

Lenguaje de Programación Orientada a Objetos

A partir de este punto, habiendo analizado las características del paradigma orientado a objetos y el lenguaje estándar que se utiliza para modelar (UML), estudiaremos el lenguaje de programación Java, que está basado en el paradigma orientado a objetos que estudiamos anteriormente.

Surgimiento del Lenguaje

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

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Es por esto que desarrollan un código “neutro” que no depende del tipo de electrodoméstico, el cual se ejecuta sobre una “máquina hipotética o virtual” denominada Java Virtual Machine (JVM). Es la JVM quien interpreta el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: “Write Once, Run Everywhere”. A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje. Java, como lenguaje de programación para computadoras, se introdujo a finales de 1995. Si bien su uso se destaca en la Web, sirve para crear todo tipo de aplicaciones (locales, intranet o internet).

Entendiendo Java

Java es un lenguaje compilado e interpretado; esto significa que el programador escribirá líneas de código utilizando un determinado programa que se denomina IDE (Ambiente de Desarrollo Integrado; Integrated Development Environment por sus siglas en inglés). Este programa le provee al programador un espacio de trabajo para crear código que se almacena en archivos de formato .java.

Luego, para poder correr el código y ejecutar el programa creado, se debe realizar primero lo que se denomina Compilación; para eso un programa denominado compilador será el encargado de revisar la sintaxis del código y si está correcto transformará el archivo .java en otro archivo de formato .class que se denominan: “bytecodes”. Una vez realizada la compilación, será la Máquina Virtual (Java Virtual Machine - JVM), la encargada de leer los archivos .class y transformarlos en el código entendido por la CPU de la computadora.

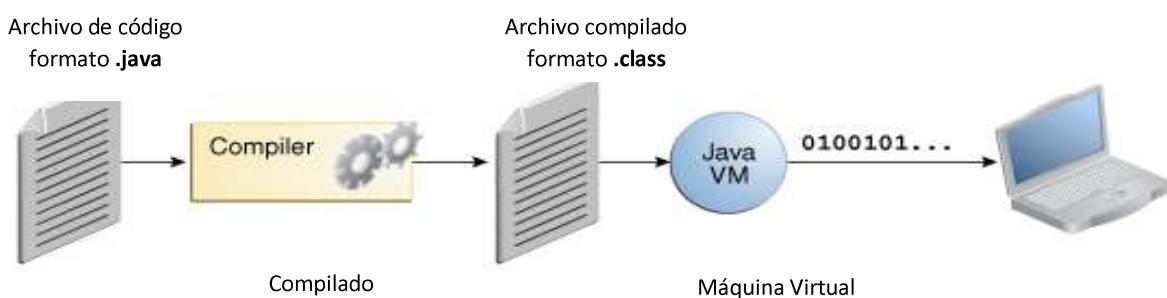


Fig. 45- Funcionamiento general del Lenguaje Java

El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. Oracle, quien compró a Sun (la creadora de Java), distribuye gratuitamente el Java Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (es el denominado Debugger). Es tarea muy común de un programador realizar validaciones y pruebas del código que construye, el debugger permite realizar una prueba de escritorio automatizada del código, lo cual ayuda a la detección y corrección de errores. En el momento de escribir este trabajo las herramientas de desarrollo: JDK, van por la versión 1.8. Estas herramientas se pueden descargar gratuitamente de <http://www.oracle.com/technetwork/java>.

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una Consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Estas herramientas brindan una interfaz gráfica para facilitar y agilizar el proceso de escritura de los programas. El entorno elegido para este módulo, por su relevancia en el mercado y su licencia gratuita se denomina Netbeans. El mismo se encuentra disponible para descargar de forma gratuita en www.netbeans.org. En la Guía Práctica abordaremos la forma de instalación y configuración de estas herramientas.

El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión *.java). Si no encuentra errores en el código genera los ficheros compilados (con extensión *.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun dicho compilador se llama javac.exe si es para el sistema operativo Windows.

La Java Virtual Machine

Tal y como se ha comentado al comienzo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se plantea la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código “neutro” el cual estuviera preparado para ser ejecutado sobre una “máquina hipotética o virtual”, denominada Java Virtual Machine (JVM). Es esta JVM quien interpreta este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

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

Características de Java

Java es un lenguaje:

- **Orientado a objetos:** Al contrario de otros lenguajes como C++, Java no es un lenguaje modificado para poder trabajar con objetos, sino que es un lenguaje creado originalmente para trabajar con objetos. De hecho, todo lo que hay en Java son objetos.
- **Independiente de la plataforma:** Debido a que existen máquinas virtuales para diversas plataformas de hardware, el mismo código Java puede funcionar prácticamente en cualquier dispositivo para el que exista una JVM.
- **Compilado e Interpretado:** La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera **bytecode** es interpretado por una máquina virtual, como se vio anteriormente. De este modo se consigue la independencia de la máquina: el código compilado se ejecuta en máquinas virtuales que sí son dependientes de la plataforma. Para cada sistema operativo distintos, ya sea Microsoft Windows, Linux, OS X, existe una máquina virtual específica que permite que el mismo programa Java pueda funcionar sin necesidad de ser recompilado.
- **Robusto:** Su diseño contempla el manejo de errores a través del mecanismo de Excepciones y fuerza al desarrollador a considerar casos de mal funcionamiento para reaccionar ante las fallas.
- **Gestiona la memoria automáticamente:** La máquina virtual de Java gestiona la memoria dinámicamente como veremos más adelante. Existe un **recolector de basura** que se encarga de liberar la memoria ocupada por los objetos que ya no están siendo utilizados.
- **No permite el uso de técnicas de programación inadecuadas:** Como veremos más adelante, todo en Java se trata de objetos y clases, por lo que para crear un programa es necesario aplicar correctamente el paradigma de objetos.
- **Multihilos (multithreading):** Soporta la creación de partes de código que podrán ser ejecutadas de forma paralela y comunicarse entre sí.
- **Cliente-servidor:** Java permite la creación de aplicaciones que pueden funcionar tanto como clientes como servidores. Además, provee bibliotecas que permiten la comunicación, el consumo y el envío de datos entre los clientes y servidores.
- **Con mecanismos de seguridad incorporados:** Java posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar. Además, posee un *gestor de seguridad* con el que puede restringir el acceso a los recursos del sistema.
- **Con herramientas de documentación incorporadas:** Como veremos más adelante, Java contempla la creación automática de documentación asociada al código mediante la herramienta Javadoc.



Aplicaciones de java

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa. Java⁸ se encuentra aplicado en un amplio rango de dispositivos desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet.

- El 97% de los escritorios empresariales ejecutan Java.
- Es la primera plataforma de desarrollo.
- 3 mil millones de teléfonos móviles ejecutan Java.
- El 100% de los reproductores de Blu-ray incluyen Java.
- 5 mil millones de Java Cards en uso.
- 125 millones de dispositivos de televisión ejecutan Java.

Antes de comenzar

Como se detalló en el apartado “Entendiendo Java”, los archivos de código escritos por el programador que se denominan: *archivos de código fuente*, en Java son los archivos .java. Éstos se podrían crear utilizando simplemente un editor de texto y guardándolo con la extensión .java. Por lo tanto, esto se puede hacer utilizando cualquier editor de texto como el bloc de notas de Windows. Para facilitar la tarea de desarrollo en la práctica, como vimos, se utilizan IDEs (entornos de desarrollo integrados) que ofrecen herramientas como coloreado de palabras clave, análisis de sintaxis en tiempo real, compilador integrado y muchas otras funciones que usaremos.

Al ser un lenguaje multiplataforma y especialmente pensado para su integración en redes, la codificación de texto utiliza el estándar Unicode, lo que implica que los programadores y programadoras hispanohablantes podemos utilizar sin problemas símbolos de nuestra lengua como la letra “ñ” o las vocales con tildes o diéresis a la hora de poner nombre a nuestras variables.

Algunos detalles importantes son:

- En java (como en C) hay diferencia entre mayúsculas y minúsculas por lo que la variable nombreCompleto es diferente a nombreCompleto.
- Cada línea de código debe terminar con un; (punto y coma)
- Una instrucción puede abarcar más de una línea. Además, se pueden dejar espacios y tabuladores a la izquierda e incluso en el interior de la instrucción para separar elementos de la misma.
- A veces se marcan bloques de código, es decir código agrupado. Cada bloque comienza con llave que abre, "{" y termina con llave que cierra, "}"

⁸ Datos obtenidos en <https://www.java.com/es/about/>

Declaración de Clases

En el caso más general, la declaración de una clase puede contener los siguientes elementos:

```
[public] [final | abstract] class NombreDeLaClase [extends ClaseMadre] [implements Interfase1 [, Interfase2 ]...]
```

Donde las porciones encerradas entre corchetes son opcionales a optar entre las posibilidades separadas por la barra vertical.

O bien, para interfaces:

```
[public] interface NombreDeLaInterface [extends InterfaceMadre1 [, InterfaceMadre2 ]...]
```

Como se ve, lo único obligatorio es la palabra reservada `class` y el nombre que queramos asignar a la clase. Las interfaces son un caso de tipo particular que veremos más adelante.

De esta forma, podríamos declarar la clase `Perro` como sigue:

```
public class Perro
{
    // esto es un comentario
    // aquí completaremos luego el cuerpo de la clase
}
```

De esta forma una instancia (objeto) de la clase `Perro` puede declararse utilizando el operador `new` de la siguiente forma, ocupando un espacio en memoria que luego podrá ser accedido mediante el nombre `miPerro`.

```
Perro miPerro = new Perro();
```

En el ejemplo anterior podemos identificar, además de la declaración de la clase `Perro` como de tipo `public` (veremos luego su significado), el uso de llaves (`{ y }`) para encerrar los componentes de la clase y las barras oblicuas para definir comentarios de una sola línea. En caso de querer escribir comentarios más extensos podemos encerrar el bloque entre `/* y */` de la siguiente forma:

```
/* Este es un comentario más extenso que ocupa más de una línea:
Al utilizar los símbolos barra oblicua y asterisco, indicamos que lo que sigue a continuación
es comentario del código. Los comentarios de código se utilizar frecuentemente en el ámbito de
la programación ya que es necesario aclarar distintos aspectos de la codificación, como, por
ejemplo: cuáles son los objetivos de los métodos, qué parámetros espera un método, explicar
una sentencia en particular, etc. Los comentarios son muy útiles para explicar utilizando el
lenguaje natural qué es lo que se está programando y de esta manera, poder en un futuro leer
rápidamente los comentarios y entender y recordar cuál era el objetivo del código. Además, en
esta disciplina realizar comentarios durante el proceso de programación es una muy buena
práctica que se utiliza frecuentemente para que en el caso de que otro programador tenga que
continuar con el código pueda entender cuál fue el pensamiento del que creó originariamente el
mismo.*/
```

Los comentarios enmarcados entre /* y */ no se pueden anidar. Los comentarios tanto sean de una línea o de bloques serán ignorados por el compilador a la hora de generar el **bytecode** a ser ejecutado en la máquina virtual, sólo sirven a los fines de lectura y comprensión del código por parte del equipo desarrollador. Más adelante veremos una forma especial de escritura de comentarios denominada **JavaDoc** que nos permitirá generar la documentación asociada al programa con las definiciones de las clases, métodos y demás de forma automatizada. Sintácticamente los comentarios JavaDoc comienzan con barra y dos asteriscos y terminan con asterisco barra:

```
/**  
...  
*/
```

Tipos de Datos

Los tipos de variables disponibles son básicamente 3:

- Tipos básicos (no son objetos)
- Arreglos (arrays o vectores)
- Clases e Interfaces

Con lo que vemos que cada vez que creamos una clase o interface estamos definiendo un nuevo tipo. Siempre es aconsejable asignar un valor por defecto en el momento de declaración de una variable. En algunos casos, incluso, se producirá un error durante la compilación si hemos olvidado inicializar el valor de alguna variable y tratamos de manipular su contenido.

Tipos básicos de datos

Nombre	Declaración	Rango	Descripción
Booleano	boolean	true - false	Define una bandera que puede tomar dos posibles valores: true o false.
Byte	byte	[-128 .. 127]	Representación del número de menor rango con signo.
Entero pequeño	short	[-32,768 .. 32,767]	Representación de un entero cuyo rango es pequeño.
Entero	int	[-2 ³¹ .. 2 ³¹ -1]	Representación de un entero estándar. Este tipo puede representarse sin signo usando su clase Integer a partir de la Java SE 8.
Entero largo	long	[-2 ⁶³ .. 2 ⁶³ -1]	Representación de un entero de rango ampliado. Este tipo puede representarse sin signo usando su clase Long a partir de la Java SE 8.
Real	float	[±3,4·10 ⁻³⁸ .. ±3,4·10 ³⁸]	Representación de un real estándar. Recordar que al ser real, la precisión del dato contenido varía en función del tamaño del número: la precisión se amplía con números más próximos a 0 y disminuye cuanto más se aleja del mismo.

