente a definir un conjunto de valores, sino que además es capaz de declarar internamente tanto atributos como constructores y métodos, pudiéndose aplicar el modificador static tanto a los atributos como a los métodos. Dichos constructores pueden utilizarse para inicializar los atributos del propio tipo enumerado (de un forma similar a lo que haría l constructor de una clase).

Cuando esto sucede, son los propios valores del tipo enumerado los que contienen los valores con los que se llamará al constructor interno:

```
public enum PokerAvanzado {

        CORAZONES("soy carta corazones"),
        DIAMANTES("soy carta diamantes"),
        PICAS("soy carta picas"),
        TREBOLES("soy carta treboles");  // ojo a este ; separa valores del resto.


  ////ATRIBUTOS ////////////////////////////////////////
        private String queSoy;
        private static int cantidad = 52;
```

```
    ////CONSTRUCTORES ////////////////////////////////////////
                private PokerAvanzado(String queSoy) {
                    this.queSoy = queSoy;
                }

                private PokerAvanzado(int x) {
                    System.out.println("numero = " +
                    x);
                }


    ////MÉTODOS ////////////////////////////////////////
                public void definirCantidad(int cantidad) {
                    this.cantidad = cantidad;
                }

                public static int cuantasCartas() {
                    return cantidad;
                }

                public String quePalo() {
                    return queSoy;
                }
        }
```

```
class Prueba {          public static void main(String[] args) {
                                PokerAvanzado pa = PokerAvanzado.CORAZONES;
                                System.out.println("pa = " + pa);
                                System.out.println(pa.cuantasCartas());
                                System.out.println(pa.quePalo());
                                System.out.println(" - - - cuantas cartas = " + PokerAvanzado.cuantasCartas());          // se definió como public y static.

                    }
}
```

Podemos utilizar los tipos enumerados con un bloque switch. A la hora de definir la condición del case, no debemos emplear el nombre del tipo enumerado, sino únicamente el valor que queremos comprobar:

```
switch(pa) {

        case CORAZONES:
                System.out.println("ES UNA CARTA DE CORAZONES");
        break;

}
```

15 BLOQUES DE INICIALIZACIÓN – ESTÁTICOS Y NO ESTATICOS.

Sirven para inicializar los atributos de la clase a un valor determinado, pudiendo dichos bloques hacer uso de métodos y constructores. Aunque lo normal es que cada constructor inicialice los atributos de la clase con los valores que le son entregados, los bloques de inicialización permiten asignar un determinado valor inicial, de forma general, a todos y cada uno de los atributos que podamos tener en la clase.

Un bloque de inicialización se define simplemente mediante una pareja de llaves de apertura y cierre {}. Han de estar declarados a nivel de clase, nunca dentro de un método. En el caso de anteponer la palabra static a dicho bloque, este pasa a ser un bloque de inicialización ESTÁTICO.

Así, podemos elegir entre dos tipos de inicialización por bloque: **estáticos** y **no estáticos**.

Los bloques de incialización estáticos:

static { código }

Sólo pueden acceder a elementos estáticos (a los no estáticos NO). El código que va dentro de dicho bloque define el conjunto de operaciones a realizar. Este tipo de bloques **SÓLO SE EJECUTA UNA VEZ** y esto es, en el momento de cargarse la clase en memoria, NO REQUIRIENDO LA INSTANCIACIÓN DE UN OBJETO PARA SU EJECUCIÓN. Siempre van por delante del resto de código, incluso de los constructores y otros bloques de inicialización no estáticos.

El número total de bloques que podemos tener en una clase puede ser cualquiera (*). En el caso de tener más de 1, se ejecutan según estén definidos en el código, de arriba hacia abajo.

Cuando tenemos herencia, serán todo el conjunto de bloques estáticos lo que se ejecute en primer lugar, empezando por la clase de jerarquía más alta (la más genérica) hasta la más baja (la más específica).

Los bloques de inicialización no estáticos:

{ código }

Pueden acceder a elementos tanto estáticos como no estáticos. Este tipo de bloques **SE EJECUTA UNA VEZ POR CADA OBJETO GENERADO** (requiere de instanciación para su ejecución). Van por delante del resto de código (incluyendo a los constructores de LA PROPIA CLASE), pero por detrás de los bloques estáticos y del constructor invocado de la clase madre (recordar que en Java siempre hay herencia y que para poder generar un objeto de la clase hija, ha de disponerse con anterioridad de toda la información de la clase madre y el resto de clases por encima de ésta).

Las variables finales estáticas (lo mismo que las finales no estáticas) requieren que se les de valor en la misma línea en la que se están declarando (no se inicializan a su valor por defecto porque una vez asignadas YA NO SE PUEDEN CAMBIAR, con cual fijarlas al valor por defecto NO RESULTA INTERESANTE). Si utilizamos un bloque de inicialización de tipo estático, podremos decidir asignarles su valor en el interior de dicho bloque, en lugar de en la propia línea de la declaración. De una u otra manera, la variable final queda inicializada antes de llamarse a un constructor.

Una variable estática, así como las no estáticas, NO FINAL, que se define como atributo o campo de la clase, no requiere inicialización explícita. Su valor por defecto es el indicado por su tipo, pudiéndose más tarde modificar dicho valor.

Así, podemos ver los bloques de inicialización, tanto los estáticos como los no estáticos, como un CÓDIGO GENÉRICO que se va a realizar independientemente del resto del código de la clase y que nos puede ayudar con la iniciación de los elementos de ésta. Podríamos por tanto, realizar la carga inicial de los valores comunes de la clase fuera de todos y cada uno de los constructores definidos en ésta, sin tener que repetir el código en cada uno de ellos. No obstante, dichos bloques son raramente utilizados, siendo una práctica más común el utilizar el cuerpo de los constructores para la inicialización de los atributos de la clase.

Ejemplo:

```
public class BloquesInicializacion {

   public static void main(String[] args) {
      Hija h = new Hija();
   }

}


class Madre {

   static {
      System.out.println("BLOQUE INI ESTÁTICO MADRE");
   }

   {
      System.out.println("BLOQUE INI NO ESTÁTICO MADRE");
   }

   public Madre() {
      System.out.println("CONSTRUCTOR MADRE");
   }

}


class Hija extends Madre {

   static {
      System.out.println("BLOQUE INI ESTÁTICO HIJA");
   }

   {
      System.out.println("BLOQUE INI NO ESTÁTICO HIJA");
   }

   public Hija() {
      System.out.println("CONSTRUCTOR HIJA");
   }

}
```

Así, el orden de ejecución podemos resumirlo en:

PRIMERO: LOS ESTÁTICOS DE LA MADRE Y LUEGO LOS DE LA HIJA (CADENA DE JERARQUÍA DE CLASES DE ARRIBA A ABAJO) UNA SÓLA VEZ:

1. BLOQUE INICIALIZACIÓN ESTÁTICO MADRE
2. BLOQUE INICIALIZACIÓN ESTÁTICO HIJA

SEGUNDO: LOS BLOQUES NO ESTÁTICOS DE LA MADRE Y A CONTINUACIÓN SUS CONSTRUCTORES (LOS DE LA MADRE) UNA VEZ POR CADA OBJETO INSTANCIADO O LLAMADA A ESTA CLASE (COMO RESULTADO DE LA INSTANCIACIÓN DE UN OBJETO HIJA):

3. BLOQUE INICIALIZACIÓN NO ESTÁTICO MADRE
4. CONSTRUCTOR MADRE

TERCERO: LOS BLOQUES NO ESTÁTICOS DE LA HIJA Y A CONTINUACIÓN SUS CONSTRUCTORES (LOS DE LA HIJA) UNA VEZ POR CADA OBJETO INSTANCIADO:

5. BLOQUE INICIALIZACIÓN NO ESTÁTICO HIJA
6. CONSTRUCTOR HIJA

16 IMPORTACIÓN ESTÁTICA.

Básicamente se trata de realizar la importación de elementos estáticos de la clase definida en un determinado paquete/s, gracias a lo cual podremos utilizar dichos elementos sin necesidad de hacer referencia a la clase que los contiene. Esta capacidad apareció con Java 5, pero a día de hoy, son muchos los programadores que reniegan de ella dado que el indicar el nombre de la clase contenedora del elemento estático:

1. Clarifica el código empleado ya que en todo momento vemos de qué clase estamos utilizando dicho atributo estático.

2. Evita los problemas de se dan cuando el mismo elemento estático aparece en dos clases distintas que se han importado estáticamente con el comodín *. A la pregunta lógica de ¿cúal uso?, el sistema responde con un error de compilación en el momento de intentar utilizar un elemento repetido (la importación en si no da problemas),indicando que la importación estática es AMBIGUA, es decir, que no sabe por cuál decidirse.

   Math.PI y Mates.PI ---> NO HAY PROBLEMA DE AMBIGUEDAD, SE SABE DE QUE CLASE ES CADA PI.

Su uso puede seguir la forma indicada a continuación:

import static paquete.subpaquete.clase.*	Importa TODO lo que sea estático en la clase del subpaquete, del paquete indicado, no siendo ya necesario referenciar su nombre para usar el atributo estático.
import static paquete.subpaquete.clase.alfa	Importa el atributo estático llamado alfa de la clase del subpaquete, del paquete indicado.

La importación estática también sirve para usar los elementos no estáticos de la clase, como si de una importación normal se tratará.

El orden de la declaración es importante. Primero se coloca import y luego static. Si lo hacemos al revés (static import) obtendremos un error de compilación.

ALGUNAS SITUACIONES A DESTACAR:

Dadas dos clases A y B, que definen un atributo estático PI, si realizamos la importación estática:

```
import static p1.A.PI;
import static p2.B.PI;
```

Error de compilación ya que un mismo atributo está siendo importado (de forma estática) dos veces.

```
import static p1.A.*;
import static p2.B.*;
```

La importación en si no da problemas, pero si más tarde se utiliza el atributo, que existe en ambas importaciones estáticas, ... System.out.println(PI); nos dará un error de compilación indicando que hay ambigüedad a la hora de utilizar dicho atributo, es decir, que no sabe que PI debe usar, si el de la clase A ó B.

```
import static p1.A.PI;
import static p2.B.*;
```

En este caso NO HABRÁ ERROR y se usará el atributo estático PI de la clase A, que indica exactamente el atributo estático a importarse. Observar como a diferencia del primer caso, la importación estática de B utiliza un *.

Si tratamos de importar estáticamente un atributo que no sea estático, obtendremos un error de compilación.

17 EXCEPCIONES.

Se pueden considerar las excepciones como errores que se producen en TIEMPO DE EJECUCIÓN (el momento en el que la JVM procesa el código de los .class).

Toda excepción ES UN OBJETO que de forma directa o indirecta hereda de la clase THROWABLE (y ésta a su vez de Object). Las dos grandes familias en las que se divide Throwable son **Error** y **Exception**.

Se puede igualmente realizar una clasificación en base al tipo de excepción lanzada, desde el punto de vista del COMPILADOR. Así, tenemos las:

NON CHECKED EXCEPTIONS ... o excepciones NO COMPROBADAS.
CHECKED EXCEPTIONS ... o excepciones COMPROBADAS.

La diferencia entre ambas se basa en el hecho de que va a ser el compilador, el que se encargue de avisarnos sobre la obligatoriedad o no del tratamiento o gestión de la excepción, que detecta al examinar el código.

La GESTIÓN DE EXCEPCIONES, consiste en tratar dichas excepciones y puede hacerse de 2 formas:

1. Capturar la excepción mediante un bloque **try-catch-finally** y sus variantes, pudiéndose relanzar la misma mediante un **throw**.

