Curso
Desarrollo de aplicaciones móviles Android Trainee

Módulo 1:

Programación básica en java
(5 unidades)

TALENTO DIGITAL
INTELIGENCIA HUMANA

AIEP

Instituto Profesional AIEP Spa.
Dirección Nacional de Educación Contínua
Programa Talento Digital 2021

Temario de unidades

1.1) Algoritmos

Definición de algoritmo

La palabra algoritmo se deriva de la traducción al latín de la palabra árabe alkhowarizmi, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX.

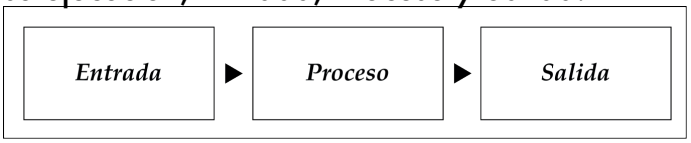
Un algoritmo se puede definir como una secuencia de instrucciones que representan un modelo de solución para determinado tipo de problemas. O bien como un conjunto de instrucciones que realizadas en orden conducen a obtener la solución de un problema o situación.

Existen dos tipos y son llamados así por su naturaleza:

Cualitativos: Son aquellos en los que se describen los pasos utilizando palabras.
Cuantitativos: Son aquellos en los que se utilizan cálculos numéricos para definir los pasos del proceso.

Partes de algoritmo

Todos los algoritmos se componen de 3 partes fundamentales que permiten su ejecución; Entrada, Proceso y Salida.



Entrada: Es toda aquella información necesaria que recibe el algoritmo.

Proceso: Son todos los cálculos necesarios para llegar a los resultados del problema o situación planteada.

Salida: Son los resultados producto de la información de entrada y los procesos aplicados a estos.

Variables

Las variables son espacios reservados en la memoria que, como su nombre indica, pueden cambiar de contenido a lo largo de la ejecución de un programa pueden ser valores proporcionados por el usuario como generados la salida de un algoritmo.

Variable numérica a través de un algoritmo de suma Valor +2

Valor 2	Valor 4	Valor 6	Valor 8	Valor 10	Valor 12

En el caso contrario cuando el dato no tendrá una variación durante la ejecución se le denomina constante como se muestra en el siguiente ejemplo de un algoritmo de conversión de temperatura:

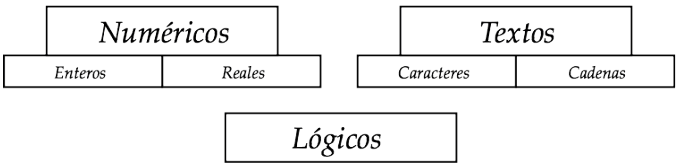
Conversión Celcius a Fahrenheit

	Proceso				
	C°			F°	
Entrada	26	x	1,8	=	46,8	Salida
	variable		constante		variable	

Tipos de Datos

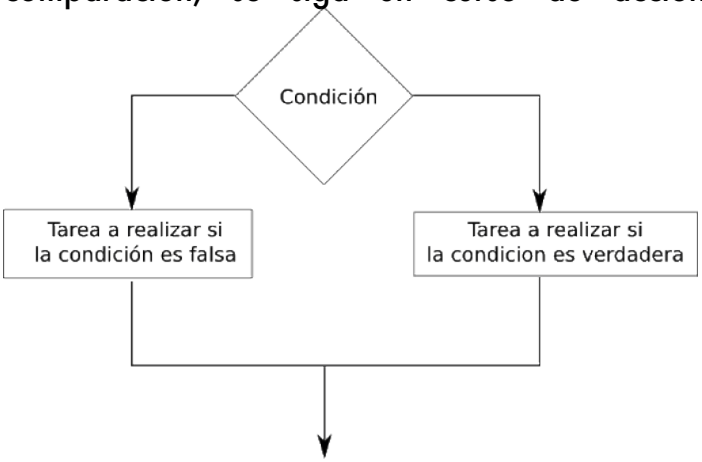
Al elaborar un algoritmo es necesario identificar a qué tipo de datos pertenecen cada una de las variables o constantes, ya sean estos números, letras, palabras, frases, entre otros y cada uno corresponde a un tipo de dato, que requerirá de una manipulación específica para obtener los resultados deseados.

Tipos básicos de datos



1.2) Estructuras de control condicional

Las estructuras condicionales comparan una variable o dato contra otro valor o valores, para que en base al resultado de esta comparación, se siga un curso de acción



dentro del algoritmo.

Teoría de Conjuntos

Los conjuntos son una de las estructuras básicas de las matemáticas, y por tanto de la informática. Se llama conjunto a toda agrupación, colección o reunión de individuos (cosas, personas o números) bien definidos que cumplen una propiedad determinada. A los objetos del conjunto se denominan "elementos".

Logica Preposicional

Es un sistema formal cuyos elementos más simples representan proposiciones, y cuyas constantes lógicas, llamadas conectivas lógicas, representan operaciones sobre proposiciones, capaces de formar otras proposiciones de mayor complejidad.

Ahora bien, una primera clasificación de las proposiciones ofrece dos tipos fundamentales de proposición, tomando en cuenta su estructura interna:

Proposiciones simples: O proposiciones atómicas, poseen una formulación sencilla desprovista de negaciones y nexos (conjunciones o disyunciones), por lo que constituyen un único término lógico.

Proposiciones simples

Una proposición simple es toda aquella en la que no hay operadores lógicos. O sea, aquellas cuya formulación es, justamente, simple, lineal, sin nexos ni negaciones, sino que expresa un contenido de manera sencilla.

Por ejemplo: "El mundo es redondo", "Las mujeres son seres humanos", "Un triángulo tiene tres lados" o "3 x 4 = 12".

Proposiciones compuestas. O proposiciones moleculares, poseen dos términos unidos por un nexo, o emplean negaciones dentro de su formulación, resultando en estructuras más complejas.

Proposiciones compuestas

Por el contrario, las proposiciones compuestas son aquellas que contienen algún tipo de operadores lógicos, como negaciones, conjunciones, disyunciones, condicionales, etc. Generalmente poseen más de un término, o sea, están formadas por dos proposiciones simples entre las cuales hay algún tipo de vínculo lógico condicionante.

Por ejemplo: "Hoy no es lunes" (˜p), "Ella es doctora y viene de España" (pˆq), "Llegué tarde porque había mucho tráfico" (p→q), "Comeré pan o me iré sin almorzar" (pˇq).v

Expresiones Lógicas

Un algoritmo, como se dijo anteriormente, es un conjunto de instrucciones, y cada una de éstas, puede considerarse como una expresión, que no es más que la combinación de variables, constantes y operadores. Dependiendo del tipo de operador, se clasifican en: Aritméticas, Relacionales y Lógicas.

Aritméticas	*Relacionales*	*Lógicas*
+	>	
-	<	*AND (&&)*
x	>=	*OR (||)*
/	<=	*NOT (!)*
	==	
	!=	

Las expresiones lógicas requieren el uso de los operadores lógicos: AND, OR, NOT y las combinaciones que se puedan generar

Para evaluar una expresión lógica se requieren conocer las tablas de verdad de la lógica de Boole.

AND

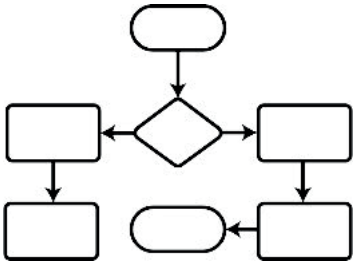
p	q	RESULTADO
V	V	V
V	F	F
F	V	F
F	F	F

OR

p	q	RESULTADO
V	V	V
V	F	V
F	V	V
F	F	F

1.3) Diagramación de algoritmos

La diagramación de algoritmos hace referencia principalmente a los diagramas de flujo los cuales proporcionan un medio para poder representar un proceso que posee una serie de pasos específicos claros para su realización estos pasos están en un algoritmo.



Los diagramas de flujo son construidos mediante cuadros de diferentes maneras los cuales son símbolos que representan una función en particular, estos símbolos están interconectados para la indicación del proceso a realizar, en ellos se encuentran los pasos determinados en el algoritmo indicando la secuencia de distintas tareas.

Arreglos

Un arreglo, es un conjunto de variables. Es como una lista de variables. Y para acceder a esas variables de dentro del arreglo se utiliza un índice o posición. En esta oportunidad nos centraremos en 2 tipos, los mas comunes; Unidimensionales y Bidimensionales.

Arreglos Unidimensionales

Un arreglo es una colección finita de datos del mismo tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común.

