Contenido

Capítulo 1: Características generales del lenguaje

Para dominar los aspectos de la Programación Orientada a Objetos, primero es necesario conocer las características propias del lenguaje, que si bien no están asociadas necesariamente a conceptos propios de **OOP** (Oriented Object Programming) nos van a ayudar en la resolución de diferentes problemas.

Tipos de datos

Los primeros lenguajes de programación no usaban objetos, solo variables. Una variable es un espacio de la memoria a la que asignamos un contenido, que puede ser un valor numérico (sólo números, con su valor de cálculo) o de tipo carácter o cadena de caracteres (valor alfanumérico que constará sólo de texto o de texto mezclado con números).

Como ejemplo podemos definir una variable "a" que contenga 32 y esto lo escribimos como a = 32. Posteriormente podemos cambiar el valor de a y hacer que a almacene el número 78 (a = 78). O hacer "a" equivalente al valor de otra variable "b" así: a = b.

Tipo de datos primitivos

Toda información que se maneja en Java puede representarse por un objeto o por un tipo de dato básico o de tipo primitivo. Java soporta ocho tipos de datos primitivos, que son, a saber:

Tipo básico	Tamaño
byte	8 bits
short	16 bits
int	32 bits
long	64 bits
char	16 bits
float	32 bits
double	64 bits
boolean	***

En el caso del tipo primitivo *boolean los* dos únicos valores que puede asumir son true o false y a diferencia de otros lenguajes no existe una equivalencia entre estos valores y números enteros. En cuanto al tamaño en número de bits del tipo *boolean*, este dependerá de la máquina virtual.

En el caso de las cadenas de caracteres, no existe un dato primitivo asociado a ellas como en otros lenguajes, sino que Java trabaja con las cadenas de caracteres como si se tratara de un objeto, concretamente objetos de la clase *String.*[1] De esto se desprende que además de usar tipos primitivos, al asignar un valor a una variable, es posible asignar un objeto.

[1] Una cadena de caracteres es un objeto. El tipo *String* en Java nos permite crear objetos que contienen texto (palabras, frases, etc.). El texto debe ir siempre entre comillas. Muchas veces se cree erróneamente que el tipo *String* es un tipo primitivo por analogía con otros lenguajes donde *String* funciona como una variable elemental. En Java no es así.

Ahora bien, ¿en qué se diferencia una variable de un tipo primitivo de una variable de tipo objeto? En que las variables de tipos primitivos son entidades elementales: un número, un carácter, un valor verdadero o falso… mientras que los objetos son entidades complejas que pueden estar formadas por la agrupación de muchas variables y métodos.

En los programas en Java puede ser necesario tanto el uso de datos elementales como de datos complejos. Por eso en Java se usa el término "Tipos de datos" para englobar a aquello que ocupa un espacio de memoria y que puede ir tomando distintos valores o características durante la ejecución del programa.

Tipo de datos Objetos

Este tipo de dato lo vamos a analizar en detalle en la sección POO cuando veamos la forma en la que Java utiliza éste paradigma. Por el momento sólo mencionamos su existencia, y lo comparamos con los tipos de datos primitivos:

<table>
<tr><td rowspan="5">TIPOS DE DATOS EN JAVA</td><td>TIPOS PRIMITIVOS</td><td>Sin métodos, no son objetos, no necesitan una invocación para ser creados.</td><td>Ejemplo: byte – short – int – long – float – double – char – boolean</td></tr>
<tr><td rowspan="4">TIPOS OBJETOS</td><td rowspan="4">Con métodos, necesitan una invocación para ser creados.</td><td>Tipos de la Biblioteca estándar de Java: String – TreeSet – Scanner - ArrayList</td></tr>
<tr><td>Tipos definidos por el programador/usuario</td></tr>
<tr><td>Arrays: Serie de elementos de tipo vector o matriz. Se los considera una clase especial porque carecen de métodos.</td></tr>
<tr><td>Tipos de Envoltorio o wrapper: equivalentes a los tipos primitivos pero tratados como objetos : Byte – Short – Integer – Long – Float – Double – Character – Boolean</td></tr>
</table>

Valores por defecto de una variable

El valor por defecto que asume una variable cuando es inicializada en forma automática se conoce como valor por defecto o predeterminado y depende del tipo de variable. Esta inicialización por defecto corresponde sólo a las variables de tipo privado (ejemplo, los campos o atributos); en caso de tratarse de variables locales (dentro de un bloque) es necesario inicializarlas antes de ser utilizadas en alguna instrucción del programa.

Según el tipo de variable los valores que asume por defecto son:

1. Numéricos enteros (*byte, short, int, long*) : 0
2. Carácter (*char*): '/u0000'
3. Numéricos decimales (*float, double*): 0.0

4. Lógicos: false
5. Objetos: null

Los tipos no primitivos corresponden a objetos creados a partir de una clase y por ser objetos asume como valor por defecto null.

Es importante destacar la diferencia que existe entre realizar una operación de asignación entre variables del tipo objeto y variables de tipo primitivo. En el caso del tipo primitivo, por ejemplo un *int*, esta operación implica que el dato contenido en una variable se copia en otra. En el caso en que se trate de un tipo objeto debemos recordar que lo que es está copiando es una referencia al objeto, no el objeto. Por lo tanto, al asignar el valor de una variable de tipo objeto a otra lo que se está copiando es la referencia al objeto, no el objeto. De esta manera, ***no tenemos dos copias del objeto, sino un único objeto referenciado por dos variables.***

Conversiones de tipo

Java es un lenguaje fuertemente tipado, o sea estricto a la hora de asignar valores a las variables. De acuerdo con esta característica, el compilador sólo admite asignar a una variable un dato del tipo declarado en la variable, aunque, en ciertas circunstancias permite realizar conversiones para almacenar en una variable un dato de tipo diferente al declarado. En java es posible realizar conversiones entre todos los tipos básicos, con excepción de *boolean*. Las conversiones pueden realizarse de dos maneras: en forma implícita o explícita.