2. Lanzar la excepción fuera del método, mediante el uso de la claúsula **throws** declarada en el mismo. Esta forma es lo que se conoce vulgarmente como “arrojar el balón fuera” o “la patata caliente”.

Las excepciones de tipo NON CHECKED, no requieren gestión, lo que no implica que llegado el momento no pueda generarse dicha excepción. Un ejemplo muy cotidiano de las mismas es la denominada: **ArrayIndexOutOfBoundsException**, que se produce siempre que superemos las dimensiones de un array a la hora de obtener sus elementos. Este tipo de excepciones, también podemos optar por gestionarlas, pero no es lo más adecuado, ya que normalmente implican que el código que la genera deba ser revisado para corregir su funcionamiento. En definitiva, que si se produce la excepción es porque estamos incurriendo en un mal diseño.

Las excepciones de tipo CHECKED, suelen ser habituales en la declaración de métodos tales como acceso a bases de datos u operaciones de entrada / salida (I/O).

Ejemplos típicos son SQLException e IOException. Tales excepciones se han de tratar, bien capturándolas (pudiendo ser lanzadas nuevamente con la forma de una nueva excepción) o lanzándolas inmediatamente fuera del método en el momento en el que estas se producen.

Igualmente, se puede definir un gestor de excepciones a modo de sumidero de éstas (todas van a parar a dicho gestor y será él el que se encargue de gestionarlas).

Al hablar de excepciones no puede perderse de vista el concepto de **GRANULARIDAD**, que nos sirve para especificar lo genérica o específica que es un tipo de excepción en el árbol de jerarquía (herencia de clases) que tiene como ya sabemos, tiene como punto inicial la clase Throwable.
Así, la excepción más genérica que podemos encontrar es la propia Throwable, seguidas de Error y Exception. Todo el resto de excepciones son subclases de alguna de las anteriores y pueden verse como tales, es decir, un bloque catch(Exception e) {} capturará todas las excepciones del tipo Exception, PERO TAMBIÉN LAS QUE SEAN SUBTIPO DE ELLA.

Podemos encontrar errores del compilador que indiquen un mensaje como:

"Exception XXX has already been caught" **... La excepción ya ha sido capturada con un catch de granularidad más gruesa.**
"unreachable statement" **... instrucción inalcanzable (algo por encima hace imposible llegar a ella).**

No es posible alcanzar un bloque catch(){} de granularidad más fino que tiene por delante uno de granularidad más amplia ... es decir, todas la excepciones subtipo de una dada se capturarán en el bloque más genérico que precede a los bloques más específicos, con lo que el código definido por ellos es totalmente inservible al no llegarse a ejecutarse nunca. La regla a seguir es escribir primero el más especifico, y a partir de ahí, ir subiendo en generales.

El hecho de NO utilizar un tipo genérico para capturar todas la excepciones, se debe a que de esta forma, **perdemos la noción / capacidad de respuesta, sobre el error que realmente se está produciendo** (a pesar de que la traza contiene información sobre ella) para simplemente saber que ha habido un error. Esta es una práctica tan extendida entre los programadores como desaconsejada. Así mismo, el bloque de gestión de la excepción está para ser usado y no simplemente para dejarlo en blanco ... algo que también se hace sobremanera. Por último, puede que en algunos casos sea conveniente lanzar la excepción fuera del método (método de la patata caliente), pero NO SIEMPRE ... nuevamente una técnica enormemente extendida e igualmente errónea.

Ejemplo:

```
try {       Connection con = DriverManager.getConnection();
} catch(Exception e) {
            /// código de gestión de la excepción.
} catch(SQLException sqle) {   ----------------------------------->  El código es inalcanzable, ya que una SQLException es también una Exception, por
            /// código de gestión de la excepción.                    lo que en el caso de producirse, se "cuela" siempre por el catch anterior, no sirviendo
}                                                                     el actual código para nada, puesto que NUNCA se va a ejecutar.
```

Igualmente, podemos definir nuestras propias excepciones, bien heredando de Throwable, bien heredando de Exception (es lo más habitual) y sobrescribir los métodos adecuados. Existe un constructor que nos va a pedir un String que constituirá el mensaje asociado a la excepción cuando ésta se lance y que será posible obtener mediante el método .getMessage().
Otro método de interés para seguir el rastro a la excepción, es decir, los métodos por lo que se ha ido propagando, es printStackTrace() que nos imprime por la consola la traza completa de la excepción.

La forma de lanzar por código una excepción es mediante la sentencia **throw new ClaseExcepción()**. Dicha instrucción puede ser capturada nuevamente o bien lanzada fuera del método para lo cual éste ha de incluir en su declaración un tipo o supertipo de dicha excepción a lanzar.

Se entiende la **PROPAGACIÓN** de excepciones como el paso de ésta desde el método que la originó hasta el método en el que finalmente es capturada. El último nivel o método posible de captura es el main (punto de entrada al programa), pudiéndose declarar éste para que también lance excepciones. En el caso de que se sobrepasarse dicho nivel, la excepción provocará la ruptura abrupta del programa, imprimiéndose por pantalla la traza con el error.

Se entiende el LANZAMIENTO de excepciones como la generación de estas por parte de un programador en un método determinado.

Para las excepciones comprobadas, el compilador siempre va a revisar que el sistema cumpla con una de las 2 posibles formas de gestión de un excepción, lanzándola nuevamente o capturándola. En el caso de que no se dé ninguna de ellas, generará un mensaje de error.

De cara a capturar una excepción hemos de utilizar los denominados bloque **try-catch-finally** y sus posibles variantes.

try { código a probar}

... delimita el código que puede generar la excepción, de forma que si esta se produce pueda ser capturada por el siguiente conjunto de bloques catch.

catch(TipoExcepcion) { código a ejecutar }

... delimita el código a ejecutarse en el caso de que se haya generado una excepción del tipo indicado por este catch. Si dicho bloque catch, no es el indicado para la excepción generada, se continua comprobando el siguiente. En el caso de ser el adecuado, tras finalizar la ejecución del código de gestión, NO SE SIGUEN REVISANDO LOS SIGUIENTES BLOQUES catch, pero si se ejecutará el bloque finally.

finally { código a ejecutar siempre, con o sin excepción}

... delimita el código a ejecutarse haya o excepción capturada.

Puede darse el caso de que la excepción no pueda ser capturada por ninguno de los bloques catch, de forma que si es comprobada, el compilador genere el error correspondiente. En tal caso, podemos añadir un nuevo bloque catch() o bien arrojar la excepción fuera del método. Así, nada nos impide el mezclar las dos formas de gestionar las excepciones (no olvidemos que una vez capturada la excepción, podemos relanzarla con el mismo tipo u otro distinto al de la captura).

Así, de forma general, el sistema empezaría a ejecutar el código definido dentro del try{}, pudiendo éste generar o no excepciones. En el caso de que se genere una excepción, se revisa la batería de bloques catch(tipo excepción) y se ejecuta el código perteneciente a aquel catch cuyo tipo o subtipo (ya que el subtipo, es también el tipo del que hereda) de excepción coincida con la que se está dando en ese determinado momento. El resto del código del try {} NO SE COMPLETA, sino que se ejecuta el código del bloque catch que captura la excepción (cabe la posibilidad de que dentro de éste se genere una nueva excepción, con lo que este último tampoco se completaría) para a continuación ejecutar el contenido COMPLETO (nuevamente, sólo si dentro del mismo no se vuelve a generar otra excepción) del bloque finally{}. El programa podría así, continuar con normalidad, de la misma forma que si no se hubiese producido la excepción.

En el caso de que el bloque de código del try no genere ninguna excepción, ninguno de los catch y sus códigos correspondientes de gestión, se activará.

Sólo puede haber 1 bloque finally{} por cada try{}, pero catch(){} los que se quiera. Así mismo, no puede existir un try{} sin al menos un catch(){} o en su defecto, un finally(){}.

Algunas de las principales combinaciones:

TRY – CATCH – FINALLY

```
try {       // CÓDIGO A EJECUTAR QUE PUEDE LANZAR EXCEPCIONES.
            // SI ESTE CÓDIGO NO GENERA NINGUNA EXCEPCIÓN COMPROBADA DE LAS CAPTURADAS MAS ABAJO,
            // EL COMPILADOR GENERARÁ UN ERROR (salvo con Exception, que también es comprobada).

            Connection con = DriverManager.getConnection(bd, user, password);

} catch(SQLException sqle) {
            // CODIGO DE GESTIÓN DE LA EXCEPCIÓN.
} catch(Exception e) {
            // CODIGO DE GESTIÓN DE LA EXCEPCIÓN.
} catch(Throwable e) {
            // CODIGO DE GESTIÓN DE LA EXCEPCIÓN.
} finally { // CÓDIGO FINAL HAYA O NO EXCEPCIÓN.

}
```

TRY – CATCH

```
try {       System.out.println("CÓDIGO A PROBAR");
} catch(Exception e) {
            // CODIGO DE GESTIÓN DE LA EXCEPCIÓN.
}
```

TRY – FINALLY

```
try {       System.out.println("CÓDIGO A PROBAR");

} finally { // CÓDIGO FINAL HAYA O NO EXCEPCIÓN.

}
```

EXCEPCIONES PERSONALIZADAS:

Podemos crear nuestro propio tipo de excepciones simplemente heredando de un tipo de excepción ya existente (aquel que elijamos en función del nivel de jerarquía que más nos convenga). Igualmente, podemos utilizar dentro del constructor de nuestra excepción, una invocación al constructor de la madre con:

```
super("MENSAJE DE LA EXCEPCIÓN PERSONALIZADA");
```

que nos permitirá establecer el atributo message a recuperar cuando se utilice el método getMessage.

Ejemplo:

```
class MiExcepcionPersonalizada extends Exception {

   public MiExcepcionPersonalizada() {
      super("MENSAJE DE MI EXCEPCIÓN PERSONALIZADA A RECUPERAR CON getMessage()");
   }

}
```

Tabla de jerarquía de excepciones. Throwable, la madre de todas es NON-CHECKED-EXCEPTION.

18 ASERCIONES.

Se utilizan para comprobar en tiempo de ejecución que se cumple una determinada condición. En caso de que no sea así, se lanza una excepción del tipo AssertionError (heredera de Error).

Las aserciones vienen desactivadas por defecto, pudiendo ser habilitadas mediante:

```
-ea / enable assertions ...  activa las aserciones en compilación
-da / disableassertions ...  las desactiva.
```

PARA ACTIVARLAS A LA HORA DE EJECUTAR:

MENU Run - Set Project Configuration - Customize - VM options: -ea

La excepción será lanzada en el caso de que la condición a evaluar resulte false, indicándose como texto del error aquello que hayamos indicado tras los :

assert (saldoRecarga >= 0) : "La recarga de saldo no se ha realizado correctamente";

En el caso de que el valor no pueda ser imprimido como texto, el compilador no dará un error.

19 COLECCIONES.

Paquete **java.util**

Pueden almacenar cualquier tipo de objeto (se introducirán éstos en la colección como Object, realizándose automáticamente el AUTOBOXING en el caso de intentar introducir un tipo primitivo). Crecen dinámicamente y permiten mezcla en el tipo de datos almacenados, aunque es algo que se desaconseja de cara a futuras operaciones con estos, como podría ser una ordenación.

