GRUPO STUDIUM FORMACIÓN

Tema 2: Introducción a JAVA

2.1 Introducción

En el presente tema vamos a empezar a trabajar con **JAVA**. Todos los conocimientos que hemos adquirido en el tema anterior de **Programación** como son las *Variables, Constantes, Sentencias condicionales, repetitivas*, etc. las vamos a ver ahora en un lenguaje de programación concreto como es el JAVA. Pero antes tendremos que ver en qué consiste la **Programación Orientada a Objetos** pues JAVA es un lenguaje que sigue a rajatabla este paradigma de programación y por tanto debemos conocer sus conceptos principales como Clases y Objetos, Herencia, Interfaces, etc.

También se nos presentará el entorno de programación donde desarrollaremos nuestro trabajo en JAVA: **Eclipse**. Y empezaremos a crear nuestros programas y veremos cómo se comportan, la forma de identificar los errores y todas las tareas necesarias para que nuestras aplicaciones sean ejecutadas por un ordenador.

2.2 Historia de JAVA

Los lenguajes de programación C y Fortran se han utilizado para diseñar algunos de los sistemas más complejos en lenguajes de programación estructurada, creciendo hasta formar complicados procedimientos difíciles de manejar y actualizar. Se hacía necesaria una nueva forma de crear software y por tanto se necesitaba también un nuevo tipo de lenguaje de programación.

No sólo se necesitaba un lenguaje de programación para tratar esta complejidad, sino un nuevo estilo de programación. Este cambio de paradigma de la programación estructurada a la programación orientada a objetos, comenzó hace 30 años con un lenguaje llamado Simula67.

El lenguaje C++ fue un intento de tomar estos principios y emplearlos dentro de las restricciones de C. Todos los compiladores de C++ eran capaces de compilar programas de C sin clases, es decir, un lenguaje capaz de interpretar dos estilos diferentes de programación. Esta compatibilidad (*"hacia atrás"*) que habitualmente se vende como una característica de C++ es precisamente su punto más débil. No es necesario utilizar un diseño orientado a objetos para programar en C++, razón por la que muchas veces las aplicaciones en este lenguaje no son realmente orientadas al objeto, perdiendo así los beneficios que este paradigma aporta.

Comienzos

Java fue diseñado en 1990 por James Gosling, de Sun Microsystems, como software para dispositivos electrónicos de consumo, electrodomésticos fundamentalmente. Curiosamente, todo este lenguaje fue diseñado antes de que diese comienzo la era World Wide Web, puesto que fue diseñado para dispositivos electrónicos como calculadoras, microondas y la televisión interactiva.

En los primeros años de la década de los noventa, Sun Microsystems decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Tras unos comienzos dudosos, Sun decidió crear una filial, denominada FirstPerson Inc., para dar margen de maniobra al equipo responsable del proyecto.

Inicialmente Java se llamó Oak (roble en inglés), aunque tuvo que cambiar de denominación, debido a que dicho nombre ya estaba registrado por otra empresa. Se dice este nombre se le puso debido a la existencia de tal árbol en los alrededores del lugar de trabajo de los promotores del lenguaje.

Tres de las principales razones que llevaron a crear Java son:

1. Creciente necesidad de interfaces mucho más cómodas e intuitivas que los sistemas de ventanas que proliferaban hasta el momento.
2. Fiabilidad del código y facilidad de desarrollo. Gosling observó que muchas de las características que ofrecían C o C++ aumentaban de forma alarmante el gran coste de pruebas y depuración. Por ello en los sus ratos libres creó

un lenguaje de programación donde intentaba solucionar los fallos que encontraba en C++.

3. Enorme diversidad de controladores electrónicos. Los dispositivos electrónicos se controlan mediante la utilización de microprocesadores de bajo precio y reducidas prestaciones, que varían cada poco tiempo y que utilizan diversos conjuntos de instrucciones. Java permite escribir un código común para todos los dispositivos.

Por todo ello, en lugar de tratar únicamente de optimizar las técnicas de desarrollo y dar por sentada la utilización de C o C++, el equipo de Gosling se planteó que tal vez los lenguajes existentes eran demasiado complicados como para conseguir reducir de forma apreciable la complejidad de desarrollo asociada a ese campo. Por este motivo, su primera propuesta fue idear un nuevo lenguaje de programación lo más sencillo posible, con el objeto de que se pudiese adaptar con facilidad a cualquier entorno de ejecución.

Basándose en el conocimiento y estudio de gran cantidad de lenguajes, este grupo decidió recoger las características esenciales que debía tener un lenguaje de programación moderno y potente, pero eliminando todas aquellas funciones que no eran absolutamente imprescindibles.

Primeros proyectos en que se aplicó Java

El proyecto **Green** fue el primero en el que se aplicó Java, y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Con este fin se construyó un ordenador experimental denominado *7 (Star Seven). El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía ya *Duke*, la actual mascota de Java.

Icono de Duke, la mascota de Java

Más tarde Java se aplicó a otro proyecto denominado VOD (Video On Demand) en el que se empleaba como interfaz para la televisión interactiva que se

pensaba iba a ser el principal campo de aplicación de Java. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo.

Una vez que en Sun se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, instaron a FirstPerson a desarrollar nuevas estrategias que produjeran beneficios. Entre ellas se encontraba la aplicación de Java a Internet, la cual no se consideró productiva en ese momento.

Resurgimiento de Java

Aunque muchas de las fuentes consultadas señalan que Java no llegó a caer en un olvido, lo cierto es que tuvo que ser Bill Joy (cofundador de Sun y uno de los desarrolladores principales del sistema operativo UNIX de Berckley) el que sacó a Java del letargo en que estaba sumido. Joy juzgó que Internet podría llegar a ser el campo adecuado para disputar a Microsoft su primacía en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes.