Conversiones implícitas

Las conversiones implícitas son las que se realizan en forma automática a través del compilador, antes de almacenarlo en una variable. Ejemplo: un dato de tipo *byte* es almacenado en la variable b y luego es convertido a *int* al asignarlo a la variable i:

```
int i;
byte b=30;
i=b;
```

Para que una conversión pueda realizarse en forma automática o sea en forma implícita, es necesario que el tipo de variable destino sea de tamaño igual o superior al tipo de origen, aunque como toda regla presenta excepciones:

- Cuando la variable destino es entera y el origen es decimal (*float o double*) la conversión no puede ser automática.
- Cuando la variable destino es char y el origen es numérico, independientemente del tipo específico, la conversión no puede ser automática.

```
int k=5, p;
short s=10;
char c=’ñ’;
float h;
```

```
p=c; // conversión implícita de char a int;
h=k; // conversión implícita de int a float;
h=s; // conversión implícita de short a int;
```

Ejemplos de conversiones implícitas que darían lugar a error:

```
int n;
long c=20;
float ft=2.4f;
char k;
byte s=4;

n=c; // error, conversión implícita de long a int;
k=s; // error, conversión implícita de byte a char;
n=ft; //error, conversión implícita de float a int;
```

Conversiones explícitas

En caso en que no se cumplan las condiciones para una conversión implícita, ésta puede realizar se en forma explícita utilizando la expresión:

***variable_destino=(tipo_destino)dato_origen*;**

Con esta expresión se le indica al compilador que convierta *dato_origen* a *tipo_destino* para que puede ser almacenado en la variable destino. A esta operación se la denomina casting o estrechamiento ya que al convertir un dato de un tipo en otro de tipo inferior se realiza una reducción, un estrechamiento que en algunos casos puede llevar a una pérdida de datos o precisión, pero que evitaría posibles errores de ejecución. Ejemplos:

```
char c;
byte k;
int p=400;
double d=34.6;

c=(char)d; // se elimina la parte decimal (truncado)
k=(byte)p;  //  se  produce  una  pérdida  de  datos,  pero  la
//conversión es posible;
```

En el caso de los objetos no es posible realizar conversiones, pero es posible asignar a un objeto de una clase a una variable de clase diferente. Esto será posible solamente en una relación de herencia entre clases.

Constantes

Una constante es una variable cuyo valor no puede ser modificado Para definir una constante en Java se utiliza la palabra reservada final, delante de la declaración del tipo, de acuerdo con la siguiente expresión:

```
final tipo nombre_constante=valor;
```

Ejemplo:

```
final double pi=3.1416;
```

Una constante se puede declarar en los mismos lugares que una variable: al principio de una clase o al interior de un método, en forma local.

Operadores

Los operadores son la herramienta del lenguaje que nos permite realizar cálculos, modificar los valores de las variables, y evaluar el contenido de estas.

En la siguiente tabla se muestran los operadores principales de Java.

Tipo	Subtipo	Operador	Descripción	Ejemplo
Aritméticos	Binarios	+	Suma	suma = numero1 + numero2
		-	Resta	resta = numero1 - numero2
		*	Multiplicación	multiplicacion = numero1 * numero2
		/	División	division = numero1 / numero2
		%	Resto	resto = numero1 % numero2
	Unarios	-	Negativo	-negativo
		++	Incremento	postIncremento++, ++preIncremento
		--	Decremento	postDecremento--, --preDecremento
	Asignación	=	Asignar	a=b
		+=	Sumar y asignar	a+=b
		-=	Restar y asignar	a-=b
		*=	Multiplicar y asignar	a*=b
		/=	Dividir y asignar	a/=b
		%=	Módulo y asignar	a%=b
Relacionales	Comparación	==	Igual	(a==b)
		!=	Diferente	(a!=b)
		>	Mayor	(a>b)
		>=	Mayor o igual	(a>=b)
		<	Menor	(a<b)
		<=	Menor o igual	(a<=b)
	Condicional	?:	Operador ternario	<condición>?<si_verdadero>:<si_falso>;
	Booleanos	&&	AND	((a<b)&&(c==d))
		||	OR	((a<b)||(c==d))
		!	NOT	(!(a==b))

Capítulo 2: POO

Una de las características principales de Java consiste en ser un lenguaje de programación orientado a objetos. En función de esto, se desprende que, para poder programar en Java, vamos a necesitar objetos. Al escribir un programa en un lenguaje orientado a objetos tratamos de modelar un problema del mundo real pensando en objetos que forman parte del problema y que se relacionan entre sí. Pero, ¿qué es objeto y una clase como conceptos fundamentales de la POO (Programación Orientada a Objetos)?

Un ***objeto*** es una entidad existente en la memoria del ordenador que tiene unas propiedades (atributos o datos sobre sí mismo almacenados por el objeto) y unas operaciones disponibles específicas (métodos).

Una ***clase*** es una abstracción que define un **tipo** de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener. Por lo tanto, una vez identificado (modelados) los objetos que van a formar parte de la solución, hay que construir las clases que van a ser la definición abstracta de esos objetos.

Clases

Supongamos que deseamos crear en Java una clase llamada Alumno. Deberíamos pensar qué atributos corresponden a ese objeto concreto que es un alumno y podríamos mencionar: su nombre, edad, cantidad de asistencias a la materia, la nota obtenida en el primer parcial y la nota obtenida en el segundo parcial, el estado del alumno al finalizar la cursada etc. Por otra parte, también sería necesario establecer los métodos que permitiría a la clase relacionarse con el mundo exterior, ya que la clase que estamos modelando tendrá la posibilidad de comunicarse con otras clases a través de mensajes.

