1. Introducción curso Java

Java, el lenguaje orientado a objetos diseñado para ser multiplataforma y poder ser empleado el mismo programa en diversos sistemas operativos.
Esta característica, junto con la posibilidad de emplearlo para crear applets, e insertarlos en páginas HTML, o mediante servlets y páginas jsp, generar código HTML dinámico. Todo ello con la capacidad de acceder a bases de datos.
Java es un lenguaje relativamente sencillo, debido a que prácticamente toda la funcionalidad se encuentra en clases que forman parte del **API de java**. Constantemente están surgiendo nuevos apis, que proporcionan nuevas extensiones a las características del lenguaje.
Estas características, junto con el hecho de que sea un lenguaje libre, pudiéndose utilizar el compilador y la máquina virtual de forma gratuita, le augura un gran futuro.

2. Características del lenguaje java

En este módulo estudiaremos las características del lenguaje java, conociendo tanto su sintaxis, como el API de ampliación del lenguaje.

Características del lenguaje

Java es un lenguaje **orientado a objetos**, eso implica que su concepción es muy próxima a la forma de pensar humana. También posee otras características muy importantes:
Es un lenguaje que es compilado, generando ficheros de clases compilados, pero estas clases compiladas, son en realidad interpretadas por la máquina virtual de java. Siendo la máquina virtual de java la que mantiene el control sobre las clases que se estén ejecutando.
Es un lenguaje **multiplataforma**: El mismo código java que funciona en un sistema operativo, funcionará en cualquier otro sistema operativo que tenga instalada la máquina virtual java.
Es un lenguaje seguro: La máquina virtual, al ejecutar el código java, realiza comprobaciones de seguridad, además el propio lenguaje carece de características inseguras, como por ejemplo los punteros.
Gracias al API de java podemos ampliar el lenguaje para que sea capaz de, por ejemplo, comunicarse con equipos mediante red, acceder a bases de datos, crear páginas HTML dinámicas, crear aplicaciones visuales al estilo window, ...
Para poder trabajar con java es necesario emplear un software que permita desarrollar en java. Existen varias alternativas comerciales en el mercado: JBuilder, Visual Age, Visual Café,... y un conjunto de herramientas shareware, e incluso freeware, que permiten trabajar con java.

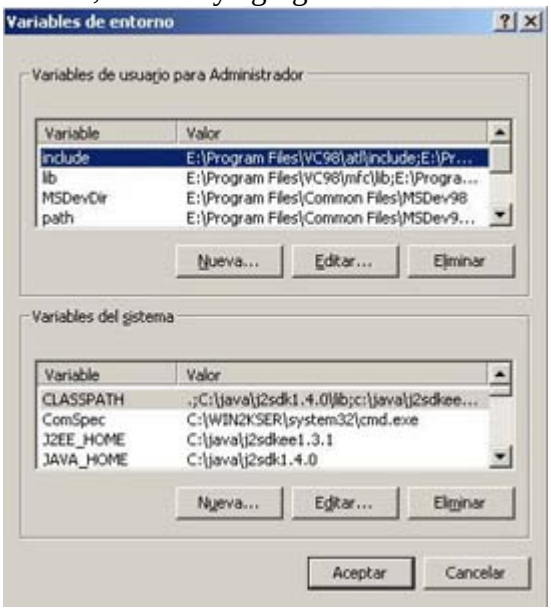
3. Características del lenguaje java (II)

Pero todas estas herramientas en realidad se basan en el uso de una herramienta proporcionada por **Sun**, el creador de java, que es el **Java Development Kit (JDK)**.
Nosotros nos centraremos en el uso de dicha herramienta.

Existen diversas versiones del JDK, siendo posible obtener cualquiera de dichas versiones desde la propia página de sun: http://java.sun.com siendo la más reciente **JSDK 1.4**
Una vez obtenida la máquina virtual hay que proceder a realizar la instalación, proceso en el que será solicitada la carpeta en la que se copiarán los ficheros del JDK. Supongamos que el nombre de dicha carpeta sea **c:\jsdk1.4**.
Tras ser instalada se generarán una serie de carpetas dentro de la carpeta **c:\jsdk1.4**, entre ellas cabe destacar:
bin: en ella se encuentran todos los programas ejecutables del jdk
lib: contiene las clases del api de java
Una vez realizado el proceso de instalación, tenemos que realizar la configuración de dos variables de entorno: **PATH** y **CLASSPATH**. Este proceso será distinto para windows 95 o 98, o para windows NT, 2000 o XP:
Windows 9X:
Localizar en el explorador el fichero c:\autoexec.bat, pulsar el botón derecho del ratón sobre el fichero y utilizar la opción Editar.
Ir al final del fichero agregar un salto de línea y teclear:
SET PATH = %PATH%;c:\jsdk1.4\bin
SET CLASSPATH = .;c:\jsdk1.4\lib
Windows NT,2000 o XP:
Pulsar sobre el icono de **MiPC** con el botón derecho del ratón, pulsar en la opción propiedades.
En windows NT pulsar sobre la pestaña Entorno, mientras que en Windows 2000 y XP hay que ir a la pestaña avanzado y sobre ella pulsar al botón variables de entorno.

4. Características del lenguaje java (III)

En la zona de variables del sistema agregar una nueva variable llamada **CLASSPATH** con el siguiente valor: .;c:\jsdk1.4\lib
También en la zona de variables del sistema localizar una variable ya existente llamada **PATH**, editarla y agregar al final del valor existente: ;c:\jsdk1.4\bin



PATH es una variable de entorno que le dice al sistema operativo donde puede encontrar los programas ejecutables, en este caso la hemos modificado para que encuentre los ejecutables del jdk.
CLASSPATH es una variable de entorno que le dice a la máquina virtual donde puede encontrar las clases que vamos a emplear o ejecutar.
1.- Escribir el código fuente correspondiente a las clases a emplear. Los ficheros generados tendrán la extensión .java
2.- Compilar el código fuente mediante la utilidad javac.exe, este paso generará las clases compiladas en ficheros con extensión .class
3.- Ejecutar la clase principal, para ello pasaremos el nombre de la clase a ejecutar a la aplicación java.exe.

5. Práctica con características del lenguaje java

Ahora veamos esos mismos pasos con más detalle:
Práctica:
Construyamos una clase que pida un número por teclado y muestre la tabla de multiplicar de dicho número. Lo primero es escribir el código de dicha clase, para ello emplearemos cualquier editor de texto, pero hay que guardar el fichero de texto como texto plano.
El fichero vamos a llamarlo **TablaMultiplicar.java** y lo vamos a almacenar en un directorio, por ejemplo c:\java
El código a escribir, y todavía no nos preocupamos por como es, será:

```
public class TablaMultiplicar{
public static void main(String arg[]){
int numero = Integer.parseInt(arg[0]);
for(int i = 1 ; i<=10 ; i++){
System.out.println(""+numero+" * "+i+" = "+(i*numero));
}
}
}
```

Una vez escrito, debemos guardar los cambios en el fichero fuente. Ahora debemos compilar el fuente, para ello iremos a una ventana de Consola y mediante cd nos moveremos hasta el directorio donde se encuentre el fichero fuente:

```
cd c:\java
```

Una vez allí debemos de compilar el código fuente:

```
javac TablaMultiplicar.java
```

Esto provocará, si no hay errores en el código, que se genere un fichero llamado TablaMultiplicar.class

Y ya podemos ejecutar la aplicación:

```
java TablaMultiplicar 6
```

Ahora probaremos a compilar un fuente con un error de sintaxis dentro del código, el fichero TablaMultiplicarError.java lo obtendremos de la carpeta de ejemplos del manual.

```
javac TablaMultiplicarError.java
```

Esto provocará que se muestre por pantalla el siguiente error:

```
TablaMultiplicarError.java:3: ';' expected
```

```
int numero = Integer.parseInt(arg[0])
^
1 error
```
dentro del cuál podemos apreciar que existe un error en el fichero TablaMultiplicarError.java en la línea 3, siendo además el error que se esperaba un ';'. Después aparece la línea en la cuál está situado el error.

6. Introducción a la programación orientada a objetos

Introducción a la programación orientada a objetos
Si nos fijamos en la vida real, podremos apreciar que casi todo se compone de objetos (coche, lapicero, casa y, por qué no, personas). Todos entendemos que tanto un BMW verde, como un SEAT rojo son coches, pero son distintos en su aspecto físico. Pero a pesar de esa diferencia de aspecto entendemos que ambos tienen unas características comunes como son poseer ruedas, un volante, asientos,... Esas características comunes son las que hacen pensar en el concepto coche en ambos casos. A partir de ahora podemos interpretar que el BMW y el SEAT son objetos, y el tipo o clase, de esos objetos es coche.
Además, podemos suponer que la clase de un objeto describe como es dicho objeto.
Por ejemplo:
clase Coche: 4 ruedas, 1 volante, 5 asientos, acelerar, frenar, girar
Como podemos ver, esta clase no sólo describe qué cosas posee un coche, sino que también describe qué acciones puede realizar un coche (acelerar, frenar y girar). Es decir, una clase define los atributos y las acciones (o métodos) que puede realizar un objeto de la clase.
Además podemos comprobar que un objeto puede estar formado por otros objetos, por ejemplo el coche posee 4 objetos de la clase Rueda.
En el mundo real los objetos se relacionan entre sí, un objeto puede pedir a otro que realice alguna acción por él. En la vida real una persona acelera, pero lo hace pidiéndoselo al coche, que es quien realmente sabe acelerar, no la persona. El hecho de que un objeto llame a un método de otro objeto, se indica diciendo que el primer objeto ha enviado un mensaje al segundo objeto, el nombre del mensaje es el nombre de la función llamada.