Para poder presentarlo en sociedad se tuvo que realizar una serie de modificaciones de diseño para poderlo adaptar al propósito mencionado. Así Java fue presentado en sociedad en agosto de 1995.

Algunas de las razones que llevaron a Bill Joy a pensar que Java podría llegar a ser rentable son:

- Java es un lenguaje orientado a objetos: Esto es lo que facilita abordar la resolución de cualquier tipo de problema.
- Es un lenguaje sencillo, aunque sin duda potente.
- La ejecución del código Java es segura y fiable: Los programas no acceden directamente a la memoria del ordenador, siendo imposible que un programa escrito en Java pueda acceder a los recursos del ordenador sin que esta operación le sea permitida de forma explícita. De este modo, los datos del usuario quedan a salvo de la existencia de virus escritos en Java. La ejecución segura y controlada del código Java es una característica única, que no puede encontrarse en ninguna otra tecnología.

- Es totalmente multiplataforma: Es un lenguaje sencillo, por lo que el entorno necesario para su ejecución es de pequeño tamaño y puede adaptarse incluso al interior de un navegador.

Las consecuencias de la utilización de Java junto a la expansión universal de Internet todavía están comenzando a vislumbrarse.

Futuro de Java

Existen muchas críticas a Java debido a su lenta velocidad de ejecución, aproximadamente unas 20 veces más lento que un programa en lenguaje C. Sun está trabajando intensamente en crear versiones de Java con una velocidad mayor.

El problema fundamental de Java es que utiliza una representación intermedia denominada *código de byte* (ByteCode) para solventar los problemas de portabilidad. Los *códigos de byte* posteriormente se tendrán que transformar en código máquina en cada máquina en que son utilizados, lo que ralentiza considerablemente el proceso de ejecución.

La solución que se deriva de esto parece bastante obvia: fabricar ordenadores capaces de comprender directamente los códigos de byte. Éstas serían unas máquinas que utilizaran Java como sistema operativo y que no requerirían en principio de disco duro porque obtendrían sus recursos de la red.

A los ordenadores que utilizan Java como sistema operativo se les llama Network Computer, WebPC o WebTop. La primera gran empresa que ha apostado por este tipo de máquinas ha sido Oracle, que en enero de 1996 presentó en Japón su primer NC (Network Computer), basado en un procesador RISC con 8 Megabytes de RAM. Tras Oracle, han sido compañías del tamaño de Sun, Apple e IBM las que han anunciado desarrollos similares.

La principal empresa en el mundo del software, Microsoft, que en los comienzos de Java no estaba a favor de su utilización, ha licenciado Java, lo ha incluido en Internet Explorer (versión 3.0 y posteriores), y ha lanzado un entorno de desarrollo para Java, que se denomina Visual J++.

El único problema aparente es la seguridad para que Java se pueda utilizar para transacciones críticas. Sun va a apostar por firmas digitales, que serán clave en el desarrollo no sólo de Java, sino de Internet.

Especulación sobre el futuro de Java

Java es una plataforma que le falta madurar, pero que a buen seguro lo va a hacer. La apuesta realizada por empresas con mucho peso específico ha sido tan grande que va a dar un impulso a Java que no le permitirá caer.

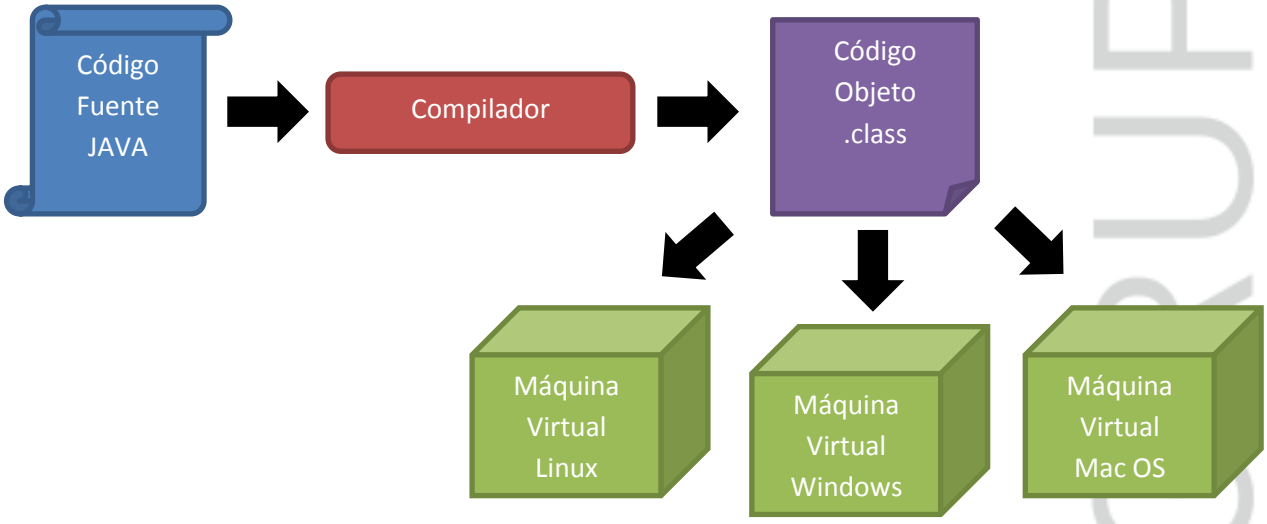
Además, el parque de productos (entornos de desarrollo, bibliotecas, elementos de conectividad...) ya disponible en la actualidad es tan amplio que es improbable que se quede en nada.