Veamos primero cómo se define una clase en Java. El cuerpo de la clase, encerrado entre { y }, corresponde a la lista de atributos y métodos que constituyen la clase. No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

```
[public] class NombreClase {

      // Declaración de campos o atributos


      // Declaración de métodos

      }
```

Volviendo a nuestro ejemplo de la clase llamada Alumno (nótese que el nombre de la clase debe iniciarse siempre con la primera letra en mayúscula), la definición de la clase quedaría así:

```java
package ar.edu.unlam.basica1;

public class Alumno {

	// Declaración de campos o atributos

	private int edad;
	private int cantidadAsistencias;
	private String nombre = "";
	private Float notasParciales[];
	private Float notasFinales[];
	private int estadoDelAlumnoEnLaMateria;

	// Declaración de métodos

	public void crecer(){
		edad++;
	}

	public void setEdad(int valor) {
		edad = valor;
	}

}
```

Package: Identifica el paquete que va a contener la clase que estamos definiendo. Toda clase debe estar contenida en un paquete. La generación de paquetes permite organizar las clases de manera estructurada y jerárquica. Otra ventaja de localizar las clases dentro de paquetes es evitar conflictos de nombres, dado que el ***nombre cualificado de la clase*** diferencia una clase de otra.

Por ejemplo ar.edu.unlam.basica1.Alumno es una clase diferente a ar.edu.unlam.basica2.Alumno.

[public] class: Lo primero [public] es el ***modificador de acceso***, sobre el cual vamos a hablar en detalle más adelante en el curso, luego, la palabra reservada class, es la que utilizamos para crear una clase.

[private] int: Al igual que en la definición de la clase, vemos que a la hora de escribir los atributos y los métodos de la misma, aparece el modificador de acceso. Por el momento respetaremos los modificadores utilizados como ejemplo, y luego veremos el porqué. A continuación del modificador de acceso, aparece el tipo de dato (en este caso del atributo). El tipo de dato puede ser cualquiera de los vistos en el Capítulo 1.

public void `setEdad(int valor)`: Como vemos, en la declaración de un método aparece una vez más el modificador de acceso, el tipo de dato que devuelve (en este ejemplo void), el nombre de método y los parámetros que recibe.

Objetos

Una vez creada la clase, para poder empezar a usarla es necesario generar objetos a partir de ella. A esta acción se la denomina ***instanciar*** una clase. Las instancias u objetos de una clase se crean con el operador new, que crea la instancia (ejemplo), la almacena en memoria y devuelve una referencia a la misma que normalmente se guarda en una variable para posteriormente invocar a los métodos del objeto.

La instrucción para instanciar nuestra clase, sería entonces:

Alumno juan;

```
juan = new Alumno();

// O en su forma abreviada

 Alumno juan = new Alumno();
```

Como podemos ver, este fragmento de código se divide en dos partes. Por un lado, la declaración de una variable (`Alumno juan`) y por el otro, la generación de un espacio en la memoria, donde esa variable pueda operar (`new Alumno()`).

De esta definición, se desprenden varias cosas. La primera, como seguramente ya lo habrán notado, es que en la declaración de la variable *juan*, estamos utilizando exactamente la misma sintaxis que para la definición de una variable común. De hecho, *juan*, en este ejemplo, es una variable común. ¿Qué quiere decir esto? Si, efectivamente podemos inferir que cuando definimos una clase, estamos de alguna manera, definiendo un ***nuevo tipo de dato*** (en este ejemplo Alumno).

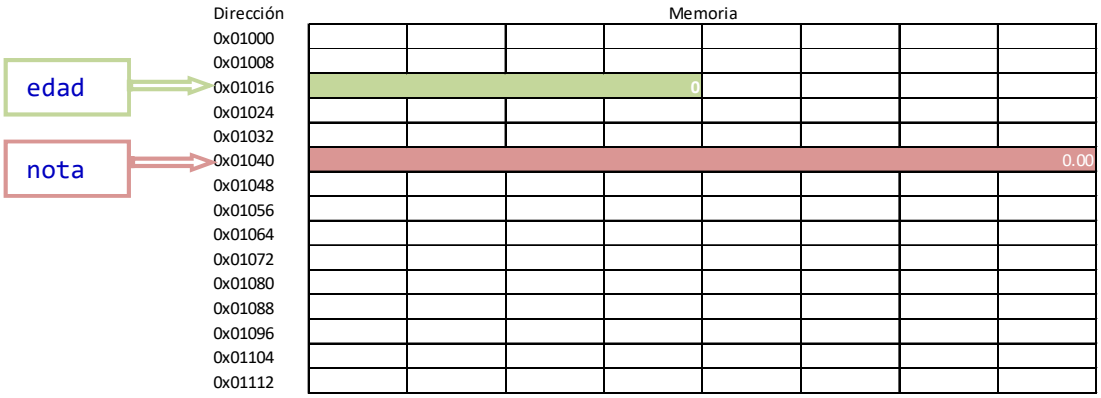
Sin embargo, si bien cuando trabajamos con objetos lo hacemos con variables de la misma manera que lo hacemos con cualquier otro tipo de dato, existe una particularidad, y es que no podemos utilizar la variable con sólo declararla, dado que obtendríamos una excepción llamada NullPointerException. ¿Por qué sucede esto?

Cuando declaramos una variable en otro lenguaje, por ejemplo C, digamos un entero, el compilador hace varias cosas:

1. Guarda el nombre de la variable, desde donde se va a acceder al valor de esta
2. Reserva un espacio en memoria donde se va a almacenar el valor de la variable
3. Establece la conexión (mapeo) entre el nombre de la variable y el valor de esta
4. Le asigna un valor por defecto. Para los enteros por ejemplo 0

```
int edad;

double nota;
```



Gracias a todas estas tareas transparentes, nosotros podemos automáticamente operar sobre la variable por ejemplo incrementando su valor, sin temor a recibir un mensaje de error.