Al obtenerse un elemento de la colección, lo que se nos da es una copia de la referencia que apunta a dicho elemento (ESTOS ESTÁN REALMENTE EN EL HEAP DE MEMORIA). Dicho elemento se recupera en forma de Object, por lo que nos veremos forzados a realizar CASTING salvo que hayamos utilizado programación de tipo GENÉRICA <tipo> (en lugar de utilizarse Object, se indica con precisión el tipo de dato a introducir en la colección. Recuperaremos con ese mismo tipo de dato).

Debemos distinguir entre lo que es una simple operación de recuperación y lo que supone una operación de extracción. En la primera, el dato a recuperarse, no se quita de la colección sino que sencillamente se devuelva una copia de la referencia que lo está apuntando (su manejador),. En la segunda, el dato extraído es igualmente recuperado pero eliminándose a continuación de la colección.

Adicionalmente, podemos utilizar el método remove(xxx) para eliminar un elemento de la colección. Habrá que estudiar en cada caso el comportamiento de extraer o eliminar un elemento dado, para ver si la posición que éste ocupaba se carga con un null o si hay una recolocación del resto de elementos para que se ajusten al nuevo tamaño de la colección.

Las colecciones en Java vienen representadas por dos grandes grupos dados por las interfaces Collection y Map:

* Aquello mostrado en negrita es una interfaz, lo que no, es una clase (también llamada implementación):

Collection: (Interfaz madre de las interfaces List y Set)

List: (SÍ admite duplicados, obtener elementos directamente de 1 en 1, con Iterator o todos a la vez en forma de array de Object)

ArrayList	// acceso aleatorio rápido, secuencial lento. Ordered.
LinkedList	// acceso aleatorio lento, secuencial rápido (es lista enlazada). Ordered.
Vector	// sincronizada. Ordered. Obsoleta.

Set: (NO admite duplicados, obtener elementos mediante Iterator o todos a la vez en forma de array de Object)

HashSet	// no es ordered.
LinkedHashSet	// ordered.

SortedSet:

TreeSet	// sorted (ordenación natural ... implica ser ordered).

Map: (Parejas clave – valor)

Hashtable	// sincronizada. No permite null ni en el key ni en el value(excepción). No ordered. Obsoleta.
HashMap	// sí permite null en el key y en los values. No ordered.
LinkedHashMap	// lista enlazada (conoce la posición del siguiente elemento). Ordered.

SortedMap:

TreeMap	// sorted (ordenación natural).

MÉTODOS DECLARADOS EN LA INTERFAZ LIST:

```
public boolean add(Object e) {}

public void add(int index, Object element) {}

public boolean addAll(Collection c) {}

public boolean addAll(int index, Collection c) {}

public void clear() {}

public boolean contains(Object o) {}

public boolean containsAll(Collection c) {}

public Object get(int index) {}
```

public int indexOf(Object o) {}

public boolean isEmpty() {}

public Iterator iterator() {}

public int lastIndexOf(Object o) {}

public ListIterator listIterator() {}

public ListIterator listIterator(int index) {}

public boolean remove(Object o) {}

public Object remove(int index) {}

public boolean removeAll(Collection c) {}

public boolean retainAll(Collection c) {}

public Object set(int index, Object element) {}

public int size() {}

public List subList(int fromIndex, int toIndex) {}

public Object[] toArray() {}

public Object[] toArray(Object[] a) {}

MÉTODOS EN LA INTERFAZ MAP:

public void clear() {}

public boolean containsKey(Object key) {}

public boolean containsValue(Object value) {}

public Set entrySet() {}

public Object get(Object key) {}

public boolean isEmpty() {}

public Set keySet() {}

public Object put(Object key, Object value) {}

public void putAll(Map m) {}

public Object remove(Object key) {}

public int size() {}

public Collection values() {}

LIST (listas):

Listas de elementos que pueden estar repetidos. Estos pueden ser recuperados 1 a 1, bien por el nombre del elemento (tipos referenciados), bien por la posición que ocupan o todos a la vez vertiéndolos en un array. En ambos casos obtendremos Object, salvo si hemos utilizado programación genérica.

El método toString() se ha sobrescrito para mostrar los elementos separados por comas: [Pepe, Juan, Ana ...]

>>>>> **ARRAYLIST** – Collection – List:

Admite duplicados y es ordered (los elementos tienen un índice asociado que indica su posición dentro de la lista, comenzando este en 0).

Acceso en modo aleatorio rápido pero lento secuencialmente.

No es sincronizada.

ArrayList al = new ArrayList();

Algunos métodos implementados:

```
al.add(elemento)            // añade un elemento primitivo(autoboxing) o referenciado. Lo coloca al final. Devuelve un boolean si se modificó la colección.
al.add(posicion, elemento)  // añade el elemento en una  posición dada. Las anteriores han de estar ocupadas. Desplaza hacia arriba al resto. Devuelve void
al.addAll(Collection)       // añadir todo el contenido de un Collection a la colección actual.
al.set(posicion, elemento)  // introduce un elemento en la posición indicada. Si ya estaba ocupada, desplaza hacia arriba a los otros.
al.add(null)                // podemos añadir null.

al.get(posicion)            // obtener el elemento de la posición indicada. Recuperamos Object, luego hacer Casting (salvo que usemos genéricos).

al.contains(elemento)       // comprueba si el elemento existe. Devuelve un true / false (podemos tener más de 1, luego utilizar bucle WHILE).
al.indexOf(elemento)        // primera posición que ocupa un determinado elemento (puede haber varios). -1 si no lo encuentra.
al.lastIndexOf(elemento)    // última posición que ocupa un determinado elemento (puede haber varios). -1 si no lo encuentra.
al.isEmpty()                // booleano que nos dice si hay elementos (también se toman en cuenta los null).
al.size()                   // tamaño del ArrayList. También cuentan los null.

al.iterator()               // obtener iterador para recorrer elementos ( hasNext() – next() ).
al.toArray()                // obtener un array de Object con los elementos contenidos en la colección.

al.remove(posicion)         // Devuelve el elemento contenido en una determinada posición y a continuación lo elimina de la colección.
al.remove(Objeto)           // Elimina el objeto de la posición (podemos tener más de 1). Devuelve true si la lista ha sido modificada.

al.clear()                  // borrar todo, incluidos los null.
```

>>>>> **LINKEDLIST –** Collection – List:

Lista simplemente enlazada. Cada elemento dispone de la dirección del elemento que le se sigue en la lista.

Aceso secuencial rápido tanto para lectura como para escritura, pero lento en acceso aleatorio.

Permite duplicados y no está sincronizada. Es ordered.

El conjunto de métodos es similar al de ArrayList, pero además incorpora algunos propios para trabajar con listas enlazadas.

Nuevos métodos offerXXX, peekXXX, pollXXX, push, pop ... etc, donde XXX es First (al principio) o Last (al final).
offer = añadir // peek = consultar // poll = extraer ..... pila FIFO: push = introducir // pop = extraer.

LinkedList ll = new LinkedList();

```
ll.addFirst(elemento)       // añade el elemento al principio de la colección.
ll.offerFirst(elemento)     // lo mismo (java 6).

ll.addLast(elemento)        // añade el elemento al final de la colección.
ll.offerLast(elemento)      // lo mismo (java 6).

ll.peekFirst()              // obtiene el primer elemento sin borrarlo de la colección
ll.peekLast()               // obtiene el último elemento sin borrarlo de la colección.

ll.pollFirst()              // extrae el primer elemento de la colección (extraer == obtener y luego quitarlo de la colección. Desplaza el resto de elementos).
ll.pollLast()               // extrae el último elemento de la colección (desplaza al resto de elementos).

ll.push(elemento)           // introduce el elemento al principio de la colección, desplazando al resto. Pila tipo LIFO.
ll.pop()                    // extrae el último elemento de la colección. Pila tipo LIFO.
```

>>>>> **VECTOR –** Collection – List:

De las tres implementaciones de la interfaz List, es la única que está sincronizada. También permite duplicados y es ordered.

Si no es necesario controlar la concurrencia en multiacceso, sería mejor utilizar un ArrayList. Dicho control sacrifica buena parte del rendimiento de la clase.

Vector v = new Vector();

```
v.capacity()                // obtiene la capacidad inicial con la que se ha creado la instancia de la clase Vector.
v.size()                    // obtiene el número total de elementos contenido en el vector (posiciones realmente ocupadas, no la capacidad inicial).

v.elementAt(posicion)       // obtiene el elemento contenido en la posición indicada. Se recupera como Object, salvo que se haya usado genéricos.
v.get(posicion)             //  igual que el anterior.
v.firstElement()            // obtiene el primer elemento de la colección.
v.lastElement()             // obtiene el último elemento de la colección.
```

v.add(posicion, elemento) // añade el elemento en la posición indicada. Operación NO SINCRONIZADA.
v.insertElementAt(elemento, posicion) // igual que el anterior, pero esta vez la operación ES SINCRONIZADA.

v.elements() // devuelve un Enumeration que contiene los elementos de la colección.
v.iterator() // devuelve un Iterator con el que poder recorrer los elementos de la colección (secuencial, hacia abajo, necesita recarga).
v.removeElementAt(posicion) // elimina el elemento de la posición indicada. Devuelve void. Operación sincronizada.

v.indexOf(elemento) // devuelve la posición que ocupa un determinado elemento (podemos tener más de 1).
v.removeAllElements() // Elimina todos los elementos de la colección.

SET (conjuntos):

Los Set no permiten duplicados, ni tienen métodos para acceder directamente a los elementos 1 a 1. En su lugar, hemos de utilizar el Iterator obtenido a partir de ésta o verter el contenido de la colección en un array.

En el caso de intentar añadirse un elemento que ya exista, el sistema se limitará a ignorar la petición. Podemos comprobar que la inserción no se ha realizado consultando el valor booleano que se nos devolverán como resultado de dicha operación, de tal forma que si se devuelve:

true – la colección ha sido modificada ... dato insertado.
false – la colección no ha sido modificada ... dato no insertado.

Se acepta null como tipo de dato contenido en la colección.

>>>>> **HASHSET** – Collection – Set:

No es ordered.

HashSet hs = new HashSet();

Los métodos más importantes son los ya estudiados con la interface List (recordar que tanto List como Set heredan de la interface Collection)

>>>>> **LINKEDHASHSET** – Collection – Set:

Es ordered.

LinkedHashSet lhs = new LinkedHashSet();

Como en el caso anterior, los métodos más importantes son los ya estudiados con la interface List.

>>>>> **TREESET** – Collection – Set – SortedSet:

Los SortedSet (y en los Map, los denominados SortedMap) organizan sus elementos en base a lo que podríamos denominar como una ordenación natural (por ejemplo, el colocar dichos elementos en orden de menor a mayor). Dicha ordenación es automática, de modo que un Iterator que se obtuviese de tal colección, recorrería los elementos que la componen de una forma ordenada.

Tal ordenación automática, NO PERMITE MEZCLAR TIPOS DE DATOS INCOMPATIBLES en la colección (como sería el caso de un int con un boolean o con un String). Es necesario que el conjunto de los mismo sea COMPARABLE, lo que obliga a que todos y cada uno de ellos implemente la interfaz Comparable y definan el método compareTo(Object obj). En caso contrario los elementos no se podrán comparar y obtendremos una excepción (no error compilación).

Es sorted.