Por otra parte, la relación simbiótica que tiene con Internet (y por derivación con las Intranets) es un punto a favor de Java de muy difícil refutación.

¿Cómo funciona JAVA?

El lenguaje JAVA es compilado e interpretado a la vez. A partir del código fuente se genera el Byte code, que es un código independiente de la máquina, un código intermedio entre el lenguaje máquina del procesador y JAVA. No es ejecutable. En cada máquina que queramos "ejecutar" el Byte code, debemos tener el Java Virtual Machine (JVM). Este JVM es el intérprete del Byte code, transformándolo al Código Máquina específico de cada máquina. Existe un JVM por cada máquina.

Lo que ganamos en portabilidad lo perdemos en velocidad.



2.3 Características de JAVA

Las principales características de JAVA son las siguientes:

- **Simple**: Los programadores de C++ no tendrán ningún problema con este nuevo lenguaje al igual que los demás programadores de cualquier lenguaje orientado a objetos. Para el resto de los programadores, no será nada difícil adaptarse a su sintaxis.
- **Orientado a Objetos**: Tiene todas las características de la Programación Orientada a Objetos: herencia, polimorfismo, sobrecarga,... Las plantillas de objetos son llamados Clases y sus copias, instancias. Como en C++, estas instancias deben ser construidas y destruidas mediante métodos.
- **Distribuido**: JAVA dispone de mecanismos para que nuestros programas sean distribuidos, que puedan acceder a librerías y herramientas remotas.
- **Robusto**: JAVA busca errores de compilación y errores de ejecución, de manera que se desarrollan aplicaciones casi libres de fallos.
- **Arquitectura Neutral**: El mismo código fuente funciona en cualquier equipo.
- **Seguro**: Previene de accesos ilegales mediante privacidad y encapsulamiento. Los punteros "desaparecen" de la sintaxis para que el programador no tenga acceso a direcciones de memoria directamente.
- **Portable**: Podemos pasar nuestros programas de un sistema a otro sin apenas modificaciones.
- **Interpretado**: El intérprete JAVA puede ejecutar directamente el código objeto.
- **Dinámico**: Los enlaces de los módulos de una aplicación se realizan en tiempo de ejecución
- **Multitarea**: JAVA permite realizar varias tareas a la vez. Las tareas son como piezas que trabajan simultáneamente y por separado para llegar a un fin común.

Herencia y Subclases

Como ya se ha visto en la POO se puede reutilizar el código mediante el uso de Clases ya creadas para crear otras y adaptarlas a las necesidades nuevas. Este proceso es denominado **Herencia**, y consiste en crear una Clase, llamada Subclase, a partir de otra, llamada Superclase de la que "heredará" todos sus atributos y métodos y a los que se le añadirán otros específicos de la nueva clase que estamos creando. Gracias a esto, la POO permite realizar aplicaciones mucho más rápidamente, sin tener que crear todas las aplicaciones desde cero.

Excepciones

Una de las características más demandadas en los nuevos lenguajes de programación es su robustez y seguridad frente a fallos a la hora de ejecutar un programa por parte del usuario final.

Las **Excepciones** son un mecanismo que permiten trabajar con errores a la hora de trabajar con programas de ordenador. En la fase de análisis de un proyecto, se deben detectar los posibles casos en que el programa puede no funcionar o bien prever los posibles malos usos del programa por parte del usuario. El típico caso de dividir por cero. En la compilación no se detectan estos fallos, sino que es en tiempo de ejecución cuando aparecen.

Las Excepciones permiten capturar los errores y tratarlos, bien mostrando un simple mensaje de error y continuando con la ejecución del programa o bien resolviendo mediante código el error.

Cuando se produce un error en un bloque de código, se lanza una excepción, que es capturada por una parte del código donde será tratado, pudiéndose volver al punto donde se dejó una vez subsanado el error.

Multitarea

La **Multitarea** consiste en realizar varias cosas a la vez. Este proceso normalmente es ficticio, ya que para poder realizar un ordenador varias cosas a la vez, es necesario más de un procesador para que cada uno de ellos se dedique a algo en concreto. Lo que se entiende por Multitarea en programación consistirá entonces en una simulación de dicha multitarea real consistente el alternar el uso

del procesador entre las distintas tareas que tiene que realizar. Es decir, si el procesador tiene tres tareas por realizar, no realizará una y a continuación otra y después otra, sino que realizará un poquito de una, otro poquito de otra y otro poco de la otra, para volver a continuación con la primera y así sucesivamente.

La POO permite la multitarea en algunos lenguajes, como el JAVA o el C++, y es una de las posibilidades que nos ofrece para poder hacer programas de control en los que haya que tener en cuenta en un mismo instante varios procesos o varios acontecimientos a la vez.

Mostremos aquí algunos ejemplos de POO, de los más sencillos a los más complejos.

Empezaremos creando una clase **ClaseA** con un atributo **atributoA** de tipo entero. Necesitaremos dos métodos constructores, uno con parámetros, otro sin parámetros, y una par de métodos inspectores, uno para establecer el valor del atributo y otro para obtener el valor del atributo.

```
CLASE ClaseA
        Atributos Privados:
                Entero atributoA
        Constructores:
        ClaseA ()
        INICIO
                atributoA = 0
        FIN
```

(Continúa en siguiente página ...)

```
        ClaseA (ENTERO a)
        INICIO
                atributoA = a
        FIN
        Inspectores:
        pon_atributoA(entero a)
        INICIO
                atributoA = a
        FIN
        ENTERO dime_atributoA()
        INICIO
                DEVOLVER atributoA
        FIN
FIN CLASE
```

Esta sería la clase, el **caso general**. Ahora crearemos un objeto de dicha clase, es decir, una **instancia** o un **caso particular**.