7. Instancias

Podemos interpretar que una clase es el plano que describe como es un objeto de la clase, por tanto podemos entender que a partir de la clase podemos fabricar objetos. A ese objeto construido se le denomina **instancia**, y al proceso de construir un objeto se le llama **instanciación**.
Cuando se construye un objeto es necesario dar un valor inicial a sus atributos, es por ello que existe un método especial en cada clase, llamado **constructor**, que es ejecutado de forma automática cada vez que es instanciada una variable. Generalmente el constructor se llama igual que la clase y no devuelve ningún valor. Análogamente, **destructor** es un método perteneciente a una clase que es ejecutado de forma automática cuando un objeto es destruido. **Java** no soporta los destructores. Es posible que exista más de un constructor en una clase, diferenciados sólo en los parámetros que recibe, pero en la instanciación sólo será utilizado uno de los constructores.

Es recomendable emplear el constructor para inicializar las variables internas del objeto, o para obtener recursos, mientras que el destructor se suele emplear para liberar esos recursos obtenidos en el constructor.
Imaginemos ahora un televisor, nosotros que poseemos el mando a distancia podemos manejar el volumen del televisor, pero no nos haría gracia que otra persona pudiese manejar el volumen a su antojo. Lo mismo sucede con los objetos: un objeto que no quiere que otro objeto llame a un método o acceda a un atributo, debe definir a dichos atributos y métodos como privados. En cambio si los definimos como públicos, cualquier objeto podrá utilizarlos.
Pensemos ahora en un coche deportivo y en un coche utilitario, ambos son coches, pero hacemos la distinción deportivo y utilitario porque son diferencias lo suficientemente importantes como para ser tenidas en cuenta, pero no tanto, como para pensar que uno es un coche, pero el otro no. Esta situación se interpreta pensando en que existen dos clases: Deportivo y Utilitario, pero que ambas, además de sus propias características (atributos y métodos), poseen las de la clase coche, esto es la herencia. Denominaremos clase base a la clase de la cuál heredamos y derivada a la clase que hereda de la clase base.

8. Herencias

Existen dos tipos de **herencia sencilla y múltiple**. Sencilla significa que sólo heredamos de una clase base, mientras que múltiple indica que tenemos varias clases base (por ejemplo un hidroavión hereda de barco y de avión). Java sólo soporta herencia simple.
Al utilizar la herencia aparecen dos conceptos: **super y this**, this representa al objeto completo, en cambio super, sólo representa la parte heredada de la clase base.
Cuando se hereda nos encontramos frente a un pequeño problema: ¿Qué sucede cuando se hereda un método de la clase base, el cuál estamos redefiniendo en la clase derivada? Esto es un caso de sobrescritura de métodos. La solución es simple, cuando estemos ejecutando el método de un objeto derivado se llamará al método de su propia clase, es decir el redefinido. Si lo que se quiere es emplear el método de la clase base, hay que emplear una técnica que consiste en usar: super.método().
Los constructores no son heredados, pero sí llamados. Es decir, cuando se construye un objeto de la clase derivada se llama al constructor de la clase derivada, pero antes de comenzar a ejecutarse se llama al constructor de la clase base, que tras ejecutarse continua la ejecución del constructor de la clase derivada.
Se puede elegir qué constructor de la clase base es llamado, generalmente llamando al método super(), que representa al constructor de la clase base, pero al pasar parámetros distintos, seleccionamos qué constructor de la clase base queremos llamar.
Supongamos ahora que tenemos algo que es capaz de encenderse, de apagarse, de iniciar una reproducción, de parar una reproducción, sin duda todos pensamos en un reproductor, pero por esa descripción encajan objetos como reproductor de cassete, reproductor de CD, el vídeo,...
Llamaremos a la descripción **interfaz**, y los objetos que cumplen ese interfaz (es decir, tienen todas las funciones que definen el interfaz) diremos que **implementan el interfaz**.
Pensemos ahora en el hecho que una persona que sabe iniciar una reproducción de un CD también sabe iniciar una reproducción de vídeo, en ambos casos debe de encender el objeto, iniciar la reproducción, parar la reproducción y apagar el reproductor. Eso significa que

para la persona es transparente el tipo (clase) real del objeto reproductor que posea, ya que la persona sabe que puede ponerlo en marcha, apagarlo, sin necesidad de conocer la clase real, tan sólo debe de saber que es un objeto del tipo reproductor.

9. Polimorfismo

Este hecho de que un objeto pertenezca a una clase, pero que pueda cumplir uno o más interfaces es muy similar al hecho de que una persona (un objeto) es hija (un interfaz), posiblemente tía (otro interfaz) o incluso abuela (otro interfaz). Con esto lo que se indica es que un interfaz sólo tiene interés para quien lo necesite usar, es decir, el interfaz tío sólo lo necesitan los objetos sobrinos, no un objeto de tipo policía.
Esta característica de que varios objetos de distintas clases puedan recibir el mismo mensaje y ser capaces de responderlo, es conocido como **polimorfismo**.

10. Sintaxis del lenguaje

Sintaxis del lenguaje
El lenguaje se basa en pensar que hay en el mundo real objetos y esos objetos tienen un tipo, o clase. Por ello el lenguaje se basa en clases, que describen como son los objetos. Por ejemplo, el lenguaje tiene una clase que describe ficheros, una que describe cadenas de texto, o bien nosotros podemos crear clases, como por ejemplo la clase Persona, que describe los datos que interesan de una persona.
Por ello siempre para comenzar a trabajar con un programa java hay que crear una clase:

```
public class TablaMultiplicar{
}
```

Además se deben de cumplir las siguientes características:
La clase se debe de llamar exactamente igual que el fichero que la contiene.
La clase que se llama igual que el fichero debe de estar precedida de la palabra public.
Cuando se intenta ejecutar una clase java, la máquina virtual lo que hace es llamar a un método especial llamado main que debe estar dentro de la clase a ejecutar:

```
public class TablaMultiplicar{
public static void main(String arg[]){
}
}
```

Y es dentro de la función main donde escribiremos el código que queremos que se ejecute:

```
public class TablaMultiplicar{
public static void main(String arg[]){
int numero = Integer.parseInt(arg[0]);
for(int i = 1 ; i<=10 ; i++){
System.out.println(""+numero+" * "+i+" = "+(i*numero));
}
}
}
```

11. Objetos y clases

Objetos y clases
Java es un lenguaje totalmente orientado a objetos, esto representa que posee instrucciones y sintaxis específicas para la programación orientada a objetos. Además, en java existe el concepto de jerarquía de herencia , es decir, que todas las clases deben de heredar de otra clase para formar todas ellas un árbol invertido. La clase raíz de este árbol es la clase java.lang.Object y todas las clases heredan de ella directa o indirectamente. Adicionalmente las clases son colocadas en carpetas para facilitar su ordenación y el trabajo con ellas, dichas carpetas (paquetes es el término más exacto) formarán parte del nombre de la propia clase, por ejemplo, java.lang.Object significa que existe una carpeta llamada lang, dentro de la cuál existe otra carpeta llamada lang, dentro de la cuál existe una clase llamada Object.
Estructura de un fichero java
Antes de ver la estructura de un fichero, veamos primero como se realizan comentarios que no se compilan, en este lenguaje hay dos tipos: hasta fin de línea (precedido por //) y el comentario de bloque (comienza por /* y acaba por */).

```
//Un comentario hasta fin de línea
/* Un comentario
en varias
líneas */
```

12. Estructura de un fichero java

En java, a la hora de crear una clase, primero se debe de crear un fichero con **extensión java**, la estructura de este fichero es:

```
[package nombrePaquete;]
[import nombreAImportar 1;]
...
[import nombreAImportar N;]
clase 1
...
[clase N]
```

Como podemos apreciar en el fichero java, puede aparecer una instrucción **package**, dicha instrucción lo que hace es indicar que la/s clase/s definida/s en este fichero estarán situadas en la carpeta (paquete), indicado por nombrePaquete. Si nombrePaquete es util.compresor significa que la clase estará dentro de una carpeta llamada compresor, que a su vez estará dentro de una carpeta llamada útil.
La instrucción import lo que indica es la intención de utilizar dentro de este fichero java, un recurso llamado nombreAImportar, donde nombreAImportar puede ser:
El nombre completo de una clase o interfaz (como java.net.Socket): se usará dicha clase dentro del fichero java.
El nombre de un paquete finalizado en .* (como java.net.*): se usarán todas las clases e interfaces que hay dentro del paquete, pero no las de las subcarpetas que haya dentro del paquete, es decir java.awt.* no incluye las clases que haya dentro de java.awt.event.
Podemos tener tantos import como consideremos necesario.