Real largo	double	$[\pm 1,7 \cdot 10^{-308} .. \pm 1,7 \cdot 10^{308}]$	Representación de un real de mayor precisión. Double tiene el mismo efecto con la precisión que float.
Carácter	char	<code>['\u0000' .. '\uffff'] o [0 .. 65.535]</code>	Carácter o símbolo. Para componer una cadena es preciso usar la clase String, no se puede hacer como tipo primitivo.

Tabla 9 – Tipos de básicos de datos en JAVA

Tipos de datos referencia

En Java los objetos, instancias de clases, se manejan a través de referencias. Cuando se crea una nueva instancia de una clase con el operador *new*, este devuelve una referencia al tipo de la clase. Para aclararlo veamos un ejemplo:

Si tenemos la clase Punto definida de la siguiente manera:

```
public class Punto {
    private float float x;
    private float y;
    ...
    public void setX(float x) {
        this.x = x;
    }
    ...
}
```

Punto unPunto = new Punto();

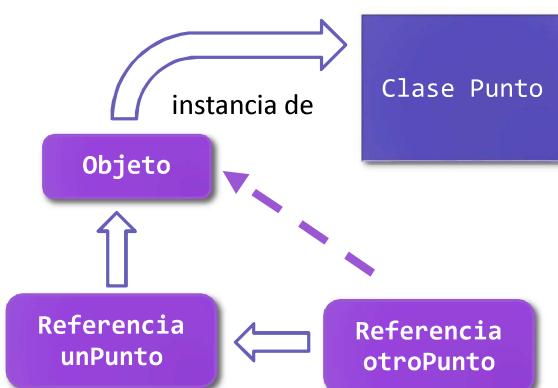
El operador *new* reserva espacio en memoria para contener un objeto del tipo Punto y devuelve una referencia que se asigna a *unPunto*, con los valores de atributos que se definieron en el constructor. A partir de aquí, accedemos al objeto a través de su referencia. Es posible, por tanto, tener varias referencias al mismo objeto. Presta atención al siguiente fragmento de código.

```
Punto unPunto = new Punto();
unPunto.print();

Punto otroPunto = unPunto;
otroPunto.setX(1.0f);
otroPunto.setY(2.0f);
otroPunto.print();
```

La salida por pantalla es:

Coordenadas del punto (0.0f,0.0f)
 Coordenadas del punto (1.0f,2.0f)



Como las dos referencias: *unPunto* y *otroPunto*, hacen referencia a la misma instancia, los cambios sobre el objeto se pueden realizar a través de cualquiera de ellas.

Conversión de tipos de datos o Casting

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo, de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia de una clase común o la implementación de una interfaz compatible. Veremos más sobre este último caso cuando hablaremos de Polimorfismo.

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

Por ejemplo:

```
int a = 5;
float b = 1.5;
float c = a * b; // producto de distintos tipos de datos primitivos convertidos
automáticamente
```

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

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

El Recolector de basura

Los objetos que dejan de estar referenciados a través de alguna variable no se pueden volver a recuperar. Para que estos objetos “desreferenciados” no ocupen memoria, un recolector de basura se encarga de «destruirlos» y liberar la memoria que estaban ocupando. Por lo tanto, para «destruir» un objeto basta con asignar a su variable de referencia el valor *null* como se puede ver en el siguiente ejemplo.

```
Punto unPunto = new Punto(1.0f, 2.0f);
Punto otroPunto = new Punto(1.0f, -1.0f);
unPunto = new Punto(2.0, 2.0f); // El punto (1.0f, 2.0f) se pierde
otroPunto = null; // El punto (1.0f, -1.0f) se pierde
```

Ámbito de las variables

Toda variable tiene un ámbito. Esto es la parte del código en la que una variable se puede utilizar. De hecho, las variables tienen un ciclo de vida:

1. En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia).
2. Se la asigna su primer valor (la variable nace).

3. Se la utiliza en diversas sentencias.
4. Cuando finaliza el bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.

Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida.

El ámbito de las variables está determinado por el bloque de código donde se declaran y todos los bloques que estén anidados por debajo de este. Presta atención al siguiente fragmento de código:

```
{
    // Aquí tengo el bloque externo
    int entero = 1;
    Punto unPunto = new Punto();
    {
        // Y aquí tengo el bloque interno
        int entero = 2; // Error ya está declarada
        unPunto = new Punto(1.0f, 1.0f); // Correcto
    }
}
```

Operadores

Las variables se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, multiplican, etc. y a veces se realizan operaciones más complejas.

Operadores aritméticos

En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros, también llamado módulo:

OPERADOR	DESCRIPCIÓN
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de una división entre enteros (en otros lenguajes denominado mod)

Tabla 10 – Tipos de operadores aritméticos en JAVA

Cabe destacar que el operador % es de uso exclusivo entre enteros. $7 \% 3$ devuelve 1 ya que el resto de dividir 7 entre 3 es 1. Al valor obtenido lo denominamos módulo (en otros lenguajes en vez del símbolo % se usa la palabra clave mod) y a este operador a veces se le denomina “operador módulo”.

Las operaciones con operadores siguen un **orden de prelación o de precedencia** que determinan el orden con el que se ejecutan. Si existen expresiones con varios operadores del mismo nivel, la operación se ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión:

$3 * 2 / 7 + 2$

se ejecutará en el orden que esperas, es posible agrupar términos utilizando paréntesis. Por ejemplo:

$3 * ((a / 7) + 2)$

Se podrían mostrar también los resultados de las dos expresiones, para ver las diferencias.

Operadores lógicos

En Java disponemos de los operadores lógicos habituales en lenguajes de programación como son “es igual”, “es distinto”, menor, menor o igual, mayor, mayor o igual, and (y), or (o) y not (no). La sintaxis se basa en símbolos como veremos a continuación y cabe destacar que hay que prestar atención a no confundir == con = porque implican distintas cosas.

OPERADOR	DESCRIPCIÓN
==	Es igual
!=	Es distinto
<, <=, >, >=	Menor, menor o igual, mayor, mayor o igual
&&	Operador and (y)
	Operador or (o)
!	Operador not (no)

Tabla 11 – Tipos de operadores lógicos en JAVA

Los conectivos lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
int edad = 21;
mayorDeEdad = edad >= 18; //mayorDeEdad será true menorDeEdad = !mayorDeEdad;
//menorDeEdad será false
```

El operador **&&** (AND) sirve para evaluar dos expresiones de modo que, si ambas son ciertas, el resultado será true sino el resultado será false. Ejemplo:

```
boolean carnetConducir=true;
int edad=20;
boolean puedeConducir= (edad>=18) && carnetConducir; //Si la edad es de al
menos 18 años y carnetConducir es //true, puedeConducir es true
```

El operador **||** (OR) sirve también para evaluar dos expresiones. El resultado será true si al menos una de las expresiones es true. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;
malTiempo= nieva || llueve || graniza;
```

Los operadores **&&** y **||** se llaman operadores en cortocircuito porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo:

$$(a == b \&\& c != d \&\& h >= k)$$

tiene tres evaluaciones: la primera comprueba si la variable a es igual a b. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

En un caso como:

$$(a < b \mid\mid c != d \mid\mid h <= k)$$

se evalúa si a es menor que b. Si se cumple esta condición el resultado de la expresión es verdadero y no se evalúan las otras dos condiciones posteriores.

Operadores de asignación

Permiten asignar valores a una variable. El fundamental es “**=**”. Pero sin embargo se pueden usar expresiones más complejas como:

```
x = 5;
x += 3; // x vale 8 ahora
```

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo **x+=3**, que **x=x+3**).

Eso se puede hacer también con todos estos operadores:

Nombre	Operador	Ejemplo	Equivalencia
Operadores aritméticos			
Suma y asignación	$+=$	$a += b$	$a = a+b$
Resta y asignación	$-=$	$a -= b$	$a = a-b$
Multiplicación y asignación	$*=$	$a *= b$	$a = a*b$
División y asignación	$/=$	$a /= b$	$a = a/b$
Resto de la división y asignación	$%=$	$a %= b$	$a = a \% b$
Operadores a nivel de bits			
AND binario y asignación	$\&=$	$a \&= b$	$a = a \& b$
OR binario y asignación	$ =$	$a = b$	$a = a b$
XOR binario y asignación	$^=$	$a ^= b$	$a = a ^ b$
Desplazamiento de bits hacia la izquierda en b posiciones y asignación	$<<=$	$a <<= b$	$a = a << b$
Desplazamiento de bits hacia la derecha en b posiciones y asignación	$>>=$	$a >>= b$	$a = a >> b$

Tabla 12 – Operadores de Asignación en JAVA

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5; // todas valen 5
```

Otros operadores de asignación son “++” (incremento) y “--”(decremento), que incrementan o decrementan en una unidad el valor de la variable. Son muy utilizados en algoritmos de recorrido o dentro de estructuras de control como el ciclo for que vimos en el módulo de Técnicas de Programación.

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo `x++` o `++x`, la diferencia estriba en el modo en el que se comporta la asignación en cuanto al orden en que es evaluada.

Ejemplo:

```
int x=5, y=5, z;
z=x++; // z vale 5, x vale 6
z=++y; // z vale 6, y vale 6
```

Operador ternario

Este operador (conocido como if de una línea) permite devolver un valor u otro según el valor de la expresión analizada. Su sintaxis es la siguiente:

Expresionlogica ? valorSiVerdadero: valorSiFalso;

Como, por ejemplo:

```
float precioDeLista = aplicaDescuento == true ? 123 : 234;
```

En el caso de que la variable `aplicaDescuento` tenga valor verdadero, el precio de lista del producto que estamos analizando será menor; es importante destacar que la asignación anterior es equivalente a utilizar condicionales como veremos en las siguientes secciones.

Arrays

En Java los arrays son objetos, instancias de la clase `Array`, la cual dispone de ciertos métodos útiles. La declaración sigue la siguiente sintaxis: se debe declarar el tipo base de los elementos del array. El tipo base puede ser un tipo primitivo o un tipo de referencia:

```
int arrayDeEnteros[] = null; // Declara un array de enteros
Punto arrayDePuntos[] = null; /* Declara un array de referencias a Puntos */
```

La creación del array se hace, como con cualquier otro objeto, mediante el uso del operador `new()`:

```
arrayDeEnteros = new int[100]; /* Crea el array con espacio para 100 enteros */
arrayDePuntos = new Punto[100]; /* Crea el array con espacio para 100 referencias
a Punto */
```

En el primer caso se reserva espacio para contener 100 enteros. En el segundo caso se crea espacio para contener 100 referencias a objetos de la clase `Punto`, pero no se crea cada uno de esos 100 objetos. En el siguiente ejemplo se muestra como se crea cada uno de esos 100 objetos de la clase `Punto` y se asignan a las referencias del array.