Si para declarar una variable de tipo entero ponemos…

ENTERO variable

…igualmente para declarar un objeto de la clase **ClaseA** escribiremos…

ClaseA objetoA

Con esto estamos declarando el objeto, pero aún no lo hemos creado, aún no existe. Este es el aspecto fundamental que tiene con respecto a las variables. Las variables, en cuanto se declaran, ya existen. El objeto en cuestión existirá cuando llamemos al constructor correspondiente mediante el operador punto (.) …

objetoA.ClaseA(4)

Estamos usando el constructor con parámetros. Más concretamente, ese número 4 se asignará al atributo atributoA

Y sobre dicho objeto podremos "aplicar" los métodos inspectores que necesitemos…

objetoA.pon_atributoA(7) → Cambiamos el valor del atributo a 7

objetoA.dime_atributoA() → Obtenemos el valor actual del atributo atributoA.

2.4 Elementos del lenguaje JAVA

Empecemos a ver todos los elementos que nos vamos a encontrar en JAVA para poder escribir código.

2.4.1 Comentarios

Existen tres tipos de comentarios:

```
/* Comentario que puede ocupar varias líneas hasta encontrarse un asterisco y una barra */
// Comentario que ocupa una sola línea
/** Comentario para documentación */
```

Este último comentario colocado justo antes de una declaración sirve para cuando se hace uso de la herramienta javadoc, que genera documentación. Cuando dicha herramienta se encuentra un comentario de documentación, dicho comentario se toma como descripción del elemento declarado a continuación.

2.4.2 Estructura de un Programa en JAVA

Los programas empiezan con la inclusión de las librerías y clases que se necesitarán tras lo cual escribiremos la palabra reservada class y a continuación el nombre de la clase que debe coincidir con el nombre del fichero en el que se encuentra.

```
// Ejemplo de programa en JAVA
class Hola_Mundo
{
	public static void main(String args[])
	{
		System.out.println("Hola mundo!!!!!!!!");
	}
}
```

Este programa muestra por pantalla la famosa frase "Hola mundo!!!!!!!!".

String args[] argumentos de la línea de comando.

System es un paquete que contiene métodos.

Out es una clase dentro del paquete.

Println es un método

El programa principal o main tiene el modificador static para que no produzca una instancia de dicha clase.

De forma general, nuestros programas (aplicaciones) en JAVA tendrán la siguiente estructura:

```
//Sentencias import
public class nombre [extends clase_Padre] [implements
Interfaces/Listeners]
{
	//Declaración de atributos-componentes de la clase
	//Inicialización de atributos

	//Constructores
	public nombre()
	{
		//Definir Distribución del Contenedor
		//Añadir componentes
		//Añadir Listeners
		//Dar tamaño a la ventana
		//Mostrar la ventana
	}

	//Inspectores
	//Listeners
	//Otras funciones y procedimientos usados

	public static void main(String args[])
	{
		new nombre();
	}
}
```

2.4.3 Identificadores de variables y Separadores

Los **identificadores** sirven para nombrar los distintos elementos creados por el programador, como variables, funciones, clases, objetos, ...

Los identificadores pueden tener cualquier nombre que no sea una palabra clave o reservada y siempre deben comenzar por una letra, un guion bajo (_) o un símbolo $. Los restantes caracteres pueden ser números y letras. Las mayúsculas y minúsculas son diferentes, es decir Hola y hola no son iguales. No existe un número máximo de caracteres para los identificadores.

Java es un lenguaje con control fuerte de Tipos (Strongly Typed). Esto significa que cada variable y cada expresión tienen un Tipo que es conocido en el momento de la compilación. El Tipo limita los valores que una variable puede contener, limita las operaciones soportadas sobre esos valores y determina el

significado de las operaciones. El control fuerte de tipos ayuda a detectar errores en tiempo de compilación. En Java existen dos categorías de Tipos: Tipos Primitivos y Referencias. Los Tipos primitivos son los que permiten manipular valores numéricos (con distintos grados de precisión), caracteres y valores booleanos (verdadero / falso). Los Tipos Primitivos son:

- **boolean**: Puede contener los valores **true** o **false**. Tamaño de 1 byte.
- **byte**: Enteros. Tamaño 8-bits. Valores entre -128 y 127.
- **short**: Enteros. Tamaño 16-bits. Entre -32768 y 32767.
- **int**: Enteros. Tamaño 32-bits. Entre -2147483648 y 2147483647.
- **long**: Enteros. Tamaño 64-bits. Entre -9223372036854775808 y 9223372036854775807.
- **float**: Números en coma flotante. Tamaño 32-bits.
- **double**: Números en coma flotante. Tamaño 64-bits.
- **char**: Caracteres. Tamaño 16-bits. Unicode. Desde '\u0000' a '\uffff' inclusive. Esto es desde 0 a 65535

El tamaño de los tipos de datos no depende de la implementación de Java. Son siempre los mismos.

Las Referencias se usan para manipular objetos.

Para declarar un identificador:

```
Tipo nombre_identificador [=valor_inicial];
```

Si lo que estamos es declarando un objeto y no propiamente una variable, debemos instanciar también dicho objeto, es decir, primero declararlo y depués instanciarlo o reservar espacio en memoria para poder usarlo mediante el operador new:

```
Clase1 objeto = new Clase1();
```

Aquí hemos declarado e instanciado un objeto de la Clase1 usando su constructor.