Ahora bien, la creación de un objeto es algo un poco más complejo, dado que no sólo estamos almacenando un valor primitivo, sino que el objeto almacena diferentes variables de distintos tipos, junto con los métodos relacionados (adicionalmente ya veremos que un objeto podrá tomar distintas formas según el momento en que lo creemos). En consecuencia, con sólo declarar la variable, nosotros estamos realizando el punto 1 descripto arriba. Nos queda pendiente:

2. Reservar un espacio en memoria para almacenar dicha variable
3. Establecer la conexión entre el nombre de la variable que creamos y el espacio de memoria donde el valor de dicha variable se va a almacenar
4. [Asignar valores por defecto a los atributos del objeto que estamos creando]

Esta parte de la operación, la realizamos con la palabra reservada new.

Luego, una vez instanciada la clase, esto es creado el objeto, podemos operar, accediendo directamente a sus ***miembros*** (métodos o atributos) públicos.

```
Alumno juan = new Alumno();
juan.crecer();
juan.setEdad(25);
```

Constructores

Detengámonos unos minutos adicionales en el proceso de creación de objetos, de manera que podamos comprender de manera completa, qué es lo que sucede cuando utilizamos la palabra reservada ***new.*** Como pudimos ver en los ejemplos, luego del new, debemos incluir el nombre de la clase que estamos instanciando seguida de los paréntesis. De estos ejemplos observamos que esta sintaxis, más allá de esta nueva instrucción (new), es muy similar a cómo vamos a invocar a los distintos métodos en Java para enviar los mensajes a los objetos que estemos utilizando. Esta semejanza con la invocación de los métodos no es mera casualidad. De

hecho, cuando instanciamos una clase, lo que estamos haciendo es invocar a un método especial, llamado constructor de la clase.

Al crear un constructor, es necesario tener en cuenta una serie de reglas prácticas:

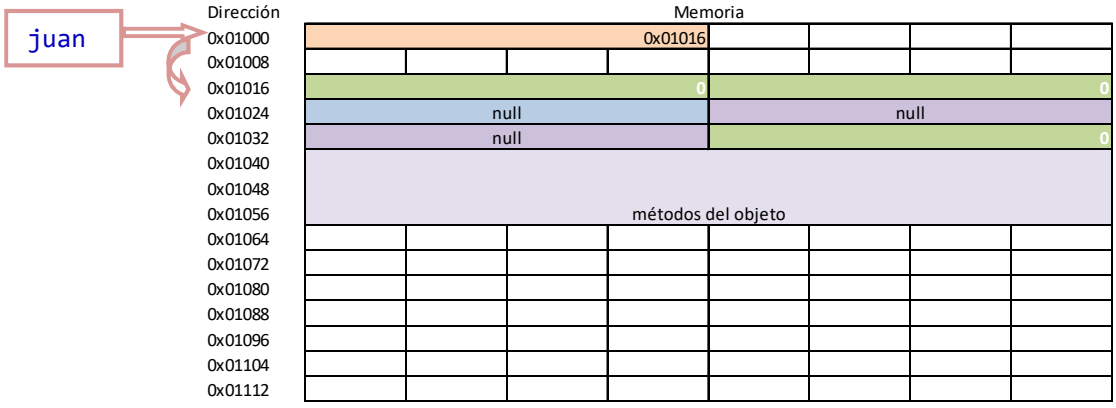
- El nombre del constructor debe ser el mismo que el de la clase.
- El constructor no debe tener tipo de devolución, ni siquiera *void*.
- Los constructores se pueden sobrecargar, lo que significa que una clase puede tener más de un constructor y en consecuencia distintas formas de inicializar sus atributos (se verá con más detalle en el próximo apartado)
- Toda clase debe tener, al menos, un constructor. En este sentido, si creamos una clase sin constructores, el compilador de Java añadirá un constructor a nuestra clase, denominado constructor por defecto, que será un constructor sin parámetros y sin código, necesario para que la clase pueda compilar y crear objetos. Su aspecto será el que a continuación se muestra:

```
public  NombreClase(){

}
```

Siempre que se defina una clase sin constructores, el compilador agregará uno por defecto sin parámetros y sin código (***sólo si la clase carece de constructor***). Cuando una clase presenta constructores, no se añade ningún constructor en forma implícita. Si se desea utilizar un constructor sin parámetros dentro de una clase que tiene definidos otros constructores (con parámetros) será necesario incluir un constructor sin parámetros como el que crea Java por defecto. Si no se crea este constructor sin parámetros y se intenta invocarlo, dará un error de compilación.

El constructor de una clase es una herramienta, a partir de la cual, reservamos la memoria necesaria para utilizar los objetos, opcionalmente establecemos el ***estado*** del nuevo objeto que estamos creando (esto sería, inicializar sus ***variables miembro***), y por último establecemos la relación entre el nombre de la variable que creamos, y la porción de memoria donde vamos a guardar el objeto (referencia al objeto).



Para comprender mejor lo anteriormente expuesto sería interesante pensar cuál sería el resultado de comparar las siguientes variables en una clase de prueba.

```
String nombre1 = new String("Hola");

String nombre2 = new String("Hola");

if(nombre2 == nombre2){
                System.out.println("Las variables son iguales");
        }
        else{
                System.out.println("Las variables son diferentes");
        }
```

Sobrecarga de métodos

Como se mencionó anteriormente, por más que no definamos ningún constructor para una clase, podemos crear objetos de esta, invocando un constructor por defecto.

```
Alumno juan = new Alumno();
```

El constructor por defecto es aquel constructor que el compilador provee para que se puedan crear objetos de cualquier clase, en el caso que no se haya definido ningún constructor dentro de la clase. Existe la posibilidad de declarar distintos constructores, dependiendo de las diferentes alternativas para instanciar la clase que estamos creando.