TreeSet ts = new TreeSet();

ts.descendingIterator(); // obtiene un Iterator de recorrido descendente (el normal tiene recorrido ascendente).
ts.ceiling(elemento) // devuelve el mismo o el siguiente mayor que se encuentre en la colección con respecto al elemento indicado.
ts.first() // obtiene el primer elemento.
ts.floor(elemento) // obtiene el elemento de la colección que sea igual o el más próximo por defecto, con respecto al indicado.
ts.higher(elemento) // obtiene el elemento de la colección que sea igual o el más próximo por exceso, con respecto al indicado.
ts.headSet(elemento) // obtiene un subconjunto inferior en ordenación al elemento indicado.
ts.tailSet(elemento) // obtiene un subconjunto superior en ordenación al elemento indicado.

MAP (mapas == parejas clave – valor)

Los mapas constituyen lo que se conoce con el nombre de diccionarios, es decir, parejas en las que cada clave (el elemento de busqueda del mapa, que ha se ser único), tiene asociado un determinado valor. Dependiendo de la implementación del mapa, se permitirán o no, valores repetidos o nulos, tanto en el campo clave (KEY) como en el campo valor (VALUE).

El hecho de introducir una pareja en la colección cuyo campo clave ya exista, hará que el sistema devuelva el campo valor, anteriormente almacenado, para a continuación sustituirlo por el nuevo. Es decir, se produce un reemplazo. Si no existiese ya, se añadirá a la colección sin más.

El método toString() se ha sobrescrito para que cumpla con el siguiente formato:

{KEY_1=VALUE_1, KEY_2=VALUE_2 ... KEY_n=VALUE_n}

La obtención de un elemento se realiza revisando el campo clave de la colección tratando de buscar aquel registro / pareja cuyo valor coincida con el del elemento buscado (la búsqueda no es por posición).

Así, La JVM utiliza el método equals del objeto indicado en get(xxx) para comprobar si alguna de las KEYS contenidas en la colección devuelve un true(de existir sólo puede haber una).

MÉTODOS DE COMPARACIÓN

CUANDO CREAMOS UNA CLASE, HAY AL MENOS 3 MÉTODOS DE OBJECT QUE DEBERÍAMOS SOBRESCRIBIR (mejor si utilizamos @Override):

1. public boolean equals(Object obj) ... utilizado para comparar con otros elementos.
2. public int hashCode() ... devuelve un int que identifica (distingue) a los objetos de esta clase.
3. public String toString() ... devuelve un String con la posible representación del objeto.

TANTO equals() COMO hashCode() SON DETERMINANTES CUANDO QUEREMOS UTILIZAR UN OBJETO DE LA CLASE COMO CAMPO KEY (CLAVE) EN UNA COLECCIÓN DE TIPO MAP.

PARA GENERAR EL ALGORITMO Y DETERMINA CUÁL ES EL VALOR DEVUELTO COMO CÓDIGO HASH, HAY QUE SEGUIR AL MENOS LAS 2 SIGUIENTES REGLAS OBLIGATORIAS:

DADOS DOS OBJETOS A y B.

SI A.equals(B) DEVUELVE true, ENTONCES LOS HASHCODE DE A Y B HAN DE SER OBLIGATORIAMENTE IGUALES.

EL HECHO DE QUE DOS OBJETOS SEAN DISTINTOS NO OBLIGA A QUE SUS HASHCODES TENGAN QUE SER TAMBIÉN DISTINTOS. CON VISTAS AL RENDIMIENTO, LO ACONSEJABLE ES QUE ASÍ FUESE, PERO RECORDAR, NO ES OBLIGATORIO(aquellas colecciones que usen Hashing, no pasarían a llamar a equals() si el hashCode() indicase valores distintos ... ahorramos ciclos de computación al no invocar el método equals()).

EL DETERMINAR CUÁNDO DOS OBJETOS SON O NO IGUALES, PERTENEZCAN O NO A LA MISMA CLASE, ES DECISIÓN DEL PROGRAMADOR QUE DISEÑA EL ALGORITMO A EMPLEAR POR LOS MÉTODOS equals() y hashCode(), SIEMPRE Y CUANDO SE SIGAN AL MENOS LAS 2 REGLAS DE OBLIGADO CUMPLIMIENTO MENCIONADAS ANTERIORMENTE.

Podría por ejemplo, establecer que para que dos objetos sean iguales:

a) Que instanceof devuelva un true.
b) Que el estado de todos y cada uno de los atributos sea el mismo en ambos objetos.
c) Son dos objetos distintos físicamente en memoria, luego no debemos prestar atención al valor de las referencias (cada una apuntará al suyo y lógicamente serán distintas entre si) ... hacemos Deep Comparation (comparación en profundidad).

Los métodos hashCode() – equals(Object obj) deben programarse en base a las premisas a, b, c, cumpliendo las dos reglas generales anteriores.

También podríamos realizar la comparación entre dos objetos mirando sólo el valor de sus referencias, es decir, para ver si ambas apuntan al mismo objeto. Este es el procedimiento usado por el equals(Object obj) heredado de la clase Object.

>>>>> **HASHTABLE** – Map:

Esta colección es de tipo sincronizada, al igual que ocurría con Vector en las listas.

No se admite null, ni para el campo key ni para el campo value (HashMap si lopermite).

Hashtable ht = new Hashtable();

ht.put(clave, valor) // introduce el elemento en la colección. Si la clave ya existe, se devuelve el valor antiguo y luego se reemplaza con el nuevo.

ht.containsKey(clave) // devuelve un booleano que indica si la clave existe.
ht.containsValue(valor) // devuelve un booleano que indica si el valor existe.

ht.get(clave) // devuelve el valor asociado a la clave indicada.
ht.keys() // devuelve un Enumeration con las claves de la colección.
ht.values() // devuelve un Collection con los valores de la colección (obtener de aquí el Iterator ***).
ht.elements() // devuelve un Enumeration con los valores de la colección.

ht.values().iterator() // *** devuelve un Iterador para recorrer la colección. Se ha de obtener primero el Collection correspondiente y de él, el Iterator.

ht.remove(clave) // devuelve el valor y a continuación elimina el registro que coincida con la clave indicada. Si la clave no existe no ocurre nada.

>>>>> **HASHMAP** – Map:

A diferencia de Hashtable, sí se permite null en el campo clave y en el campo valor.

HashMap hm = new HashMap();

A diferencia de Hashtable, HashMap sí permite null en el campo clave y en el campo valor.

Los métodos son muy similares a los de Hashtable, pero a diferencia de éste, HashMap no tiene el método elements() que nos devolvía un Enumeration con los valores de la colección.

>>>>> **LINKEDHASHMAP** – Map:

Un mapa que se basa en una lista enlazada.

LinkedHashMap lhm = new LinkedHashMap();

Los métodos son los ya vistos hasta ahora.

>>>>> **TREEMAP** – Map – SortedMap :

La clase TreeMap representa una mapa ordenado. A la hora de mostrarse los elementos, se hará de menor a mayor.

TreeMap tm = new TreeMap();

Esta clase incorpora nuevos métodos:

tm.descendingMap() // obtiene un **NavigableMap** para recorrer la colección en orden inverso al natural (mayor a menor).
tm.firstKey() // obtiene la primera clave de la colección.
tm.lastKey() // obtiene la última clave de la colección.
tm.floorKey(clave) // obtiene la clave indicada o la más próxima a ésta por defecto.
tm.lowerKey(clave) // igual que la anterior.
tm.higherKey(clave) // obtiene la clave indicada o la más próxima a ésta por exceso.

Tabla 08A - COLECCIONES JAVA 5.0
Tabla 08B - COLECCIONES JAVA 6.0

20 PROGRAMACION GENÉRICA <Tipo>.

I) APLICADA A LOS DATOS DE UNA COLECCIÓN:

La programación genérica, también llamada de tipo, fue introducida en Java a partir de la versión 5 (v1.5). Esta nueva forma de crear código, viene a resolver buena parte de los problemas que se producen cuando al recuperarse los elementos de una determinada colección, éstos no se corresponden con el tipo esperado.

Recordemos que todos y cada uno de ellos se guardan con el tipo genérico Object y que al recuperarse, estamos obligados a realizar un casting para poder utilizar la completa funcionalidad de los mismos, ya que utilizar una referencia de tipo Object limita su funcionalidad y características a las propias de Object.
Si al realizarse dicho casting los tipos no son compatibles, la JVM generará una ClassCastException, para notificar el error de conversión.

Al utilizar genéricos, estamos obligando al compilador a que revise los elementos que se van a introducir en la colección, de manera que si el tipo de estos no corresponde con el tipo (o subtipo) declarado en la colección, se genere un error. Así, si ya se conoce con exactitud el tipo de datos que se van a poder guardar en una determinada colección, a la hora de recuperar éstos, ya no será necesario realizar el casting. Los datos recuperados vendrán ya “casteados” al tipo con el que se declaró inicialmente la colección.

La forma de establecer el tipo de datos a manejar por una colección es mediante la notación <tipoDato>:

tipoColección <tipoDato> nombreColección = new tipoColección <tipoDato>();

ArrayList <Gato> misGatos = new ArrayList <Gato>();

II) APLICADA A LA DEFINICIÓN DE CLASES:

También es posible utilizar genéricos para crear nuestra propias clases. De esta forma podemos obligar a que sólo puedan manejarse los tipos de datos con los que las hemos definido (aunque también tenemos la posibilidad de utilizar conjuntamente con estos, cualquier otro tipo de dato).
Así, definimos en la clase lo que se conoce como comodines, que representarán un determinado tipo de dato. El número de comodines a usar es en principio ilimitado, y será el compilador el que sustituya los mismos por el valor real con el que se defina la colección.

Aunque podemos utilizar cualquier nombre para los comodines, usos habituales son:

1. <E> Cuando sólo tenemos 1 (listas).
2. <K,V> Cuando tenemos 2 (mapas).

Hay que aclarar que no estamos obligados a especificar el tipo de dato de la clase a al que hemos aplicado definición genérica, pero en este caso el compilador no realizará la comprobación:

Ejemplo:

```
public class MiClaseGenerica<E> {

   private E dato;
   private static int contador;

   public MiClaseGenerica(E dato) {
      this.dato = dato;
      contador++;
   }

   public int cuantos() {
         return contador;
   }

   public void setDato(E dato) {
      this.dato = dato;
   }

   public E getDato() {
      return dato;
   }

}
```

```
class Prueba {

   public static void main(String[] args) {

      ///////// SIN ESPECIFICAR EL TIPO DE DATO GENÉRICO /////////
      MiClaseGenerica mcg1 = new MiClaseGenerica(5);
      System.out.println(mcg1.getDato());
      mcg1.setDato("a");          // NO DA ERROR. NO HEMOS DEFINIDO EL TIPO GENÉRICO.
      System.out.println(mcg1.getDato());

      System.out.println("\n");

      ///////// ESPECIFICANDO EL TIPO DE DATO GENÉRICO /////////
      MiClaseGenerica <Integer>mcg2 = new MiClaseGenerica <Integer>(5);
      System.out.println(mcg2.getDato());
      mcg2.setDato(100);
      //mcg2.setDato("a");        // ERROR, NECESARIO UN Integer, NO UN String.

      System.out.println("\n");

      System.out.println("OBJETOS CREADOS = " + mcg1.cuantos());
   }

}
```

III) APLICADA A LA DEFINICIÓN DE MÉTODOS Y SUS PARÁMETROS:

Por último, también podemos hacer uso de métodos genéricos en los que los parámetros puedan ser una colección y se verifique el tipo de datos contenidos en la misma. Así, se establecen dos posibilidades válidas tanto para clases como para interfaces como parámetros de entrada al método:

(tipoColeccion <? extends tipoDato> nombre)

Sólo serán válidas aquellas colecciones que sean tipo o subtipo de la especificada como parámetro de entrada al método y cuya declaración de elementos sea de tipo o subtipo del indicado para dicha colección.