Los separadores en JAVA son los siguientes:

- ( ) → Paréntesis: listas de parámetros en la definición, llamada e implementación de módulos. También sirve para definir la precedencia con los operadores.

- { } → Llaves: para inicializar tablas en la declaración de las mismas y para delimitar un bloque de código.
- [ ] → Corchetes: para indicar tamaño y posición en las tablas.
- ; → Punto y Coma: para acabar una sentencia.
- , → Coma: para separar identificadores en una declaración. También se usa para encadenar sentencias en las sentencia for
- . → Punto: para separar paquetes de subpaquetes, clases, objetos y métodos.

2.4.4 Constantes

Las **constantes** nos sirven para definir unos valores que no cambiarán a lo largo del programa, pero que al tratarse de valores numéricos o caracteres complejos de escribir, podemos darles unos nombres para así referirnos en todo el programa.

En Java, se utiliza la palabra clave final para indicar que una variable debe comportarse como si fuese constante.

Como es una constante, se le ha de proporcionar un valor en el momento en que se declare, por ejemplo:

```
final float PI = 3.14159;
```

Si se intenta modificar el valor de una variable final desde el código de la aplicación, se generará un error de compilación.

Si se usa la palabra clave final con una variable o clase estática, se pueden crear constantes de clase, haciendo de esto modo un uso altamente eficiente de la memoria, porque no se necesitarían múltiples copias de las constantes.

La palabra clave final también se puede aplicar a métodos, significando en este caso que los métodos no pueden ser sobreescritos.

2.4.5 Palabras claves

Existen una serie de **palabras claves** que forman el lenguaje, que son las siguientes:

abstract	boolean	break	byte	byvalue	case
catch	char	class	const	continue	default
do	double	else	extends	false	final
finally	float	for	goto	if	implements
import	instanceof[1]	int	interface	long	native
new	null	package	private	protected	public
return	short	static	super	switch	synchronized
this	threadsafe	throw	transient	true	try
void	while				

2.4.6 Palabras reservadas

Y también nos encontramos con palabras que aunque no tienen un uso, están reservadas para su posterior utilización.

cast future generic inner operator outer rest var

2.4.7 Operadores aritméticos

A continuación, veremos la forma de operar con las distintas variables y constantes que hemos visto. Empezaremos por los Operadores Aritméticos. Los que tenemos disponibles en JAVA son los siguientes:

Operación	Operador	Número de Operadores
Suma	+	2
Valor positivo	+	1
Resta	-	2
Negativo	-	1
Producto	*	2
División entera	/	2
Módulo	%	2
Incremento	++	1
Decremento	--	1

[1] Operador que devuelve true o false según pertenezca o no a una clase el objeto sobre el que se aplica: objeto. InstanceOf clase;

2.4.8 Operadores relacionales

Los operadores relacionales que soporta JAVA son los siguientes:

Operación	Operador
Mayor que	>
Mayor o igual que	>=
Menor	<
Menor o igual que	<=
Igual	= =
Distinto	!=

2.4.9 Operadores condicionales

Los operadores condicionales (o lógicos) son los siguientes:

Operación	Operador
Y	&&
O	||
NO	!

También existe en JAVA el operador condicional ternario siguiente:

```
Expresión ? sentencia1 : sentencia2
```

Su interpretación es que se evalúa la expresión. Si resulta ser verdadera, se realiza la sentencia1. En caso de ser falsa la expresión, se realiza la sentencia2.

2.4.10 Operadores a nivel de Bits

Los operadores para trabajar a nivel de bits con los datos son los siguientes:

Operador	Uso	Operación
>>	Operando >> Desplazamiento	Desplaza bits hacia la derecha las posiciones indicadas (con signo)
<<	Operando << Desplazamiento	Desplaza bits hacia la izquierda las posiciones indicadas
>>>	Operando >>> Desplazamiento	Desplaza bits hacia la derecha las posiciones indicadas (sin signo)
&	Operando & Operando	Realizar operación Y lógica
^	Operando ^ Operando	Realizar operación O lógica
~	~ Operando	Complementario del operando

En la operación >>> se rellena con ceros los lugares que quedan libres.

En la operación << debemos tener en cuenta que podemos poner un 1 en la posición más alta, con lo que transformaremos el número en negativo.

2.4.11 Operadores de asignación

Los distintos operadores de asignación que podemos encontrar en JAVA son los siguientes:

=	+=	-=	*=	/=	%=	&=	|=	^=	<<=	>>=	>>>=

2.4.12 Precedencia de Operadores

En las expresiones que escribamos podremos combinar los distintos operadores, pero puede que el ordenador interprete una cosa diferente a la que nosotros queríamos expresar. Por ejemplo, si quisiéramos hacer una suma y un producto, podríamos poner 2+3*4, que sería igual a 14. Pero también podríamos querer (2+3)*4, que sería 20, caso muy diferente a la anterior operación. Lo mismo puede ocurrir con los demás operadores, no sólo con los matemáticos. Tenemos la opción de usar los paréntesis, pero también existe predefinido un orden en que deben ser tomados los operadores. A esto se le denomina **Precedencia de Operadores**. Si el ordenador encuentra diversos operadores en una expresión sin paréntesis (con paréntesis, nosotros decimos qué se hace primero y qué se hace después) tomará los operadores según el cuadro siguiente, teniendo mayor precedencia los operadores más altos del esquema. Para operadores en la misma fila, se tomará el operador que primero aparezca en la expresión.