Esto es posible gracias a una característica que nos permite Java (la cual está presente en varios lenguajes), que se denomina sobrecarga de métodos. Gracias a la sobrecarga de métodos, es posible escribir métodos que tengan el mismo nombre, siendo su comportamiento diferente. La restricción que debemos cumplir a la hora de utilizar esta característica es que la firma de cada uno de estos métodos sobrecargados no puede ser igual.

Se conoce como la firma de un método al nombre del método junto con los parámetros que recibe. Esto significa que podemos escribir métodos con el mismo nombre, pero que se diferencien uno de otro por la cantidad de parámetros, o al menos el tipo de dato de alguno de sus parámetros.

La ventaja de la sobrecarga de métodos es que si tenemos varios métodos que van a realizar la misma operación no necesitamos asignarle un nombre distinto a cada uno. Un ejemplo concreto de sobrecarga de métodos lo constituye el método *valueOf()* de la clase *String,* que permite convertir tipos básicos en cadenas de caracteres, donde existe una versión del mismo método para cada uno de los tipos básicos de Java (todo dependerá del tipo de dato que se le pase)

Es importante tener en cuenta que una vez definido un constructor, ya no es posible utilizar el constructor por defecto, es decir que si para una clase determinada, declaramos un

único constructor, que reciba por ejemplo un parámetro, ya no podremos crear objetos de dicha clase con su constructor sin parámetros.

Encapsulamiento

Uno de los pilares fundamentales de la programación orientada a objetos es el encapsulamiento (los otros pilares son la herencia y el polimorfismo). Recordemos que una clase está compuesta por ***métodos*** que determinan el ***comportamiento*** de los objetos y ***atributos*** que representan las ***características de los objetos*** de la clase. El concepto de encapsulamiento se fundamenta en el hecho de mantener los atributos de los objetos como privados y proporcionar acceso a los mismos a través de los métodos públicos (métodos de acceso). Los métodos, al exponer su funcionalidad al exterior llevan el modificador de acceso *public* (en general en casi todos los casos), en tanto que los atributos suelen tener acceso privado, de modo que sólo pueden ser accesibles desde el interior de la clase.

Para lograr el encapsulamiento en forma apropiada es importante la correcta utilización de los ***modificadores de acceso***. Ya vimos algunos modificadores de acceso al declarar las clases y los miembros de estas (atributos y métodos); veamos todos los modificadores de acceso que tiene Java.

Private: Este modificador sólo es aplicable a los miembros de una clase (atributos y/o métodos), y no a la clase en sí. El declarar un miembro como private, hace que el uso del miembro esté restringido al ***interior de la clase***, no pudiendo ser utilizado desde fuera de la misma.

Default (Ninguno): La no utilización de un modificador de acceso, le da al elemento un acceso “por default”. Si un elemento (clase, método o atributo) tiene acceso por defecto, únicamente las clases de su mismo ***paquete*** tendrán acceso al mismo.

Protected: Se trata de un modificador de acceso empleado en la herencia, por lo que será estudiado con detenimiento más adelante. Por el momento digamos que un método o atributo definido como protected en una clase puede ser accedido por cualquier otra clase de su mismo ***paquete*** y además por cualquier ***subclase*** de ella, independiente del paquete en donde se encuentre. Una clase no puede ser protected, sólo sus miembros.

Public: El modificador public ofrece el máximo nivel de visibilidad. Un elemento (clase, método o atributo) public será visible desde ***cualquier clase***, independientemente del paquete en que se encuentre.

Nota: Los modificadores de acceso, se utilizan para los atributos de una clase, pero las variables locales (incluidos los parámetros de los métodos), no incluyen este tipo de clasificación.

Habiendo descripto los modificadores de acceso en Java, volvemos al tema del encapsulamiento. El mismo proporciona grandes beneficios a la hora de programar: por un lado

permite la protección de datos sensibles y por otro una gran facilidad y flexibilidad en el mantenimiento de las aplicaciones, veamos por qué.

Protección de datos

Imaginemos que vamos a trabajar con una nueva clase: Televisor que cuenta con un par de atributos de tipo int: volumen y canal. Supongamos que desarrollamos la aplicación sin tener en cuenta el principio de encapsulamiento de esta manera:

```
public class Televisor{

	public int volumen;

	public int canal;

		//métodos de la clase
}
```

El utilizar esta clase desde otro programa e intentar asignar los valores a los atributos, podría ocurrir algo así:

```
Televisor televisorSony= new Televisor();

televisorSony.canal=-10;
```

El ingresar un valor negativo como valor del atributo canal no tendría ningún sentido y por el contrario generaría resultados incoherentes en la ejecución de los métodos de una clase. A este hecho, llamado corrupción de los datos se lo puede evitar haciendo uso del principio de encapsulamiento por el cual se protegen los atributos del acceso directo del exterior declarándolos como privados y obligando a acceder a ellos por medio de un método de acceso.

Métodos getter() y setter()

En consecuencia, para asignar o recuperar el valor de un atributo particular, vamos a necesitar métodos que estén expuestos al exterior. En general vamos a llamar a estos métodos "get" + el nombre del atributo para recuperar su valor y "set" + el nombre del atributo para asignarle un valor. Esta nomenclatura hace que el código sea más legible, simplifica el trabajo en equipo, y además nos permitirá aprovechar el uso de las herramientas de soporte, como ser los IDEs.

```
	public class Televisor {

		private int volumen;
		private int canal;

		public int getVolumen(){
			return this.volumen;
		}

		public void setVolumen(int volumen){
```

```
                if(volumen>=0){
                        this.volumen = volumen;
                }
        }

    }

        public void setCanal(int canal){
                if (canal<0 && canal>200){
                        this.canal = canal;
                }
    }

                public int getCanal(){
                return this.canal;
    }

}
```

Como puede observarse el método de acceso que permite la escritura se denomina setNombre_Atributo, en tanto que el método que permite la lectura se denomina getNombre_Atributo. En el caso de los dos métodos de escritura (setCanal y setVolumen), existe un control del valor que el atributo puede asumir (en el caso del atributo volumen que sea mayor o igual a 0 y en el caso de canal que el valor se encuentre entre 1 y 199), esto evita que al almacenar el valor en el atributo el mismo se corrompa por asumir un valor incoherente.