Por ejemplo, supongamos que queremos guardar las notas de los 20 alumnos de una clase. Podemos representar gráficamente el arreglo de notas de la siguiente forma:

Notas	6,4	6,1	5,3	6,7	6,2	5,8	5,9	6,0
	Notas[0]	Notas[1]	Notas[2]	Notas[3]	Notas[4]	Notas[5]	Notas[6]	Notas[7]

Para acceder a cada elemento del arreglo se utiliza el nombre del arreglo y un índice que indica la posición que ocupa el elemento dentro del arreglo. El índice se escribe entre corchetes.

El primer elemento del arreglo ocupa la posición 0, el segundo la posición 1, etc
En un arreglo de N elementos el último ocupará la posición N-1. En el ejemplo anterior, **notas[0]** contiene la nota del primer alumno y **notas[7]** contiene la del último.

Los índices deben ser enteros no negativos.

Arreglos Bidimensionales

Los arreglos bidimensionales representan tablas de datos o valores. Para acceder a cada elemento del arreglo se utiliza el nombre del arreglo y los índices que indica la posición que ocupa el elemento dentro del arreglo. Los índices se escribe entre corchetes.

Notas

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]
[1]	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]
[2]	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]
[3]	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]

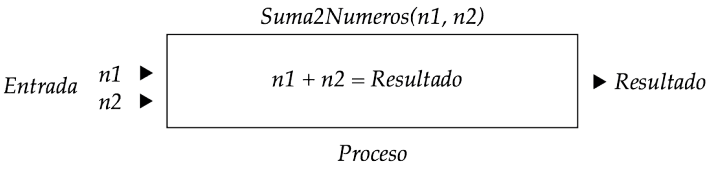
En el ejemplo anterior, podremos acceder a una posición especifica del arreglo utilizando los indices o coordenadas y el nombre del arreglo como **notas[3][2]**

Funciones

En programación, una función es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:
· los parámetros, que son los valores que recibe la función como entrada;
· el código de la función, que son las operaciones que hace la función; y
· el resultado (o valor de retorno), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.



2.1) El Entorno Java para la programación

Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems.

Los entornos de desarrollo Java son aplicaciones que permiten al programador implementar las abstracciones del mundo real en un aplicación concreta mediante la introducción de secuencias de código con sus estructuras de programación.

Para empezar vamos a instalar el entorno de desarrollo para Windows. Podemos descargar el software gratuitamente de la página oficial de Oracle.

Actualmente los paquetes de desarrollo que ofrece Oracle vienen con una herramienta **IDE** (Integrated Development Environment) llamada **Netbeans**. Esta herramienta es muy potente, a lo mejor demasiado para empezar con Java, nosotros nos centraremos en otra llamada **Eclipse** también gratuita.

Primero es necesario descargar e instalar el entorno de desarrollo de JAVA, JDK (Java Develpment Kit)

Link:

https://www.oracle.com/technetwork/es/java/javase/downloads/index.html

Una vez descargado archivo. Para instalar hay que ejecutar el archivo, seguir las instrucciones y darle una ruta de instalación en el momento en que nos lo requiera.

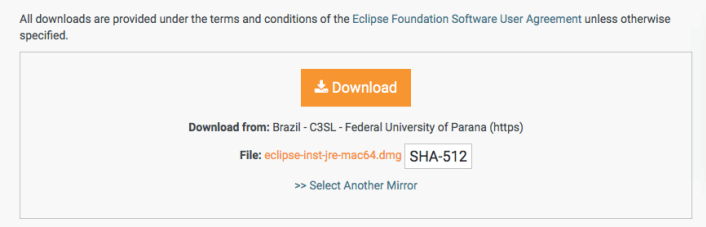
El Entorno Integrado de Desarrollo

Completada la instalación del JDK procederemos a la instalación del IDE (Integrated Development Environment) o en español Entorno Integrado de Desarrollo es aquel que nos permitirá de una forma simplificada con mayores prestaciones la elaboración de código, prueba y depuración.

Para programación y desarrollo en JAVA existen 2 principales IDE; **Netbeans y Eclipse.** En esta oportunidad nos enfocaremos en la instalación y puesta en marcha de Eclipse por ser un IDE recomendado para la iniciación en JAVA.

Link:

https://www.eclipse.org/downloads/



Una vez descargado archivo. Para instalar hay que ejecutar el archivo, seguir las instrucciones y darle una ruta de instalación en el momento en que nos lo requiera.

Antes de arrancar el programa debemos seleccionar el espacio de trabajo. Eclipse guarda todos nuestros proyectos en el espacio de trabajo. Podemos dejarlo como está o poner la ruta que queramos. Cada vez que arranque eclipse nos preguntará esto, a menos que marquemos la casilla "Use this as the default and do not ask again". Luego hacemos click en "OK".

Tipos de datos en Java

Java cuenta con un pequeño conjunto de tipos de datos primitivos. Podríamos considerarlos fundamentales, ya que la mayor parte de los demás tipos, los tipos estructurados o complejos, son composiciones a partir de estos más básicos.

Estos tipos de datos primitivos sirven para gestionar los tipos de información más básicos, como números de diversas clases o datos de tipo verdadero/falso (también conocidos como "valores booleanos").

De estos tipos primitivos, ocho en total, seis de ellos están destinados a facilitar el trabajo con números. Podemos agruparlos en dos categorías: tipos numéricos enteros y tipos numéricos en punto flotante. Los primeros permiten operar exclusivamente con números enteros, sin parte decimal, mientras que el segundo grupo contempla también números racionales o con parte decimal.

Tipos numéricos enteros

En Java existen cuatro tipos destinados a almacenar números enteros. La única diferencia entre ellos es el número de bytes usados para su almacenamiento y, en consecuencia, el rango de valores que es posible representar con ellos. Todos ellos emplean una representación que permite el almacenamiento de números negativos y positivos. El nombre y características de estos tipos son los siguientes:

byte: como su propio nombre denota, emplea un solo byte (8 bits) de almacenamiento. Esto permite almacenar valores en el rango [-128, 127].

short: usa el doble de almacenamiento que el anterior, lo cual hace posible representar cualquier valor en el rango [-32.768, 32.767].

int: emplea 4 bytes de almacenamiento y es el tipo de dato entero más empleado. El rango de valores que puede representar va de -231 a 231-1.

long: es el tipo entero de mayor tamaño, 8 bytes (64 bits), con un rango de valores desde -263 a 263-1.

Tipos numéricos en punto flotante

Los tipos numéricos en punto flotante permiten representar números tanto muy grandes como muy pequeños además de números decimales. Java dispone de 2 tipos concretos en esta categoría:

float: conocido como tipo de precisión simple, emplea un total de 32 bits. Con este tipo de datos es posible representar números en el rango de 1.4x10-45 a 3.4028235x1038.

double: sigue un esquema de almacenamiento similar al anterior, pero usando 64 bits en lugar de 32. Esto le permite representar valores en el rango de 4.9x10-324 a 1.7976931348623157x10308.

Booleanos y caracteres

Aparte de los 6 tipos de datos que acabamos de ver, destinados a trabajar con números en distintos rangos, Java define otros dos tipos primitivos más:

boolean: tiene la finalidad de facilitar el trabajo con valores "verdadero/falso" (booleanos), resultantes por regla general de evaluar expresiones. Los dos valores posibles de este tipo son true y false.

char: se utiliza para almacenar caracteres individuales (letras, para entendernos). En realidad está considerado también un tipo numérico, si bien su representación habitual es la del carácter cuyo código almacena. Utiliza 16 bits y se usa la codificación UTF-16 de Unicode.

Operadores en Java

Los operadores son símbolos que indican cómo se deben manipular los operandos. Los operadores junto con los operandos forman una expresión, que es una fórmula que define el cálculo de un valor. Los operandos pueden ser constantes, variables o llamadas a funciones, siempre que éstas devuelvan algún valor.

Expresión = Operandos + Operadores

Operadores Aritméticos

Se utilizan para realizar operaciones aritméticas simples en tipos de datos primitivos.

Operador de asignación

El operador de asignación se usa para asignar un valor a cualquier variable. Tiene una asociación de derecha a izquierda, es decir, el valor dado en el lado derecho del operador se asigna a la variable de la izquierda y, por lo tanto, el valor del lado derecho debe declararse antes de usarlo o debe ser una constante.

El formato general del operador de asignación es, en muchos casos, el operador de asignación se puede combinar con otros operadores para construir una versión más corta de la declaración llamada Declaración Compuesta (Compound Statement).

Por ejemplo, en lugar de a = a + 5, podemos escribir a + = 5.