.	[]	( )			
++	--				
!	~	Instanceof			
*	/	%			
+	-				
<<	>>	>>>			
<	<	<=	>=	= =	!=
&	^	|			
&&	||				
?:					

2.4.13 Casting

En ciertas ocasiones tendremos que asignar una variable de un tipo a otra variable de otro tipo diferente. JAVA lo realiza automáticamente. Si asignamos un INT a un FLOAT, JAVA automáticamente lo convierte poniendo los decimales y la coma.

Pero hay ciertos casos en que JAVA no hace dicha transformación, y debemos nosotros realizar "a mano" dicha transformación anteponiendo, entre paréntesis, el tipo al que queremos transformar. Ejemplo:

```
int a;
long b;
a=3;
b=(long) a;
```

2.4.14 Sentencias de control

Estudiaremos a continuación las sentencias de control que son las instrucciones que hacen los caminos por los que pasará la ejecución del programa dependiendo de valores, variables, condiciones, etc.

2.4.14.1 Estructuras condicionales

Las **estructuras condicionales** sirven para bifurcar la ejecución de un programa según unas condiciones en concreto. Llegado a un punto, evaluamos una condición. Si se cumple haremos una cosa y si es falsa haremos otra u otras.

En JAVA tenemos varias instrucciones condicionales:

```
if (expresión)
{
      sentencias;
}
```

Si se cumple la expresión, realizamos las sentencias entre las llaves. Si no se cumple, seguimos por las sentencias que haya después del bloque de las llaves.

```
if (expresión)
{
      sentencias;
}
else
{
      sentencias_alternativas;
}
```

Si se cumple la expresión, realizamos las sentencias entre las llaves. Si no se cumple, realizamos las sentencias_alternativas que hay después del else. Tanto en un caso como el otro, seguiremos después tras el bloque de llaves del else. Dentro

del bloque else, podemos tener a su vez más sentencias if-else anidados, pero si anidamos demasiados, el código empieza a hacerse confuso. Si debemos evaluar una condición con muchos estados posibles, lo mejor es utilizar la instrucción condicional múltiple, que es de la forma siguiente:

```
switch (expresión)
{
      case valor1:
            sentencias1;
            break;
      case valor2:
            sentencias2;
            break;
      ...
      default:
            sentenciasN;
}
```

Al llegar a esta estructura, el programa evalúa la expresión entre paréntesis, que puede tener varios valores. En función del valor que tenga (valor1, valor2,...) se ejecutarán las sentencias correspondientes al case que posee el valor de la expresión. En caso de que la expresión no coincida con ningún case, se ejecutarán las sentencias agrupadas en el default. La sentencia break, sirve para que una vez realizadas las sentencias de un case, no realice las sentencias de los siguientes case, sino que sigue por las sentencias que haya a continuación del bloque default.

Para acabar este apartado de sentencias condicionales vamos a hablar de la evaluación de las condiciones que dilucidan qué camino se debe seguir en la ejecución de un programa.

Si tenemos más de una condición, la evaluación de las mismas se hace en **cortocircuito**. Esto significa que mientras la condición completa no sea verdadera o falsa, seguir evaluando más condiciones. Pero en el momento que ya se tenga claro el sentido de la condición, dejar de evaluar. Por ejemplo,

```
if((a==3)&&(b==4)&&(c==6))
```

La ejecución del programa al llegar a esta sentencia, evalúa en primer lugar el valor de a. Si vale 3, evalúan la siguiente condición, si b vale 4 o no. Pero si a, no vale 3, al tener operadores AND, con que uno sea falso, todo lo demás será falso. Es por ello que, si a no vale 3, toda la condición es falsa. Por ello, para qué seguir evaluando lo demás. En el caso de OR, en cuanto una condición sea verdadera, se para la evaluación del resto, pues ya será verdadera la condición al completo.

Debido a este comportamiento, en la construcción de condiciones compuestas, debemos tener en cuenta el orden de evaluación de las condiciones simples. Debemos buscar siempre que el cortocircuito se ejecute lo antes posible, haciendo la evaluación de las instrucciones condicionales lo más eficientes posible.

2.4.14.2 Estructuras repetitivas

Las estructuras repetitivas sirven para realizar un número de veces, una serie de instrucciones o sentencias. La repetición se puede realizar un número fijo de veces o bien estar repitiéndose hasta que se cumpla (o deje de cumplirse) una condición.

En primer lugar nos encontramos con la estructura for, que nos permitirá repetir un número de veces preestablecido una serie de sentencias. Se construye de la siguiente manera:

```
for(iterador=valor_inicial; condición; incremento)
{
      sentencias;
}
```

La estructura for funciona así: a una variable llamada iterador se le da un valor inicial. Las sentencias se estarán repitiendo mientras se cumpla la condición. El iterador se va actualizando según se indique en el incremento.

La estructura while nos repetirá unas sentencias mientras se cumpla una condición.

```
while (condición)
{
      sentencias;
}
```

Las sentencias se repiten indefinidamente mientras se cumpla la condición. En las sentencias, habrá algún caso que haga que en algún momento, la condición deje de ser verdadera. En otro caso tendremos un bucle infinito.

La estructura do/while tiene comportamiento parecido a while, pero con la diferencia que éste evalúa primero la condición y si se cumple, entra en el bucle. La estructura do/while realiza una vez las sentencias del bloque y luego evalúa la condición para saber si debe seguir haciendo las sentencias o no.

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

2.4.15 Clases

En JAVA todo está dentro de clases. Las clases se definen de la siguiente forma:

```
class nombre_clase
{
      cuerpo clase;
}
```

Cada clase deriva directa o indirectamente de la clase Object. Si al declarar una clase no es especifica de qué clase deriva, se supone que lo hace de esta clase Object.