En cuanto a la palabra ***this*** que aparece en los métodos de acceso a la lectura, esta palabra reservada se utiliza en el interior de una clase para invocar métodos y atributos propios de un objeto. Si bien su uso es redundante, ya que los métodos y atributos propios de una clase pueden ser llamados sin necesidad de utilizar ***this***, su uso será necesario para invocar a un miembro del propio objeto en caso que una variable local y un atributo posean el mismo nombre (para distinguir cuál es el atributo), como en el caso de los ejemplos previamente mencionados.

Tips de Programación: Una forma de generar getters y setters en forma automática desde Eclipse consiste en ir a la opción de menú *Source* (o bien desde el menú contextual que se despliega pulsando botón derecho sobre el espacio de edición del código) y seleccionar *Generate Setters and Getters*…

Facilidad en el mantenimiento de una clase

Si una vez creada una clase queremos cambiar un posible criterio sobre los valores que puede asumir un atributo (supongamos que a futuro existieran más de 199 canales, por ejemplo), bastaría con modificar el código de los métodos de acceso. De esta manera, los detalles de implementación quedan ocultos, manteniendo la interfaz, ya que el formato y la utilización del método no cambian.

Finalmente, para cerrar los conceptos de encapsulamiento, veremos en el ejemplo de la clase Televisor la sobrecarga en el método constructor:

```
public class Televisor {
	private int volumen;
	private int canal;

	public Televisor(){
		this.volumen = 10;
		this.canal = 2;
	}

	public Televisor(int volumen, int canal){
		this.volumen = volumen;
		this.canal = canal;
	}
	//Otros métodos
}
```

En el ejemplo anterior podemos observar una sobrecarga del método constructor, en el primer caso se trata del constructor por defecto, al cual no se le han pasado parámetros, y a través del cual se inicializan los atributos volumen y canal. En la segunda versión del método constructor se pasan parámetros de tal modo que ambos atributos asumirán el valor que se les ha pasado a través del parámetro.

Modelo final de la clase Televisor con la aplicación de todos los métodos:

```
package ar.edu.unlam.basica2;
public class Televisor {

	private int volumen;
	private int canal;

	public int getVolumen(){
		return this.volumen;
	}

	public void setVolumen(int volumen){
		if(volumen>=0){
			this.volumen = volumen;
		}
	}

}

	public void setCanal(int canal){
		if (canal<0 && canal>200){
			this.canal = canal;
		}
```

```
	}

		public int getCanal(){
		return this.canal;
	}


		public void cambiarVolumen(String modo){
			if (modo.equals("subir")){
				volumen++;
			}
			else{
				volumen--;
			}
	}

		public void cambiarCanal(String modo){
			if (modo.equals("subir")){
				canal++;
			}
			else{
				canal--;
	}
}
```

Es importante considerar que al comparar el parámetro modo, que es de tipo *String* contra la cadena de caracteres "subir" debe emplearse el método ***equals*** y no el operador== dado que en este último caso sólo se podrían comparar dos cadenas de caracteres. Y en el caso del parámetro modo, el mismo es un objeto de la clase *String* y como objeto, no guarda el valor de la cadena, sino ***una referencia a la posición de la memoria donde se almacena dicho contenido (la cadena guardada en el parámetro modo) y no el contenido mismo.***

Javabeans

En muchos tipos de aplicaciones puede resultar útil crear clases cuya única finalidad sea encapsular datos dentro de la misma. Esas clases se denominan ***JavaBeans*** y los datos que encapsula están asociados a una entidad (información de un empleado, de un libro, de un producto) y además de campos y constructores solamente dispone de métodos setter y getter.4

El siguiente código corresponde a una clase de esas características:

```
	public class Empleado {

		private String nombre;
		private String dni;
		public Empleado(String nombre,String dni)    {
			this.nombre = nombre;
			this.dni = dni;
```

```
        }

        public void setNombre(String n){
              nombre = n;
        }


        public String getNombre(){
              return nombre;
        }

        public void setDni(String d){
              dni = d;
        }


        public String getDni(){
              return dni;
        }

}
```

Herencia

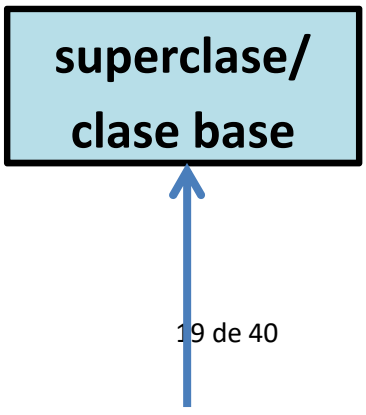
La herencia constituye uno de los pilares más más importantes y potentes de la POO.

Concepto de herencia: Capacidad de crear clases que adquieran de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

La herencia permite la **reutilización de código**, al evitar tener que reescribir todos los métodos en la nueva clase y el **mantenimiento de las aplicaciones existentes**. En este caso, si tenemos una clase con determinada funcionalidad y necesitamos ampliar dicha funcionalidad, no es necesario modificar la clase existente, sino que podemos crear una clase que herede a la primera, aprovechando toda su funcionalidad y añadiendo la suya propia.