Después de las instrucciones import, aparecen las definiciones de las clases que estemos construyendo, pero con las siguientes características:
Tiene que haber definida al menos una clase o un interfaz.
Puede haber tantas clases como queramos, pero sólo puede haber una de ellas con el modificador public (lo veremos más adelante).
En caso de haber una clase definida public, esta clase debe llamarse obligatoriamente igual que el nombre del fichero, respetando mayúsculas y minúsculas, pero sin la extensión del fichero.

13. Definición de una clase

Definición de una clase
Aquí tenemos la sintaxis para definir una clase:
modifAcceso modifClase class nombreClase [extends nombreBase] [implements listaInterfaces]
{
Atributo 1
Atributo N
método 1
método N
}
Donde **nombreClase** es el nombre de la clase, cualquier nombre, pero respetando las reglas de nomenclatura del lenguaje.
modifAcceso puede ser uno de los siguientes valores:
public: indica que la clase es pública, y por tanto que puede ser utilizada desde cualquier otra clase, con independencia de si están en el mismo paquete o no.
Sin especificar: indica que la clase tiene visibilidad de paquete, es decir, sólo la pueden usar las clases que se encuentren en el mismo paquete que dicha clase.
modifClase indica características específicas de la clase que estamos construyendo, los posibles valores son:
abstract: indica que a la clase le falta, al menos uno, el código de algún método. Posee el método (abstracto), pero no tiene el código de ese método, siendo responsabilidad de las clases derivadas proporcionar el código de dicha clase. Una clase abstracta no se puede instanciar.
final: se emplea para evitar que esta clase pueda ser derivada.
extends: se utiliza para indicar que la clase hereda de nombreBase, en java sólo se permite heredar de una única clase base. En caso de no incluir la cláusula extends, se asumirá que se está heredando directamente de la clase java.lang.Object
implements: indica que esta clase es de los tipos de interfaz indicados por listaInterfaces, pudiendo existir tantos como queramos separados por comas. Esta cláusula es opcional.

14. Definición de atributos de una clase (I)

Definición de atributos de una clase
Los atributos de una clase se definen según esta sintaxis:
[modifVisibilidad] [modifAtributo] tipo nombreVariable [= valorInicial] ;

Donde **nombreVariable** es el nombre que daremos a la variable, siendo un nombre válido según las normas del lenguaje: por convención, en Java, los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra mayúscula).
Un nombre de variable Java: debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. Unicode es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos. Unicode permite la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.
No puede ser el mismo que una palabra clave
No deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.
tipo es el tipo de la variable, pudiendo ser un tipo básico o un objeto de una clase o de un interfaz. También puede ser una matriz o vector.

15. Definición de atributos de una clase (II)

Veamos ahora que tipos de datos básicos existen en el lenguaje y sus características:
Tipo Tamaño/Formato Descripción
(Números enteros)
byte 8-bit complemento a 2 Entero de un Byte.
short 16-bit complemento a 2 Entero corto.
int 32-bit complemento a 2 Entero.
long 64-bit complemento a 2 Entero largo.
(Números reales)
float 32-bit IEEE 754 Coma flotante de precisión simple.
double 64-bit IEEE 754 Coma flotante de precisión doble.
(otros tipos)
char 16-bit Carácter Un sólo carácter (Unicote).
boolean true o false Un valor booleano (verdadero o falso).
Tenemos un ejemplo del uso de los tipos de datos en Tipos.java
modifVisibilidad indica desde que parte del código se puede acceder a la variable:
public: indica que es un atributo accesible a través de una instancia del objeto.
private: indica que a través de una instancia no es accesible el atributo. Al heredar el atributo se convierte en inaccesible.
protected: indica que a través de una instancia no es accesible el atributo. Al heredar si se puede usar desde la clase derivada.
Sin especificar: indica visibilidad de paquete, se puede acceder a través de una instancia, pero sólo desde clases que se encuentren en el mismo paquete.
valorInicial permite inicializar la variable con un valor.
Se permite definir más de una variable, separándolas por coma, por ejemplo:
public int a = 5, b, c = 4;

16. Definición de atributos de una clase (III)

modifAtributos son características específicas del atributo, son:
static: El atributo pertenece a la clase, no a los objetos creados a partir de ella.
final: El atributo es una constante, en ese caso debe de tener valor inicial obligatoriamente. Por convenio en java las constantes se escriben en mayúsculas.
transient: Marca al atributo como transitorio, para no ser serializado. Lo emplearemos en java beans.
volatile: es un atributo accedido de forma asíncrona mediante hilos, con este atributo se lo notificamos a java.
En java definir un atributo de un tipo básico o tipo String significa que podemos acceder a dichas variables de forma directa (ejemplo PruebaVariable1.java):
int a = 25;
a = 34;
en cambio intentemos definir y emplear una variable del tipo, por ejemplo, Thread:
Socket c=null;
c.close();
si tratamos de compilarlo (fichero PruebaVariable2.java) todo irá bien, pero al ejecutarlo aparecerá un error:
Exception in thread "main" java.lang.NullPointerException
esto nos quiere indicar que hemos intentado emplear una variable que no apuntaba a un objeto válido, sino a null, y hemos intentado llamar a una función del objeto inexistente.

17. Definición de atributos de una clase (IV)

Es decir, en java las variables de tipo básico son el nombre de una zona de memoria en la cuál podemos almacenar valores, pero que en cambio, las variables de tipo objeto son en realidad referencias (punteros o alias) de objetos.
Una variable de tipo objeto no es un objeto completo, sino tan solo almacena la situación del objeto en la memoria del equipo. Esto es muy similar a lo que ocurre con las casas y las direcciones de dichas casas: la dirección calle Alcalá 950 es una dirección válida, pero no podemos mandar cartas a dicha dirección porque es...un descampado!!!
Lo mismo sucede con los objetos, podemos tener una variable para referirnos a objetos, pero la variable puede que no apunte a ningún objeto y por tanto no la puedo emplear para intentar acceder a un método o a un atributo del objeto referenciado por la variable, sencillamente porque no existe el objeto referenciado.
Una variable que no apunta a un objeto se asume que tiene un valor especial llamado null, e incluso podemos asignar el valor null a la variable:
Thread t = null;
Es por ello que se deben construir objetos y asignárselos a las referencias, usando la palabra clave new. new permite crear un objeto a partir de la descripción de la clase que le pasamos como argumento, por ejempo:
new Persona()
Conseguimos crear un objeto de la clase Persona, los paréntesis permiten especificar qué constructor estamos llamando al crear el objeto (veremos constructores más adelante).

Pero al crear un objeto persona como en el código anterior lo estamos creando como un objeto anónimo, es decir sin asignar el objeto a una variable de tipo referencia, desde la cuál poder referirnos al objeto y poder llamar a sus métodos y atributo, por ello lo más habitual será asignar el objeto a una variable como en:
Persona p = new Persona();
y así poder acceder al objeto Persona recien creado:
p.nombre = "Alberto";

18. Práctica con definición de atributos de una clase

Práctica
Vamos a construir la clase Persona, pero no estará completa hasta que no la completemos con características que iremos viendo en capítulos posteriores. Fabricaremos la clase Persona poco a poco, para ello:
Construir el fichero Persona.java
Agreguemos la clase dentro del fichero:
public class Persona
{
}
Agreguemos a la clase los atributos de persona (en este caso seleccionaremos dos atributos):
private String nombre = null;
private int edad;

Como resultado final obtendremos:
public class Persona
{
private String nombre = null;
private int edad;
}
Ahora vamos a crear una clase que emplearemos única y exclusivamente para que tenga la función main, y así poder escribir código que queremos ejecutar. Llamaremos a dicha clase Arranque, por tanto:
Creamos el fichero Arranque.java
Agreguemos la clase y el método main en su interior:
public class Arranque {
public static void main (String arg[]){
}
}
Ahora agregaremos dentro de la función main el siguiente código:
Persona per1 = new Persona();
//per1.nombre = "Luis";
//System.out.println(per1.nombre);
en el que declaramos una variable de tipo Persona y le damos el valor de un nuevo objeto de la clase Persona, en la segunda línea asignamos al atributo nombre del objeto per1 el

valor Luis, y en la tercera línea mostramos por pantalla (gracias al método System.out.println) el valor de dicha variable.
La segunda y tercera líneas están comentadas, debido a que el atributo nombre está definido como privado en la clase Persona.
Ejercicio: Descomentar la segunda y tercera línea y compilar las clases, leer el error de compilación provocado. Cambiar el tipo del atributo nombre de private a public y compilar y ejecutar. Volver a dejar todo como al comienzo (Comentada la línea y con nombre como variable privada).