```
for(int i = 0; i < 100; i++)
    arrayDePuntos[i] = new Punto();
```

Los arrays se pueden iniciar en el momento de la creación, como en el siguiente ejemplo:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
Punto arrayDePuntos[] = {new Punto(), new Punto(1.0f, 1.0f)};
```

Los arrays disponen de un atributo llamado `length`, que significa longitud, indica el número de elementos que contiene, al que se puede acceder como sigue:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
int tamano = arrayDeEnteros.length; // el valor es 5
```

Cadenas de caracteres

En Java existe una clase para representar y manipular cadenas, la clase `String`. Una vez creado un `String` no se puede modificar. Se pueden crear instancias de una manera abreviada y sobre ellas se puede utilizar el operador de concatenación `+`:

```
String frase = "Esta cadena es una frase "
String larga = frase + "que se puede convertir en una frase larga."
```

Constantes

Una constante es una “variable” de solo lectura. Dicho de otro modo, más correcto, es un valor que no puede variar (por lo tanto, no es una variable en sí).

La forma de declarar constantes es la misma que la de crear variables, sólo que hay que anteponer la palabra final que es la que indica que estamos declarando una constante y por tanto no podremos variar su valor inicial durante la ejecución del programa:

```
final double PI = 3.141591;
```

Es una buena práctica comúnmente aceptada el asignar nombres en mayúscula sostenida para las constantes a fin de diferenciarlas luego en el código del programa.

Estructuras de control

Estructuras condicionales

Dos son las estructuras de control condicionales en Java: bifurcación y selección múltiple.

Bifurcación: if-else, if-else-if

Su sintaxis es:

```
if (condicion)
{
    instruccion1();
    instruccion2();
    // etc
} else
{
    instruccion1();
    instruccion2();
    // etc
}
```

Es necesario que la condición sea una variable o expresión booleana. Si sólo existe una instrucción en el bloque, las llaves no son necesarias. No es necesario que exista un bloque else, como en el siguiente ejemplo:

```
if (condicion)
{
    bloqueDeInstrucciones();
}

// continua la ejecución del programa
```

```

    }
else
{
    if(condicion2)
    {
        bloqueDeInstrucciones();
    }
    else
    {
        bloqueDeInstrucciones();
    }
}

```

Cada cláusula `else` corresponde al último `if` inmediato anterior que se haya ejecutado, es por eso que debemos tener especial consideración de encerrar correctamente entre llaves los bloques para determinar exactamente a qué cláusula corresponde. En este caso, es de especial utilidad indentar nuestro código utilizando espacios o tabulaciones como se muestra en el ejemplo anterior."

Un ejemplo del uso para un caso en particular es el siguiente:

```

final int totalMaterias = 4;
int materiasAprobadas = 2;

if (materiasAprobadas == totalMaterias)
{
    otorgarCertificado();
}
else
{
    continuarCapacitacion();
}

```

Selección múltiple: switch

Su sintaxis es la siguiente:

```

switch (expresion)
{
    case valor1:
        instrucciones();
        break;
    case valor2:
        instrucciones();
        break;
    default:
        instrucciones();
}

```

La expresión ha de ser una variable de tipo `int` o una expresión que devuelva un valor entero. Cuando se encuentra coincidencia con un `case` se ejecutan las instrucciones a él asociadas hasta encontrar el primer `break`. Si no se encuentra ninguna coincidencia se ejecutan las instrucciones del bloque `default`, la cual es opcional. El `break` no es exigido en la sintaxis y entonces si no se pone, el código se ejecuta atravesando todos los cases hasta que encuentra uno.

Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también lazo o loop. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (expresión booleana) no se llega a cumplir nunca.

Bucle While

Las sentencias encerradas por llaves se ejecutan mientras **condición** tenga un valor verdadero:

```
while (condicion)
{
    accion();
}
```

Bucle For

La forma general del bucle for es la siguiente:

```
for (inicio; condicion; incremento)
{
    accion();
}
```

que es equivalente a utilizar while en la siguiente forma,

```
inicio();
while (condicion)
{
    accion();
    incremento();
}
```

Bucle Do While

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

```
do
{
    acciones();
}
while (condicion);
```

Sentencias break y continue

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

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

Bloques para manejo de excepciones

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente solo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

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

Existen algunos tipos de excepciones que Java obliga a tener en cuenta. Esto se hace mediante el uso de bloques *try*, *catch* y *finally*.

```
try {  
    // código que puede arrojar una excepcion  
}  
catch (Exception ex) { // atrapamos el error producido  
    // lo procesamos  
}  
finally {  
    // y finalmente lo post procesamos  
}
```

El código dentro del bloque *try* está *vigilado*: si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque *catch* que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques *catch* como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque *finally*, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error. Es importante destacar que el parámetro del bloque *catch* es un objeto de una clase que hereda de *Exception* (veremos esto más adelante), pero por ahora debemos saber que este objeto puede contener información importante acerca de la causa del error.

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

En el peor de los casos que nadie se haga cargo de la excepción ésta llegará al usuario en forma de un mensaje de “Excepción no capturada” lo cual no es una buena práctica.

Como veremos más adelante para algunos tipos de excepciones es obligatorio el uso de los bloques try-catch para manejar los posibles resultados, o en el último de los casos, el método de ejecución main(String[] args) deberá arrojarla explícitamente usando la palabra reservada throws y llegará al usuario final, lo cual no es en absoluto recomendable. En cambio, algunas excepciones propias del lenguaje Java heredan de la clase RuntimeException y se utilizan para tratar errores que pueden ocurrir al momento de ejecutar el programa posiblemente derivados del manejo de datos que no pudieron ser previstos en el momento de la programación. Estas excepciones no necesitan ser capturadas en un bloque try-catch de forma explícita.

Clases y Objetos

Como vimos anteriormente, las *clases* son el centro del paradigma de *Programación Orientada a Objetos(POO)*. Algunos conceptos importantes de la POO son los siguientes:

1. **Encapsulamiento:** Las clases pueden ser declaradas como públicas (*public*) y como paquete (*package*) (accesibles sólo para otras clases del mismo paquete). Las variables miembros y los métodos pueden ser *public*, *private*, *protected* y *package*. De esta forma se puede controlar el acceso entre objetos y evitar un uso inadecuado.
2. **Herencia:** Una clase puede衍生 de otra (*extends*), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede *añadir* nuevas variables y métodos y/o *redefinir* las variables y métodos heredados.
3. **Polimorfismo:** Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface, pueden responder de forma indistinta a un mismo método. Esto, como se ha visto anteriormente, facilita la programación y el mantenimiento del código.

A continuación, veremos cómo se declaran tanto clases como interfaces, y cuál es el proceso para crear sus instancias.

Intuitivamente, una clase es una agrupación de *datos* (variables o campos) y de *funciones* (métodos) que operan sobre esos datos. Todos los métodos y variables deben ser definidos dentro del *bloque* {...} de la clase.

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

```
NombreDeLaClase unObjeto;
NombreDeLaClase otroObjeto;
```

A continuación, se enumeran algunas características importantes de las clases:

1. Todas las variables y métodos de *Java* deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (*extends*), hereda todas sus variables y métodos.
3. *Java* tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios. Es decir que toda clase definida por el programador es heredada de la clase Object definida por el lenguaje de programación.

4. Una clase sólo puede heredar de una única clase (en *Java* no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de *Object*. La clase *Object* es la base de toda la jerarquía de clases de *Java*.
5. En un archivo de código fuente se pueden definir varias clases, pero en un mismo archivo no puede haber más que una clase definida como *public*. Este archivo se debe llamar como la clase *public* que debe tener extensión *.java*. Con algunas excepciones, lo habitual es escribir una sola clase por archivo.
6. Si una clase contenida en un fichero no es *public*, no es necesario que el fichero se llame como la clase.
7. Los métodos y variables de una clase pueden referirse de modo global a un *objeto* de esa clase a la que se aplican por medio de la referencia *this*. Al utilizar la palabra reservada 'this' para referirse tanto a métodos como atributos se restringe el ámbito al objeto que hace la declaración.
8. Las clases se pueden agrupar en *packages* que significa paquetes, introduciendo una línea al comienzo del fichero (*package packageName;*). Esta agrupación en *packages* está relacionada con la jerarquía de carpetas y archivos en la que se guardan las clases.
En la práctica usamos paquetes para agrupar clases con un mismo propósito usando jerarquía de paquetes; esta decisión es muy importante a la hora de diseñar la estructura de nuestro programa.

Variables miembros de objeto

La programación orientada a objetos está *centrada en los datos*. Una clase son *datos y métodos* que operan sobre esos datos.

Cada objeto, es decir cada instancia concreta de una clase, tiene su propia copia de las variables miembro. Las variables miembros de una clase (también llamadas *atributos*) pueden ser de *tipos primitivos* (*boolean*, *int*, *long*, *double*, ...) o referencias a *objetos* de otra clase (*agregación* y *composición*).

Un aspecto muy importante para el correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembros de *tipos primitivos* se inicializan siempre de modo automático, incluso antes de llamar al *constructor* (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la *declaración*, como las variables locales, por medio de constantes o llamadas a métodos. Por ejemplo,

```
public class Alumno
{
    int edad = 18;
}
```

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

Cada *objeto* que se crea de una clase tiene *su propia copia* de las variables miembro. Por ejemplo, cada objeto de la clase *Circulo* tiene sus propias coordenadas del centro *x* e *y*, y su propio valor del radio *r*. Se puede aplicar un método a un objeto concreto poniendo el nombre del objeto y luego el nombre del

método separados por un punto. Por ejemplo, para calcular el área de un objeto de la clase *Circulo* llamado *c1* se escribe: *c1.area()*;

La definición de cada atributo debe empezar con un modificador de acceso. Los modificadores de acceso indican la visibilidad, es decir, si se puede tener acceso sólo desde la clase (*private*), desde la clase y las clases que heredan de ella (*protected*) y desde cualquier clase definida en el mismo paquete o desde cualquier clase (*public*).

Tras el modificador de acceso se escribe el tipo del argumento, este puede ser un tipo primitivo a de tipo referencia. Tras el modificador de acceso, un atributo se puede declarar como *static*. Esto implica que no existe una copia de este atributo en cada instancia de la clase, si no que existe uno único común a todas las instancias. A los atributos *static* también se les llama atributos de clase.

Otro modificador que puede afectar al comportamiento de los atributos de una clase es *final*. Si un atributo se declara como *final*, implica que no se puede cambiar su valor una vez definido. Un ejemplo de uso de este modificador son las constantes de clase:

```
public static final float E = 2.8182f;
```

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

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

Si no se les da valor en la declaración, las variables miembros *static* se inicializan con los valores por defecto (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos) para los tipos primitivos, y con *null* si es una referencia.

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

Métodos

Los métodos especifican el comportamiento de la clase y sus instancias. Los modificadores de acceso y su significado son los mismos que al operar sobre atributos. En particular, al declarar un método estático implica que es un método de la clase, y por lo tanto no es necesario crear ninguna instancia de la clase para poder llamarlo. El conjunto de los métodos públicos de una clase forma su interface.

Un método declarado *final* implica que no se puede redefinir en ninguna clase que herede de esta, veremos el tema de Herencia más adelante, pero es importante destacar que el cuerpo de un método definido como *final* no podrá ser modificado por ninguna clase hijo.

En el momento de la declaración hay que indicar cuál es el tipo del parámetro que devolverá el método o *void* en caso de que no devuelva nada. En otros lenguajes, estos tipos de métodos o funciones se denominan procedimientos.

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

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

También se ha de especificar el tipo y nombre de cada uno de los argumentos del método entre paréntesis. Si un método no tiene argumentos el paréntesis queda vacío, no es necesario escribir void. El tipo y número de estos argumentos identifican al método, ya que varios métodos pueden tener el mismo nombre, con independencia del tipo devuelto, y se distinguirán entre sí por el número y tipo de sus argumentos, como veremos a continuación.

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public String getNombreCompleto ()
    {
        return nombre + " " + this.apellido;
    }
}
```

En el ejemplo anterior hemos definido la clase Alumno con dos atributos de tipo cadena de texto: nombre y apellido. Es importante destacar que estos atributos son definidos como privados para poder respetar el principio de encapsulamiento del paradigma orientado a objetos y lo tomaremos como una buena práctica: los atributos de un objeto deberían ser accesibles sólo a través de métodos públicos. Además hemos definido el método público *getNombreCompleto()* que no recibe parámetros y devuelve una cadena de texto formada por el nombre y el apellido del alumno separados por un espacio. Es importante destacar que para acceder a los atributos de la instancia de esta clase puede usarse o no la palabra reservada *this* ya que en el cuerpo del método no existe otra variable local con el mismo nombre que pueda resultar en ambigüedad.

Sobre el paso de parámetros a un método

En Java los argumentos de los *tipos primitivos* se pasan siempre *por valor*. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. Para modificar un argumento de un tipo primitivo dentro del cuerpo del método puede incluirse como variable miembro o ser retornado para luego realizar la asignación en el momento de la llamada. Las *referencias* se pasan también *por valor*, pero a través de ellas se pueden modificar los objetos referenciados.

Sobre los métodos de seteo (get y set)

En el ejemplo anterior hemos visto que el prefijo del nombre es get, y en la práctica veremos que es de uso común nombrar métodos de obtención y modificación con los prefijos get y set respectivamente. Para poder respetar la buena práctica antes mencionada relacionada con el encapsulamiento de la orientación a objetos cada atributo de la clase debería ser definido como privado y existir métodos get y set para poder obtener y modificar sus valores.

Así, la definición de un método get para obtener el valor del atributo de un objeto tendría la siguiente forma:

```
public TipoDeDatos getAtributo ()
{
    return this.atributo;
}
```

Donde TipoDeDatos puede ser tanto un tipo primitivo como una clase o interfaz si estamos referenciando a otros objetos. Otro punto importante es que por convención el nombre del método luego del prefijo get o set normalmente se escribe en CamelCase como mencionamos en el módulo anterior a la hora de nombrar variables.