+ = , para sumar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

– = , para restar el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

*** =** , para multiplicar el operando izquierdo con el operando derecho y luego asignándolo a la variable de la izquierda.

/ = , para dividir el operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

^ = , para aumentar la potencia del operando izquierdo al operando derecho y asignarlo a la variable de la izquierda.

% = , para asignar el módulo del operando izquierdo con el operando derecho y luego asignarlo a la variable de la izquierda.

Operadores relacionales

Estos operadores se utilizan para verificar relaciones como igualdad, mayor que, menor que. Devuelven el resultado booleano después de la comparación y se usan ampliamente en las instrucciones de bucle, así como en las sentencias condicionales if/else. El formato general es, variable operador_relacion valor

Algunos de los operadores relacionales son:

==, Igual a: devuelve verdadero si el valor del lado izquierdo es igual al lado derecho.

! =, No igual a: devuelve verdadero si el valor del lado izquierdo no es igual al lado derecho.

<, menos que: el resultado verdadero si el valor del lado izquierdo es inferior al del lado derecho.

<=, menor o igual que: devuelve verdadero si el valor del lado izquierdo es menor o igual que el lado derecho.

>, Mayor que: devuelve verdadero si el valor del lado izquierdo es mayor que el lado derecho.

> =, Mayor que o igual a: regresa verdadero si el valor del lado izquierdo es mayor o igual que el lado derecho.

Operadores de incremento

Estos operadores son utilizados principalmente en los ciclos para realizar los conteos y evaluaciones de la cantidad de iteraciones estableciendo la continuidad o termino del ciclo o bucle.

Incremento, ++
Decremento, ++

Expresión	Valor inicial de X	Valor final de Y	Valor final de X
y = x++	5	5	6
y = ++x	5	6	6
y = x - -	5	5	4
y = - - x	5	4	4

Operadores lógicos

Utilizados para evaluaciones lógicas en ciclos o sentencias condicionales como if/else/or. Realizan operaciones sobre datos booleanos y tienen como resultado un valor booleano.

AND (&&)
OR (||)
NOT (!)

2.2) Sentencias de control

Ciclos en Java

Un ciclo en Java o bucle en Java permite repetir una o varias instrucciones cuantas veces lo necesitemos o sea necesario, por ejemplo, si quisiéramos escribir los números del uno al cien no tendría sentido escribir cien líneas de código mostrando un número en cada una de estas, para eso y para varias cosas más, es útil un ciclo. Un ciclo nos ayuda a llevar a cabo una tarea repetitiva en una cantidad de líneas muy pequeña y de forma prácticamente automática.

Existen diferentes tipos de ciclos o bucles en Java, cada uno tiene una utilidad para casos específicos y depende de nuestra habilidad y conocimientos poder determinar en qué momento o situación es bueno usar alguno de ellos. Tenemos entonces a nuestra disposición los siguientes tipos de ciclos en Java:

FOR
un ciclo for es una estructura iterativa para ejecutar un mismo segmento de código una cantidad de veces deseada; conociendo previamente un valor de inicio, un tamaño de paso y un valor final para el ciclo realizando finalmente el incremento en su contador para la siguiente iteración.
	for (**int** i = valor inicial; i <= valor **final**; i = i ++)
	{
	....
	Instrucciones....
	....
	}

WHILE
El bucle while presenta ciertas similitudes y ciertas diferencias con el bucle for. La repetición en este caso se produce no un número predeterminado de veces, sino mientras se cumpla una condición. Conceptualmente el esquema más habitual es el siguiente:
	Int i = 0;
	while (i <= valor **final**)
	{
	i++;
	}

DO-WHILE
El bucle do ... while es muy similar al bucle while. La diferencia radica en cuándo se evalúa la condición de salida del ciclo. En el bucle while esta evaluación se realiza antes de entrar al ciclo, lo que significa que el bucle puede no llegar ejecutarse. En cambio, en un bucle do ... while, la evaluación se hace después de la primera ejecución del ciclo, lo que significa que el bucle obligatoriamente se ejecuta al menos en una ocasión. A modo de ejercicio, escribe este código y comprueba los resultados que se obtienen con él.
	do {
	contador += 1;
	} **while** (contador < 10);

Arreglos y Colecciones

Arreglos

Las características principales de los arreglos son que una vez dimensionales no pueden variar su tamaño. En este sentido la estructura se manipula directamente y por esto es también ofrece un mejor rendimiento en los accesos directos e iteraciones.

Para definir un arreglo o array en java indicamos el tipo de datos que usaremos, el nombre del arreglo y los corchetes de la siguiente forma:

"**tipoArray** nombreArray[];"

int temperaturas[];

Personas listado[];

Colecciones

Java ofrece este tipo de almacenamiento para objetos las cuales son estructuras completas basadas en clases de Java que realizan una gestión interna del almacenamiento y recuperación de los elementos. Tienen también una gestión dinámica de la memoria pero solo permite el almacenamiento de objetos.

Básicamente una colección es un objeto que permite almacenar y agrupar otros objetos. En una colección se pueden realizar operaciones sobre los objetos que están almacenados en ella.

Para trabajar con colecciones se utiliza un marco de trabajo en Java llamado Collections. Las clases e interfaces que componen este marco se encuentran en las librerías básicas **java.util.**

Convenciones

Las convenciones de nombrado hacen los programas más entendibles haciéndolos más fáciles de leer. También pueden proporcionar información sobre la función del identificador, por ejemplo, si es una constante, un paquete o una clase, lo que puede ayudarnos a entender el código.

Java define una serie de palabras para la identificación de operaciones, métodos, clases etc. con el fin de que el compilador pueda entender los procesos que se están desarrollando. Estas palabras no pueden ser usadas por el desarrollador para nombres de métodos, variables, clases entre otras

Convenciones Clases e Interfaces

· La primer letra debe ser mayúscula

· Utiliza nomenclatura camelCase

· Para las clases, los nombres deben de ser sustantivos (Sujeto) y van después de la palabra reservada **class**

· Para las interfaces, los nombres deben de ser adjetivos (Califica el sustantivo) y van después de la palabra reservada **interface**

Ejemplo:
class Persona
class ClasePrincipal
class VentanaRegistro
interface ActionListener
interface MouseInputListener

Convenciones en Paquetes

· Deben ser escritos todo en minúscula.
· Van después de la palabra reservada package
· Si se van a usar paquetes dentro de otros paquetes, se unen mediante un punto (.)
· Finalizan con ;

Ejemplo
package ventanas;
package vo;
package dao;
package imagenes.iconos;

Convenciones en Métodos

· La primer letra debe ser minúscula
· Utiliza nomenclatura camelCase
· Los nombres deben conformarse por el par verbo + sustantivo
· el nombre va después del tipo de método (void, int, double, String)
· al finalizar el nombre del método debe indicarse mediante paréntesis con o sin argumentos ()

Ejemplo
void miMetodo()
int sumaEnteros(int a, int b)
Sting mensaje(String saludo)
boolean retornaPermisos(int tipoUsuario)

Convenciones en Variables

· La primer letra debe ser minúscula
· Utiliza nomenclatura camelCase

·el nombre va después del tipo de dato (int, String, double, boolean)

· Es recomendable utilizar nombres con un significado explícito, y en lo posible, cortos

Ejemplo
int edad
String nombre
String direccionCasa
boolean resultadoPrueba

Convenciones en Constantes

· Todas las letras de cada palabra deben estar en mayúsculas
· Se separa cada palabra con un _
· se declaran similar a las variables, con la diferencia de que el tipo de dato va después de la palabra reservada **final.**

Ejemplo
final int EDAD
final String CODIGO_CIUDAD
final double PI

Estilos de codificación

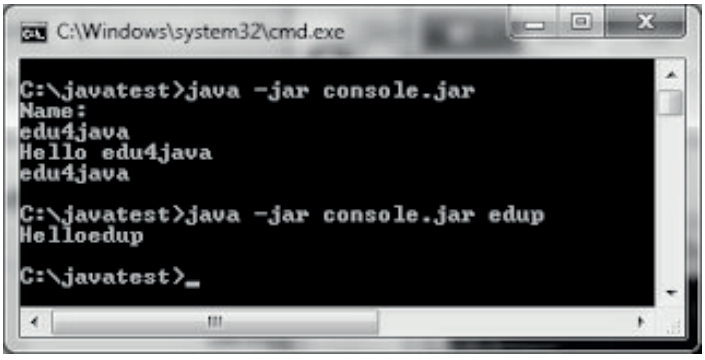
Como todo lenguaje, Java tiene su propio estilo de codificación. Los estilos de codificación (cómo tabular, dónde ubicar las llaves, cómo nombrar las variables, etc) nos ayudan a crear código que se vea consistente y sea fácil de interpretar por distintos desarrolladores. Un estilo de codificación no es una cuestión de gustos, sino una convención que ayuda a los desarrolladores.