19. Definición de métodos de una clase (I)

Definición de métodos de una clase
Para definir los métodos se emplea la siguiente sintaxis:
[modifVisibilidad] [modifFunción] tipo nombreFunción (listaParámetros) [throws listaExcepciones]
{
}
Para **modifVisibilidad** se aplica las mismas normas que para atributos:
public: indica que es un método accesible a través de una instancia del objeto.
private: indica que a través de una instancia no es accesible el método. Al heredar el método se convierte en inaccesible.
protected: indica que a través de una instancia no es accesible el método. Al heredar si se puede usar desde la clase derivada.
Sin especificar: indica visibilidad de paquete, se puede acceder a través de una instancia, pero sólo de clases que se encuentren en el mismo paquete.
nombreFunc debe de ser un identificador válido en el lenguaje.
tipo es el tipo del valor devuelto por la función, pudiendo ser:
Un tipo básico.
Un objeto de una clase o interfaz. En este tipo de objetos se incluyen las matrices o vectores.
void, en el caso de no devolver ningún valor.

20. Definición de métodos de una clase (II)

listaParámetros es la lista de los parámetros que tomará la función separados por comas y definidos cada uno de ellos como:
tipo nombreParámetro
modifFunción puede tener los siguientes valores:
static: el método pertenece a la clase, no a los objetos creados a partir de la clase.
final: el método no puede ser sobrescrito en una clase derivada.
abstract: En esta clase no se proporciona el código para la función, se debe de proporcionar en alguna clase derivada. En el caso de poseer un método abstracto la clase debe de llevar a su vez el modificador abstract. En caso de ser abstracto un método, se debe de sustituir las llaves que contienen el código por un punto y coma.
native: Es un método no escrito en java, sino en código nativo, que será usado en java como un método propio de java.

synchronized: Es un método que sólo puede ser ejecutado por un hilo, y hasta que ese hilo no acabe la llamada al método, no puede comenzar la llamada al método otro hilo. Lo emplearemos al trabajar con hilos.
La cláusula opcional throws es empleada para indicar que dentro del método se pueden generar errores en su ejecución, y que debemos estar preparados para tratarlos. listaExcepciones es el nombre de todos esos posibles errores, su utilización la veremos en el punto dedicado a la gestión de errores mediante try y catch.
El método posee un par de llaves, dentro de las cuales estará el código que se ejecutará al ser llamada la función. Dicho código estará formado por instrucciones válidas en el lenguaje, finalizadas generalmente por punto y coma.

21. Práctica con Definición de métodos de una clase

Práctica
Continuamos con la clase Persona:
Vamos a crear 4 métodos que permitan acceder a los atributos privados de la clase persona.
Los agregamos dentro de la clase:

```
public int getEdad()
{
return edad;
}
public void setEdad(int laEdad)
{
edad = laEdad;
}
public String getNombre()
{
return nombre;
}
public void setNombre(String elNombre)
{
nombre = elNombre;
}
```

22. Práctica con definición de métodos de una clase (II)

Regresemos a la clase Arranque, y localicemos dentro del método main la línea comentada
//per1.nombre = "Luis"
Vamos a sustituirla (como está comentada no es necesario borrarla) por:
per1.setNombre("Luis");
Lo mismo vamos realizar en la segunda línea comentada, sustituyamosla por:
System.out.println(per1.getNombre());
Compilamos y ejecutamos la clase Arranque.
Ahora estaremos pensando que para que necesitamos métodos públicos para acceder a variables privadas cuando es más sencillo tener variables públicas y acceder a ellas

libremente sin tener que emplear funciones. Vamos a modificar algunas cosas para entender porque es más interesante emplear funciones de acceso a atributos privados.
Modificar el código de la función setEdad para que sea:

```
if(laEdad<0){
System.out.println("Una persona no puede tener una edad negativa.");
}
else
{
edad = laEdad;
}
```

Volvamos a la clase Arranque, en el método main, y además del código que ya hay en su interior agregaremos:

```
per1.setEdad(4);
System.out.println(per1.getEdad());
per1.setEdad(-35);
```

Compilar y ejecutar la clase Arranque.

23. Definición de constructores de una clase

Definición de constructores de una clase.
Cuando se construye un objeto es necesario inicializar sus variables con valores coherentes, imaginemos un objeto de la clase Persona cuyo atributo color de pelo al nacer sea verde, un estado incorrecto tras construir el objeto persona. La solución en los lenguajes orientados a objetos es emplear los constructores. Un constructor es un método perteneciente a la clase que posee unas características especiales:
Se llama igual que la clase.
No devuelve nada, ni siquiera void.
Pueden existir varios, pero siguiendo las reglas de la sobrecarga de funciones.
De entre los que existan, tan sólo uno se ejecutará al crear un objeto de la clase.
Dentro del código de un constructor generalmente suele existir inicializaciones de variables y objetos, para conseguir que el objeto sea creado con dichos valores iniciales.
Para definir los constructores se emplea la siguiente sintaxis:

```
[modifVisibilidad] nombreConstructor (listaParámetros) [throws listaExcepciones]
{
}
```

24. Definición de constructores de una clase (II)

Para modifVisibilidad se aplica las mismas normas que para atributos y métodos:
public: indica que es un método accesible a través de una instancia del objeto.
private: indica que a través de una instancia no es accesible el método. Al heredar el método se convierte en inaccesible.
protected: indica que a través de una instancia no es accesible el método. Al heredar si se puede usar desde la clase derivada.
Sin especificar: indica visibilidad de paquete, se puede acceder a través de una instancia, pero sólo de clases que se encuentren en el mismo paquete.

nombreConstructor debe de coincidir con el nombre de la clase.
listaParámetros es la lista de los parámetros que tomará la función separados por comas y definidos cada uno de ellos como:
tipo nombreParámetro
La cláusula opcional throws es empleada para indicar que, dentro del método, se pueden generar errores en su ejecución, y que debemos estar preparados para tratarlos.
listaExcepciones es el nombre de todos esos posibles errores, su utilización la veremos en el punto dedicado a la gestión de errores mediante try y catch.
El constructor posee un par de llaves, dentro de las cuales estará el código que se ejecutará al ser llamada la función. Dicho código estará formado por instrucciones válidas en el lenguaje, finalizadas generalmente por punto y coma.

25. Prácticas con Definición de constructores de una clase

Practicas:
Vamos a agregar a la clase Persona un par de constructores, uno que la inicialice asignando a la edad un valor 0 y al nombre "anónimo", y otro que permita asignar al nombre un parámetro recibido en el constructor:

```
public Persona(){
edad = 0;
nombre = "anónimo";
}
public Persona(String nuevoNombre){
edad = 0;
nombre = nuevoNombre;
}
```

y vamos a crear una clase ArranqueConstructor con el siguiente código:

```
public class ArranqueConstructor {
public static void main (String arg[]){
Persona per1 = new Persona();
System.out.println( per1.getNombre());
System.out.println(per1.getEdad());
Persona per2 = new Persona("Luis");
System.out.println( per2.getNombre());
System.out.println(per2.getEdad());
}
}
```

26. La herencia en java (I)

La herencia en java
Java permite el empleo de la herencia , característica muy potente que permite definir una clase tomando como base a otra clase ya existente. Esto es una de las bases de la reutilización de código, en lugar de copiar y pegar.

En java, como ya vimos la herencia se especifica agregando la claúsula extends después del nombre de la clase. En la claúsula extends indicaremos el nombre de la clase base de la cuál queremos heredar.
Al heredar de una clase base, heredaremos tanto los atributos como los métodos, mientras que los constructores son utilizados, pero no heredados.
Practicas:
Construyamos la clase Taxista.java con el siguiente código:

```
public class Taxista extends Persona {
private int nLicencia;
public void setNLicencia(int num)
{
nLicencia = num;
}
public int getLicencia()
{
return nLicencia;
}
}
```

27. La herencia en java (II)

Y construyamos ArranqueTaxista.java:

```
public class ArranqueTaxista {
public static void main (String arg[]){
Taxista tax1 = new Taxista();
tax1.setNombre("Luis");
tax1.setEdad(50);
System.out.println( tax1.getNombre());
System.out.println(tax1.getEdad());
}
}
```

Ahora intentemos usar el constructor que existía en la clase Persona que recibia el nombre de la persona y vamos a usarlo para la clase Taxista. Para ello construyamos la clase ArranqueTaxista2.java:

```
public class ArranqueTaxista2 {
public static void main (String arg[]){
Taxista tax1 = new Taxista("Jose");
tax1.setEdad(50);
System.out.println( tax1.getNombre());
System.out.println(tax1.getEdad());
System.out.println(tax1.getNLicencia());
}
}
```

28. La herencia en java (III)

Se genera un error de compilación, debido a que los constructores no se heredan, sino que hay que definir nuestros propios constructores. Agreguemos en la clase Taxista los siguientes constructores:

```
public Taxista(int licencia)
{
super();
nLicencia = licencia;
}
public Taxista(String nombre,int licencia)
{
super(nombre);
nLicencia = licencia;
}
```

Ahora si podremos compilar y ejecutar la clase ArranqueTaxista2. La llamada al método super indica que estamos llamando a un constructor de la clase base (pensemos que un Taxista antes que Taxista es Persona y por tanto tiene sentido llamar al constructor de Persona antes que al de Taxista). Además gracias al número de parámetros de la llamada a super podemos especificar cuál de los constructores de la clase base queremos llamar.
En java se pueden emplear dos palabras clave: this y super .
Como vimos en la introducción a la programación orientada a objetos, this hace alusión a todo el objeto y super hace alusión a la parte heredada, por ello empleamos super para referenciar al constructor de la clase base.