De esta forma, los métodos set reciben como parámetro el nuevo valor y no retornan nada, por lo que el tipo especificado es void.

```
public void setAtributo (TipoDeDatos nuevoValor)
{
    this.atributo = nuevoValor;
}
```

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public String getNombre ()
    {
        return this.nombre;
    }

    public void setNombre (String nombre)
    {
        this.nombre = nombre;
    }

    public String getApellido ()
    {
        return this.apellido;
    }

    public void setApellido (String apellido)
    {
        this.apellido = apellido;
    }
}
```

```

public String getNombreCompleto ()
{
    return this.nombre + " " + this.apellido;
}
}

```

Entonces ahora podríamos crear un par de instancias de alumnos y mostrar en pantalla sus nombres completos de la siguiente forma:

```

Alumno alumno1 = new Alumno();
alumno1.setNombre("Pablo");
alumno1.setApellido("Filippo");

Alumno alumno2 = new Alumno();
alumno2.setNombre("Florencia");
alumno2.setApellido("Venne");

System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
System.out.println("Nombre del alumno 2: " + alumno2.getNombreCompleto());

```

En programación orientada a objetos las llamadas a los métodos se les llama paso de mensajes, llamar a un método es análogo a pasarle un mensaje.

El método main()

Existe un nombre de método que está reservado en Java y otros lenguajes: el método main. Este método es especial ya que es el que da lugar al inicio del programa y será llamado por la máquina virtual al momento de la ejecución.

Es importante tener claro que el método main() no es el elemento principal en el desarrollo del programa. El programa, de acuerdo con el paradigma de programación orientada a objetos, se desarrolla mediante la interacción entre objetos por lo que el objetivo de este método normalmente es iniciar el programa delegar el comportamiento a los distintos los objetos correspondientes.

Este método debe pertenecer a una clase pública y su definición es la siguiente:

```

public static void main(String[] args)
{
    // cuerpo de nuestro programa
}

```

Es estático ya que no depende de una instancia en particular de la clase en la que se declara y no tiene ningún valor de retorno. Podemos ver que recibe un array de parámetros de tipo String que representan los argumentos pasados a la hora de ejecutar el programa. En el caso de realizar la ejecución mediante la línea de comando, cada elemento del array será una cada cadena de texto luego de la llamada a nuestro programa, separadas por espacios, ejemplo:

>> java ejemplo1 paso parámetros a mi programa

En este caso, al ejecutar el programa llamada ejemplo1, el método main() recibirá el array args[] compuesto de la siguiente forma:

```
args[0] => "paso"
args[1] => "parámetros"
args[2] => "a"
args[3] => "mi"
args[4] => "programa"
```

Métodos sobrecargados y redefinición (overload y override)

Java permite métodos *sobre cargados* (*overloaded*), es decir métodos distintos con *el mismo nombre* que se diferencian por el número y/o tipo de datos de los argumentos.

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

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, *long* en vez de *int*), el programador debe hacer un *cast* explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobre cargado. En realidad, es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobre cargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

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

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    // se omiten los métodos get y set por simplicidad

    public String getNombreCompleto (String titulo)
    {
        return titulo + " " + this.nombre + " " + this.apellido;
    }
    public String getNombreCompleto ()
    {
```

```

        return this.getNombreCompleto("Sr/a.");
    // usamos un título por defecto para cuando no se especifica por parámetros
}
}

```

Entonces para la ejecución del ejemplo anterior que coloca como prefijo del nombre del alumno el título en caso de tenerlo o "Sr/a." en su defecto tenemos lo siguiente:

```
System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
// muestra en pantalla "Sr/a. Pablo Filippo"
```

```
System.out.println("Nombre del alumno 2: " +
alumno2.getNombreCompleto("Contadora."));
// muestra en pantalla "Contadora. Paula Filippi"
```

Constructores de Objetos

Los métodos que tienen el mismo nombre que la clase tienen un comportamiento especial, sirven para crear las instancias de la clase y se les denomina constructores. Un *constructor* es un operador que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del *constructor* es reservar memoria e inicializar las variables miembros de la clase. A continuación, se listan los principales aspectos de los constructores en Java que es necesario entender:

- Cuando se llama al constructor de una clase para instanciarla y crear el objeto, se invoca al constructor.
- Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase.
- Su argumento implícito es el objeto que se está creando.
- Si no se define explícitamente un constructor, Java lo hará por nosotros ya que siempre es necesario que exista. Se creará un constructor sin argumentos.
- Una clase puede tener *varios constructores*, que se diferencian por el tipo y número de sus argumentos (los constructores justamente son un ejemplo típico de métodos *sobrecargados* que vimos anteriormente).

Si es necesario que un constructor llame a otro constructor lo debe hacer antes que cualquier otra cosa. Se llama *constructor por defecto* al constructor que no tiene argumentos. El programador debe proporcionar en el código, valores iniciales adecuados para todas las variables miembro. En caso de que sólo definamos un constructor con parámetros el constructor por defecto no será creado por Java, por lo que deberemos definirlo explícitamente en caso de ser necesario.

Por ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public Alumno ()
    {

    }

    public Alumno (String nombre, String apellido)
```

```

{
    this.nombre = nombre;
    this.apellido = apellido;
}

// se omiten los métodos get y set por simplicidad
}

```

De esta forma podemos crear los mismos alumnos anteriores de la forma:

```

Alumno alumno1 = new Alumno("Pablo", "Filippo");
Alumno alumno2 = new Alumno("Florencia", "Venne");

```

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

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

Finalización y Destrucción de Objetos

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

Antes de que un objeto sea completamente eliminado de la memoria por el *recolector de basura*, se llama a su método *finalize()*. Este método está definido en la clase *Object* de la que hereda implícitamente cualquier nueva clase.

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

Agrupando Clases en Paquetes

Un *package* es una agrupación de clases que sirve para establecer una jerarquía lógica en la organización de las clases. Además, tiene una relación directa con la organización física de nuestro código ya que también se representa en la estructura de archivos y carpetas que conforman nuestro programa. Al estructurar de este modo las clases, estamos estableciendo un dominio de nombres que la máquina virtual de *Java* utiliza, por ejemplo, cuando nuestra aplicación utiliza clases distribuidas en una red de computadores.

Todas las clases dentro de un mismo paquete tienen acceso al resto de clases declaradas como públicas. Para poder acceder a una clase de otro paquete se ha de importar anteriormente mediante la sentencia *import*.

Para que una clase pase a formar parte de un *package* llamado *nombreDelPaquete*, hay que introducir en ella la sentencia:

```
package nombreDePaquete;
```

Debe ser la primera sentencia del archivo sin contar comentarios y líneas en blanco.

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

Todas las clases que forman parte de un *package* deben estar en la misma carpeta. Los nombres compuestos de los *packages* están relacionados con la jerarquía de carpetas en que se guardan las clases. Es recomendable que los *nombres de las clases* de Java sean únicos, es el nombre del *package* lo que permite obtener esta característica.

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

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

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

```
import poo.cine.Actor;
import poo.cine.*;
```

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

Imaginemos la siguiente estructura de paquetes y clases:

```
futsal
- torneos
  -- Torneo.java
  -- Fecha.java
  -- Encuentro.java
- equipos
  -- Equipo.java
  -- Jugador.java
- predio
  -- Cancha.java
```

Podríamos definir la clase Equipo de la siguiente forma:

```
package futsal.equipos;

public class Equipo
{
    private String nombre;
    private Jugador[] jugadores;
    // Podríamos seguir listando atributos, pero por simplificación dejamos los establecidos anteriormente.
}
```

Es importante destacar la primera línea "package" en el ejemplo anterior, que es la que indica cuál es el paquete en el que está ubicada la clase Equipo. Además, vemos que nuestra clase tiene relación con la clase Jugador, que al estar en el mismo paquete no debe ser importada ya que comparte la visibilidad. Otro dato importante es que la definición de visibilidad de la clase Equipo es public, lo que nos permitirá utilizarla en otros paquetes.

Por otro lado, podríamos escribir la clase Encuentro entre 2 equipos como sigue:

```
package futsal.torneos;

import futsal.equipos.Equipo;
import futsal.predio.Cancha;

public class Encuentro
{
    private Date fechaHora;
    private Cancha cancha;
    private Equipo equipo1;
    private Equipo equipo2;
    private int golesEquipo1 = 0;
    private int golesEquipo2 = 0;
    private boolean jugado = false;
    . . .
}
```

En este caso es necesaria la sentencia import para poder utilizar la clase Equipo ya que se encuentra en otro paquete sobre el que no tenemos visibilidad. Como práctica podrías definir el método obtenerEquipoGanador() que nos retorne el Equipo ganador en caso de que el partido ya se haya jugado?

Herencia de Clases

Como mencionamos en la sección anterior Modelo de Objetos anterior, el concepto de Herencia es de suma importancia en el Paradigma Orientado a Objetos y es una herramienta muy útil a la hora de diseñar nuestros programas. La Herencia define relaciones del tipo "es un" como, por ejemplo "un Alumno es una Persona" por lo que comparte tanto sus atributos como nombre, apellido, dni y su comportamiento, así como tiene sus propios atributos como las materias a las que está inscripto, cuáles ha rendido, entre otras.

Es posible construir una nueva clase a partir de otra mediante el mecanismo de la *herencia*. Mediante el uso de la herencia las clases pueden extender el comportamiento de otra clase permitiendo con ello un gran aprovechamiento del código. Aunque su principal virtud es el Polimorfismo, que veremos luego. Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser *redefinidas (overridden)* en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la *sub-clase* (la clase derivada) “contuviera” un objeto de la *super-clase*; en realidad lo “amplía” con nuevas variables y métodos.

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

La herencia induce una jerarquía en forma de árbol sobre las clases con raíz en la clase *Object*. Una clase se dice que hereda o extiende a otra clase antecesora. La palabra reservada *extends* sirve para indicar que una clase extiende a otra. La clase que extiende a otra, hereda todos los atributos y métodos de la clase antecesora, los atributos y métodos privados no tienen visibilidad. La clase antecesora puede extender a su vez otra clase.

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

Un ejemplo de herencia puede ser el siguiente:

```
Persona.java
public class Persona {
    private String nombre;
    private String apellido;
    private String dni;

    public String getNombreCompleto () {
        return this.nombre + " " + this.apellido;
    }

    // ...
}
```

```
Alumno.java
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    // ...
}
```

De esta forma cualquier objeto de la clase *Alumno* también tendrá los atributos *nombre*, *apellido* y *dni* y podrá hacer uso del método público *getNombreCompleto()*.

Sobrescritura de variables y métodos

Una clase puede *redefinir* (volver a definir) cualquiera de los métodos heredados de su *superclase* que no sean *final*. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido. Además, una clase que extiende a otra puede declarar atributos con el mismo nombre que algún atributo de la clase a la que extiende; se dice que el atributo se ha sobreescrito u ocultado. Un uso de sobreescritura de atributos es la extensión del tipo.

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

Finalmente, un método declarado *static* en una clase antecesora puede sobreescibirse en una clase que la extienda, pero también se debe declarar *static*; de lo contrario se producirá un error en tiempo de compilación.

En el ejemplo anterior podríamos redefinir el método *getNombreCompleto()* para que incluya el número de legajo del Alumno a la hora de mostrar su nombre en pantalla.

```
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    public String getNombreCompleto () {
        return this.legajo + ":" + super.getNombreCompleto();
    }

    // ...
}
```

Otro ejemplo puede ser el de las Cuentas Bancarias, donde nuestra clase *CuentaBancaria* es heredada por *CuentaCorriente* y *CajaDeAhorro*. De esta forma la clase heredad contiene el atributo *saldo* que es común a ambos tipos de cuenta y define ciertos métodos también comunes como *depositar()* y *extraer()* que reciben un monto como parámetros. Al ser conocida por nosotros la interfaz de la clase *CuentaBancaria* podríamos acceder a estos métodos comunes sin conocer explícitamente la clase que la hereda, esto se conoce como Polimorfismo.