2.3) Creando aplicaciones de consola en Java

Lo primero que tenemos que saber es que tipos de programas podemos realizar con Java, fundamentalmente son tres:

· Aplicaciones de consola
· Aplicaciones de propósito general
· Applets

Las aplicaciones de consola son aquellos programas en Java que se van a ejecutar en una ventana de comandos o de Shell, como vemos a continuación:



Cuando se utiliza un entorno de desarrollo como Eclipse o Netbeans estos incluyen ya su ventana de consola para no tener que ejecutar la que viene con el sistema operativo.

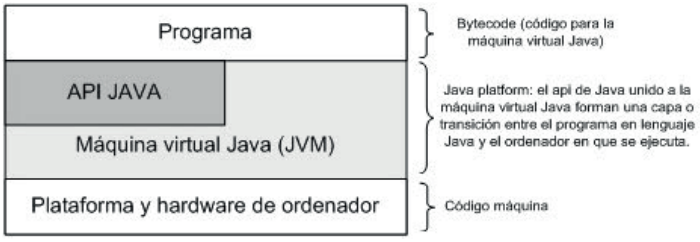
Las aplicaciones de propósito general, es decir, programas que pueden construirse para diversos objetivos o para cubrir diferentes necesidades, un ejemplo de un programa hecho en Java sería Netbeans o Eclipse.

Y los applets son programas creados en Java que se ejecutan dentro de un navegador como si fuera un plugin.

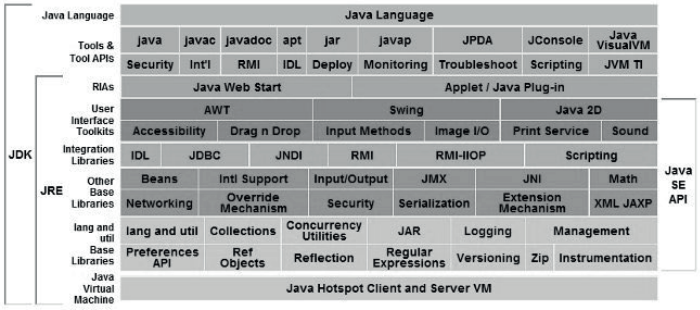
El API de Java y las bibliotecas String y Math

Al instalar Java (el paquete JDK) en nuestro ordenador, además del compilador y la máquina virtual de Java se instalan bastantes más elementos. Entre ellos, una cantidad muy importante de clases que ofrece Java y que están a disposición de todos los programadores listas para ser usadas. Estas clases junto a otros elementos forman lo que se denomina API (Application Programming Interface) de Java.

La mayoría de los lenguajes orientados a objetos ofrecen a los programadores bibliotecas de clases que facilitan el trabajo con el lenguaje.



Cuando Java es instalado en nuestro ordenador también instalamos múltiples herramientas, entre ellas una serie de "librerías" (paquetes) a cuyo conjunto solemos referirnos como "biblioteca estándar de Java". Las librerías contienen código Java listo para ser usado por nosotros. Ese es el motivo de que podamos usar clases como System, String o Math sin necesidad de programarlas.



Aunque no podamos acceder al código fuente, sí podemos crear objetos de los tipos definidos en la librería e invocar sus métodos. Para ello lo único que hemos de hacer es conocer las clases y la signatura de los métodos, y esto lo tenemos disponible en la documentación de Java accesible para todos los programadores a través de internet.

String

La clase **String** está orientada a manejar cadenas de caracteres. Hasta este momento hemos utilizado algunos métodos de la clase String (equals, compareTo)

Métodos

boolean equals(String s1)

Como vimos el método equals retorna true si el contenido de caracteres del parámetro s1 es exactamente igual a la cadena de caracteres del objeto que llama al método equals.

boolean equalsIgnoreCase(String s1)

El funcionamiento es casi exactamente igual que el método equals con la diferencia que no tiene en cuenta mayúsculas y minúsculas (si comparamos 'Ana' y 'ana' luego el método equalsIgnoreCase retorna true)

int compareTo(String s1)

Este método retorna un 0 si el contenido de s1 es exactamente igual al String contenido por el objeto que llama al método compareTo. Re-

torna un valor >0 si el contenido del String que llama al método compareTo es mayor alfabéticamente al parámetro s1.

char charAt(int pos)

Retorna un caracter del String, llega al método la posición del caracter a extraer.

int length()

Retorna la cantidad de caracteres almacenados en el String.

String substring(int pos1,int pos2)

Retorna un substring a partir de la posición indicada en el parámetro pos1 hasta la posición pos2 sin incluir dicha posición.

int indexOf(String s1)

Retorna -1 si el String que le pasamos como parámetro no está contenida en la cadena del objeto que llama al método. En caso que se encuentre contenido el String s1 retorna la posición donde comienza a repetirse.

String toUpperCase()

Retorna un String con el contenido convertido todo a mayúsculas.

String toLowerCase()

Retorna un String con el contenido convertido todo a minúsculas.

Math

La clase **Math** representa la librería matemática de Java. Las funciones que contiene son las de todos los lenguajes, parece que se han metido en una clase solamente a propósito de agrupación, por eso se encapsulan en Math, y lo mismo sucede con las demás clases que corresponden a objetos que tienen un tipo equivalente (Character, Float, etc.). El constructor de la clase es privado, por los que no se pueden crear instancias de la clase. Sin embargo, Math es public para que se pueda llamar desde cualquier sitio y static para que no haya que inicializarla.

Funciones matemáticas

Si se importa la clase, se tiene acceso al conjunto de funciones matemáticas estándar:

Math.abs(x)	para int, long, float y double
Math.sin(	double)
Math.cos(	double)
Math.tan(	double)
Math.asin(	double)
Math.acos(	double)
Math.atan(	double)
Math.atan2(	double,double)
Math.exp(	double)
Math.log(	double)
Math.sqrt(	double)
Math.ceil(	double)
Math.floor(	double)
Math.rint(	double)
Math.pow(	a,b)
Math.round(x)	para double y float
Math.random()	devuelve un double
Math.max(a,b)	para int, long, float y double
Math.min(a,b)	para int, long, float y double
Math.E	para la base exponencial
Math.PI	para PI

Javadoc

Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML 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. BlueJ también utiliza javadoc y permite la generación automática de

documentación, y visionarla bien de forma completa para todas las clases de un proyecto, o bien de forma particular para una de las clases de un proyecto.

Este sistema consiste en incluir comentarios en el código, utilizando unas etiquetas especiales, que después pueden procesarse y generar un juego navegable de documento HTML.

Estos comentarios están incluidos en bloques entre las etiquetas /** y */

Para poder generar la documentación de alguna clase o método se tienen que usar etiquetas HTML precedidas del carácter "@". Las etiquetas tienen que estar dentro de un comentario java iniciando con "/**" y terminando con "*/".
Estos comentarios con etiquetas de documentación pueden ir al inicio de:

Una clase.
Un atributo.
Un método.

Etiquetas javaDoc

Estas son las **etiquetas** más comunes para generar **JavaDoc**

Etiqueta	Descripción
@author	Indica el nombre de quien desarrollo el componente.
@deprecated	Indica que algún método o clase u otro componente está obsoleto.
@param	Indica un parámetro de un método, se tiene que usar para todos los parámetros del método.
@return	Indica el valor de retorno de un método
@see	Indica que el componente puede hacer referencia a otro. Ejemplo: #método(); clase#método();
@throw	Indica que excepción lanza el método.
@version	Indica la versión actual del componente.

Ejemplo

```
/**
 * Esta clase contiene los atributos y métodos de una persona
 * @author Mario
 * @version 1.0
 * @see Persona
 */
```

3.1) El Paradigma de la Orientación a Objetos (POO)

Un paradigma de programación es una manera o estilo de programación de software. Existen diferentes formas de diseñar un lenguaje de programación y varios modos de trabajar para obtener los resultados que necesitan los programadores.

En este modelo de paradigma se construyen modelos de objetos que representan elementos (objetos) del problema a resolver, que tienen características y funciones. Permite separar los diferentes componentes de un programa, simplificando así su creación, depuración y posteriores mejoras. La programación orientada a objetos disminuye los errores y permite la reutilización del código. Es una manera especial de programar, que se acerca de alguna manera a cómo expresaríamos las cosas en la vida real.