29. Prácticas con la herencia en java

Practicas:
Ahora vamos a agregar la función getNombre dentro de la clase Taxista, es decir, tenemos la misma función en Persona y en Taxista:

```
public String getNombre()
{
return "Soy un taxista y me llamo: " + super.getNombre();
}
```

Compilamos Taxista y ejecutamos ArranqueTaxista2. Veremos que el mensaje que aparece en pantalla demuestra que la función getNombre llamada es la de del tipo real del objeto construido, en este caso la de la clase derivada que es Taxista.
Tambien apreciamos que para acceder al atributo nombre es necesario acceder al método getNombre de la clase base (y por ello emplear super).
En java los atributos y métodos de la clase base pueden cambiar su modificador de visibilidad dentro de la clase derivada, la siguiente tabla recoge dichos cambios:
Modificadores en la clase base
public
private
protected
paquete

En la clase derivada se transforman en
public
inaccesible
protected
paquete
Inaccesible significa que, a pesar de haber sido heredado, no hay permisos en la clase derivada para poder acceder a dicho elemento inaccesible, pero aún así, se pueden llamar a métodos de la clase base que si pueden acceder y modificar al elemento.
Recordemos que protected significa que es private, pero que al heredar no se hace inaccesible, es decir que desde la clase derivada se puede acceder.

30. Matrices, arrays o vectores en java

Matrices, arrays o vectores en java.
Java posee la capacidad de definir un conjunto de variables del mismo tipo agrupadas todas ellas bajo un mismo nombre, y distinguiéndolas mediante un índice numérico.
Para definir un array en java es como definir una variable o atributo, pero al especificar el tipo lo que hacemos es colocar un par de corchetes [] para indicar que lo que estamos definiendo es un array. Por ejemplo:
public int [] losValores;
en la que definimos un array de enteros llamado losValores. Vamos a intentar realizar un ejemplo para ver como funciona:
Práctica:
Crear el fichero Array.java
Agregar el siguiente código en el fichero:

```
public class Array
{
public static void main(String arg[])
{
int [] losValores = null;
losValores[4] = 100;
System.out.println(losValores[4]);
}
}
```

Compilamos el código, ejecutemos y...error!!! Parece extraño que el error sea NullPointerException, pero tiene sentido, recordemos que una variable java, que no sea de tipo básico es una referencia que puede apuntar a objetos y por tanto losValores también es una referencia y debe de apuntar a objetos de tipo array de enteros. Es decir, el código de la función main es necesario modificarlo:

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

31. Práctica con matrices, arrays o vectores (I)

Práctica:
Crear el fichero Array.java
Agregar el siguiente código en el fichero:

```
public class Array
{
public static void main(String arg[])
{
int [] losValores = null;
losValores[4] = 100;
System.out.println(losValores[4]);
}
}
```

Compilamos el código, ejecutemos y...error!!! Parece extraño que el error sea **NullPointerException**, pero tiene sentido, recordemos que una variable java, que no sea de tipo básico es una referencia que puede apuntar a objetos y por tanto losValores también es una referencia y debe de apuntar a objetos de tipo array de enteros. Es decir, el código de la función main es necesario modificarlo:

```
int [] losValores = new int[10];
losValores[4] = 100;
```

La modificación consiste básicamente en asignar a la variable losValores un objeto de tipo array de enteros. La sintaxis para crear un objeto de tipo array es:

```
new tipo[cantidad]
```

Donde tipo es el tipo de datos que contendrá el array. cantidad es el número máximo de elementos que podemos almacenar dentro del array. A la hora de acceder a las posiciones del array hay que tener en cuenta que la primera posición es 0 y la última cantidad-1.

32. Práctica con matrices, arrays o vectores (II)

Ahora vamos a tratar de escribir el mismo código, pero empleando en lugar del tipo int, el tipo Persona:
Práctica:
Crear el fichero Array2.java
Agregar el siguiente código en el fichero:

```
public class Array2
{
public static void main(String arg[])
{
Persona [] lasPersonas = new Persona[10];
lasPersonas[4].setNombre("Luis");
System.out.println(lasPersonas [4].getNombre());
}
}
```

Veamos: definimos un array de 10 objetos de tipo Persona y lo asignamos a la variable lasPersonas. Accedemos a la persona 4 del array y la asignamos un nombre al objeto 4

mediante el método setNombre de la clase Persona. Posteriormente mostramos por pantalla el nombre del objeto Persona 4.
Compilemos y ejecutemos: ERROR!!! Al ejecutar genera el error NullPointerException. La explicación es sencilla: Al crear un array de objetos en realidad el array no contiene objetos, sino que contiene variables de tipo referencia para apuntar a los objetos.
Arreglemos nuestro código para poder ejecutarlo de nuevo:

```
Persona [] lasPersonas = new Persona[10];
lasPersonas[4] = new Persona();
lasPersonas[4].setNombre("Luis");
System.out.println(lasPersonas [4].getNombre());
```

Si es necesario se puede averiguar el tamaño de cualquier array java, mediante de un atributo público que poseen todos los objetos de tipo array, independientemente del tipo de objetos que almacene dicho array en su interior. Este atributo se denomina length y para usarlo:

```
miArray.length
```

En la siguiente práctica usaremos dicho atributo para crear un bucle (los veremos más adelante) que permita mostrar por pantalla todos los elementos que hay en un array llamado arg.

33. Práctica con matrices, arrays o vectores (III)

Práctica:
Crear el fichero Array3.java
Lo que vamos a hacer es tratar de ejecutar nuestra clase Array3, pero en lugar de ejecutarla mediante: "java Array3" vamos a ejecutarla mediante "java Array3 Esto es una prueba", es decir, vamos a pasar argumentos a nuestro programa. Dichos argumentos son recibidos dentro del array llamado arg que existe en la definición del método main.
Agreguemos el siguiente código dentro del fichero Array3.java:

```
public class Array3
{
public static void main(String arg[])
{
System.out.println( "Hay " + arg.length + " parametros." );
System.out.println( "Los parámetros son: " );
int i = 0;
while(i<arg.length){
System.out.println( "parámetro " + i + ": " + arg[i]);
i++;
}
}
}
```

34. Instrucciones del lenguaje, instrucción simple

Instrucciones del lenguaje
Existen varios tipos de instrucciones en java: Instrucción simple, Instrucción condicional, Instrucción iterativa y Instrucción simple
Una instrucción simple debe finalizar en punto y coma, y puede ser una expresión con operadores, una llamada a un método, una declaración de variable o una instrucción compuesta por varias instrucciones simples:

```
int a = 5; //Declaración de variable
System.out.println(a); //Llamada a método
a = a+4; // Expresión con operadores
System.out.println(++a); //instrucción compuesta de llamada a método y operador ++
```

Para definir una variable se emplea la misma sintaxis que para la definición de un atributo en una clase, pero sin utilizar los modificadores de visibilidad, ni los modificadores de atributos.

35. Operadores aritméticos

Además, en una instrucción simple pueden aparecer operadores, los hay de dos tipos: los que actúan sobre un operador, o los que lo hacen sobre dos. Los operadores los clasificaremos por su empleo:
Operadores aritméticos

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2 (*)
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

(*) En java también se emplea el operador + para concatenar cadenas de texto.

36. Operadores relacionales

Operadores relacionales:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Nota: Los operadores relacionales siempre devuelven un valor booleano.
Hay que hacer especial mención a los operadores == y !=, además de ser empleados en los tipos básicos del lenguaje pueden ser utilizados para comparar dos objetos, o más específicamente, comparar dos referencias a objetos. Al comparar dos referencias a objetos

lo que realmente se comprueba es si ambas referencias apuntan al mismo objeto o no. No confundir esto con comparar dos referencias de tipo String, no se compararía si el texto es el mismo, sino si ambas referencias apuntan al mismo objeto String. Para comparar String es necesario hacerlo mediante los métodos que para ellos existen dentro de la clase String, como por ejemplo equals:

```
String cad1 = "Texto";
String cad2 = "Texto2";
boolean resultado = cad1.equals(cad2);
```

37. Operadores lógicos, nivel de bit y asignación

Operadores lógicos:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso (niega op)

Nota: Los operadores logicos siempre devuelven un valor booleano.

Operadores a nivel de bit:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1(sin signo)
&	op1 & op2	operación and
|	op1 | op2	operación or
^	op1 ^ op2	operación xor
~	~ op	operación complemento a 1

Los operadores a nivel a bit toman los operadores, los transforman a binario y realizan las operaciones trabajando con los bits uno a uno.

Operadores de asignación:

Operador	Uso	Equivale a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Nota: Estos operadores son en realidad abreviaturas de otros operadores unidos junto al operador asignación.