Existen 4 tipos de clases:

- **Public**: Clases accesibles desde otras clases
- **Abstract**: Clase que tiene al menos un método abstracto. Un método abstracto es un método que solamente se define en una clase. Su implementación se realizará en una subclase de ésta otra.
- **Final**: Clase que finaliza una serie de herencias. De esta clase no hereda nadie.
- **Synchronizable**: Todos los métodos de la clase son sincronizados, es decir, que deben ser accedidos de uno en uno por varias tareas.

Para cada clase disponemos de 4 niveles de acceso o alcance aplicables tanto a los atributos como a los métodos:

- **Private**: Los atributos y los métodos privados sólo pueden ser accedidos desde dentro la propia clase. Normalmente, los atributos son Privados.
- **Public**: Cualquier clase desde cualquier sitio puede acceder a los atributos y a los métodos de esta clase declarada con este acceso. Normalmente, los métodos son en su mayoría Públicos.
- **Protected**: Sólo las clases que han heredado de ésta pueden acceder a los atributos y a los métodos.
- **Package**: Las clases dentro de un mismo paquete pueden acceder a los atributos y a los métodos de todas las clases de dicho paquete.

Tenemos una serie de características de las clases JAVA:

- Todas las variables y funciones o procedimientos deben pertenecer a una clase, es decir no existen elementos globales.

- Si una clase deriva de otra, hereda todas sus variables y todos sus métodos.
- Una clase solamente puede heredar de una clase. Si al definir una clase, no se indica de qué clase deriva, se considerará que lo hace de Object.
- En un fichero se pueden declarar varias clases pero solamente una puede ser pública, y su nombre dará nombre al fichero en conjunto.
- Si una clase contenida en un fichero no es pública, no tienen por qué tener el mismo nombre.
- Las clases se pueden agrupar en paquetes introduciendo una línea al principio del fichero de la forma: **package** *nombre_paquete*;

Existe una palabra clave usada para hacer referencia a los miembros de la propia clase, tanto atributos como métodos. Esa palabra es *this*. Veremos su uso con el ejemplo:

```
public class Clase
{
	int atributo;
	//Constructor sin parámetros
	public Clase()
	{
		atributo=10;
	}
	//Constructor con parámetro
	public Clase(int i)
	{
		this.atributo=i;
	}
}
```

Otra palabra clave que haremos uso cuando estemos utilizando la herencia es *super*. Esta palabra clave sirve para hacer referencia a algún método de la clase padre de la que estamos trabajando. Siguiendo el ejemplo anterior:

```
import Clase;
public class NuevaClase extends Clase
{
	public void NuevaClase( int valor)
	{
		super.Clase(valor);
	}
}
```

Esta palabra resulta bastante útil cuando existen dos métodos sobrecargados, uno en la propia clase y otra en la clase de la que heredó,

permitiéndonos hacer referencia al método de la clase padre y no al método de la propia clase que sería el usado por defecto.

2.4.16 Herencia

La herencia se implementa de la siguiente manera:

```
class Clase_hija extends Clase_padre
{
      cuerpo clase;
}
```

La herencia es usada para crear nuevas clases a partir de otras ya existente. De esta manera reutilizamos código de forma bastante eficiente. La forma de trabajar de la herencia es la siguiente:

- Se crea una clase con unos atributos y unos métodos.
- Al crear otra clase (clase hija) a partir de la clase anterior (clase padre), la clase hija "hereda" todos los atributos y métodos de la clase padre, además de poder definirse, dentro de la clase hija, nuevos atributos y métodos.

2.5 Programando en JAVA

Existen muchos compiladores y entornos de programación de JAVA, pero en términos generales todos disponen de las mismas utilidades y se manejan de manera similar.

Los ficheros fuentes se deberán llamar con el mismo nombre de la clase que contiene con la extensión *.java*.

Sea cual sea nuestra forma de programar, partiremos de un fichero fuente donde escribiremos las instrucciones JAVA que guardaremos con extensión *.java.*

A continuación, compilaremos dicho fichero, obteniendo otro fichero llamado igual que el anterior pero con extensión *.class* (el antes llamado Byte Code).

Dependiendo del entorno de programación en el que estemos, se nos indicarán los errores de alguna forma u otra, normalmente en una pantalla con las líneas donde se producen los errores y una breve descripción de los mismos.

Una vez obtenido el *.class* sin errores ya lo podremos ejecutar.

En la práctica obligatoria de este tema se va a trabajar con el entorno de desarrollo para JAVA, con lo cual, dejamos para dicha actividad todo lo concerniente a la programación práctica de JAVA.

2.6 Clases de JAVA

A continuación, podemos ver una serie de clases definidas en JAVA de las que podremos hacer uso, bien para usar directamente, bien para crear, a partir de ellas de nuestras propias clases. Estas clases poseen una serie de métodos ya definidos, que además nosotros podremos sobrecargar en nuestras propias clases.

2.6.1 Clase Object

La clase Object es la clase de la que heredan todas las demás clases, ya sea directamente, ya sea como clase raíz de un árbol de herencias.

Esta clase posee toda una serie de métodos que son comunes, debido a la herencia, a todas las clases que creemos o que ya estén implementadas.

- Public boolean equals (Object o) → Nos indicará si dos objetos son iguales en tipo y en contenido.
- Public final native Class getClass()→ Nos indicará la clase a la que pertenece un objeto.
- Public String toString()→ Para convertir cualquier objeto de cualquier clase a una cadena.