```
CuentaBancaria miCuenta = new CajaDeAhorros();
// realizamos una extracción usando el objeto miCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
miCuenta.extraer(100);
CuentaBancaria miOtraCuenta = new CuentaCorriente();
// realizamos una extracción usando el objeto miOtraCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
// aunque es una instancia de CuentaCorriente
miOtraCuenta.extraer(100);
```

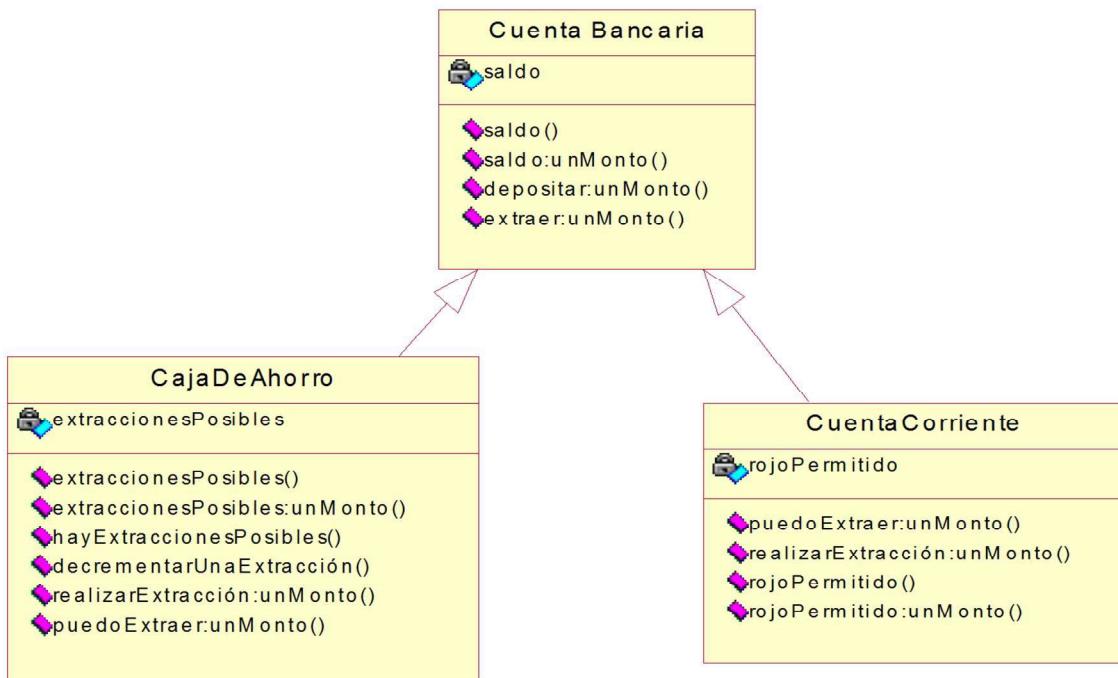


Fig. 46- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria

Clases y métodos abstractos

Las clases abstract funcionan como plantillas para la creación de otras clases, son clases de las que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra *abstract*, como, por ejemplo

```
public abstract class Geometria { ... }
```

Un método se puede declarar como *abstract*. Un método declarado de esta manera no implementa nada. Si una clase contiene uno o más métodos declarados como *abstract*, ella a su vez debe ser declarada como *abstract*. Las clases que la extiendan deben *obligatoriamente* sobrescribir los métodos declarados como *abstract* o la clase también debe declararse *abstract*.

Interfaces

Así como la Herencia representa relaciones del tipo “es un”, las interfaces nos permiten representar relaciones “se comporta como un”. El concepto de la implementación de interfaces garantiza que objetos de una Clase con esa interfaz tendrán disponibles ciertos métodos definidos. Es en este punto que entra el concepto de Polimorfismo, ya que cualquier objeto independientemente de su clase que implemente una interfaz determinada podrá responder ante la llamada a esos métodos definidos. Las *interfaces* permiten “publicar” el comportamiento de una clase develando un mínimo de información. El nombre de una *interface* se puede utilizar como un *nuevo tipo de referencia*. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la *interface*. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Si queremos que una determinada clase vaya a tener cierto comportamiento, hacemos que implemente una determinada interface. En la interface no se implementa el comportamiento, únicamente se especifica cuál va a ser, es decir, se definirán los métodos, pero no se implementan. No se pueden crear instancias de una interface. Todos los métodos declarados en una interface son públicos, así como sus atributos y la misma interface.

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

Veamos un ejemplo:

```
public interface DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio);
}
```

La interfaz anterior define un método para poder diagramar un Torneo para el cual tenemos diferentes formas de hacerlo, puede ser por eliminatorias, todos contra todos, entre otras formas. Este método debe recibir una lista con los Equipos que se enfrentarán y una fecha que representa el inicio del Torneo. Por otro lado, retorna una lista con los partidos a jugar para la forma de diagramación correspondiente. Así podríamos tener dos clases que implementen esta interfaz:

```
public class TodosContraTodos implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con la combinación de todos los
        equipos
    }
}

public class PorEliminatorias implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con llaves de eliminatorias
    }
}
```

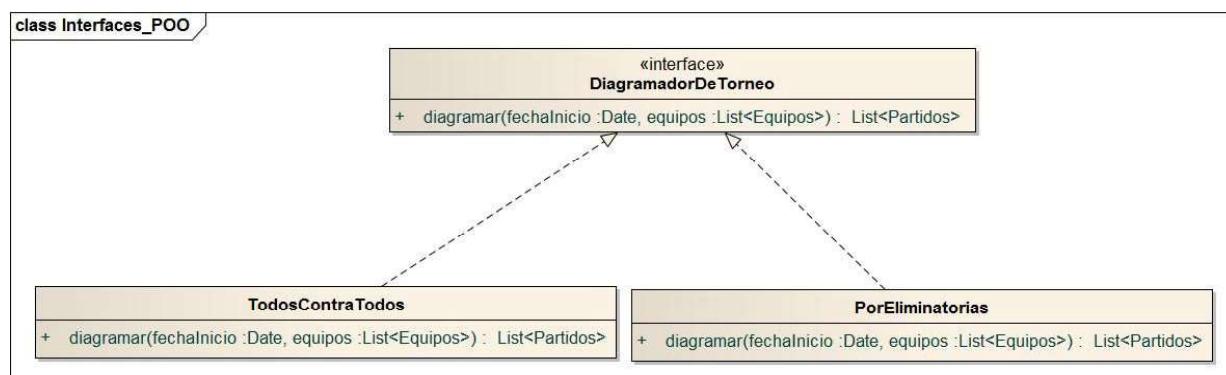


Fig. 47- Ejemplo de interfaces – Diagramación de Torneos

Documentación de Clases y Métodos

Documentar un proyecto es algo fundamental de cara a su futuro mantenimiento. Cuando programamos una clase, debemos generar documentación lo suficientemente detallada sobre ella como para que otros programadores sean capaces de usarla sólo con su interfaz. No debe existir necesidad de leer o estudiar su implementación, lo mismo que nosotros para usar una clase la biblioteca de Java no leemos ni estudiamos su código fuente.

Javadoc es una utilidad de Oracle para la generación de documentación en formato de página web a partir de código fuente Java. Javadoc es el estándar para documentar clases de Java. La mayoría de los IDEs utilizan javadoc para generar de forma automática documentación de clases.

La documentación a ser utilizada por javadoc se escribe en comentarios que comienzan con `/**` (notar el doble `*`) y que terminan con `*/`. A la vez, dentro de estos comentarios se puede escribir código HTML y operadores para que interprete javadoc (generalmente precedidos por `@`).

Tag	Descripción	Uso
<code>@author</code>	Nombre del desarrollador creador de la clase o interfaz	nombre
<code>@deprecated</code>	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores	descripción
<code>@param</code>	Definición de un parámetro de un método, es requerido para todos los parámetros del método	nombre_parametro descripción
<code>@return</code>	Informa lo que devuelve el método, no se puede usar en constructores o métodos "void"	descripción
<code>@see</code>	Asocia con otro método o clase, brinda más información o elementos relacionados	referencia (<code>#método();</code> <code>clase#método();</code> paquete.clase; <code>paquete.clase#método();</code>).
<code>@throws</code>	Excepción lanzada por el método (lo veremos más adelante)	nombre_clase descripción
<code>@version</code>	Versión del método o clase	versión

Tabla 13 – Tags utilizados en Javadoc

Las etiquetas `@author` y `@version` se usan para documentar clases e interfaces. Por tanto no son válidas en cabecera de constructores ni métodos. La etiqueta `@param` se usa para documentar constructores y métodos. La etiqueta `@return` se usa solo en métodos de tipo función.

Dentro de los comentarios se admiten etiquetas HTML, por ejemplo con @see se puede referenciar una página web como link para recomendar su visita de cara a ampliar información.

Un ejemplo de su uso en un archivo de código puede ser:

```
package futsal.equipos;

import java.util.Date;

/**
 * Clase que representa a cualquier Jugador registrado en
 * el Torneo. Es importante considerar que sólo puede pertenecer
 * a un Equipo inscripto.
 *
 * @author joaquinleonelrobles
 * @see futsal.equipos.Equipo
 */
public class Jugador {

    private String nombre;
    private String apellido;
    private Date fechaNacimiento;

    /**
     * Calculamos la edad del jugador en base a su fecha de nacimiento
     *
     * @return Edad
     */
    public int calcularEdad () {
        // TODO recordar implementar este método
        return 0;
    }

    /**
     * Comprobamos si el Jugador puede participar en un torneo en
     * base a la edad mínima pasada por parámetros.
     *
     * @param edadMinima Edad mínima (inclusive) en años
     * @return Verdadero si puede participar
     */
    public boolean puedeParticiparPorSuEdad (int edadMinima) {
        return edadMinima >= this.calcularEdad();
    }
}
```

En el ejemplo anterior puede notarse un comentario que inicia con las letras “TODO”, se trata de una convención utilizada por los desarrolladores para denotar código que aún debe implementarse, del inglés “To Do” o “Pendiente”. Algunos IDE agregan automáticamente estos comentarios y además permiten resaltarlos para recordarnos de las porciones de código que nos faltan escribir.

```

24  public int calcularEdad () {
25      // TODO recordar implementar este metodo
26      return 0;
27 }

```

Fig. 48- Resaltado de comentarios TODO en la IDE Eclipse

La documentación generada por JavaDoc tendrá una apariencia similar a la siguiente captura de pantalla:

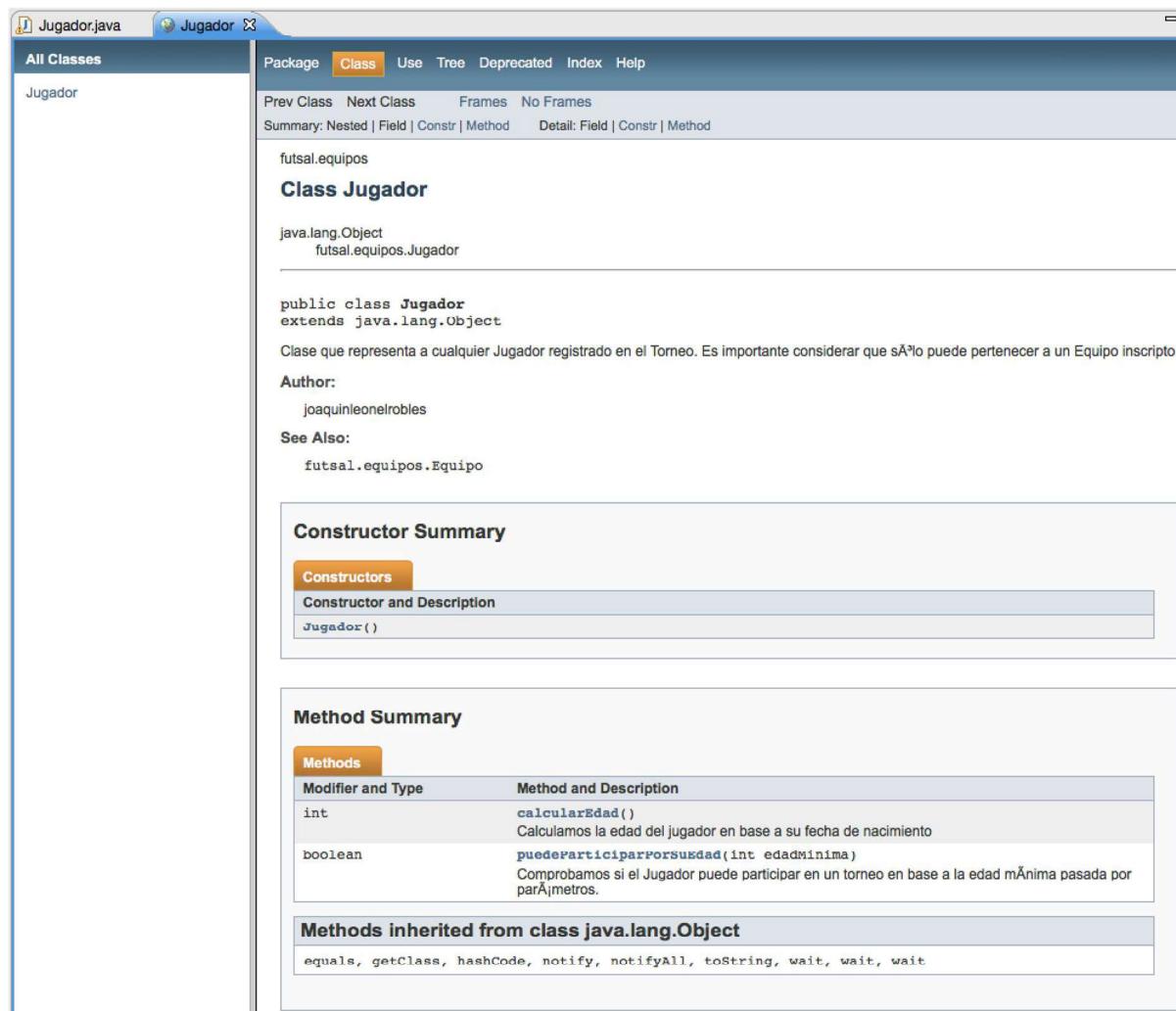


Fig. 49-. Documentación JavaDoc generada automáticamente en base a comentarios

Colecciones de tamaño variable

Anteriormente hemos mencionado los arrays de datos, los cuales son inicializados con un tamaño fijo que no podremos modificar a lo largo del desarrollo del programa. En algunos casos es útil poder tener colecciones de tamaño variable, a las cuales podamos aplicar operaciones de agregado o remoción de elementos sin tener que preocuparnos si el espacio en memoria está o no reservado. Java incorpora en su biblioteca de clases algunas alternativas muy útiles para trabajar con este tipo de colecciones, a continuación, veremos algunas de ellas.

Listas

Las listas son estructuras de datos que permiten tener cierta flexibilidad en su manejo, pueden crecer o acortarse según se lo requiera. Existen varias formas de implementar una lista en Java: en este caso se presenta un ejemplo en código utilizando punteros mediante la referencia a objetos. Una lista es una secuencia de elementos dispuesto en un cierto orden, en la que cada elemento tiene como mucho un predecesor y un sucesor. El número de elementos de la lista no suele estar fijado, ni suele estar limitado por anticipado.

Representaremos la estructura de datos de forma gráfica con cajas y flechas. Las cajas son los elementos y las flechas simbolizan el orden de los elementos.



Fig. 50- Representación de nodos en una lista ordenada

La estructura de datos deberá permitirnos determinar cuál es el primer elemento y el último de la estructura, cuál es su predecesor y su sucesor (si existen de cualquier elemento dado). Cada uno de los elementos de información suele denominarse **nodo**.

La interfaz java.util.List

Esta interfaz normalmente acepta elementos repetidos o duplicados y al igual que los arrays es lo que se llama “basada en 0”, esto quiere decir que el primer elemento no es el que está en la posición “1”, sino en la posición “0”.

Esta interfaz proporciona iterador especial que nos permite recorrer los distintos elementos de la lista. Este iterador permite además de los métodos definidos por cualquier iterador (estos métodos son hasNext, next y remove) métodos para inserción de elementos y reemplazo, acceso bidireccional para recorrer la lista y un método proporcionado para obtener un iterador empezando en una posición específica de la lista.

Debido a la gran variedad y tipo de listas que puede haber con distintas características como permitir que contengan o no elementos nulos, o que tengan restricciones en los tipos de sus elementos, hay una gran cantidad de clases que implementan esta interfaz.

Es posible restringir de qué clase serán instancia los objetos que pertenezcan a la lista utilizando los símbolos < y > en la definición de la variable, como, por ejemplo:

```
List<Jugador> jugadores;
```

De esta forma la lista jugadores sólo podrá estar compuesta por elementos que sean instancias de la clase Jugador.

A continuación, veremos las siguientes clases que implementan esta interfaz:

- ArrayList
- Stack

La Clase java.util.ArrayList

ArrayList, como su nombre lo indica, basa la implementación de la lista en un array. Eso sí, un array dinámico en tamaño (es decir, de tamaño variable), pudiendo agrandarse o disminuirse el número de elementos. Implementa todos los métodos de la interfaz List y permite incluir elementos null.

Un beneficio de usar esta implementación de List es que las operaciones de acceso a elementos, capacidad y saber si es vacía o no se realizan de forma eficiente y rápida. Todo arraylist tiene una propiedad de capacidad, aunque cuando se añade un elemento esta capacidad puede incrementarse. Java amplía automáticamente la capacidad de un arraylist cuando sea necesario. A través del código podemos incrementar la capacidad del arraylist antes de que este llegue a llenarse, usando el método ensureCapacity.

A continuación, veremos un ejemplo en el cual calculamos la edad promedio de un grupo de Jugadores.

```
package futsal.equipos;

public class Jugador {

    private String nombre;
    private String apellido;
    private int edad;

    public Jugador (nombre, apellido, edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
}
```

En el cuerpo de nuestro programa creamos algunos jugadores con sus edades, los incluimos en un ArrayList y luego lo recorremos para obtener el promedio de edades.

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houlebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la variable acumuladora de promedios
float promedio = 0;

// inicializamos la lista de jugadores
// se especifica el tipo de datos de los elementos de la lista
// entre cocodrilos (< y >) para evitar insertar otro tipo de objetos
List<Jugador> jugadores = new ArrayList<Jugador>();

// agregamos un elemento a la lista con el método add()
jugadores.add (j1);
jugadores.add (j2);
```

```

jugadores.add (j3);
jugadores.add (j4);

// obtenemos un objeto Iterador que nos permita recorrer la lista
Iterator<Jugador> iter = jugadores.iterator();

// mientras exista un elemento siguiente por recorrer
while (iter.hasNext()) {
    // obtenemos el Jugador siendo recorrido
    Jugador j = iter.next();

    // acumulamos su edad promedio sobre el total de jugadores
    // que obtenemos con el método size() de la lista
    promedio += j.getEdad() / jugadores.size();
}

// mostramos el resultado en pantalla
System.out.println("Promedio: " + promedio);

```

A continuación, se listan algunos de los métodos más relevantes de la clase ArrayList:

Retorno	Método y descripción
boolean	add(E e) Agrega el elemento al final de la lista
void	add(int index, E element) Agrega el elemento en la posición especificada
boolean	addAll(Collection<? extends E> c) Agrega todos los elementos de otra colección al final de esta lista
boolean	addAll(int index, Collection<? extends E> c) Agrega todos los elementos de otra colección en la posición especificada
void	clear() Elimina todos los elementos de la lista
boolean	contains(Object o) Devuelve true si el objeto o existe en la lista
void	ensureCapacity(int minCapacity) Incrementa la capacidad del array subyacente para asegurar la capacidad especificada
E	get(int index) Devuelve el elemento de la posición solicitada
int	indexOf(Object o) Devuelve la posición del objeto solicitado
boolean	isEmpty() Devuelve true si la lista no contiene elementos
Iterator<E>	iterator() Devuelve un objeto iterador para recorrer la lista
E	remove(int index)

	Elimina el elemento de la posición especificada
boolean	<u>remove(Object o)</u>
	Elimina la primera ocurrencia del objeto en la lista
E	<u>set(int index, E element)</u>
	Reemplaza el elemento en la posición especificada
int	<u>size()</u>
	Devuelve el tamaño de la lista

Tabla 14. Extracto del JavaDoc para la clase ArrayList

La clase java.util.Stack

Esta clase es una implementación de la estructura de datos que vimos en el módulo de Técnicas de Programación denominada Pila, una estructura de tipo LIFO (Last In First Out, último en entrar primero en salir). Provee las operaciones de push (colocar) y pop (extraer) así como otros métodos accesorios como el size(), peek (consulta el primer elemento de la cima de la pila), empty (que comprueba si la pila está vacía) y search (que busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella).

Veamos un ejemplo apilando Jugadores:

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houellebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la pila de jugadores
Stack<Jugador> pila = new Stack<Jugador>();

// agregamos un elemento a la lista con el método add()
pila.push (j1);
pila.push (j2);
pila.push (j3);
pila.push (j4);

// mientras existan elementos por extraer
while (!pila.empty()) {
    // mostramos el nombre del jugador en pantalla
    System.out.println (pila.pop().getNombreCompleto());
}
```

La salida de nuestro programa será la siguiente, respetando el orden de extracción Último en Entrar Primero en Salir:

Alejandro Baricco
 Aldo Huxley
 Pablo Auster
 Miguel Houellebecq

A continuación, presentamos un extracto del JavaDoc de la clase Stack con los métodos más relevantes:

Retorno	Método y Descripción
boolean	empty() Devuelve true si la pila está vacía
E	peek() Consultamos el primer elemento de la pila sin extraerlo
E	pop() Extraemos el primer elemento de la pila
E	push(E item) Colocamos un elemento en el tope de la pila
int	search(Object o) Devuelve la posición del elemento en la pila (basado en 1)

Tabla 15. Extracto del JavaDoc para la clase Stack

Excepciones

Como mencionamos anteriormente, en Java existe un tipo especial de errores denominado Excepción. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

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

Para poder trabajar con excepciones y capturarlas para poder desarrollar un programa que pueda ser tolerante a errores se utilizan los bloques try-catch-finally que enunciamos anteriormente cuando hablamos de Estructuras de Control. Ahora veremos con mayor profundidad cómo se utilizan.

Los errores se representan mediante dos tipos de clases derivadas de la clase *Throwable*: *Error* y *Exception*, sobre estas últimas es que trabajaremos en nuestros programas.

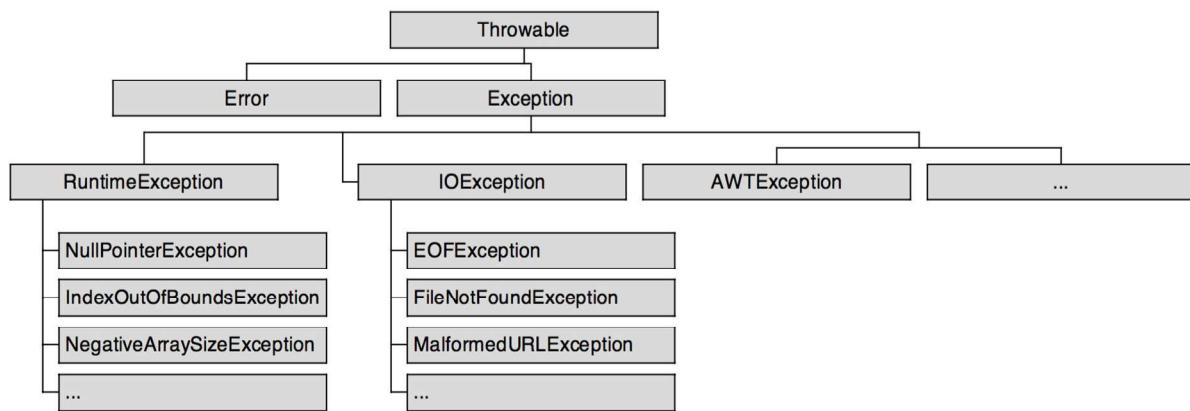


Fig. 51- Jerarquía de Excepciones en el lenguaje Java

Las clases derivadas de *Exception* pueden pertenecer a distintos packages de Java. Algunas pertenecen a *java.lang* (*Throwable*, *Exception*, *RuntimeException*, ...); otras a *java.io* (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de *Throwable* todos los tipos de excepciones pueden usar los siguientes métodos:

1. `String getMessage()`
Extrae el mensaje asociado con la excepción
2. `String toString()`
Devuelve un String que describe la excepción
3. `void printStackTrace()`
Indica el método donde se lanzó la excepción

Además disponemos de los siguientes constructores propios de la clase *java.lang.Exception* que podemos invocar en nuestra clase:

1. `Exception()`
2. `Exception(String message)`

El primero determina un mensaje nulo para la causa de la excepción mientras que el segundo nos permite especificar el mensaje en la invocación al constructor de la clase padre.

En base a lo anterior, podemos declarar nuestras propias excepciones a la hora de desarrollar nuestros programas para manejar los errores propios de la funcionalidad que implementemos; por ejemplo, para el caso de un Encuentro entre dos Equipos, a la hora de intentar obtener el equipo ganador podríamos lanzar una excepción en el caso de que el partido aún no se haya jugado. Para crear nuestra propia excepción sólo tenemos que hacer lo siguiente:

```

public class PartidoNoJugadoException extends Exception
{
    // acá podríamos declarar y redefinir los métodos enunciados más arriba
    // por ahora sólo nos ocuparemos del mensaje de error

    public String getMessage ()
    {
        return "El partido aún no se ha jugado";
    }
}
  
```

```

    }
}
```

De la misma forma podemos especificar el mensaje de la causa declarando el constructor de nuestra clase invocando al constructor de la clase Exception:

```

public class PartidoNoJugadoException extends Exception
{
    public PartidoNoJugadoException()
    {
        super("El partido aún no se ha jugado");
    }
}
```

Es importante notar que es una buena práctica agregar como sufijo al nombre del método la palabra Exception para saber de antemano la naturaleza de esta clase.

En la implementación del método obtenerEquipoGanador() deberíamos agregar la cláusula throws para indicar que el método puede arrojar una excepción de la que explícitamente deberemos hacernos cargo:

```

public Equipo obtenerEquipoGanador() throws PartidoNoJugadoException
{
    // comprobamos si el partido ya se ha jugado
    if (jugado)
    {
        // comparamos la cantidad de goles
        if (golesEquipo1 > golesEquipo2)
            return equipo1; //Cuando el bloque if tiene una única sentencia se
                           //puede evitar el uso de {}
        else
        {
            if (golesEquipo1 < golesEquipo2)
                return equipo2;
            else
                return null;// hubo un empate por lo que devolvemos nulo
        }
    }
    // el partido aún no se ha jugado por lo que lanzamos una excepcion
    else
    {
        throw new PartidoNoJugadoException();
    }
}
```

Por otro lado, también podríamos redefinir el constructor de la clase de excepción y agregar parámetros propios del dominio de nuestro problema. En este caso podríamos, por ejemplo, agregar cuál fue el partido del cual intentamos obtener el resultado.

Durante la ejecución de nuestro programa podríamos tratar de obtener el equipo ganador de la siguiente forma:

```

Encuentro encuentro = new Encuentro();
encuentro.setEquipo1(losQuirquinchosVerdes);
encuentro.setEquipo2(losPadecientesFC);

// como la llamada a nuestro método puede arrojar una excepción debemos
// encerrarla en un bloque try-catch
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque: " +
ex.getMessage());
}

```

En el ejemplo anterior podemos notar que al capturar la excepción con el bloque catch() además obtenemos una instancia, un objeto de la clase PartidoNoJugadoException, lo que nos permite llamar al método getMessage() para conocer la causa del problema.

La salida del ejemplo anterior sería:

No se puede obtener el equipo ganador porque: El Partido aún no se ha jugado

Ahora que tenemos nuestras excepciones definidas y creadas es necesario arrojarlas en el momento en que se producen durante la ejecución de nuestro programa, para ello disponemos de la palabra reservada throw cuya sintaxis es la siguiente:

```

// alguna condición que genera nuestra excepción
throw objetoDeExcepcion;

```

A los fines de abreviar el código es frecuente ver la creación de los objetos instancia de la clase de excepción en la misma línea en que son arrojados; como, por ejemplo:

```

// alguna condición que genera nuestra excepción
throw new PartidoNoJugadoException();

```

Es importante destacar la diferencia con la palabra reservada throws que informa que un método en particular puede arrojar una excepción de un tipo determinado y deberá ser manejada por el método que lo invoque.

Por otro lado, al reformular el ejemplo agregando el siguiente bloque antes del try-catch la ejecución tomaría el camino feliz mostrándonos al flamante ganador del encuentro: Los Padecientes FC.

```

encuentro.setJugado (true);
encuentro.setGolesEquipo1 (1);
encuentro.setGolesEquipo2 (9); // <-- goleada!

```

Jerarquía de Excepciones

Como vimos anteriormente todas las excepciones heredan de la clase `java.lang.Exception`, pero además podemos heredar de otras excepciones que finalmente hereden ésta clase. De esta forma podemos diseñar una jerarquía de excepciones para ir del caso más general al más particular en la definición de nuestros errores, veamos un ejemplo expandiendo la jerarquía de la excepción `PartidoNoJugadoException`:

```
public class PartidoNoJugadoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class PartidoCanceladoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class ResultadoDePartidoException extends Exception
{}
```

Entonces ahora tenemos dos excepciones para tratar los posibles errores a la hora de intentar obtener el resultado de un partido en base su estado. Además, a la hora de capturarlas podemos hacer uso de múltiples bloques `catch`, de la siguiente forma:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}
catch (PartidoCanceladoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido fue cancelado");
}
```

O utilizando la propiedad del polimorfismo propia del lenguaje de programación orientado a objetos podemos capturar todas las excepciones que heredan de `ResultadoDePartidoException` con sólo un bloque `catch`, usando:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
```

```
catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Y usando una combinación de ambas estrategias podemos tomar acciones para algunas excepciones en particular y, en los demás casos, realizar una operación para todos los demás casos como, por ejemplo:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}
catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Es importante destacar que cada bloque catch() es evaluado en orden y sólo se ejecutará el primero que corresponda a la excepción arrojada, por lo cual deberemos declararlos en orden de las excepciones más particulares a las más generales. Si en el ejemplo anterior hubiésemos capturado primero ResultadoDePartidoException antes de PartidoNoJugadoException, el último bloque catch nunca sería evaluado ya que todos los casos serían capturados con el primer bloque más general.

La API de Java

Como el lenguaje Java es un lenguaje orientado a objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. Se trata de un conjunto de clases de propósito general ya desarrolladas y probadas, que nos permiten ahorrar tiempo a la hora de programar nuestras aplicaciones. Algunas de ellas ya las hemos mencionado y hasta utilizado en ejemplos anterior, como ArrayList, Stack y Exception. La API es parte integral del lenguaje de programación y está implementada para cada máquina virtual independientemente de la plataforma donde ejecutemos nuestro código, por lo cual podemos confiar en su portabilidad.

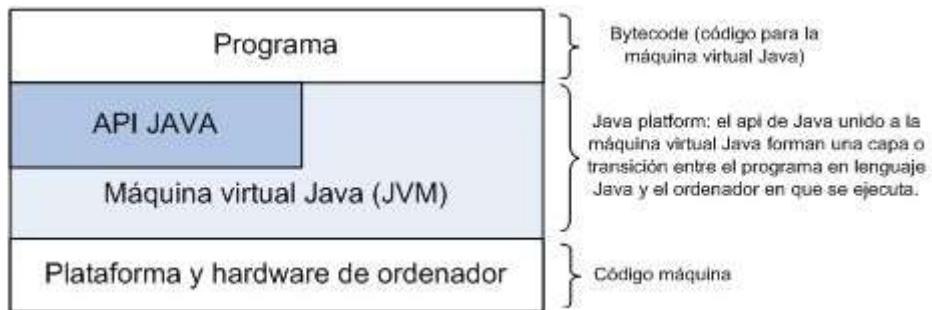


Fig. 52- La API de Java en contexto

La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente. A continuación, se muestra un esquema que describe en términos generales la estructura de la API.

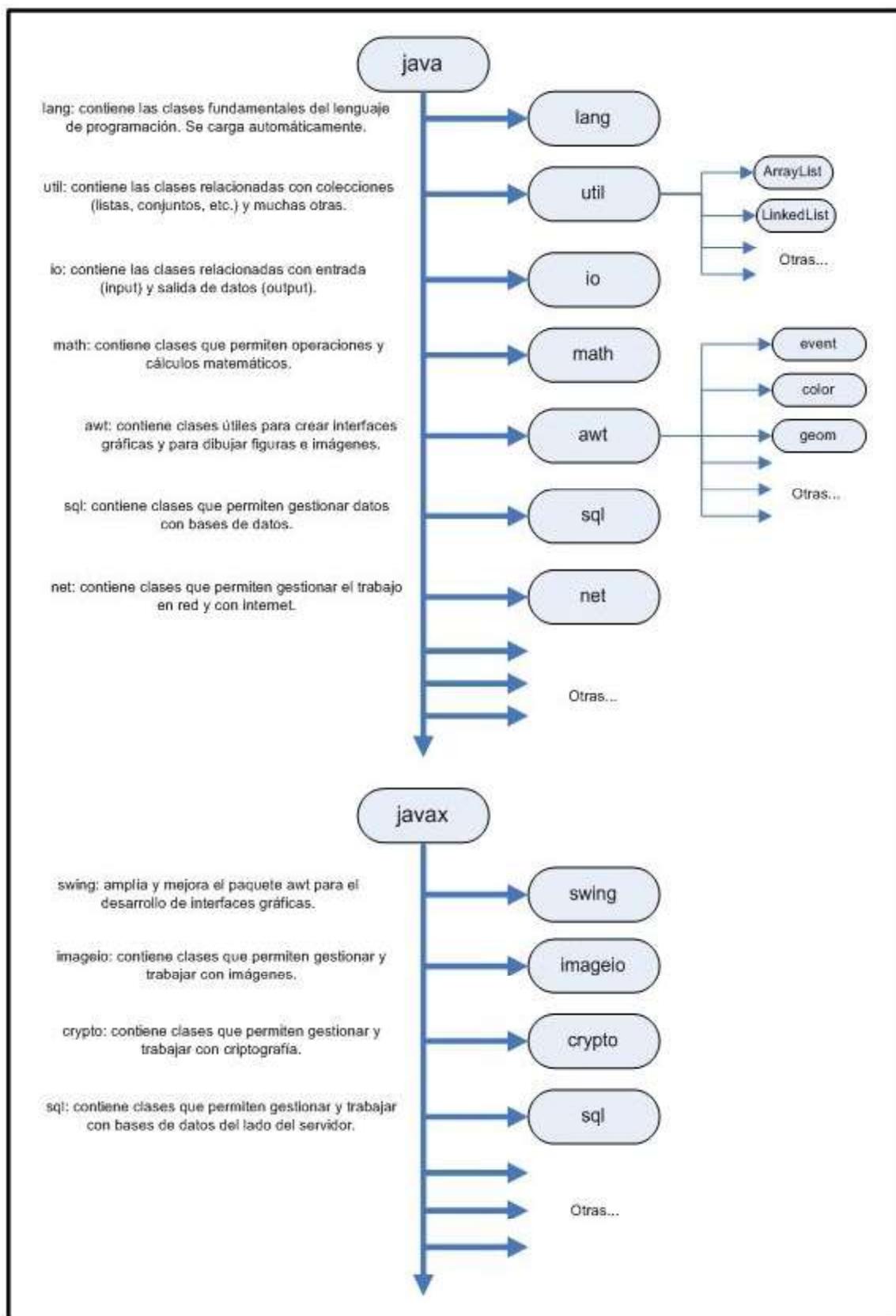


Fig. 53- Estructura de paquetes general para la API de Java 8.

Las librerías podemos decir que se organizan en ramas como si fueran las ramas de un árbol. Vamos a fijarnos en dos grandes ramas: la rama “java” y la rama “javax”. La rama java parte de los orígenes de Java y contiene clases más bien generales, mientras que la rama javax es más moderna y tiene paquetes con fines mucho más específicos.

Encontrar un listado de librerías o clases más usadas es una tarea casi imposible. Cada programador, dependiendo de su actividad, utiliza ciertas librerías que posiblemente no usen otros programadores. Los programadores centrados en programación de escritorio usarán clases diferentes a las que usan programadores web o de gestión de bases de datos.

La especificación de la API de forma completa (en JavaDoc por supuesto) está disponible en <http://www.oracle.com/technetwork/java/api-141528.html> y es propia de cada versión del lenguaje.

Programando interfaces gráficas

Hasta ahora hemos desarrollado aplicaciones que corren en la línea de comandos del sistema operativo y cuya comunicación con el usuario se realiza mediante mensajes en formato de texto que se ingresan o muestran en la consola. Pero la gran mayoría de las aplicaciones modernas requieren interfaces visuales y gráficas para interactuar con los usuarios y cada vez están tomando mayor peso con la predominancia de los dispositivos táctiles. El AWT (Abstract Windows Toolkit) es la parte de Java que se ocupa de construir interfaces gráficas de usuario. Aunque el AWT ha estado presente en Java desde la versión 1.0, la versión 1.2 ha incorporado un modelo distinto de componentes llamado Swing con componentes modernos e independencia del sistema operativo. Esto significa que nuestras aplicaciones con interfaz gráfica podrán utilizarse tanto en sistemas Microsoft Windows, Linux y otros dibujándose con las herramientas propias de cada plataforma.

Java Swing pertenece a las JFC (Java Foundation Classes), está contenido en el paquete javax.swing y fue creada a partir de java.awt. A continuación podemos ver su estructura jerárquica de componentes.