38. Precedencia de Operadores en Java

Por el hecho de poder emplear varios operadores en la misma expresión nos encontramos con la necesidad de conocer el orden de evaluación de los operadores:

Precedencia de Operadores en Java

operadores sufijo	[] . (*params*) *expr* ++ *expr* --
operadores unarios	++ *expr* -- *expr* + *expr* - *expr* ~ !
creación o tipo	new (*type*) *expr*
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
AND lógico	&&
OR lógico	||
condicional	? :
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

Existe una palabra clave llamada instanceof que puede ser interpretada como un operador, encargado de comprobar si un objeto es una instancia de una determinada clase, por ejemplo:
String cad = "unTexto";
Bolean resultado = cad instanceof String;

39. Instrucciones Condicionales (I)

Instrucciones Condicionales
Java proporciona las instrucciones if/else, switch y el operador ternario para poder tomar decisiones en función del resultado de la evaluación de una condición o variable.
La instrucción if tiene la siguiente estructura:

```
if (condición)
{
//Código a ejecutar si condición es true
}
else
{
//Código a ejecutar si condición es false
}
```

Esta instrucción evalúa la expresión condición, y si es true, ejecuta el código que hay entre las llaves que hay debajo de if. Si condición fuese false, el código a ejecutar sería el contenido entre las llaves que existen debajo de else.
La parte else es opcional, es decir, esto también es correcto:

```
if (condición)
{
//Código a ejecutar si condición es true
}
```

En este caso si condición es false no sucede nada, la instrucción no ejecuta ninguna instrucción.
Otra simplificación también correcta es que en caso de que sólo exista una instrucción dentro de las llaves (del if, o del else) se pueden eliminar las llaves, es decir:

```
if (condición) //instrucción a ejecutar si condición es true;
else //instrucción a ejecutar si condición es false;
```

Lo que no está permitido eliminar en ningún caso, son los puntos y coma de las instrucciones que empleemos en la parte verdadera (if) o falsa (else), tengamos o no las llaves.

40. Prácticas con instrucciones condicionales (I)

Práctica:
Vamos a construir una clase java que permita recibir un parámetro al ejecutar la clase y comprobemos que existe al menos un parámetro, y en caso de que exista que lo visualice.
Construyamos la clase if1.java
Agreguemos el siguiente código:

```
public class If1{
public static void main(String arg[]){
if ( arg.length>0) {
System.out.println(" Al menos hay un parámetro, y el primero es: " + arg[0] );
}
else
{
System.out.println("No hay ningún parámetro");
}
}
}
```

Ejecutar sin pasar nigún parámetro, o pasando al menos un parámetro y comprobemos lo que sucede. La instrucción switch permite ejecutar código, pero dependiente del valor de una variable:

```
switch (variable)
{
case val1:
------
------
break;
.
.
.
case valn:
------
------
break;
default:
------
------
}
```

41. Práctica completa con Instrucción Condicionales (I)

La instrucción switch toma la variable que le pasamos como argumento, y obtiene el valor de dicha variable. Después lo compara con los valores que hay junto a las etiquetas case, comenzando a ejecutar el código que hay debajo de la etiqueta case que coincida. Cuando se encuentre con la instrucción break finalizará la ejecución del switch.

En caso de no encontrar una etiqueta case coincidente, ejecutará el código que existe dentro de la etiqueta default. La etiqueta default es opcional, por tanto en caso de no tenerla simplemente no hace nada la instrucción.

Práctica:

Construir una aplicación que tome un valor entero como argumento, y que muestre el mes equivalente al entero introducido.

Crear el fichero Switch1.java

Agregar el siguiente código:

```
public class Switch1{
public static void main(String arg[])
{
if ( arg.length == 0 )
{
System.out.println("Uso: \n\tjava Switch1 entero");
return;
}
int mes = Integer.parseInt(arg[0]);
String nombreMes;
switch(mes){
case 1:
nombreMes = "Enero";
break;
case 2:
nombreMes = "Febrero";
break;
case 3:
nombreMes = "Marzo";
break;
```

42. Práctica completa con Instrucción Condicionales (II)

```
case 4:
nombreMes = "Abril";
break;
case 5:
nombreMes = "Mayo";
break;
case 6:
nombreMes = "Junio";
break;
case 7:
nombreMes = "Julio";
break;
case 8:
nombreMes = "Agosto";
break;
```

```
case 9:
nombreMes = "Septiembre";
break;
case 10:
nombreMes = "Octubre";
break;
case 11:
nombreMes = "Noviembre";
break;
case 12:
nombreMes = "Diciembre";
break;
default:
nombreMes = "desconocido";
}
System.out.println("El mes es " + nombreMes);
}
}
```

Ejecutar y comprobar su correcto funcionamiento.
Eliminar todas las instrucciones break del código, y comprobar que siempre se devuelve el mismo valor, independientemente del valor pasado como argumento.

43. Instrucción Condicionales (II)

El operador ternario tiene la misión de devolver un valor, dependiendo del valor de una condición booleana. La sintaxis es:
((condición)?valor1:valor2)
Se evalúa la condición y si es true se devuelve valor1, y si es false se devuelve valor2.
Practica:
Construir una aplicación que tome un valor entero como argumento y que muestre si es par o no por pantalla.
Crear el fichero Ternario1.java
Agregar el siguiente código:

```
public class Ternario1{
public static void main(String arg[]){
if ( arg.length>0) {
int valor = Integer.parseInt(arg[0]);
String resultado = ((valor%2==0)?"par":"impar");
System.out.println("El número es "+resultado);
}
else
{
System.out.println("No hay ningún parámetro");
}
}
}
```

44. Instrucciones iterativas (I)

Instrucciones iterativas
También conocidas como bucles, las instrucciones iterativas tienen la misión de ejecutar las mismas instrucciones de código un número de veces, determinado por una condición.
En java tenemos tres bucles: while, do while y for.
while :
La sintaxis es:

```
while (condición)
{
//Instrucciones a ejecutar
}
```

Este bucle evalúa la condición, si es cierta ejecutará las instrucciones de su interior, una vez ejecutadas regresará al comienzo y se repetirá el proceso de evaluar/ejecutar. Este proceso sólo finalizará cuando en la evaluación la condición de como resultado false.

45. Práctica con instrucciones iterativas

Practicas:
Vamos a construir una aplicación que tome un número como argumento y muestre la tabla de multiplicar de dicho número.
Crear While1.java
Agregar el siguiente código:

```
public class While1{
public static void main(String arg[]){
if ( arg.length>0) {
int valor = Integer.parseInt(arg[0]);
int contador = 1;
while (contador<=9)
{
System.out.println("" + valor + " * " + contador + " = " + (valor*contador));
contador++;
}
}
else
{
System.out.println("No hay ningún parámetro");
}
}
}
```

do while :
La sintaxis es:

```
do
{
//Instrucciones a ejecutar
} while (condición);
```

46. Instrucciones iterativas (II)

Este bucle comienza ejecutando las instrucciones que hay en su interior, una vez ejecutadas comprueba el valor de condición, si es true vuelve de nuevo a repetir el proceso ejecutar/evaluar. Si la evaluación de condición es false, entonces finaliza la ejecución del bucle.
Un bucle while ejecutará sus instrucciones de 0 a n veces, dependiendo del valor de las condiciones. Mientras que un bucle do while ejecutará sus instrucciones de 1 a n veces dependiendo del valor de la condición.
Prácticas:
Crear una aplicación que solicite una contraseña al usuario, si es correcta mostrará que es correcta, si el usuario se equivoca tres veces finalizará el programa.
Crear el fichero DoWhile1.java
Agregar el siguiente código:

```
public class DoWhile1{
public static void main(String arg[]){
String clave = "Sin clave";
String candidata;
int nVeces = 0;
do
{
System.out.println("Introduzca la clave: ");
candidata = leerLinea();
nVeces++;
}while(!clave.equals(candidata) && nVeces<3);
if(nVeces==3 && !clave.equals(candidata))
{
System.out.println("Lo siento no acertó.");
}
else
{
System.out.println("Clave correcta.");
}
}
/* Esta función permite leer una línea de texto, veremos su funcionamiento cuando
lleguemos al capítulo de flujos de entrada y salida */
```

47. Segunda práctica con Instrucciones iterativas

```
public static String leerLinea(){
try{
java.io.BufferedReader d = new java.io.BufferedReader(new
java.io.InputStreamReader(System.in));
return d.readLine();
}catch(Exception e) {}
return "";
```

```
}
}
```
for
La sintaxis del bucle for es:
```
for (inicialización;condición;evaluación){
//instrucciones a ejecutar
}
```
Antes de nada, veamos una equivalencia de un bucle while configurado como un bucle for:
```
inicialización;
while (condición)
{
//Instrucciones a ejecutar
evaluación;
}
```

48. Tercera práctica con instrucciones iterativas

En esta equivalencia veremos que inicialización es empleada para dar un valor inicial a la variable que utilizaremos para ser evaluada posteriormente en la condición. La condición, que será evaluada para comprobar si se detiene o no el bucle for, y la evaluación, empleada para indicar los incrementos que se realizarán sobre la variable.
Prácticas:
Crear una aplicación que muestre los primeros 256 caracteres por pantalla, en cada fila aparecerán 5 caracteres.
Crear el fichero For1.java
Agregar el siguiente código:
```
public class For1{
public static void main(String arg[]){
for (int nLetra = 0; nLetra<=255 ; nLetra ++)
{
System.out.print(" " + nLetra + ": " + (char)nLetra);
if (nLetra%4==0) System.out.println("");
}
}
}
```
Para los tres tipos de bucles existen dos palabras claves que se pueden emplear: break y continue.break consigue que se detenga el bucle y siga el flujo de ejecución después del bucle, mientras que continue provoca que se regrese al comienzo del bucle. Este regreso no provoca que se reinicialicen las variables empleadas.