(tipoColeccion <? super tipoDato> nombre)

Sólo serán válidas aquellas colecciones que sean tipo o subtipo de la especificada como parámetro de entrada al método y cuya declaración de elementos sea de tipo o supertipo del indicado para dicha colección.

Ejemplo:

```
public class MetodosGenericos {

   private static void metodoSuperMineral(ArrayList <? super Mineral> al) {

      // CON super PODEMOS AÑADIR NUEVOS ELEMENTOS A LA COLECCIÓN, PERO SIEMPRE Y CUANDO
      // SEAN TIPO O SUBTIPO DEL GENÉRICO CON EL QUE SE MARCARON LOS ELEMENTOS DE DICHA
      // COLECCIÓN

      //al.add(new Atomo()); // ERROR, ES SUPERTIPO DE MINERAL ...sólo valen Mineral y Diamente. Una colección super de Mineral, sabrá manejarlos.
      al.add(new Mineral()); // pero dicha colección super de Mineral no sabrá nada de otras colecciónes que también sean super de Mineral.
      al.add(new Diamante());
   }

   private static void metodoExtendsMineral(ArrayList <? extends Mineral> al) {

/* ERROR, CON extends NO PODEMOS AÑADIR NUEVOS ELEMENTOS A LA COLECCIÓN (No puedo mezclar clases herederas de Mineral, como sería el caso de Diamente y
Rubí.
El <género> es "un apaño" del compilador del que no sabe nade la Máquina Virtual Java y ésta, a diferencia de los arrays (son Type-Safe, disponiendo de una excepción para avisar de la
incompatiblidad entre los tipos de datos que almacenan), no dispone de ningún mecanismo para en tiempo de ejecución alertar de la susodicha incompatibilidad de clases, como sería el
caso de mezcla de clases herederas del mismo nivel (hermanas) ... no hay excepción para este caso ya que las colecciónes admiten heterogeneidad (mezcla de tipos) ). Así, el género de la
colección recibida como parámetro será de tipo Mineral, algo que en principio nos permitiría contener tanto Diamentes como Rubíes, incurriendo en esa mezcolanza que en compilación
queremos evitar para ahorrarnos el casteo posterior o los posibles error de ClassCastException). Prohibiendo el poder añadir nuevos elementos a esa colección de tipo Mineral,
PREVENIMOS EL HECHO DE QUE EL PROGRAMADOR PUEDA AÑADIR TIPOS INCOMPATIBLES ENTRE SÍ, PERO TODOS ELLOS HEREDEROS DE Mineral ... Probar a
crear un ArrayList de tipo Mineral. Introducir Rubies y Diamantes. Obtenerlos (salen como Mineral). Realizar Casting directo a uno de los tipos, sin el instanceof y comprobar como se
lanzarán excepciones*/

      //al.add(new Atomo());
      //al.add(new Mineral());
      //al.add(new Diamante());
   }

   public static void main(String[] args) {

      ArrayList <Atomo>      a   = new ArrayList <Atomo>();
         a.add(new Atomo());
         a.add(new Mineral());
         a.add(new Diamante());

      ArrayList <Mineral>    m   = new ArrayList <Mineral>();
         //m.add(new Atomo()); // ERROR, ATOMO NO ES SUBTIPO DE MINERAL.
         m.add(new Mineral());
         m.add(new Diamante());

      ArrayList <Diamante>   d   = new ArrayList <Diamante>();
         //d.add(new Atomo()); // ERROR, ATOMO NO ES SUBTIPO DE DIAMANTE.
         //d.add(new Mineral()); // ERROR, MINERAL NO ES SUBTIPO DE DIAMANTE.
         d.add(new Diamante());

      ArrayList          sinTipo = new ArrayList();
         a.add(new Atomo());
         a.add(new Mineral());
         a.add(new Diamante());


      metodoSuperMineral(a);
      metodoSuperMineral(m);
      //metodoSuperMineral(d); // ERROR, SÓLO VALE UN ARRAYLIST CON ELEMENTOS TIPO O SUPERTIPO DE MINERAL.
      metodoSuperMineral(sinTipo);

      //metodoExtendsMineral(a); // ERROR, SÓLO VALE UN ARRAYLIST CON ELEMENTOS TIPO O SUBTIPO DE MINERAL.
      metodoExtendsMineral(m);
      metodoExtendsMineral(d);
      metodoExtendsMineral(sinTipo);
   }

}


class Atomo {}

class Mineral extends Atomo {}

class Diamante extends Mineral {}          // clase hija de Mineral y hermana de Rubí.

class Rubi extends Mineral {}              // clase hija de Mineral y hermana de Diamente.
```

También es posible indicar exactamente el tipo de datos de la colección que se espera recibir en el método. A la hora de invocar al método, es correcto utilizar el tipo o subtipo utilizado para la colección, pero está ha de estar marcada <tipo> con el mismo tipo con el que se ha declarado en el método, a pesar de que a la hora de rellenar la misma se admitan el tipo o subtipo de elemento con el que se marco la misma.

(tipoColeccion <tipoDato> nombre)

Ejemplo (la clase GatoPersa hereda de la clase Gato):

```
   public void metodoGato(ArrayList <Gato> alGato) {
      System.out.println(alGato);
   }

ArrayList <Gato>al1                = new ArrayList<Gato>();
al1.add(new Gato());
al1.add(new GatoPersa());
metodoGato(al1);       // CORRECTO. OBSERVAR COMO EL OBJETO GatoPersa SE HA PODIDO AÑADIR A LA COLECCIÓN <Gato>.

ArrayList <GatoPersa>al2           = new ArrayLIst<GatoPersa>();
al2.add(new GatoPersa());
//metodoGato(al2)      // INCORRECTO, PIDE <Gato> Y NO <GatoPersa>.
```

IMPORTANTE:

DISTINGUIR ENTRE LO QUE ES EL TIPO DE LA COLECCIÓN (ArrayList) Y EL DE LOS DATOS QUE ÉSTA CONTIENE (Atomo, Mineral, Diamante).

AL PASAR LA COLECCIÓN COMO PARÁMETRO AL MÉTODO, **NO SE COMPRUEBAN INTERNAMENTE LOS ELEMENTOS QUE ESTA CONTIENE**, SINO LA POSIBLE (RECORDAR QUE NO ES OBLIGATORIA) DECLARACIÓN DE ELEMENTOS EMPLEADA PARA DICHA COLECCIÓN. ASÍ, LO UNICO QUE SE COMPRUEBA ES EL **<TIPO>**.

ASÍ MISMO, SI NO DECLARAMOS EL TIPO DE DATOS EMPLEADO EN LA COLECCIÓN, EL COMPILADOR NO GENERARÁ NINGÚN ERROR YA QUE NO REALIZARÁ NINGUNA COMPROBACIÓN.

EN ÚLTIMO CASO, ES LA JVM LA QUE SE ENCARGARÁ DE GENERAR O NO LAS EXCEPCIONES DE CONVERSIÓN.

RESUMEN:

```
   // <Tipo>
   // ADMITE SÓLO TIPO O SUBTIPO PARA EL GÉNERO. Cumple relación IS-A.
   // PERMITE AÑADIR ELEMENTOS (tipo o subtipo, pero no  supertipo).

   // <? super  tipo>
   // ADMITE SÓLO TIPO O SUPERTIVO PARA EL GÉNERO.
   // PERMITE AÑADIR ELEMENTOS (tipo o subtipo, pero no  supertipo).

   // <? extends tipo>
   // ADMITE SÓLO TIPO O SUBTIPO PARA EL GÉNERO.
   // NO ME PERMITE AÑADIR NIGÚN ELEMENTO.

   // <?>
   // ADMITE CUALQUIER TIPO PARA EL GÉNERO.
   // NO PERMITE AÑADIR NINGÚN ELEMENTO.
```

// LA DIFERENCIA ENTRE extends y super, ES QUE extends NO PERMITE AÑADIR NUEVOS ELEMENTOS Y super SÍ, pero tipo o subtipo del indicado.
// PARA ELLO, LA COLECCIÓN HA DE TENER UN GÉNERO DE TIPO IGUAL O MADRE (POR ENCIMA DEL INDICADO EN LA CADENA DE JERARQUÍA DE CLASES), PARA PODER CUMPLIR QUE EL HECHO DE AÑADIR DICHOS ELEMENTOS (Rubí o Diamante) NO PROVOCA MEZCLA DE TIPOS INCOMPATIBLES ENTRE SÍ. **NO SE PUEDEN AÑADIR ELEMENTOS SUPERTIPO DEL INDICADO POR EL GÉNERO, SÓLO TIPO O SUBTIPO DEL INDICADO.**

21 I/O (INPUT – OUTPUT). java.util.Scanner.

Paquete **java.io**

El API de Java proporciona una serie de clases e interfaces para poder leer o escribir de un determinado dispositivo de entrada (I/O) ya sea un File o un Socket, así como para crear o eliminar elementos para organización, como serían los directorios del sistema operativo.

Un File representa lo que conocemos como un fichero o un directorio convencional, mientras que un Socket es una conexión con otra máquina a través de una determinada IP y un puerto lógico.

Para que nuestra aplicación Java pueda leer o enviar datos desde o hacia un dispositivo I/O, ésta requiere establecer con dicho dispositivo, lo que se conoce con el nombre de un FLUJO o STREAM. Así, es OBLIGATORIO el uso de un flujo para poder leer o escribir de un dispositivo I/O. Como funcionalidad adicional, los flujos (el constructor de la clase que lo implementa) TAMBIÉN SON CAPACES DE CREAR O REUTILIZAR un dispositivo I/O.

Un flujo representa una conexión para el intercambio de datos entre dos puntos. El API de programación dispone de una amplia variedad de clases capaces de implementar un flujo de comunicación.

Cada una de dichas clases está especializa en trabajar con un determinado tipo o tipos de datos, lo que da lugar a los denominados MODOS DE FUNCIONAMIENTO. Se establecen 3 modos, denominados: Byte, Texto y Objeto.

En cada modo de funcionamiento, encontramos los conectores, los cuales, pueden clasificarse a su vez en BAJO Y ALTO NIVEL.

Un conector de bajo nivel es aquel que es capaz de escribir y/o leer directamente de un dispositivo I/O. Su funcionalidad, lo que es capaz de hacer, es muy reducida ya que sólo dispone de unos pocos métodos para realizar operaciones de lectura y o escritura. Se leen y escriben bytes(0 1) o characteres.

Un conector de alto nivel es aquel que NO es capaz de conectarse directamente a un dispositivo I/O, pero que a cambio, ofrece una amplía funcionalidad, un conjunto más elevado de métodos en comparación con el de bajo nivel, para poder realizar operaciones de lectura y o escritura.

Adicionalmente se dispone de conectores capaces de actuar en bajo y alto nivel a un mismo tiempo, como es el caso de la clase PrintWriter (escritor de alto nivel en modo texto), ofreciendo además capacidades de buffer (leemos y escribimos lotes de datos en lugar de 1 en 1) que mejoran considerablemente el rendimiento del sistema.

Una vez establecido el flujo se pueden realizar las operaciones de lectura o escritura sobre el dispositivo I/O asociado (podemos tener más de un flujo asociado, lo cuál puede incurrir en problemas de concurrencia o inconsistencia de datos). Terminado el trabajo, debemos CERRAR EL FLUJO, invocando al método close() del mismo el cual puede lanzar una IOException que al ser comprobada, tendremos que gestionar, capturándola o lanzándola fuera del método.