Podemos definir un objeto como una estructura abstracta que, de manera más fiable, describe un posible objeto del mundo real y su relación con el resto del mundo que lo rodea a través de interfaces. Ejemplos de lenguajes de programación orientados a objetos serían **Java, Python o C#.**

La programación orientada a objetos se sirve de diferentes conceptos como:

- Abstracción de datos
- Encapsulación
- Eventos
- Modularidad
- Herencia
- Polimorfismo

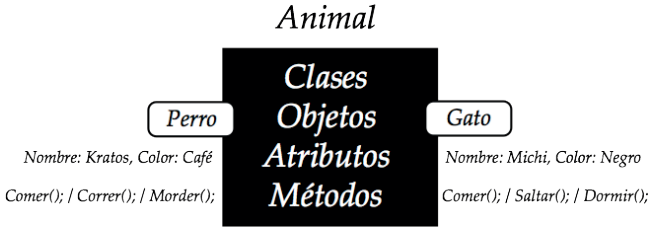
Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores que busca evitar duplicar el código y crear de esta manera programas eficientes.

3.2) Clases y Objetos

Una clase es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse 'animal' y tener una serie de atributos, como 'nombre' o 'edad' (que normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

Un ejemplo sencillo de un objeto, como decíamos antes, podría ser un animal. Un animal tiene una edad, por lo que creamos un nuevo atributo de 'edad' y, además, puede envejecer, por lo que definimos un nuevo método. Datos y lógica. Esto es lo que se define en muchos programas como la definición de una clase, que es la definición global y genérica de muchos objetos. Con la clase se pueden crear instancias de un objeto, cada uno de ellos con sus atributos definidos de forma independiente. Con esto podríamos crear un gato llamado Michi, con 3 años de edad, y otro animal, este tipo perro y llamado Kratos, con una de edad de 4 años. Los dos están definidos por la clase animal, pero son dos instancias distintas. Por lo tanto, llamar a sus métodos puede tener resultados diferentes. Los dos comparten la lógica, pero cada uno tiene su estado de forma independiente.



3.3) Métodos Constructores

Cuando se construye un objeto es necesario inicializar sus variables con valores coherentes, imaginemos un objeto de la clase Persona cuyo atributo color de pelo al nacer sea verde, un estado incorrecto tras construir el objeto persona.

La solución en los lenguajes orientados a objetos es emplear los constructores. Un constructor es un método perteneciente a la clase que posee unas características especiales:

· Se llama igual que la clase.
· No devuelve nada, ni siquiera void.
· Pueden existir varios, pero siguiendo las reglas de la sobrecarga de funciones.
· De entre los que existan, tan sólo uno se ejecutará al crear un objeto de la clase.

Dentro del código de un constructor generalmente suele existir inicializaciones de variables y objetos, para conseguir que el objeto sea creado con dichos valores iniciales.

Accesadores y Mutadores

El papel de los descriptores de acceso y los mutadores es devolver y establecer los valores del estado de un objeto

Accesadores

Se utiliza un método de acceso para devolver el valor de un campo privado. Sigue un esquema de nomenclatura que antepone la palabra "get" al comienzo del nombre del método. Por ejemplo, agreguemos métodos de acceso para firstname, middleNames y lastname:

Estos métodos siempre devuelven el mismo tipo de datos que su campo privado correspondiente (por ejemplo, String) y luego simplemente devuelven el valor de ese campo privado.

Mutadores

Se utiliza un método mutador para establecer un valor de un campo privado. Sigue un esquema de nomenclatura que antepone la palabra "set" al comienzo del nombre del método. Por ejemplo, agreguemos campos mutantes para la dirección y el nombre de usuario:

Estos métodos no tienen un tipo de retorno y aceptan un parámetro que es el mismo tipo de datos que su campo privado correspondiente. Luego, el parámetro se usa para establecer el valor de ese campo privado.

Modificadores de Acceso

Como su nombre indica, los modificadores de acceso en Java ayudan a restringir el alcance de una clase, constructor, variable, método o miembro de datos.

Hay cuatro tipos de modificadores de acceso disponibles en Java:

· Default – No se requiere palabra clave
· Private
· Protected
· Public

Tabla de Modificadores de Acceso en Java

Modificador/Acceso	Clase	Paquete	Subclase	Todos
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

Modificador de acceso por defecto (default)

Cuando no se especifica ningún modificador de acceso para una clase, método o miembro de datos, se dice estar teniendo **modificador de acceso default** por defecto.

Los miembros de datos, clase o métodos que no se declaran utilizando ningún modificador de acceso, es decir, que tengan un modificador de acceso predeterminado, solo son accesibles dentro del mismo paquete.

Modificador de acceso privado (private)

El modificador de acceso privado se especifica con la palabra clave . Los métodos o los miembros de datos declarados como privados solo son accesibles dentro de la clase en la que se declaran.

· Cualquier otra clase del mismo paquete no podrá acceder a estos miembros.
· Las clases e interfaces no se pueden declarar como privadas (private).

Modificador de acceso protegido (protected)

El modificador de acceso protegido se especifica con la palabra clave .

· Los métodos o miembros de datos declarados como son accesibles dentro del mismo paquete o sub-clases en paquetes diferentes.

Modificador de acceso público (public)

El modificador de acceso público se especifica con la palabra clave .
· El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso.
· Las clases, métodos o miembros de datos que se declaran como públicos son accesibles desde cualquier lugar del programa. No hay restricciones en el alcance de los miembros de datos públicos.

Modificadores que no son de acceso

En Java, tenemos 7 modificadores que no son de acceso o, a veces, también llamados **especificadores**. Se usan con clases, métodos, variables, constructores, etc. para proporcionar información sobre su comportamiento a la JVM. Y son:

· static
· final
· abstract
· synchronized
· transient
· volatile
· native

Colaboración entre objetos

Una aplicación orientada a objetos es una colección de objetos, que colaboran entre sí para dar una funcionalidad global. Cada objeto debe asumir una parte de las responsabilidades y delegar otras a los objetos colaboradores. Existirá un objeto que asume las funciones de la aplicación principal, será el encargado de crear las instancias oportunas y tener un único punto de entrada (main).

Cabe hacernos una pregunta fundamental, ¿en cuántos objetos debemos descomponer la aplicación? Si realizamos objetos muy pequeños, cada uno de ellos será fácil de desarrollar y mantener, pero resultará difícil la integración. Si realizamos objetos grandes, la integración será fácil, pero el desarrollo de los objetos complicado. Se debe buscar un equilibrio. En definitiva, y sin adentrar más en la estructuración de las clases de una aplicación, nuestra aplicación tendrá un conjunto de clases que están relacionadas entre ellas. Se considera que existe una relación entre clases si sus objetos colaboran, es decir, se mandan mensajes.

Composición

Hay dos formas de reutilizar el código, mediante la composición y mediante la herencia. La composición significa utilizar objetos dentro de otros objetos. Por ejemplo, un applet es un objeto que contiene en su interior otros objetos como botones, etiquetas, etc. Cada uno de los controles está descrito por una clase.

Diagramas de Clases

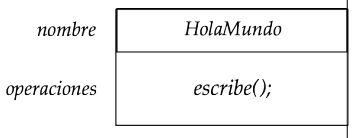
Una clase es una descripción de conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.

Las clases son gráficamente representadas por cajas con compartimentos para:

- Nombre de la clase, atributos y operaciones / métodos y operaciones
- Responsabilidades, Reglas, Historia de Modificaciones, etc.

Los diseñadores desarrollan clases como conjuntos de compartimentos que crecen en el tiempo de agregando incrementalmente aspectos y aspectos y funcionalidades.

Representación Gráfica	**Código**

	import java.awt.Graphics;
	class **HolaMundo** extends java.applet.Applet {
	public void **escribe** (Graphics g) {
	g.drawString (“¡Hola Mundo!”, 10, 10);
	}
	}

4.1) Herencia y Polimorfismo

Herencia

La **herencia** define relaciones jerárquicas entre clases, de forma que atributos y métodos comunes puedan ser reutilizados. Las clases principales extienden atributos y comportamientos a las clases secundarias. A través de la definición en una clase de los atributos y comportamientos básicos, se pueden crear clases secundarias, ampliando así la funcionalidad de la clase principal y agregando atributos y comportamientos adicionales.

Volviendo al ejemplo de los animales, se puede usar una sola clase de animal y agregar un atributo de tipo de animal que especifique el tipo de animal. Los diferentes tipos de animales necesitarán diferentes métodos, por ejemplo, las aves deben poder poner huevos y los peces, nadan. Incluso cuando los animales tienen un método en común, como moverse, la implementación necesitaría muchas declaraciones "si" para garantizar el comportamiento de movimiento correcto. Por ejemplo, las ranas saltan, mientras que las serpientes se deslizan. El principio de herencia nos permite solucionar este problema.