49. Tercera práctica con instrucciones iterativas (II)

Prácticas:
Crear el fichero BreakContinue.java y agregar el siguiente código:

```
public class BreakContinue1{
public static void main(String arg[]){
for (int contador = 0; contador<=15 ; contador ++)
{
System.out.println(contador);
// if(contador==10) break;
// if(contador==10) continue;
System.out.println("Despues del if");
}
}
}
```

Al ejecutar aparecerá:
0
Despues del if
pero variando de 0 a 14.
Ahora vamos a descomentar la línea del break y ejecutamos, el resultado será el mismo, pero ahora variará de 0 a 10 y además para el caso 10 no aparecerá el texto después de if.
Ahora comentamos la línea del break y descomentamos la línea del continue, ejecutamos. Podremos apreciar que aparecen los valores de 0 a 14, pero en el caso 10 no aparece el mensaje "Después de if".

50. Introducción a la entrada por teclado

Introducción a la entrada por teclado
En java para poder escribir se emplea el objeto **System.o**ut, pero para leer del teclado es necesario emplear **System.in**. Este objeto pertenece a la clase **InputStream**, esto significa que para leer tenemos que emplear sus métodos, el más básico es read, que permite leer un carácter:

```
char caracter = (char) System.in.read();
```

Pero como podemos comprobar es muy incómodo leer de letra en letra, por ello para poder leer una línea completa emplearemos el siguiente código:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String línea = br.readLine();
```

En el cuál creamos un InputStreamReader a partir de System.in y pasamos dicho InputStreamReader al constructor de BufferedReader, el resultado es que las lecturas que hagamos sobre br son en realidad realizadas sobre System.in, pero con la ventaja de que se permite leer una línea completa.
Es necesario realizar un import de java.io para poder emplear esta lectura de líneas.
Además la línea del readLine puede lanzar Excepciones, es por ello que hay que meterla entre instrucciones try/catch para poder gestionar el posible error:

```
String línea;
try{
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
línea = br.readLine();
}catch(Exception e){ e.printStackTrace();}
System.out.println(línea);
```

51. Introducción a la transformación entre tipos de datos (I)

Introducción a la transformación entre tipos de datos

En java será necesario transformar entre tipos de datos básicos, generalmente entre String hacia cualquier otro tipo tipo básico, o viceversa. Si es necesario transformar un tipo básico hacia String, la forma más rápida consiste en concatenar el valor a un objeto String, por ejemplo:

```
"" + 34
```

Con ello conseguiremos que se cree el literal "34".

Si en cambio queremos transformar el literal "2345" a un valor int para poder operar con el será necesario emplear:

```
int valor = Integer.parseInt("2345");
```

Integer es una clase empleada para representar al tipo básico int, y para transformar de String a int, pero puede lanzar excepciones, es por ello que siempre hay que usarla con try/catch:

```
int valor;
try{
valor = Integer.parseInt( "2345");
}catch(Exception e){ e.printStackTrace();}
System.out.println(valor*10);
```

52. Introducción a la transformación entre tipos de datos (II)

En ocasiones nos interesará transformar un tipo, por ejemplo entero, a otro tipo entero de menos precisión, es decir, transformar de long a int, para ello habrá que emplear el operador de moldeo:

```
(tipo)
```

para forzar la conversión de un tipo a otro:

```
long b =-234;
int a = (int)b;
```

Funcionará mientras que el valor almacenado en el long esté dentro del rango permitido al tipo int, si no, se truncará el valor. Lo mismo se puede indicar para los tipos float y double.

También se puede emplear el operador de moldeo para objetos. Veamos un ejemplo:

```
Coche c = new Coche();
Vector v = new Vector();
v.add(c);
Object o = v.get(0);
Coche recuperado = (Coche) o;
```

En este ejemplo creamos un objeto de la clase Coche, y un objeto de la clase Vector. Un Vector en java se emplea como un array dinámico.

En este caso agregamos al vector el coche mediante la función add. Para recuperar el objeto que está en la posición 0 usamos la función get, esta función devuelve un Object, cuando pensamos que debería devolver un objeto Coche. En realidad lo que devuelve es una referencia de la clase base Object, que apunta al objeto de la posición 0, es decir nuestro objeto coche. Pero necesitamos que nuestro objeto sea apuntado por una referencia de tipo Coche para poder llamar a las funciones de la clase Coche. Por ello hacemos una conversión de referencias en la última línea.

53. Interfaces (I)

Interfaces
Un **interfaz** es una lista de acciones que puede llevar a cabo un determinado objeto. Sorpresa, ¿eso no eran los métodos que se definen en una clase? Casi, en una clase además de aparecer los métodos aparecía el código para dichos métodos, en cambio en un interfaz sólo existe el prototipo de una función, no su código.
Veámoslo con un ejemplo: Pensemos en un interfaz en el que en su lista de métodos aparecen los métodos despegar, aterrizar, servirComida y volar. Todos pensamos en un avión, ¿verdad? El motivo es sencillamente que avión es el concepto que engloba las acciones que hemos detallado antes, a pesar que existan muchos objetos avión diferentes entre sí, por ejemplo Boeing 747, Boeing 737, MacDonell-Douglas.
Lo realmente interesante es que todos ellos, a pesar de pertenecer a clases distintas, poseen el interfaz avión, es decir poseen los métodos detallados en la lista del interfaz avión.
Esto significa también que a cualquier avión le podemos pedir que vuele, sin importarnos a que clase real pertenezca el avión, evidentemente cada clase especificará como volará el avión (porque proporciona el código de la función volar).
En java un interfaz define la lista de métodos, pero para que una clase posea un interfaz hay que indicar explícitamente que lo implementa mediante la claúsula implements. Pero veamos primero la estructura de un interfaz:

```
[modif.visibilidad] interface nombreInterfaz [extends listaInterfaces]
{
prototipo método1;
.....
prototipo método1;
}
```

54. Interfaces (II)

Donde modif.visibilidad puede ser public o bien sin especificar, es decir visibilidad pública (desde cualquier clase se puede emplear el interfaz) o de paquete (sólo se puede emplear desde clases del mismo paquete).
nombreInterfaz por convenio, sigue las mismas reglas de nomenclatura que las clases, y en muchos casos acaba en able (que podíamos traducir como: 'ser capaz de').
La claúsula opcional extends, se emplea para conseguir que un interfaz herede las funciones de otro/s interfaces, simplemente listaInterfaces es una lista separada por coma de interfaces de los que se desea heredar.

En muchas ocasiones un interfaz es empleado para definir un comportamiento, que posteriormente será implementado por diversas clases, que podrán no tener nada que ver entre ellas, pero que todas se comportarán igual de cara al interfaz. Es decir, todas tendrán las funciones indicadas por el interfaz.
Cuando varios objetos de distintas clases pueden responder al mismo mensaje (función), aún realizando cosas distintas se denomina **polimorfismo**.

55. Práctica con interfaces

Prácticas:
Vamos a definir el interfaz **Cantante**, un interfaz muy simple que sólo posee un método: cantar.
Crear el fichero **Cantante.java**
Agregar el siguiente código:

```
public interface Cantante
{
public void cantar();
}
```

Cojamos la clase Persona y hagamos que implemente el interfaz Cantante:

```
public class Persona implements Cantante
```

Además agreguemos el código para el método que define el interfaz cantante:

```
public void cantar()
{
System.out.println("La laa la raa laaa!");
}
```

Construyamos ahora una clase con función main (ArranqueInterfaz.java)para ejecutar:

```
public class ArranqueInterfaz
{
public static void main(String arg[])
{
Persona p = new Persona();
hacerCantar(p);
}
public static void hacerCantar(Cantante c)
{
c.cantar();
}
}
```

56. Interfaces (III)

Podemos ver que construimos un objeto (p) de tipo persona y se lo pasamos a la función hacerCantar. Esta función espera recibir un objeto Cantante, y una persona lo es, por tanto la recibe y llama al método cantar del objeto recibido.
Probemos a intentar pasar a la función hacerCantar en lugar del objeto Persona (p) un objeto String (texto), resultado: error de compilación.

Contruyamos ahora la clase **Canario** (**Canario.java**), pensando que también sabe cantar:

```
public class Canario implements Cantante
{
private int peso;
/* Aqui vendrían el resto de atributos y funciones propias de un canario */
public void cantar()
{
System.out.println("Pio Pio Pio");
}
}
```

Y ahora agreguemos en la clase ArranqueInterfaz el siguiente código, para crear un objeto canario y pasarselo a la función hacerCantar:

```
Canario c = new Canario();
hacerCantar(c);
```

Tras ejecutar comprobaremos que podemos pasar tanto una Persona como un Canario a la función hacerCantar, de tal manera que dentro de dicha función sólo accedamos a las funciones del interfaz y no habrá problemas. Por ejemplo, si pusiéramos:

```
c.SetNombre("Luis")
```

dentro de la función hacerPersona, podría funcionar si pasásemos un objeto Persona, pero no si pasamos uno de tipo Canario.

57. Excepciones

Excepciones

Excepcion es, o sencillamente problemas. En la programación siempre se producen errores, más o menos graves, pero que hay que gestionar y tratar correctamente. Por ello en java disponemos de un mecanismo consistente en el uso de bloques try/catch/finally . La técnica básica consiste en colocar las instrucciones que podrían provocar problemas dentro de un bloque try, y colocar a continuación uno o más bloques catch, de tal forma que si se provoca un error de un determinado tipo, lo que haremos será saltar al bloque catch capaz de gestionar ese tipo de error específico. El bloque catch contendrá el codigo necesario para gestionar ese tipo específico de error. Suponiendo que no se hubiesen provocado errores en el bloque try, nunca se ejecutarían los bloques catch.

Veamos ahora la estructura del bloque **try/catch/finally**:

```
try
{
//Código que puede provocar errores
}
catch(Tipo1 var1)
{
//Gestión del error var1, de tipo Tipo1
}
[ ...
catch(TipoN varN)
{
//Gestión del error varN, de tipo TipoN
```

}]
[
finally
{
//Código de finally
}
]
Como podemos ver es obligatorio que exista la zona try, o zona de pruebas, donde pondremos las instrucciones problemáticas. Después vienen una o más zonas catch, cada una especializada en un tipo de error o excepción. Por último está la zona finally, encargada de tener un código que se ejecutará siempre, independientemente de si se produjeron o no errores.
Se puede apreciar que cada catch se parece a una función en la cuál sólo recibimos un objeto de un determinado tipo, precisamente el tipo del error. Es decir sólo se llamará al catch cuyo argumento sea coincidente en tipo con el tipo del error generado.

58. Prácticas con excepciones (I)

Prácticas:
Crear el fichero **Try1.java**
Agregar el siguiente código:

```
public class Try1
{
public static void main(String arg[])
{
int [] array = new int[20];
array[-3] = 24;
}
}
```

Como podremos comprobar al ejecutar se generará el siguiente error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
at Try1.main(Try1.java:6)
```

Que indica que se ha generado una excepción del tipo java.lang.ArrayIndexOutOfBoundsException en la función Try1.main, dentro del fichero Try1.java y en la línea 6 del código. Esta excepción en particular, se lanza cuando intentamos acceder a una posición de un array y no existe dicha posición.

59. Prácticas con excepciones (II)

Vamos a gestionar esta interrupción mediante un bloque try/catch, el fichero crear es **Try2.java**, con el siguiente código:

```
public class Try2
{
public static void main(String arg[])
{
int [] array = new int[20];
```

```
try
{
array[-3] = 24;
}
catch(ArrayIndexOutOfBoundsException excepcion)
{
System.out.println(" Error de índice en un array");
}
}
}
```

60. Prácticas con excepciones (III)

Intentemos provocar también un error de tipo división por cero y pongamos un catch específico para dicho error (fichero **Try3.java**):

```
public class Try3
{
public static void main(String arg[])
{
int [] array = new int[20];
try
{
// array[-3] = 24;
int b = 0;
int a = 23/b;
}
catch(ArrayIndexOutOfBoundsException excepcion)
{
System.out.println(" Error de índice en un array");
}
catch(ArithmeticException excepcion)
{
System.out.println(" Error de índice en un array");
}
}
}
```

Podemos comprobar que se ejecuta el catch correspondiente al tipo de error generado. La línea que lanza el error de índice la hemos comentado para que no lo genere y podamos generar el error de división por cero.

Los tipos de error que se generan, son todos ellos clases, que heredan de la clase java.lang.Exception, que a su vez hereda de java.lang.Throwable, por lo tanto podríamos crear nuestros propios errores personalizados. Al igual que podríamos tener un solo catch que capture todos los errores, independientemente del tipo del error.

61. Prácticas con excepciones (IV)

Prácticas:
Construyamos la clase **Try4** y agreguemos el siguiente código:

```
public class Try4
{
public static void main(String arg[])
{
int [] array = new int[20];
try
{
// array[-3] = 24;
/* int b = 0;
int a = 23/b;
*/
String s = null;
s.equals("QQQQ");
}
catch(ArrayIndexOutOfBoundsException excepcion)
{
System.out.println(" Error de índice en un array");
}
catch(ArithmeticException excepcion)
{
System.out.println(" Error de índice en un array");
}
catch(Exception excepcion)
{
System.out.println("Se ha generado un error que no es de índices, ni Aritmético");
System.out.println("El objeto error es de tipo " + excepcion);
}
}
}
```

62. Prácticas con excepciones (V)

Podemos comprobar que el catch que captura el error es el correspondiente a la clase base **Exception**, el orden de selección del catch es de arriba a abajo, se comprueba el tipo del error con el del argumento del primer catch, luego con el del segundo,...Hasta que uno de los catch tenga el mismo tipo y pueda gestionarlo. Por ello el catch de la clase base debe de ser el último, y los de las clases derivadas deben de estar antes.
Al transformar un objeto que herede de **Exception** a un **String**, el resultado es el nombre de la excepción.
La clase **Exception** proporciona algunos métodos de utilidad, por ejemplo **printStackException** que muestra el volcado de pila con todas las funciones que están siendo llamadas en el momento en el que se lanzó el error:

```
try
{
array[-3] = 24;
}
catch(Exception excepcion)
{
excepcion.printStackTrace();
}
```
Si es necesario podemos lanzar un error siempre que lo consideremos oportuno, para ello tenemos que usar la palabra clave throw y especificar un objeto que sea un objeto de la clase **Exception**, o de una clase derivada.

63. Prácticas completa con excepciones

Practicas:
Creemos Try5.java con el siguiente código:

```
public class Try5
{
public static void main(String arg[])
{
try
{
Exception e = new Exception("Este es mi propio error.");
throw e;
}
catch(Exception excepcion)
{
excepcion.printStackTrace();
}
}
}
```

Ahora construyamos nuestra propia clase personalizada de error y lancemos un error, como en el código del fichero Try6.java:

```
public class Try6
{
public static void main(String arg[])
{
try
{
MiPropioError e = new MiPropioError("Este es mi propio error.");
throw e;
}
catch(Exception excepcion)
{
excepcion.printStackTrace();
}
```

```
}
}
class MiPropioError extends Exception{
public MiPropioError(String mensaje){
super(mensaje);
}
}
```

64. Práctica con clausula Finally

Habitualmente dentro de try solicitamos recursos y trabajamos con ellos, en esos casos se pueden lanzar excepciones, pero la única manera de liberar recursos, independientemente de si se lanza una excepción o no, consiste en emplear una claúsula **finally**, que será ejecutada siempre, haya o no lanzamiento de excepciones.
Prácticas:
Construyamos el fichero Tray7.java:

```
public class Try7
{
public static void main(String arg[])
{
try
{
Exception e = new Exception("Este es mi propio error.");
throw e;
}
catch(Exception excepcion)
{
excepcion.printStackTrace();
}
finally
{
System.out.println("Se ejecuta finally");
}
}
}
```

65. Glosario (I)

Glosario de términos
abstract: Abstracto.Aplicable a clases o métodos.
array: Variable que posee varias posiciones para almacenar un valor en cada posición. Las posiciones son accedidas mediante un índice numérico.
break: Palabra clave que finaliza la ejecución de un bucle o de una instrucción switch.
bucles: Tipo de estructura iterativa, que permite repetir un conjunto de instrucciones un número variable de veces.
clase: Estructura que define como son los objetos, indicando sus atributos y sus acciones.

clase base: Clase de la cuál se hereda para construir otra clase, denominada derivada.
CLASSPATH: Variable de entorno que permite a la máquina virtual java saber donde localizar sus clases.
constructor: Función especial empleada para inicializar a los objetos, cada clase posee sus propios constructores.
derivada: Clase que hereda de una clase base.
Excepcion: Objeto empleado para representar una situación de excepción (error) dentro de una aplicación java.

66. Glosario (II)

herencia: Característica que permite que una clase posea las características de otra, sin tener que reescribir el código.
herencia sencilla y múltiple: Dos tipos de herencia, con una sóla clase base, o con varias.
instancia: Un objeto creado a partir de una clase.
instanciación: Proceso de creación de un objeto a partir de una clase.
interfaz: Define un tipo de datos, pero sólo indica el prototipo de sus métodos, nunca la implementación.
JDK: Java Development Kit, es el conjunto de herramientas proporcionadas por sun, que permite compilar y ejecutar código java.
jerarquía de herencia: Árbol construido mediante las relaciones de herencia en las clases java.
máquina virtual: Es la encargada de ejecutar el código java.
multiplataforma: Posibilidad de existir en varias plataformas (sistemas operativos)
package: Paquete. Carpeta creada para contener clases java, y así poder organizarlas.
PATH: Variable de entorno, empleada por los sistemas operativos para saber donde localizar sus programas ejecutables.
Sobrescritura: Poseer el mismo método, pero con código distinto, en una clase base y en una clase que deriva de ella.
transformación de datos: Cómo cambiar el tipo de una información, por ejemplo cambiar el literal "23" al valor numérico 23.
try/catch/finally: Instrucciones empleadas para gestionar los posibles errores que se puedan provocar en un programa java.