En este apartado se estudian las clases e interfaces más relevantes para el tratamiento de ficheros (Files), dejándose el estudio de los Socket para más adelante.

* DISPOSITIVOS I/O (entrada y salida):

Files (ficheros y directorios).
Sockets.

* SUPERTIPO DE FLUJOS - CLASES ABSTRACTAS BASE. (a partir de estas se crean las demás):

--- PARA LEER FLUJOS DE BYTES ---
InputStream
OutputStream

--- PARA LEER FLUJOS DE CARACTERES ---
Reader
Writer

* FICHERO / DIRECTORIO:

File * el fichero o directorio físico. Para leer o escribir sobre él necesitaremos los flujos.

```
File fichero = new File("miFichero.txt");          // Objeto File.
fichero.createNewFile();                           // HAY QUE CREAR EL FICHERO FÍSICO --> IOException.
```

* EXCEPCIONES:

IOException // operaciones de escritura y lectura., cierre de flujo ... etc
FileNotFoundException // error al asociar el flujo a un fichero / directorio.
EOFException // detección del final del fichero.

Todas estas excepciones son COMPROBADAS, luego tendremos que gestionarlas (capturar o lanzar).

* INTERFAZ:

Serializable

* La marca que indica que el objeto se puede serializar (no declara ningún método).
* LOS ATRIBUTOS static Y transient NO SE INCLUYEN EN LA SERIALIZACIÓN
(Guardar el estado de un objeto, el valor de sus atributos en un fichero)

- MODOS DE LOS FLUJOS -

EL CONSTRUCTOR QUE NOS PIDE UN BOOLEANO A LA HORA DE ESTABLECER EL FLUJO DE ESCRITURA CON EL FICHERO, NOS PERMITIRÁ ABRIR ESTE DE FORMA QUE LOS NUEVOS DATOS SE AÑADAN A LO QUE YA PUDIESE TENER O POR CONTRARIO, SE IGNOREN LOS MISMOS Y SE EMPIECE EN BLANCO.

LA CODIFICACIÓN ES SIEMPRE EN OCTETOS, CONJUNTOS DE 8 BITS DE 0 Y 1. AUNQUE TENDREMOS QUE UTILIZAR LOS TIPOS DE JAVA, TODOS ELLOS SE REPRESENTARÁN EN BASE AL TIPO UNIVERSAL BYTE (8 BITS SIN SIGNO => 0 ... 255), **QUE NO ES EL TIPO PRIMITIVO byte DE JAVA** (7+1 BITS => -128 ... 127).

DE HECHO, EL MÉTODO HABITUAL write(), NOS PEDIRÁ UN int EN LUGAR DE UN byte, DE FORMA QUE SE PUEDA CODIFICAR EL CITADO VALOR 0 ... 255 (RECORDEMOS COMO UN VALOR MAYOR QUE 127 NOS DARÍA UN ERROR DE COMPILACIÓN AL USAR EL TIPO PRIMITIVO byte), PERO DEL QUE SÓLO SE USARÁN LOS 8 BITS MÁS BAJOS (LSB), IGNORÁNDOSE EL RESTO. ASÍ, HAY QUE VER EL DATO EN CODIFICACIÓN BINARIA (JAVA NO DISPONE DE UN TIPO PRIMITIVO PARA REPRESENTACIÓN BINARIA) Y NO COMO EL TIPO PRIMITIVO PROPIO DE JAVA.

EJ: SI ESCRIBIMOS Y LEEMOS EL DATO (write(dato) ... read(dato):
16714241 decimal == 111111110000101000000001 binario == FF0A01 hexadecial ESTE SE VERÁ COMO 1. SÓLO SE COGEN LOS 8 ÚLTIMOS BITS.

POR OTRA PARTE, LA LECTURA REQUIERE QUE DETECTEMOS EL FINAL DE FICHERO. ESTE VENDRÁ DADO EN FUNCIÓN DEL TIPO DE IMPLEMENTACIÓN QUE ESTEMOS UTILIZANDO Y DEL MODO Y MÉTODO DE LECTURA UTILIZADO. ASÍ:

UTILIZANDO LOS MÉTODOS SIGUIENTES, EL FIN DE FICHERO SE DETECTARÁ CON (utilizar un bucle while(true) { } a romper al detectar el fin de fichero) :

read(), read(byte[] b)	**... UN –1.**
readBoolean(), readByte(), readShort(), readChar(), readInt(), readLong(), readFloat(), readDouble()	**... UNA EOFException.**
readLine()	**... UN null.**

NOTA: read() devuelve un BYTE (no es tipo Java) de datos (8 bits sin signo) mientras que readInt() devuelve un int (si es tipo Java = 32 bits con signo).

LA LECTURA DEL FICHERO POR PARTE DEL FLUJO, ES SECUENCIAL. ASÍ, SI UNA VEZ COMPLETADA LA MISMA, QUEREMOS VOLVER A LEER DICHO FICHERO, TENDREMOS QUE CREAR UN NUEVO FLUJO (EL ANTERIOR YA NO ES ÚTIL) . SI ESTAMOS UTILIZANDO UN LECTOR DE ALTO NIVEL ASOCIADO A SU CORRESPONDIENTE LECTOR DE BAJO NIVEL, HABREMOS DE VOLVER A CREAR AMBOS, SI DESEAMOS REPETIR LA LECTURA (SE USAN AMBOS CONJUNTAMENTE).

EL HECHO DE QUE LOS FLUJOS DE LECTURA SEAN DE UN ÚNICO USO OCURRE TAMBIÉN CON LAS IMPLEMENTACIONES DE LAS INTERFACES Iterator Y ResultSet.

PRINTWRITER, **AL IGUAL QUE EL RESTO DE FLUJOS ESCRITORES DE ALTO NIVEL,** ESTÁ DISEÑADA PARA CERRAR TODOS Y CADA UNO DE LOS FLUJOS QUE TENGA ASOCIADA AL INVOCARSE A SU MÉTODO .close().

ASÍ, IMAGINAR QUE A UN PRINTWRITER SE LE ASOCIA UN FLUJO DE ESCRITURA DE BAJO NIVEL EN MODO BINARIO, POR EJEMPLO UN FILEOUTPUTSTREAM, QUE A SU VEZ, ATACA A UN FICHERO FÍSICO.

SI INVOCAMOS AL MÉTODO .close() DEL PRINTWRITER, NO SÓLO SE CERRARÁ DICHO FLUJO DEL PRINTWRITER HACIA EL FILEOUTPUTSTREAM, SINO QUE TAMBIÉN SE CERRARÁ EL FLUJO DEL FILEOUTPUTSTREAM HACIA EL FICHERO FÍSICO.

SI A CONTINUACIÓN INTENTAMOS USAR CUALQUIERA DE LOS 2 FLUJOS, OBTENDREMOS UNA EXCEPCIÓN.

TODAS LAS CLASES ESCRITORAS DISPONEN DE UN MÉTODO .flush().

¡¡¡ATENCIÓN!!!:

LAS CLASES ESCRITORAS DE LOS MODOS TEXTO Y OBJETO, **TENDRÁN QUE INVOCAR A SU MÉTODO flush()** PARA QUE EL DISPOSITIVO I/O RECIBA LOS DATOS. SI NO SE HACE ÉSTO Y TRATAMOS, A CONTINUACIÓN, DE LEER DICHO DISPOSITIVO I/O NO SE DETECTARÁ NINGÚN DATO PUESTO QUE REALMENTE NO SE HA ESCRITO NADA. TAMBIÉN DECIR, QUE UN CIERRE VOLUNTARIO DEL FLUJO MEDIANTE EL MÉTODO .close() O UN LLENADO DEL BUFFER, PROVOCA EL VACIADO (VOLCADO DE DATOS) AUTOMÁTICO DE ÉSTE.

[MODO BYTE]

BAJO NIVEL (tipos básicos): byte[], int. Se leen byte crudos, no el tipo primitivo byte de Java.

LECTURA	**FileInputStream**	**fin fichero con –1.**
ESCRITURA	**FileOutputStream**	**true/false. (borra no/sí los datos anteriores). no requiere hacer flush().**

ALTO NIVEL (permite más tipos): byte[], int, boolean, byte, short, long, float, double, String.

LECTURA	**DataInputStream**	**fin fichero con –1, EOFException.**
ESCRITURA	**DataOutputStream**	**no requiere hacer flush().**

[MODO TEXTO]

BAJO NIVEL (utilizamos characteres): String, char[], int.

LECTURA	**FileReader**	**fin fichero con –1.**
ESCRITURA	**FileWriter**	**true/false. (borra no/sí los datos anteriores). requiere hacer flush() para enviar los datos.**

ALTO NIVEL (utilizamos cadenas de texto): String, char[], int.

LECTURA	**BufferedReader**	**fin fichero con –1, null.**
ESCRITURA	**BufferedWriter**	**requiere hacer flush() para enviar los datos.**
ESCRITURA	**PrintWriter**	**conexión directa a dispositivos I/O. requiere hacer flush() para enviar los datos. puede escribir en System.out (consola). cierra TODOS los flujos asociados con .close().**

[MODO OBJETO - SERIALIZACIÓN]

ALTO NIVEL: Todos los tipos primitivos y tipos referenciados.

Leer en el mismo orden en el que se guardaron. Recuperar los objetos mediante el uso del Casting.
En este modo no existen los conectores de bajo nivel. Así, deberemos usar el bajo nivel del modo binario (modo texto no vale).
Una clase sólo se puede serializar si implementa la interfaz Serializable (está vacía, sólo es una marca).
Lo que se guarda en el fichero es el estado del objeto (conjunto de atributos y sus valores en un momento determinado).
Los métodos y constructores no forman parte del estado del objeto, luego no se serializan.
Los atributos **static** y los declarados como **transient**, no formarán parte de la serialización.

LECTURA	**ObjectInputStream**	**Debemos saber cuántos objetos hemos escrito.**
ESCRITURA	**ObjectOutputStream**	**requiere hacer flush() para enviar los datos.**

LA CLASE SCANNER:

Paquete **java.util**

La clase Scanner se incorporó al API de Java para facilitar la adquisición de datos a través de distintas fuentes. Así, podemos utilizar como entrada el teclado (System.in), una cadena de texto (String) o un determinado fichero (File) gracias a la sobrecarga del constructor que nos va a proporcionar una instancia de la clase Scanner.

```
Scanner sc1 = new Scanner(System.in);
Scanner sc2 = new Scanner(new File("ficheroDatos1.txt"));
Scanner sc3 = new Scanner("¿América fué descubierta por Cristobal Colón?");
```

Se dispone de diversos métodos que nos permiten comprobar si existen más datos y el tipo de estos en la entrada, así como de poder elegir el delimitador para la separación de los mismos (cada dato es también llamado token). Así, los métodos habituales a utilizar son.

useDelimiter(“delimitador”)	Especifica el delimitador a usar para separar los elementos o tokens. Por defecto es un espacio en blanco. Sirve cualquier palabra, frase o número “abc” “5” “#”. No utilizar elementos de expresiónes regulares (+,?...).
hasNext()	Comprueba si quedan más elementos (del tipo que sean) por extraer. Devuelve un boolean con la respuesta.
hasNextBoolean()	Comprueba si el siguiente elemento puede verse como un boolean (no importa mayúsculas o minúsculas).
hasNextInt()	Comprueba si el siguiente dato puede verse como un int.
.....	Así con todos los datos primitivos.
nextBoolean()	Obtiene el siguiente dato en forma de tipo boolean.
nextInt()	Obtiene el siguiente dato en forma de tipo int.
.....	Así con todos los datos primitivos.
next()	Obtiene el siguiente dato, no importa como se pueda ver, en forma de tipo String.
nextLine()	Obtiene y devuelve la línea completa de tokens (los que queden por extraer) como un String.