El polimorfismo consiste en diseñar objetos para compartir comportamientos, lo que nos permite procesar objetos de diferentes maneras. Es la capacidad de presentar la misma interfaz para diferentes formas subyacentes o tipos de datos. Al utilizar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute diferentes comportamientos de dos formas: anulación de método y sobrecarga de método.

Interfaces

Las interfaces son una forma de especificar qué debe hacer una clase sin especificar el cómo.
Las interfaces tienen una semejanza con las clases abstractas, en el sentido que no tiene sentido definir objetos de instancia de una interfaz. Igual que las clases abstractas clase asociada se comprometa a implementar todos los métodos en ellas definidos, pero en este caso la relaciones no es de herencia en su totalidad, dado que no hay atributos en la definición de una interfaz.

Las interfaces no son clases, sólo especifican requerimientos para la clase que las implementa o, desde su uso, los servicios para la función que manipula un objeto que se dice cumplir con la interfaz.

¿Cómo se usan las interfaces?

Debemos crear una clase que implementa la interfaz. Implementarla implementar cada uno de los métodos de la interfaz.

Podemos definir métodos que usen como parámetro objetos que implementen la interfaz. Basta usar el nombre de la interfaz como el tipo del parámetro.

Implementación del Poliformismo

El polimorfismo en Java y en la POO se refiere a que un objeto puede tomar diferentes formas de comportarse, es decir, que las subclases de una clase pueden definir su propio comportamiento.

Estas heredan del mismo padre, cada clase sobrescribirá un método del padre para ver cómo se comporta el polimorfismo.
Luego, las instancias de una clase que implemente la Interfaz, pueden tomar el lugar del argumento donde se espere alguna que implemente la interfaz.

Clase padre

Esta clase tiene un método llamado printMensaje(), el cual solo va a imprimir un
mensaje en la consola de salida.

```
public class Animal {
      private String especie;
      public Animal(String especie){
            this.especie = especie;
      }
      public void printMensaje(){
            System.out.println("Soy un animal de la especie: "
+  this.getEspecie());
      }   public String getEspecie() {
            return especie;
            }   public void setEspecie(String especie) {
            this.especie = especie;
      }
}
```

Clases hijas

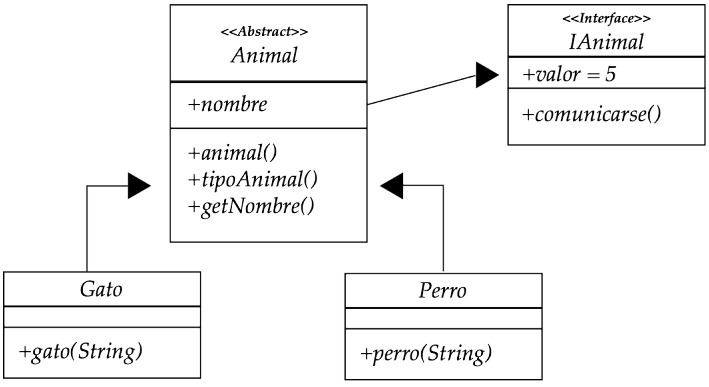
Cada una tiene sus métodos y atributos, las dos heredan la funcionalidad del padre, también las dos hijas están sobrescribiendo el método **printMensaje()** del padre.**Polimorfismo en un diagrama de clases**

```
public class Perro extends Animal {

   private String nombre;
   public Perro(String especie, String nombre) {
      super(especie);
      this.nombre = nombre;
   }
   public void printMensaje() {
      super.printMensaje();
      System.out.println("Soy un perro que ladra");
   }
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
}

public class Gato extends Animal {
   private String nombre;
   public Gato(String especie, String nombre) {
      super(especie);
      this.nombre = nombre;
   }
   public void printMensaje() {
      super.printMensaje();
      System.out.println("Soy un gato que maulla");
   }
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
```

Podemos ver que se tiene un árbol de herencia definido, donde la clase Abstracta Animal implementa la interface IAnimal y al mismo tiempo es clase Padre de Gato y Perro, los cuales implementaran no solo los métodos abstractos de Animal sino también el método comunicarse() de la interface IAnimal. Una de las reglas al trabajar con clases abstractas o interfaces es que todas las clases concretas que desciendan de ellas, están obligadas a implementar sus métodos. En este caso la clase Animal al ser abstracta no esta obligada a hacerlo, pero sus hijas si.



La clase Abstracta que implementa la Interface IAnimal no esta obligada a implementar el método comunicarse() ya que también es clase abstracta.

Tenemos el método tipoAnimal() el cual si debe ser declarado como abstracto, evidenciamos también el atributo nombre el cual lo declaramos como private y solo accederemos a el en nuestro árbol de herencia, con el aplicamos el concepto de Encapsulación.

Las Clases Gato y Perro heredan de la clase abstracta Animal, por ende implementan

el método tipoAnimal(), y como Animal implementa la interface IAnimal, entonces tanto Gato como Perro al ser clases concretas están obligadas a implementar el método comunicarse().

En las clases también se utiliza la propiedad nombre que es enviada al constructor de Animal mediante el llamado a super(nombre).

La importancia de trabajar con un paradigma Orientado a Objetos da facilidades tanto a nivel de optimización como a nivel de estructura y lógica de la aplicación.

4.2) Principios básicos de diseño Orientado a Objetos

Introducción a los principios SOLID

SOLID es una palabra que debes conocer como uno de los fundamentos de la arquitectura y desarrollo de software.

SOLID es el acrónimo que acuñó Michael Feathers, basándose en los principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su paper "Design Principles and Design Patterns". Ocho años más tarde, el tío Bob siguió compendiando consejos y buenas prácticas de desarrollo y se convirtió en el padre del código limpio con su célebre libro Clean Code.

Los 5 principios SOLID de diseño de aplicaciones de software son:

S – Single Responsibility Principle (SRP)
O – Open/Closed Principle (OCP)
L – Liskov Substitution Principle (LSP)
I – Interface Segregation Principle (ISP)
D – Dependency Inversion Principle (DIP)

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

· Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
· Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
· Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

Acoplamiento

El acoplamiento se refiere al grado de interdependencia que tienen dos unidades de software entre sí, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etc.
Si dos unidades de software son completamente independientes la una de la otra, decimos que están desacopladas.

Cohesión

La cohesión de software es el grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado. Se refiere a la forma en que podemos agrupar diversas unidades de software para crear una unidad mayor.

S - Principio de Responsabilidad Única

"A class should have one, and only one, reason to change."

La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio "una clase debería tener una, y solo una, razón para cambiar". Es esto, precisamente, "razón para cambiar", lo que Robert C. Martin identifica como "responsabilidad".

· El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

O- Principio de Abierto/Cerrado

"You should be able to extend a classes behavior, without modifying it."

· El segundo principio de SOLID lo formuló Ber-

trand Meyer en 1988 en su libro "Object Oriented Software Construction" y dice: "Deberías ser capaz de extender el comportamiento de una clase, sin modificarla". En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.
· En su blog Robert C. Martin defendió este principio que a priori puede parecer una paradoja.
· Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks.

L - Principio de Sustitución de Liskov

"Derived classes must be substitutable for their base classes."

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que "las clases derivadas deben poder sustituirse por sus clases base".

· Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.
· Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.

I - Principio de Segregación de la Interfaz

"Make fine grained interfaces that are client specific."

· En el cuarto principio de SOLID, el tío Bob sugiere: "Haz interfaces que sean específicas para un tipo de cliente", es decir, para una finalidad concreta.

· En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.

D - Principio de Inversión de Dependencias

"Depend on abstractions, not on concretions."

Llegamos al último principio: "Depende de abstracciones, no de clases concretas".
Así, Robert C. Martin recomienda:

· Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.

· Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

5.1) Pruebas Unitarias en Java

Prueban una funcionalidad única y se basan en el principio de responsabilidad única (la S de los principios de diseño S.O.L.I.D.)

Las pruebas unitarias son una forma de probar el buen funcionamiento de un modulo o una parte de sistema con el fin de asegurar el correcto funcionamiento de todos los modelos por separado y evitar si errores futuros en el momento de la integración de todas sus partes.

Características de las pruebas unitarias

Los tests unitarios prueban las funcionalidades implementadas en el SUT (System Under Test). Para los desarrolladores Java, el SUT será la clase Java.