2.6.2 Clases Abstractas

Las clases abstractas se declaran simplemente sin implementar los métodos que la definen. Serán en las clases que hereden de estas clases abstractas, en las que se implementarán los métodos antes declarados. Los Métodos declarados en las Clases Abstractas se denominan Virtuales o Puras. Una Clase será Abstracta si todos sus métodos son Puros. Un ejemplo de estas clases es la clase Graphics, que tiene todos sus métodos declarados, pero no implementados. Nosotros tendremos que implementarlos en nuestras clases que hagan uso de aquellas.

Las Clases Abstractas NO tienen objetos.

2.6.3 Interfaces

Las clases Interfaces son la respuesta de JAVA a la Herencia múltiple. Estas clases proporcionan un mecanismo para abstraer los métodos a un nivel superior. Una Interfaz contiene una serie de métodos que se implementan en otro lugar. Estos métodos son public, static y final.

Una Clase Interfaz se declara así:

```
public interface ClaseInterfaz
{
      ...
}
```

Las clases que quieran utilizar los métodos de una clase Interfaz deben declararse de la siguiente forma:

```
class Clase1 implements ClaseInterfaz
{
      ...
}
```

La clase Clase1 implementa los métodos declarados en la interfaz.

Entonces, ¿en qué se diferencia una clase Interfaz de una clase Abstracta? Pues que una clase Interfaz puede ser implementada por varias clases diferentes, mientras que una clase Abstracta sólo podrá ser implementada por una clase que herede de ella. Hemos conseguido así la Herencia Múltiple sin usar Herencia.

Una clase puede ser implementada por varias clases interfaces, mientras que sólo puede ser heredada de otra única clase.

2.6.4 Paquetes

Los Paquetes de JAVA contienen una serie de clases muy útiles que nos permitirán realizar muchos tipos de trabajo sin necesidad de realizar nosotros mismo las implementaciones. El entorno de desarrollo estándar de Java comprende ocho paquetes:

1 - El Paquete de Lenguaje Java: El paquete de lenguaje Java, también conocido como java.lang, contiene las clases que son el corazón del lenguaje Java. Las clases de este paquete se agrupan de la siguiente manera:

- **Object**: La abuela de todas las clases: la clase de la que parten todas las demás.
- **Tipos de Datos Encubiertos**: Una colección de clases utilizadas para encubrir variables de tipos primitivos: Boolean, Character, Double, Float, Integer y Long. Cada una de estas clases es una subclase de la clase abstracta Number.
- **Strings**: Dos clases que implementan los datos de caracteres: Las Clases String y StringBuffer
- **System y Runtime**: Estas dos clases permiten a los programas utilizar los recursos del sistema. System proporciona un interface de programación

independiente del sistema para recursos del sistema y Runtime da acceso directo al entorno de ejecución específico de un sistema.

- **Thread**: Las clases Thread, ThreadDeath y ThreadGroup implementan las capacidades multitareas tan importantes en el lenguaje Java. El paquete java.lang también define el interface Runnable. Esta interface es conveniente para activar la clase Java sin subclasificar la clase Thread.
- **Class**: La clase Class proporciona una descripción en tiempo de ejecución de una clase y la clase ClassLoader permite cargar clases en los programas durante la ejecución.
- **Math**: Una librería de rutinas y valores matemáticos como pi.
- **Exceptions, Errors y Throwable**: Cuando ocurre un error en un programa Java, el programa lanza un objeto que indica qué problema era y el estado del intérprete cuando ocurrió el error. Sólo los objetos derivados de la clase Throwable pueden ser lanzados. Existen dos subclases principales de Throwable: Exception y Error. Exception es la forma que deben intentar capturar los programas normales. Error se utiliza para los errores catastróficos: los programas normales no capturan Errores. El paquete java.lang contiene las clases Throwable, Exception y Error, y numerosas subclases de Exception y Error que representan problemas específicos.
- **Process**: Los objetos Process representan el proceso del sistema que se crea cuando se utiliza el sistema en tiempo de ejecución para ejecutar comandos del sistema. El paquete java.lang define e implementa la clase genérica Process.

El compilador importa automáticamente este paquete. Ningún otro paquete se importa de forma automática.

2 - El Paquete I/O de Java: El paquete I/O de Java (java.io) proporciona un juego de canales de entrada y salida utilizados para leer y escribir ficheros de datos y otras fuentes de entrada y salida.

3 - El Paquete de Utilidades de Java: Este paquete, java.util, contiene una colección de clases útiles. Entre ellas se encuentran muchas estructuras de datos

genéricas (Dictionary, Stack, Vector, Hashtable) un objeto muy útil para dividir cadenas y otro para la manipulación de calendarios. El paquete java.util también contiene el interface Observer y la clase Observable que permiten a los objetos notificarse unos a otros cuando han cambiado.

4 - El Paquete de Red de Java: El paquete java.net contiene definiciones de clases e interfaces que implementan varias capacidades de red. Las clases de este paquete incluyen una clase que implementa una conexión URL. Se pueden utilizar estas clases para implementar aplicaciones cliente-servidor y otras aplicaciones de comunicaciones

5 - El Paquete Applet: Este paquete contiene la clase Applet. La clase que se debe utilizar si se quiere escribir un applet. En este paquete se incluye el interface AudioClip que proporciona una abstracción de alto nivel para audio.

6, 7 y 8 - Los Paquetes de Herramientas para Ventanas Abstractas: Tres paquetes componen las herramientas para Ventanas Abstractas: java.awt, java.awt.image, y java.awt.peer.

2.6.5 El paquete AWT

El paquete java.awt proporciona elementos GUI utilizados para obtener información y mostrarla en la pantalla como ventanas, botones, barras de desplazamiento, etc. será estudiado en profundidad a partir del segundo trimestre.

13/08/2017