En el caso de utilizarse el teclado, el objeto Scanner se quedará esperando hasta que el usuario pulse la tecla intro, haya o no introducido datos. Hay que tener presente el delimitador usado y el nextXXX empleado.

NO DEBEMOS CERRAR EL FLUJO System.in, TAN SÓLO EL Scanner.

// USANDO UNA CADENA DE TEXTO:

```
String texto = "HOLA 123 555 CÓMO TRUE ESTAS false TrUe 22";          // OBSERVAR QUE TODO ES TEXTO. ESPACIOS BLANCOS.
Scanner escaner = new Scanner(texto);

String palabras = "";
String numeros = "";
String booleanos = "";

while (escaner.hasNext()) {                                           // ¿TIENES MÁS ELEMENTOS (DEL TIPO QUE SEA)?
   if (escaner.hasNextInt()) {                                        // ¿EL SIGUIENTE ES UN int?
      numeros += escaner.nextInt() + ",";                             // CÓGELO COMO UN int.
   } else if (escaner.hasNextBoolean()){                              // ¿EL SIGUIENTE ES UN boolean?
      booleanos += escaner.nextBoolean() + ",";                       // CÓGELO COMO UN boolean.
   } else {
      palabras += escaner.next() + ",";                               // CÓGELO COMO STRING, SEA LO QUE SEA.
   }
}

System.out.println(" numeros ---> " + numeros);
System.out.println(" booleanos ---> " + booleanos);
System.out.println(" letras ---> " + palabras);
```

// USANDO EL TECLADO:

```
System.out.print("\nINTRODUCE TU NOMBRE ... ");
Scanner s = new Scanner(System.in);
String nombre = "";

//nombre = s.next();                          // COGE SÓLO EL PRIMER TOKEN (SEPARADOR ESPACIO BLANCO POR DEFECTO).
nombre = s.nextLine();                        // COGE TODO.
System.out.println("\nHOLA ... " + nombre);
```

23 THREADS (HILOS).

La forma en la que Java es capaz de implementar varios procesos simultáneamente (multiprocesamiento o multithreading) es a través de hilos.

FORMAS PRINCIPALES DE INSTANCIAR UN HILO:

1. Creando una clase que herede de Thread y sobrescribiendo el método public void run() {} ... no puede lanzar excepciones porque el original no lanza ninguna. La herencia hace que dicha clase sea también un Thread (recordar que ya no podremos usar más la herencia directa).

```
MiHilo h1 = new MiHilo();
```

2. Creando una clase que implemente la interfaz Runnable (obliga a implementar el método public void run()) y pasando una instancia de la misma al constructor de Thread (sobrecargado). Hay que destacar que dicha clase NO ES UN THREAD, sino un RUNNABLE, por lo que no tendremos los métodos heredados de Thread. A cambio, seguiremos teniendo disponible el uso de la herencia directa.

```
Thread t1 = new Thread(new MiRunnable());

// LO MISMO PERO CON DECLARACIÓN EN LÍNEA DE LA INTERFAZ Y SU CONTENIDO:
Thread t3 = new Thread(
                    new Runnable() {
                        public void run() {
                            // código de ejecución del hilo //
                        }
                    }
                );
```

También puede utilizarse el constructor que pida el nombre con el que designar al hilo o bien utilizarse más tarde el método setName(“”).

```
Thread(Runnable r, String nombre)
Thread(String nombre)
```

Todo hilo es gestionado por un elemento denominado **SCHEDULER** o planificador, que se encargará de activar o desactivar estos (ejecutar a lo largo del tiempo el contenido de su método run()) en lo que se conoce como CICLO DE VIDA DE UN HILO.

El funcionamiento, el plan de ejecución de dicho planificador ES TOTALMENTE INDEPENDIENTE DE LA VOLUNTAD DEL PROGRAMADOR y aunque existen algunos métodos con los que tratar de “sugerirle” cómo debe actuar, será éste el que finalmente decida cuál es el hilo que debe estarse ejecutando ... en resumen, “EL PLANIFICADOR ES EL QUE MANDA”.

Así, podríamos decir que cuando se trata de hilos, NO HAY NADA 100% SEGURO RESPECTO A LA EJECUCIÓN DE LOS MISMOS. Con hilos suele ser habitual que ejecutar dos veces el mismo programa, en la misma máquina virtual, genere distintos resultados. Aparte, cada versión de dicha máquina virtual, puede tener su propia versión del planificador con lo que los resultados entre ellos variará casi con toda seguridad.

EL CICLO DE VIDA DE UN HILO:

ESTADOS NORMALES:

1.	NEW	- INSTANCIADO:	Instanciar un objeto Thread. El futuro hilo aún no está vivo ni tiene pila de ejecución. El método isAlive() indicaría false.
2.	RUNNABLE	- DISPONIBLE:	Llamar a su método start(). Hace que el planificador gestione y entregue al objeto Thread una pila de ejecución (Execution Stack) y lo marque como hilo elegible. El hilo está vivo y a punto para poder ejecutar el contenido de su run() al ser elegido. El método isAlive indicaría true.
3.	RUNNING	- EJECUTÁNDOSE:	Ejecuta el método run() del hilo seleccionado (variables a su pila de ejecución).

ESTADOS DE NO EJECUCIÓN (NON RUNNING STATES):

4.	WAITING	- ESPERANDO:	Hilo en espera de que otro le despierte o pase un cierto tiempo. Suelta la llave.
5.	SLEEPING	- DURMIENDO:	Hilo dormido hasta que pase como mínimo el tiempo indicado. No suelta la llave.
6.	BLOCKED	- BLOQUEADO:	Hilo en espera hasta que se produzca situación de desbloqueo. Acceso a dispositivo.

ESTADO FINAL:

7.	DEAD	- MUERTO	El método run() del hilo ha sido completado. No se puede volver a activar. El método isAlive indicaría false.

Como mención especial, tener en cuenta que un hilo pasa siempre del estado RUNNING al estado RUNNABLE o en su defecto a uno de los NON RUNNING STATES o al DEAD. De igual forma, pasará de un NON RUNNING STATE al estado RUNNABLE (siempre y cuando se produzca el adecuado evento que provoque el cambio).

Lo que nunca va a poder hacer es pasar al estado de ejecución (RUNNING) directamente desde un estado que no sea el de disponible (RUNNABLE) y sin la actuación del planificador.

En definitiva, el paso al estado de ejecución (RUNNING) sólo puede hacerse a través del planificador, debiendo estar el hilo previamente en estado RUNNABLE (disponible o elegible).

HERRAMIENTAS PARA SINCRONIZACIÓN SOBRE UN RECURSO COMPARTIDO (BLOQUE O MÉTODO synchronized):

DEBEMOS USARLOS SOBRE EL RECURSO COMPARTIDO, o de lo contrario obtendremos una IllegalThreadStateException:

Si bloque de código sincronizado a través de un objeto compartido ---> **objeto.**wait();
Si método sincronizado (se usa la propia instancia) ---> wait(); ... es this implícito (no hace falta ponerlo).

wait() Poner el hilo en espera, WAITING, hasta que éste reciba una notificación por parte del planificador (como solicitud de otro hilo sobre el recurso compartido por ambos) para pasar a RUNNABLE.
Puede lanzar InterruptedException.

wait(tiempo ms)
Igual que el anterior, pero pasará al estado RUNNABLE bien por notificación bien porque haya pasado como mínimo el tiempo indicado.
Puede lanzar InterruptedException.

notify() La forma en la que un hilo notifica al planificador que ponga en RUNNABLE otro hilo de los que comparten con él, un determinado recurso.

notifyAll()
La forma en la que un hilo notifica al planificador que ponga en RUNNABLE todos aquellos hilos que comparten con el un determinado recurso.

ESTOS MÉTODOS NO SON PROPIOS DE THREAD, SINO DE OBJECT.

HAN DE USARSE DENTRO DE UN BLOQUE O MÉTODO SINCRONIZADO (palabra reservada synchronized u obtendremos un error de compilación)

El utilizar la sincronización con un recurso compartido evita los problemas propios del multiacceso, como podría ser la inconsistencia de datos. Para ello, Java provee los denominados cerrojos o llaves, de tal forma que sólo existe una llave por cada elemento compartido y sólo un hilo puede estar en posesión de dicha llave. Aquel hilo que disponga de la llave, podrá ejecutar el código declarado con la palabra synchronized. El resto de hilos, a pesar de estar en un determinado momento en ejecución, no podrán ejecutar el código del bloque o método compartido.

Hay que resaltar el hecho de que una llamada a sleep() provocará que el hilo que "se pone a dormir", se lleve consigo la llave sobre el recurso compartido, en el caso de tenerla. Sin embargo, una llamada a wait(), hará que el hilo "se ponga en espera" pero previamente entregando la llave al planificador para que éste se la pueda dar al siguiente hilo que entre en ejecución, pudiendo así, hacer uso del código sincronizado (sólo si este trabaja también con dicho recurso compartido).

Tanto wait como sleep (además del método de encolamiento join) pueden provocar una excepción de tipo InterruptedException.

Las denominadas aplicaciones productor – consumidor hacen uso de la sincronización para que el hilo productor (el que genera datos) actué conjuntamente con el hilo consumidor (el que usa dichos datos) sin caer en los problemas de consistencia propios de un acceso concurrente o simultaneo. Así, se define el recurso compartido, que será un objeto de una determinada clase (como por ejemplo un File) que le será pasado a ambos hilos a la hora de construirlos. La invocación a los métodos de sincronización, principalmente wait() y notify(), se hará sobre dicho recurso compartido por ambos hilos (podemos tener más de 2 hilos usándolo, en cuyo caso también podemos optar por notifyAll()).

En términos generales y suponiendo que el hilo actual de ejecución posee la llave sobre un determinado recurso compartido:

Si éste pasa a WAITING, **suelta la llave** del recurso compartido. El planificador puede dársela a otro hilo para que lo pueda usar.

Si se pone en SLEEPING o vuelve a RUNNABLE, **la llave la no la suelta**, imposibilitando al resto de hilos que también trabajen sobre dicho recurso compartido, el poder hacer uso de él.

No debemos olvidar que por cada elemento compartido existe una única llave, pero que un mismo hilo puede hacer uso simultáneamente de 2 o más recursos compartidos (también ninguno), es decir, estar en posesión de dos o más llaves. Así, sólo en el momento de pasar a WAITING, es cuando soltará todas y cada una de dichas llaves.

SINCRONIZACIÓN DE UN BLOQUE DE CÓDIGO CON RESPECTO A UN RECURSO COMPARTIDO (OBJETO):

Se hace necesario utilizar un objeto, el recurso compartido, a partir del cual obtener la llave (key lock) que permita o deniegue la ejecución del bloque de código sincronizado del hilo (cada hilo tiene el suyo y lo que se busca es que se pueda o no ejecutar el mismo en un momento determinado).

En el instante en el que un hilo adquiere la llave, consigue el permiso de ejecución de su bloque sincronizado, denegándose la misma al resto de hilos (cada uno de su propio bloque). El hilo elegido, puede decidir ponerse en WAITING (habrá que vigilar la notificación previa notify() – notifyAll() que debe enviar al planificador para evitar dead locks) y permitir así, que otro hilo ejecute su bloque sincronizado.