Nomenclatura y reglas

En POO a la clase que va a ser heredada se la llama superclase o clase base y a la que hereda se la denomina subclase o clase derivada. Gráficamente la herencia entre dos clases se representa con una flecha saliendo desde la subclase hacia la superclase, del siguiente modo:



subclase/ clase

Existen una serie de reglas básicas en Java, relacionadas con la herencia que hay que tenerlas en cuenta:

- En Java no está permitida la herencia múltiple, es decir, una subclase no puede heredar más de una clase.
- Si es posible la herencia multinivel, o sea, A puede ser heredada por B y C puede heredar B.
- Una clase puede ser heredada por varias clases.

La herencia entre dos clases establece una relación entre las mismas de tipo “es un”, lo que significa que un objeto de una subclase es también un objeto de la superclase. Así, vehículo es la superclase de Coche, por lo que Coche también es un Vehículo. De la misma forma Persona es la superclase de Alumno y esta a su vez de la superclase AlumnoUnlam por lo cual AlumnoUnlam “es un” Alumno y “es una” (en este caso por ser femenino el nombre de la clase) Persona.

Creación de herencia en Java

Veamos un ejemplo, para que sea más fácil la comprensión de este pilar de la POO. Sea la clase Cuenta y la subclase CuentaCorriente derivada de ella:

```
package ar.edu.unlam.basica2;

public class Cuenta{
	private Double saldo;

	public Cuenta(Double saldo){
		this.saldo = saldo;
	}

	public Cuenta(){
		this.saldo = 0.0D;
	}

	public void depositarDinero(Double importe){
		this.saldo+=importe;
	}

	public Double extraerDinero(Double importeARetirar){
```

```
		Double importeRetirado;

		if(importeARetirar<=this.saldo){
			saldo-=importeARetirar;
			importeRetirado = importeARetirar;
		}
		else{
			importeRetirado = 0.0D;
		}

		return importeRetirado;
	}

	public Double consultarSaldo(){
		return (getSaldo());
	}

	public Double getSaldo(){
		return this.saldo;
	}
}
```

Para definir que una clase va a heredar a otra clase se utiliza la palabra ***extends,*** seguida del nombre de la superclase en la cabecera de la declaración.

```
public class subclase extends superclase
{
	//código de la subclase
}
```

En nuestro ejemplo, la clase CuentaCorriente heredaría de la siguiente manera:

```
package ar.edu.unlam.basica2;

public class CuentaCorriente extends Cuenta {

	private Double sobregiro;

	public Double getSobregiro() {
		return sobregiro;
	}

	public CuentaCorriente(){
		sobregiro = 300.0;
	}

	public CuentaCorriente(Double saldoInicial){
		super(saldoInicial);
```

```
                sobregiro = 300.0;
        }

        public Double extraerDinero(Double importeARetirar){
                Double importeRetirado;

                if(importeARetirar<=super.getSaldo()){
                        importeRetirado =
super.extraerDinero(importeARetirar);
                }
                else if(importeARetirar<=(super.getSaldo() +
this.sobregiro)){
                        sobregiro -= (importeARetirar - super.getSaldo());
                        super.extraerDinero(super.getSaldo());
                        importeRetirado = importeARetirar;
                }
                else{
                        importeRetirado = 0.0;
                }
                return importeRetirado;
        }
}
```

Como se observa en el ejemplo, la nueva clase CuentaCorriente agrega un atributo (sobregiro) y métodos propios para completar su función (dos constructores; uno donde simplemente se inicializa al atributo sobregiro y otro constructor donde además de inicializar el atributo sobregiro se accede al valor de saldo, que es un atributo propio de la clase Cuenta, además de un método *get* y un método que permite evaluar si es posible retirar un determinado importe.

Todas las clases en Java heredan alguna clase, implícitamente todas heredan la clase **Object** (sin necesidad de especificarlo con extends). En la clase Object se encuentra el paquete java.lang y constituye el soporte básico para cualquier clase en Java. La clase Object, por lo tanto es la superclase de todas las clases de Java.

Aunque una subclase hereda todos los miembros de la superclase, incluidos los privados, no tiene acceso directo a éstos ya que son privados de la clase y solamente accesibles desde el interior de ésta (por el principio de encapsulamiento).

Para poder acceder a los atributos privados de una superclase pueden utilizarse los métodos *set* y *get* de la misma desde la subclase, al ser heredados por ésta. Aunque lo ideal a la hora de inicializar atributos es el uso de constructores. En Java, cada vez que se crea un objeto de una clase, antes de ejecutarse el constructor de dicha clase, se ejecuta primero el de su superclase, dado que Java añade como primera línea de código en todos los constructores de una clase la siguiente instrucción:

```
super();
```

que provoca la llamada al constructor sin parámetros de la superclase. Si en vez de llamar al constructor sin parámetros deseáramos invocar a un constructor de la superclase, se lo debería hacer en forma explícita, añadiendo como primera línea de código del constructor de la subclase la instrucción:

```
super(argumentos);
```

Los argumentos son los parámetros que utiliza el constructor de la superclase. De esta manera el constructor de la subclase puede pasarle al constructor de la superclase los datos necesarios para la inicialización de los atributos privados, que no son accesibles desde la subclase.

Volviendo nuevamente a nuestro ejemplo, comparemos cómo operan los dos constructores:

```
public CuentaCorriente(){
      sobregiro = 300.0;
}

public CuentaCorriente(Double saldoInicial){
      super(saldoInicial);
      sobregiro = 300.0;
```

El primer constructor simplemente se utiliza para inicializar el valor del atributo sobregiro, en tanto que el segundo constructor llama al constructor de la superclase Cuenta y le pasa el parámetro saldoInicial para que inicialice el saldo de la cuenta (atributo propio de la superclase). Veamos el código en la superclase:

```
public Cuenta(Double saldo){
            this.saldo = saldo;
      }
```

donde el constructor recibe el parámetro saldoInicial y lo asigna como valor al atributo privado saldo de la superclase Cuenta. Por tanto existe una forma de invocar a métodos y atributos propios de la superclase desde la clase derivada haciendo uso de la palabra reservada super, en tanto que los atributos propios de la clase derivada serán accedidos usando la palabra reservada *this*.