Los tests unitarios deben cumplir las siguientes características:

Principio F.I.R.S.T.
· Fast: Rápida ejecución.
· Isolated: Independiente de otros test.
· Repeatable: Se puede repetir en el tiempo.
· Self-Validating: Cada test debe poder validar si es correcto o no a sí mismo.
· Timely: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación (TDD: Test-driven development), pero es lo suyo para centrarnos en lo que realmente se desea implementar.
· Además podemos añadir estos dos puntos más:
· Sólo pruebas los métodos públicos de cada clase.
· No se debe hacer uso de las dependencias de la clase a probar. Esto quizás es discutible porque en algunos casos donde la dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
· Un test no debe implementar ninguna lógica de negocio (nada de if...else...for...etc)
· Los tests unitarios tienen la siguiente estructura:
· Preparación de datos de entrada.
· Ejecución del test.
· Comprobación del test (assert). No debe haber más de 1 assert en cada test.

Ventajas y limitaciones

Ventajas de JUnit

· Es gratuito, open source (código abierto) y puede ser utilizado en desarrollos comerciales.

· Se encuentra actualmente bien mantenido y en constante mejora.

· Reglas de diseño de JUnit pueden crear una biblioteca modularizada de casos de pruebas para cada clase bajo construcción

· Permite un desarrollo de código mas rápido y de mayor calidad.

· Comprueba resultados operativos propios y para proveer una retroalimentación puntual.

· Las pruebas son escritas utilizando Java

Limitaciones o Desventajas

· Reglas de diseño JUnit tienden a dirigir a un numero masivo de pruebas de clases

· El código de las pruebas no se aislara de los datos de pruebas

· JUnit no hace pruebas unitarias en interfaces Swing, JSPs, HTML.

· Las pruebas de JUnit no pueden reemplazar las pruebas funcionales.

· Las pruebas unitarias no pueden ser código Java del lado del servidor.

5.2) Introducción a JUnit

JUnit es un framework Java para implementar test en Java.

Se basa en anotaciones:

- **@Test:** indica que el método que la contiene es un test: expected y timeout.
- **@Before:** ejecuta el método que la contiene justo antes de cada test.
- **@After:** ejecuta el método que la contiene justo después de cada test.
- **@BeforeClass:** ejecuta el método (estático) que la contiene justo antes del **primer test.**
- **@AfterClass:** ejecuta el método (estático) que la contiene justo después del **último test.**
- **@Ignore:** evita la ejecución del tests. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.

Las condiciones de aceptación del test se implementa con los asserts. Los más comunes son los siguientes:

- **assertTrue/assertFalse** (condición a testear): Comprueba que la condición es cierta o falsa.
- **assertEquals/assertNotEquals** (valor esperado, valor obtenido). Es importante el orden de **los valores esperado y obtenido.**
- **assertNull/assertNotNull (object):** Comprueba que el objeto obtenido es nulo o no.
- **assertSame/assertNotSame(object1, object2):** Comprueba si dos objetos son iguales o no.
- **fail():** Fuerza que el test termine con fallo. Se puede indicar un mensaje.

Caso de Prueba

Para realizar un test automático de prueba, debemos realizar una clase o un trozo de código que se encargue de hacer new de las clases que queremos probar y que se encargue de llamar a los métodos de dichas clases, pasando parámetros concretos y viendo que el resultado es el esperado. Es posible que dicho resultado nos venga por el return del método, o es posible que ese método no devuelva nada, pero haga algo en otro sitio y es ahí donde debemos ir a verificar si se ha hecho. Por ejemplo, un método suma() al que se pasan dos parámetros, devuelve un resultado, este método es fácil de testear.

Un método insertaEnBaseDeDatos() al que se pasan varios parámetros, no devuelve nada (quizás un true o false si la operación ha tenido éxito), pero nuestro programa de pruebas tendrá que ir a la base de datos y comprobar que se ha hecho la inserción.

Se utilizara una clase Suma.java y otra clase Resta.java. Ambas tienen dos métodos, la primera getSuma() e incrementa(), la segunda getDiferencia() y decrementa(). En los enlaces puedes ver el código completo de estas clases.

Para los programas de prueba podemos hacer una o más clases de prueba, pero lo normal es hacer una clase de prueba por cada clase a probar o bien, una clase de prueba por cada conjunto de pruebas que esté relacionado de alguna manera.

Tenemos dos clases; Suma y Resta, así que haremos dos clases de prueba TestSuma y TestResta.

```
import junit.framework.TestCase

public class TestSuma extends TestCase {

}
```

Ahora tenemos que hacer los métodos de **test.** Cada método probará alguna cosa de la clase. **JUnit** 3.8.1 requiere que estos métodos empiecen por test***. En el caso de **TestSuma,** vamos a hacer dos métodos de **test,** de forma que cada uno pruebe uno de los métodos de la clase Suma.

Por ejemplo, para probar el método getSuma() de la clase Suma, vamos a hacer un método de prueba testGetSuma(). Este método sólo tiene que instanciar la clase Suma, llamar al método getSuma() con algunos parámetros y comprobar que el resultado es el esperado.

```
import junit.framework.TestCase

public class TestSuma extends TestCase {

   Suma suma = new Suma();

   double resultado = suma.getSuma(1.0,
1.0);

   // Aqui se debe comprobar resultado

}
```

Sumaremos 1 más 1 donde conocemos el resultado; 2.

¿Cómo comprobamos ahora que el resultado es el esperado?

Al heredar de TestCase, tenemos disponibles de esta clase padre un montón de métodos assert***. Esto métodos son los que nos permiten comprobar que el resultado obtenido es igual al esperado. Uno de los métodos más usados es assertEquals(), que en general admite dos parámetros: El primero es el valor esperado y el segundo el valor que hemos obtenido. Así, por encima, la comprobación del resultado podría ser tan simple como esto.

```
import junit.framework.TestCase

public class TestSuma extends TestCase {

   Suma suma = new Suma();

   double resultado = suma.getSuma(1.0,
1.0);

   assertEquals(2.0, resultado);

}
```

Si realizamos la ejecución de los tests obtendremos el siguiente resultado:

```
C:\> set CLASSPATH=path\mi.jar;path\junit.jar
C:\> java junit.textui.TestRunner com.chuidiang.
ejemplos.junit38.TestResta
..
Time: 0,006

OK (2 tests)
```

Integración de JUnit en Eclipse

Considerando que en eclipse ya tenemos creado nuestro proyecto con la clase Suma.java del ejemplo anterior. En el directorio donde queramos crear el test, damos al botón derecho del ratón para obtener el menú y elegimos new -> JUnit test case. Si no hemos hecho todavía ningún test, eclipse nos preguntará la versión de JUnit que deseamos usar. Elegimos la que más nos guste y obtenemos la siguiente ventana

Debemos comenzar el llenado de campos necesarios para comenzar las pruebas:

· En primer lugar, aparece la versión de JUnit que vamos a usar.

· El directorio de fuentes donde queremos dejar el test. Este sale relleno por defecto en el lugar donde hemos hecho el new -> JUnit test case.

· El paquete para el test de JUnit. Podemos poner lo que queramos, pero en general es buena idea poner el mismo paquete que la clase que queremos probar. De esta forma, los Test quedarán organizados de forma similar a las clases.

Quizás, para que los test no queden mezclados con las clases, se puede añadir un subpaquete test.

· El nombre de la clase de test que vamos a crear. Podemos poner cualquier nombre, pero suele ser aconsejable usar algún criterio para reconocer los test fácilmente. Algunas herramientas consideran que son test aquellas clases que comiencen por Test***, por lo que es un criterio bastante aceptado en general.

· Marcamos si queremos que la clase de test tenga constructor, método setUp() y/o método tearDown(). setUp() es un método que se ejecutará antes de cada uno de nuestros test, por lo que nos servirá para incializar lo que queramos antes del test. Y tearDown() es un método que se ejecutará después de ejecutar cada uno de nuestros tests, por lo que servirá para liberar recursos o borrar ficheros u otros rastros que haya dejado el test.

· Podemos poner la clase de la que vamos a hacer test. Si lo hacemos así, en el siguiente paso de la creación tendremos más ayuda.

Al haber seleccionado la clase que queremos testear (Suma.java) en el paso anterior, nos aparece aquí la clase con todos sus métodos. Podemos marcar aquellos de los que prevemos hacer test. Pulsamos Finish y veremos que se genera la siguiente clase de test:

```
import junit.framework.TestCase;


public class TestDeSuma extends TestCase {

  protected void setUp() throws Exception {

    super.setUp();

  }

  protected void tearDown() throws Exception {

    super.tearDown();

  }


  public final void testGetSuma() {

    fail("Not yet implemented"); // TODO

  }


  public final void testIncrementa() {

    fail("Not yet implemented"); // TODO

  }

}
```

Vemos que se han creado los métodos setUp() y tearDown(), según habíamos marcado. No se ha generado constructor, ya que no lo habíamos marcado. Y se han generado un método de test para cada uno de los métodos de nuestra clase Suma que habíamos marcado. Estos métodos llaman a fail(), que es un método de la clase padre TestCase que provoca un fallo en el test. El fallo, obviamente, es que el test todavía no está implementado.