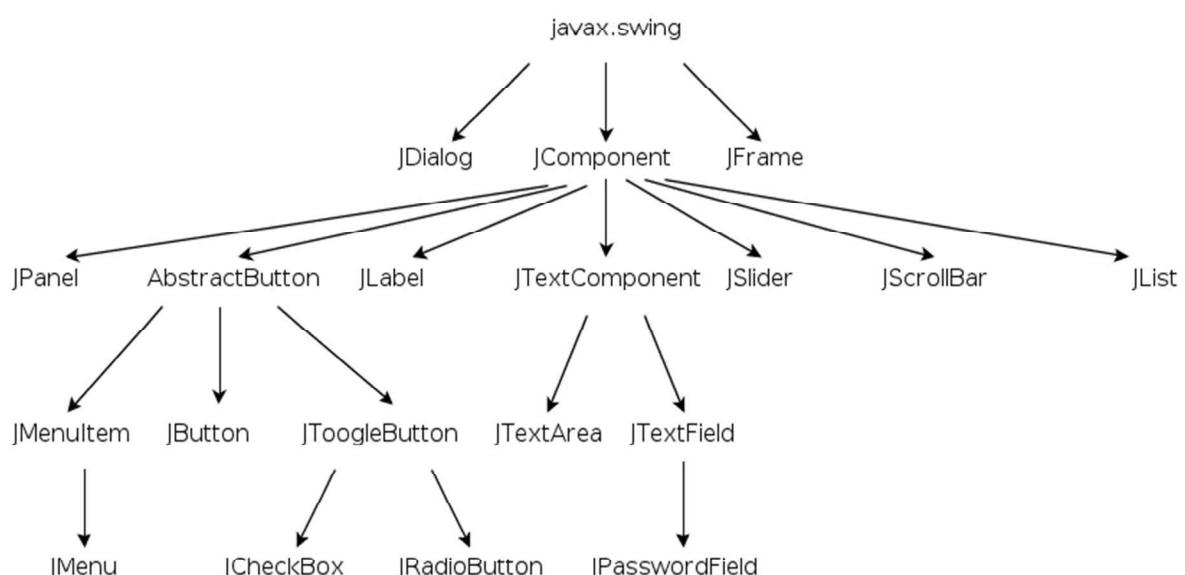


Fig. 54- Jerarquía de componentes de la biblioteca Java Swing para interfaces gráficas

En la imagen anterior es importante destacar que todos los componentes como los campos de ingreso de texto, botones, menues y demás heredan de la clase JComponent. Además, nuestras ventanas serán clases que hereden de JFrame.

El modelo de Eventos para interactuar con la aplicación

Para poder comunicarnos con nuestra aplicación mediante sus componentes dibujados en la ventana, Java utiliza un modelo de eventos donde un evento es alguna acción realizada por el usuario, como un click con el mouse, la presión de una tecla, arrastrar un componente, entre otros, que puede desencadenar una funcionalidad.

El modelo de eventos de Java está basado en que los objetos sobre los que se producen los eventos (event sources) “registran” los objetos que habrán de gestionarlos (event listeners), para lo cual los event listeners habrán de disponer de los métodos adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los event listeners disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface Listener. Las interfaces Listener se corresponden con los tipos de eventos que se pueden producir. De esta forma al producirse alguna acción del usuario que tenga un “oyente” a la espera, se llamará a la funcionalidad definida en la implementación de la interfaz Listener correspondiente. En caso de que no haya ningún oyente registrado para ese evento será descartado sin más.

Veamos un ejemplo a continuación:

HolaMundo.java

```
package ui;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HolaMundo extends JFrame {

    private JLabel texto1;
    private JButton boton;

    public HolaMundo() {
        setLayout (null);

        texto1 = new JLabel("Hola Mundo!");
        texto1.setBounds(100,100,200,40);

        boton = new JButton("Soy un boton");
        boton.setBounds(70,140,200,40);
        boton.addActionListener(new MiClickListener(this));

        add(texto1);
        add(boton);
    }

    public static void main(String[] args) {
        HolaMundo ventana = new HolaMundo();
    }
}
```

```

        ventana.setBounds(500,250,300,250);
        ventana.setVisible(true);
        ventana.setResizable(false);
    }
}

MiClickListener.java
package ui;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JDialog;
import javax.swing.JFrame;

public class MiClickListener implements ActionListener {

    private JFrame padreFrame;

    public MiClickListener(JFrame padreFrame) {
        this.padreFrame = padreFrame;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        JDialog d = new JDialog (padreFrame, "Hola", true);
        d.setLocationRelativeTo(padreFrame);
        d.setVisible(true);
    }
}

```

En este ejemplo mostramos una ventana principal con un texto estático y un botón de acción que al ser presionado abre un cuadro de diálogo vacío.

La primera clase extiende a `JFrame` la cual viene a ser la ventana principal de nuestra aplicación que es inicializada en el método `main()` al igual que hacímos con las aplicaciones de línea de comandos. En el constructor declaramos dos componentes, uno de tipo `JLabel` que es un texto estático y un `JButton`, nuestro botón con llamado a la acción. Además de asignarles las etiquetas de texto al inicializarlos definimos su posición en la pantalla y para el caso del botón, registramos un `ActionListener`, es decir una clase específica que estará a la escucha de acciones realizadas sobre ese botón.

Por otro lado, hemos creado la clase `MiClickListener` que, como su nombre lo indica, será la encargada de estar a la escucha de clicks realizados sobre nuestro botón. Es importante destacar que esta clase debe implementar la interfaz `ActionListener`, para poder reescribir el método `actionPerformed()` que será invocado al realizar alguna acción sobre el botón. En el cuerpo de este método incializamos y mostramos dentro de la ventana principal un cuadro de diálogo vacío.

Java Swing es una biblioteca con muchos componentes y cada uno puede ser configurado y utilizado para fines específicos dentro de nuestras aplicaciones gráficas. Su documentación está disponible en formato JavaDoc en <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>.

Referencia de Figuras y Tablas

Figuras

Fig. 1 – Tarjetas perforadas para programación de computadoras	8
Fig. 2 – Instrucciones en Assembly en la vista de Terminator (1984).....	9
Fig. 3 - Clases y Objetos.....	22
Fig. 4. Estado, comportamiento e identidad de un objeto	24
Fig. 5 - Herencia entre Clases	29
Fig. 6 Representación de la herencia de diamante.....	30
Fig. 7 - Abstracción en el Modelo de Objetos	32
Fig. 8- Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo.....	33
Fig. 9 - El encapsulamiento en el Modelo de Objetos	34
Fig. 10 - Modularidad, cohesión y acoplamiento en el Modelo de Objetos	35
Fig. 11 - Las Abstracciones forman una Jerarquía	37
Fig. 12- La comprobación estricta de tipos impide que se mezclen abstracciones	38
Fig. 13 - La concurrencia permite a dos objetos actuar al mismo tiempo	39
Fig. 14 - Conserva el estado de un objeto en el tiempo y el espacio.....	41
Fig.15- La clasificación es el medio por el cual ordenamos el conocimiento.....	42
Fig. 16 - Diferentes observadores pueden clasificar el mismo objeto de distintas formas	43
Fig. 17 - Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior	44
Fig. 18- Un caso de uso y los escenarios que lo conforman.....	53
Fig. 19- Generalización entre actores	56
Fig. 20 – Ejemplo de una vista de un diagrama de casos de uso para un Complejo de Cines	59
Fig.21- Notación para una clase en UML	62
Fig. 22- Representación icónica de los tres tipos de clases de UML.....	62
Fig. 23- Notaciones para una Interface en UML, representación etiquetada e icónica	64
Fig. 24- Visualización de una clase con interfaces asociadas	65
Fig. 25- Notación para representar la relación de asociación entre clases en UML.....	65
Fig. 26- Notación para representar la relación de agregación entre clases en UML.....	65
Fig. 27- Notación para representar la relación de composición entre clases en UML	66
Fig. 28. Notación para representar la relación de generalización entre clases en UML, herencia simple	66

Fig. 29- Notación para representar la relación de generalización entre clases en UML, herencia múltiple	66
Fig. 30- Notación para representar la relación de dependencia entre clases en UML.....	67
Fig. 31. Notación para representar la relación de realización entre clases en UML	67
Fig. 32- Notación para representar la relación de contención entre clases en UML	67
Fig. 33- Vista parcial del Modelo de Dominio para el caso de estudio del Complejo de Cines.....	69
Fig. 34-. Notaciones para representar estados en UML.	70
Fig. 35- Transición entre estados en un Diagrama de Máquina de Estados.	71
Fig. 36-. Diagrama de Máquina de estados para la clase Sala, utilizada en el caso del Complejo de Cines	74
Fig. 37- Diagrama de Máquina de estados que muestra el uso de una sub-máquina y historia para los estados	74
Fig. 38-. Elementos de modelado utilizados en un Diagrama de Secuencia	75
Fig. 39- Líneas de vida de los objetos en un Diagrama de Secuencia	76
Fig. 40- Tipos de mensajes que pueden utilizarse en un Diagrama de Secuencia	76
Fig. 41- Creación y destrucción de objetos en un Diagrama de Secuencia.....	78
Fig. 42- Parámetros en un Diagrama de Secuencia	78
Fig. 43-. Diagrama de secuencia para el escenario del curso normal del caso de uso Registrar Película	80
Fig. 44: Vista del Diagrama de clases que muestra las clases necesarias para modelar el escenario del curso normal del caso de uso Registrar Película.....	81
Fig. 45- Funcionamiento general del Lenguaje Java	82
Fig. 46- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria.....	112
Fig. 47- Ejemplo de interfaces – Diagramación de Torneos.....	113
Fig. 48- Resaltado de comentarios TODO en la IDE Eclipse	116
Fig. 49-. Documentación JavaDoc generada automáticamente en base a comentarios	116
Fig. 50- Representación de nodos en una lista ordenada.....	117
Fig. 51- Jerarquía de Excepciones en el lenguaje Java	122
Fig. 52- La API de Java en contexto.....	127
Fig. 53- Estructura de paquetes general para la API de Java 8.....	128
Fig. 54- Jerarquía de componentes de la biblioteca Java Swing para interfaces gráficas.....	129

Tablas

Tabla 1 – Ejemplo de Clasificación de clases del dominio del problema	43
Tabla 2 – Diagramas de UML 2.0	49
Tabla 3 – Elementos básicos de modelado para un diagramas de Casos de Uso en UML 2.0.....	50
Tabla 4 – Tipos de Narrativas para la descripción de los Casos de Uso.....	52
Tabla 5 – Tipos de relaciones utilizadas para estructurar un diagrama de Casos de Uso	58
Tabla 6 – Ejemplo de descripción de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.	60
Tabla 7 – Ejemplo de narrativa de los actores que participan en la vista del diagrama de casos de uso presentado en la Figura 14.	61
Tabla 8 – Tipos de Mensajes que pueden utilizarse para modelar diagramas de secuencia.....	77
Tabla 9 – Tipos de básicos de datos en JAVA	88
Tabla 10 – Tipos de operadores aritméticos en JAVA.....	90
Tabla 11 – Tipos de operadores lógicos en JAVA	91
Tabla 12 – Operadores de Asignación en JAVA.....	93
Tabla 13 – Tags utilizados en Javadoc.....	114
Tabla 14. Extracto del JavaDoc para la clase ArrayList	120
Tabla 15. Extracto del JavaDoc para la clase Stack	121

Fuentes de Información

- **Armour Frank & Miller Granville** – ADVANCED USE CASE MODELING- EDITORIAL ADDISON WESLEY, AÑO 2000.
- **Shneider, Geri & Winters, Jason**- APPLYING USE CASES- A PRACTICAL GUIDE- ADDISON WESLEY, AÑO 1998.
- **Sommerville, Ian** - "INGENIERÍA DE SOFTWARE" 9na Edición (Editorial Addison-Wesley Año 2011).
- **Pressman Roger** - "Ingeniería de Software" 7ma. Edición - (Editorial Mc Graw Hill Año 2010).
- **Jacobson, Booch y Rumbaugh** - "EL PROCESO UNIFICADO DE DESARROLLO" (Editorial Addison-Wesley - Año 2000 1^a edición).
- **Booch, Rumbaugh y Jacobson** - "Lenguaje de Modelado Unificado" - (Editorial Addison-Wesley-Pearson Educación – 2da edición - Año 2006).
- **Booch, Grady** - Análisis y Diseño Orientado a Objetos. (Editorial Addison-Wesley/Diaz de Santos Año 1996).
- **Jacobson, Ivar** - Object-Oriented Software Engineering. (Editorial Addison-Wesley Año 1994).
- https://es.wikipedia.org/wiki/Tipo_de_dato#Java
- **Belmonte Fernández Oscar** - Introducción al lenguaje de programación Java. Una guía básica
- **Sánchez Asenjo Jorge** – Programación Básica en Lenguaje Java. (<http://www.jorgesanchez.net> Año 2009)
- **García de Jalón Javier, Rodríguez José Ignacio, Mingo Iñigo, Imaz Aitor, Brazález Alfonso, Larzabal Alberto, Calleja Jesús, García Jon** – Aprenda Java como si estuviera en primero (Escuela Superior de Ingenieros Industriales, Universidad de Navarra Año 1999)
- <http://java-white-box.blogspot.com.ar/2012/08/javadoc-que-es-el-javadoc-como-utilizar.html>
- <http://www.ecured.cu/Javadoc>
- http://www.ciberaula.com/articulo/listas_en_java
- http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=603:interface-list-del-api-java-clases-arraylist-linkedlist-stack-vector-ejemplo-con-arraylist-cu00917c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180
- <https://geekytheory.com/tutorial-14-java-swing-interfaces-graficas/>