Así, la sincronización se obtiene mediante el uso de los métodos wait(), notify() y notifyAll() sobre el recurso compartido.

Esta forma de trabajo es propia de las aplicaciones productor – consumidor.

Ejemplo de un hilo productor, sobre un recurso compartido vaso:

```
public class Camarero extends Thread {

   private Vaso vaso;

   Camarero(String nombre, Vaso vaso) {
      super(nombre);
      this.vaso = vaso;
   }

   @Override
   public void run() {
      //System.out.println("SOY CAMARERO .... " + Thread.currentThread().getName());

      synchronized (vaso) {
         while (Vaso.contador < 5) {
         if ("lleno".equals(vaso.consultar())) {
            vaso.notify();
            try {
               vaso.wait();
            } catch (InterruptedException ex) {
               // nada
            }
         } else {
            vaso.llenar();
            System.out.println("EL CAMARERO LLENA EL VASO");
            vaso.notify();
            try {
               vaso.wait();
            } catch (InterruptedException ex) {
               // nada
            }
         }
         vaso.notify();
         }
      }
   }
}
```

Si un hilo no dispone de la llave para poder ejecutar su bloque de código sincronizado, no se salta dicho bloque para seguir ejecutando el resto del código de su método run(), sino que se detiene hasta poder conseguir la llave.

SINCRONIZACIÓN DE UN MÉTODO:

De la misma forma que se pueden sincronizar dos o más hilos con respecto a un recurso compartido, podemos también realizar dicha sincronización utilizando un método. Para ello, simplemente marcaremos el mismo como synchronized y pasaremos una copia de la referencia del objeto instanciado para invocarle, a todos y cada uno de los hilos que quieran usarlo.

En el momento en que un hilo acceda a dicho método, intentará adquirir la llave de sincronización y no la devolverá hasta que no termine la completa ejecución del mismo. Si obtenida la llave, dicho hilo se desactiva sin haber finalizado el método (paso a Runnable por conmutación de hilo), el resto de hilos tendrán que esperar a que éste termine, antes de poder acceder al método.

Para ello, el hilo poseedor de la llave tendrá que volver a ser elegido, entrar en el estado Running y finalizar por completo la ejecución del código del método antes de devolver la misma. Es este bloqueo el que asegura la sincronización.

En una clase podemos combinar métodos sincronizados con métodos no sincronizados.

Ejemplo:

```
synchronized void escribir() { System.out.println("entrando en
      escribir"); System.out.println("HILO : " +
      Thread.currentThread().getName());
      System.out.println("BLOQUEO EL ACCESO A LOS OTROS HILOS A ESTE MÉTODO HASTA QUE YO TERMINE");

      try {                                          // PONER A DORMIR EL HILO ACTUAL, NO SUELTA LA LLAVE.
                  Thread.sleep(4*1000);
            } catch (InterruptedException ie) {
                        ie.printStackTrace();
      }

      System.out.println("YA HE TERMINADO, LIBERO EL ACCESO (KEY LOCK - LLAVE)");
      System.out.println("saliendo de escribir");
      System.out.println("\n");
   }
```

SITUACIONES DE BLOQUEO (BLOCKED) Y BLOQUES MUERTOS (DEAD LOCKS):

Como ya sabemos, dos o más hilos pueden compartir un determinado recurso. Para evitar los problemas de sincronía o multiacceso, obligamos a los hilos a coordinarse entre ellos, a través del planificador, empleando los métodos wait(xxx), notify() y notifyAll(). Si dicha coordinación no está bien planteada, podríamos entrar en la generación de una situación de bloqueo o en un bloque muerto(es un caso particular).

Una situación de bloqueo (hilos "Blocked") podríamos entenderlo como una espera infinita. Se da la circunstancia de que **TODOS** los hilos sincronizados están en situación **WAITING** a la espera de que el planificador decida levantar uno de ellos (o todos a la vez) para poder pasarle al estado RUNNABLE y a continuación poder ser elegido para continuar con la ejecución de su método run(), en el ya conocido estado RUNNING.

El problema viene dado porque el planificador debe a su vez esperar a que sea uno de dichos hilos, el que le envié una petición, aviso o notificación de que puede proceder al levantamiento de otro hilo (puede que sea este mismo hilo que ha lanzado la petición el que resulte elegido). Dado que todos ellos están esperando a que sea otro el que genere el aviso, se concluye en que NADIE finalmente lo provocará, con lo que el planificador no podrá elegir ningún hilo para ponerlo en RUNNING, ya que no tiene ninguno para escoger (recordemos que todos estaban en WAITING).

Así, nunca se sale del estado de eterna búsqueda de un hilo disponible (RUNNABLE) por parte del planificador, lo que nos obligará a detener el programa de una forma abrupta.

Los bloques muertos pueden verse como "cruces en las llamadas de bloques sincronizados" sin que los hilos estén en waiting. Veamos cómo pueden producirse:

El hilo1 entra al método sincronizado m1() de la clase A, para lo cual adquiere su llave, comenzando a ejecutar su código. Llega un momento en el que el Scheduler conmuta el hilo con lo que se activa el hilo2, que entra al método sincronizado m2() de la clase B, adquiriendo su llave (observar como hay dos objetos, luego hay dos llaves, y que los métodos están sicronizados a nivel de objeto (this)). El hilo2 empieza a ejecutar el m2(). Un tiempo después, se vuelve a producir un cambio de hilo, con lo que hilo1 vuelve a estar activo. Éste, sigue ejecutando el método m1() y llega a una línea de código en el que debe ejecutar el método m2() del objeto de clase B, objeto compartido con el hilo2 y del que éste último tiene su llave. El Scheduler detecta esta situación, hilo1 no puede continuar porque no tiene acceso a m2() (la llave la tiene el hilo2), forzando esta situación al Scheduler a nuevamente conmutar el hilo. Así, se vuelve a activar el hilo2 y continua con su ejecución del m2() llegando a una línea en la que tiene que invocar el método m1() del objeto de clase A, que comparte con el hilo1. Aquí se repite la situación anterior, hilo2 no puede usar m1() porque la llave de acceso la tiene el hilo1, ya que fué él el que entró primero y aún no ha terminado de ejecutar el método en cuestión.

Vemos por tanto, como hilo1 espera a que hilo2 suelte la llave de m2 del objeto compartido de clase B, mientras que hilo2 espera a que hilo1 suelte la llave del m1() del objeto compartido de clase A ... ESTAMOS EN UN CALLEJÓN SIN SALIDA O DEAD BLOCK. El Scheduler no puede resolver esta situación, NO LE QUITA LA LLAVE A NINGUNO DE LOS DOS HILOS, con lo que el programa NÚNCA PUEDE CONCLUIR. Podríamos decir aquello de "yo te doy y tu me das, pero dame tú primero".

Para simular esta situación necesitamos por ejemplo, 2 clases A y B, con sus métodos m1() y m2(), respectivamente, sincronizados a nivel de objeto (se usa el propio objeto ... this implícito) y pasar la referencia de los objetos a y b creados, a ambos hilos hilo1 e hilo2. Usaremos un Thread.sleep(2000); para provocar la conmutación una vez que los hilos entran en sus respectivos métodos. Observaremos como aquello no acaba núnca.

MÉTODOS PROPIOS DE THREAD:

join() Aquel hilo del que se llama a su método join(), debe completarse para poder continuar la ejecución del hilo actual (el que hizo la llamada).
Puede lanzar InterruptedException.

join(long ms)
El hilo que llama al método join(ms) de otro hilo, esperará al menos el tiempo indicado, antes de continuar su propia ejecución.
Puede lanzar InterruptedException.

sleep(ms) El hilo que invoca al método sleep(ms) (ha de ser sobre el hilo en ejecución, no sobre otro hilo) pasará al estado SLEEPING y permanecerá así, al menos el tiempo indicado. El planificador lo volverá a colocar automáticamente en RUNNABLE.
Es estático así que podemos invocarlo sobre el hilo actual siendo lo mejor llamarlo a partir de la clase Thread.
Puede lanzar InterruptedException.

yield() Pasa el hilo a RUNNABLE, sugiriendo al planificador que ponga en ejecución otro hilo de su misma prioridad.
Es estático así que podemos invocarlo sobre el hilo actual siendo lo mejor llamarlo a partir de la clase Thread.

setName() Da un nombre o indentificador a cada objeto de la clase Thread creado. Esto también puede hacerse utilizando el constructor o mediante un super("nombre") en la clase heredera de Thread.

getName()
Devuelve un String con el nombre que se utilizó para designar el hilo. Si no se le dió nombre, obtendremos algo como ... Thread-0.

setPriority()
Establece la prioridad de ejecución de un hilo ante el planificador. En realidad se trata de un valor int que cubre el rango 1 ... 10 y que puede verse también mediante las constantes MIN_PRIORITY(1), NORM_PRIORITY(5) y MAX_PRIORITY(10).

Como ya hemos indicado, el planificador evalúa muchos parámetros a la hora de decidir cuál va a ser el hilo que efectivamente sea elegido para ejecución. El asignar una prioridad alta, SÓLO SUPONE aumentar las posibiliades de que un determinado hilo sea elegido.
Como norma a seguir, el planificador tratará siempre de ejecutar el hilo de mayor prioridad.

isAlive() Mediante un booleano, se indica si el hilo en cuestión sigue vivo o no, es decir, si ha finalizado completamente su método run().
Un hilo en estado new (sólo se ha creado el objeto de la clase que hereda de Thread) dará false al invocar sobre él a isAlive().

currentThread()
Devuelve una referencia al hilo que se está ejecutando en el momento actual.

EXCEPCIONES DE THREAD:

InterruptedException Métodos join(), wait(xxx) y sleep(xxx)

IllegalThreadStateException Si tratamos de volver a activar un hilo muerto o arrancar un hilo ya iniciado.

Los métodos notify() y notifyAll() no generan ninguna excepción, pero recordar que junto con wait(), SÓLO PUEDEN UTILIZARSE EN UN CONTEXTO SINCRONIZADO (el compilador no se quejará pero obtendremos una IllegalThreadStateException, lo mismo que si no sincronizamos sobre el elemento compartido).

EL MÉTODO MAIN

El propio método main (publis static void main(String args[]) {}) es en si mismo un hilo. Así, podemos crear en su interior otros hilos y lanzarlos a través de su correspondiente método star(). Igualmente, podemos hacer que main no finalice hasta que no se complete uno o más hilos, usando el join() sobre los mismos.

Cuando desde main invocamos el método run() de una determinada instancia de Thread, no estamos indicando al planificador que se encargue de gestionar dicho objeto como un hilo. En su lugar, el cuerpo o código del método run() invocado se ejecutará como parte del propio hilo main y NO como un hilo independiente.

Así, es necesario usar el método start() para que el planificador gestione la instancia del Thread como un verdadero hilo.

La ejecución del método run() de un hilo suele ser extremadamente rápida, por lo que si arrancamos distintos hilos para comprobar su funcionamiento, puede darse el caso de que el tiempo de CPU que asigne el planificador a cada de ellos, sea lo suficientemente alto como para que estos se completen, sin observarse en pantalla una mezcla de las distintas respuestas (cada una correspondiente a la ejecución de un determinado hilo). Si esto sucede, podemos optar por utilizar retardos (con sleep(xxx)) para comprobar cómo el planificador conmuta la ejecución de los mismos produciéndose dicha mezcla de respuestas.