Con esto, podemos símplemente rellenar el código de los métodos de test.

5.3) Utilización de Fixtures en las unidades de prueba

Utilización de objetos simulados (mocks)

Cuando hablamos de unit tests debemos hablar de "Mocking", este es una de las habilidades principales que debemos tener a la hora de hacer tests en nuestras aplicaciones.

Al momento de desarrollar aplicaciones las dividimos en capas, web, negocio y datos, entonces al escribir tests unitarios en una capa, no queremos preocuparnos por otra, así que la simulamos, a este proceso se le conoce como Mocking.

Es posible crear mocks manualmente, pero ya existen frameworks que pueden hacerlo por nosotros, estos permiten crear objetos mock en tiempo de ejecución y definir su comportamiento.

Un objeto mock es un proveedor de datos que cuando se ejecuta la aplicación el componente se conectará a una base de datos y proveerá datos reales, pero cuando se ejecuta un test unitario lo que buscamos es aislarlo y para esto necesitamos un objeto mock que simulará la fuente de datos, esto asegurará que las condiciones de prueba sean siempre las misma.

Suites de pruebas

Mediante las suites podemos asignar métodos de prueba a grupos, y así poder ejecutar todas nuestras pruebas (o un grupo de ellas) de forma conjunta.

Para crear la suite con JUnit simplemente deberemos crear una nueva clase Java, y anotarla como se muestra a continuación:

```
import org.junit.*;

import org.junit.runner.*;

import org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@SuiteClasses( { SumaTest.class, SumaBOTest.class })

public class MiSuite {}
```

La anotación @RunWith simplemente hace referencia a la clase de JUnit que se encarga de ejecutar las suites. Por otro lado, la anotación SuiteClasses sirve para poner entre llaves, y separadas por comas, todas las clases de prueba que queramos incorporar a la suite. De esta forma, la clase, como se ve, puede quedar vacía perfectamente. Luego el compilador, al procesar las anotaciones, ya se encarga de construir la suite correspondiente con las clases de prueba.

El Desarrollo Dirigido por Test (TDD)

TDD son las siglas de Test Driven Development. Es un proceso de desarrollo que consiste en codificar pruebas, desarrollar y refactorizar de forma continua el código construido.

La idea principal de esta metodología es realizar de forma inicial las pruebas unitarias para el código que tenemos que implementar. Es decir, primero codificamos la prueba y, posteriormente, se desarrolla la lógica de negocio.

Es una visión algo simplificada de lo que supone, ya que bajo mi punto de vista también nos aporta una visión más amplia de lo que vamos a desarrollar. Y en cierta manera nos ayuda a diseñar mejor nuestro sistema (al menos, siempre ha sido así desde mi experiencia).

· Para que un test unitario sea útil y esta metodología tenga éxito, previamente a comenzar a codificar, necesitamos cumplir los siguientes puntos:

· Tener bien definidos los requisitos de la función

a realizar. Sin una definición de requisitos no podemos comenzar a codificar. Debemos saber qué se quiere y qué posibles implicaciones puede tener en el código a desarrollar.

· Criterios de aceptación, contemplando todos los casos posibles, tanto exitosos como de error. Imaginemos un sistema de gestión de alta de fichas de jugadores. ¿Qué posibles criterios de aceptación podríamos tener?:

· Si el jugador se da de alta correctamente debe de devolver un mensaje satisfactorio como "El jugador con ID "X"* se ha dado de alta correctamente"*.

· Si se encuentra un jugador con el ID duplicado debe devolver un mensaje de error indicando "El jugador con ID "X"* no ha podido ser dado de alta debido a que ya existe otro jugador con el mismo ID"*.

· Si alguno de los campos queda vacío, debe devolver un mensaje de validación indicando qué campo es obligatorio o qué error de formato es el causante del problema.

· Cómo vamos a diseñar la prueba. Para realizar un buen test unitario debemos ceñirnos únicamente a testear la lógica de negocio que queremos implementar, abstrayéndonos en cierto modo de otras capas o servicios que puedan interactuar con nuestra lógica, simulando el resultado de dichas interacciones (Mocks). Aquí siempre hay diferentes perspectivas, con sus ventajas y desventajas, aunque en mi opinión al final debe ser el propio desarrollador quien debe decidir la opción con la se encuentre más cómodo y le aporte mayor información y eficiencia, tanto a nivel técnico como a nivel de concepto (aunque eso ya entra dentro de otro tipo de discusión).

· Qué queremos probar. El ejemplo expuesto en el punto 2, nos da pistas sobre qué deberíamos probar antes de codificar. Cada casuística para cada criterio de aceptación debería llevar su prueba asociada.

· Por ejemplo, si en el caso de "error por validación" es por un campo obligatorio, deberíamos hacer una prueba para este caso. Si es por un "error de validación de formato", deberíamos hacer una prueba para este otro caso.

·¿Cuántos test son necesarios? Tantos como casos nos encontremos. De esta manera aseguramos que nuestra cobertura de pruebas es lo suficientemente fuerte como para asegurar el correcto funcionamiento del código desarrollado.

ENLACES Y BIBLIOGRAFÍA

Unidad 1

Correa U, Guillermo. Desarrollo de Algoritmos y sus aplicaciones. Editorial Mc-Graw Hill
Farrell Joyce. Introducción a la programación. Editorial Thomson

Unidad 2

“5 mejores IDE para programadores y desarrolladores Java”
https://javadesdecero.es/fundamentos/mejores-ide-para-programadores-java/

Unidad 3

“PARADIGMA ORIENTADO A OBJETOS. FUNDAMENTOS Y ORIGEN DE JAVA”
http://dis.um.es/~lopezquesada/documentos/IES_1415/IAW/curso/UT3/ActividadesAlumnos/java7/paginas/pag1.html

Unidad 3

“Java Polimorfismo, Herencia y simplicidad”
https://www.arquitecturajava.com/java-polimorfismo-herencia-y-simplicidad/

Unidad 3

“Introducción a los Test Unitarios (Unit Testing)“
http://javadesde0.com/introduccion-a-los-test-unitarios-unit-testing/

– Una vez realizado todos los ejercicios que sean necesarios para la investigación, se discuten los resultados con los participantes con el fin de que den su parecer sobre la orgánica y la lógica del contenido. Existe una gran posibilidad de que surja información que no estaba contemplada, o de que sobre información que desde ahora se considere innecesaria.

– Si se realiza un Card Sorting abierto, las tarjetas deberán ser llenadas en la sesión con ayuda de quien lo dirige. Se debe guiar para que se cumplan los puntos anteriores. Por sobre todo recordar que debe ser una guía, no una dictadura

Otras consideraciones importantes:

– Llenar las tarjetas con poco texto, idealmente palabras que describan acciones, tareas, opciones y contenido.

– En sitios muy grandes, lo mejor es trabajar con categorías e ignorar cierto contenido. Hay cosas que están establecidas y no requieren de la opinión del usuario para ser diseñadas o manejadas dentro del contenido.

–Comenzar con un ensayo, estar dispuestos a que los primeros ejercicios no serán de mucha ayuda. Intentar cosas nuevas, corregir y aumentar las tarjetas si es necesario.

– Realizar el primer Card Sorting siempre con el equipo interno de la empresa, con las personas que crean y manejan el contenido, con los directores y empleados. Luego realizarlo con el usuario en grupos de no más de 10 personas. Repetir toda las veces que sea necesario.

– Finalmente, no confiar en que los resultados del Card Sorting son en realidad la directriz que condicionará el diseño del sitio web. Es sólo un método dentro de la investigación que servirá de puente entre el contexto, el contenido y el usuario.

Jacobson (ed.) (2000). "Information Design"
Morville, Rosenfeld. (2007). "Information architecture for the world wide web"
Nielsen, Jacob (2007). "Usabilidad: prioridad en el diseño web"
Spencer, Donna. (2010). "A practical guide to information architecture" Wurman, Richard Saul (1989). "Information Anxiety"
Garrett, Jesse James. (2010). "The elements of user experience" Spencer, Donna. (2009). "Card sorting: designing usable categories"

TALENTO DIGITAL
INTELIGENCIA HUMANA

AIEP

Instituto Profesional AIEP Spa.
Dirección Nacional de Educación Continua
Programa Talento Digital 2021