Métodos y atributos protegidos: Existe un modificador de acceso aplicable a atributos y métodos de una clase pensado para ser utilizado con el manejo de herencia: el modificador ***protected.*** Este modificador de acceso permite que un miembro de una clase (atributo o método) sea accesible desde cualquier subclase dependiente de ésta, independientemente de los paquetes en que esas clases se encuentren.

Clases finales: Si queremos evitar que una clase sea heredada por otra, debemos declararla como clase final, colocando el modificador final antes de class, de esta manera:

```
public final class ClaseA{
```

```
}
```

Si otra clase intenta heredar una clase final se producirá un error de compilación

```
Public class ClaseB extends ClaseA{
}
```

Sobreescritura de métodos

Cuando una clase hereda a otra puede suceder que el comportamiento de los métodos que hereda no se ajuste a las necesidades de la nueva clase. En este caso, la subclase puede reescribir el método heredado, lo que se conoce como sobreescritura de un método. Con una salvedad: cuando se sobreescribe el método de una subclase, ésta debe tener exactamente el mismo formato que el método de la superclase que sobreescribe. O sea, que deben llamarse igual, tener los mismos parámetros y el mismo tipo de devolución. Veamos qué sucede en nuestro ejemplo:

En a superclase Cuenta:

```
public Double extraerDinero(Double importeARetirar){
            Double importeRetirado;

            if(importeARetirar<=this.saldo){
                  saldo-=importeARetirar;
                  importeRetirado = importeARetirar;
            }
            else{
                  importeRetirado = 0.0D;
            }

            return importeRetirado;
      }
```

En la clase derivada CuentaCorriente:

```
public Double extraerDinero(Double importeARetirar){
            Double importeRetirado;

            if(importeARetirar<=super.getSaldo()){
                  importeRetirado =
super.extraerDinero(importeARetirar);
            }
            else if(importeARetirar<=(super.getSaldo() +
this.sobregiro)){
                  sobregiro -= (importeARetirar - super.getSaldo());
                  super.extraerDinero(super.getSaldo());
                  importeRetirado = importeARetirar;
            }
            else{
                  importeRetirado = 0.0;
            }
```

```
            return importeRetirado;
        }
    }
```

En caso que al sobreescribir un método de una subclase manteniendo el mismo nombre, pero modificando los parámetros, el nuevo método no sobreescribe el de la superclase, pero tampoco se produce un error de compilación, dado que nos encontramos ante un caso de sobrecarga de métodos: dos métodos con el mismo nombre y distintos parámetros.

El método sobreescrito puede tener un modificador de acceso menos restrictivo que el de la superclase. Por ejemplo, un método de la superclase puede ser protected y la versión sobreescrita de la subclase puede ser public (pero nunca uno más restrictivo).

Métodos y campos estáticos

Una variable *static* representa información en toda la clase (todos los objetos de la clase comparten el mismo dato). La declaración de una variable static comienza con la palabra clave *static*. Ahora bien, un método declarado como *static* no puede tener acceso a los miembros no *static* de una clase, ya que un método *static* puede llamarse aun cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, esta referencia *this* no puede usarse en un método *static*; debe referirse a un objeto específico de la clase, y a la hora de llamar a un método *static*, podría no haber objetos de su clase en la memoria. La referencia *this* se requiere para permitir a un método de una clase acceder a otros miembros no *static* de la misma clase.

La clase **Math** cuenta con métodos *static* para realizar cálculos matemáticos comunes; además, declara dos campos que representan constantes matemáticas de uso común: Math.PI y Math.E. La constante Math.PI (3.14159265358979323846) es la relación entre la circunferencia de un círculo y su diámetro. La constante Math.E (2.7182818284590452354) es el valor de la base para los logaritmos naturales (que se calculan con el método *static* Math log). Math.PI y Math.E se declaran con los modificadores *public*, *final* y *static*. Al hacerlos *public*, otros programadores pueden usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave *final* es constante; su valor no se puede modificar una vez que se inicializa el campo. Tanto PI como E se declaran *final,* ya que sus valores nunca cambian. Al hacer a estos campos *static*, se puede acceder a ellos a través del nombre de la clase **Math** y un separador punto (.), justo igual que con los métodos de la clase **Math**.

La sintaxis para incluir atributos o métodos estáticos en una clase es la siguiente:

```
static tipo campon;
static tipo métodoX(parámetros)
      {
            //codigo métodoX
      }
```

A diferencia de los atributos en general que suelen ser de tipo privados, los atributos estáticos suelen llevar el modificador de acceso *public(*o *protected* o ninguno) lo cual permite que

puedan ser accedidos por afuera de la clase. **Dado que el método estático no hace referencia a ningún objeto en particular, no puede hacer referencia a campos y métodos que si dependan de un objeto.**

A continuación se detalla el código de la clase Alumno donde puede observarse un atributo de tipo estático llamado CANTIDAD_DE_CLASES_BRINDADAS, inicializado en 0 y una constante de tipo estático llamada CANTIDAD_DIAS_DE_CLASE, inicializada en 20. Mientras el primero es de tipo *private*, la constante estática es de tipo *public*, por lo cual puede ser accedida en forma directa desde fuera de la clase Alumno. Luego puede observarse es el método habilitarDiaDeClase, de tipo estático y público que permite incrementar la variable CANTIDAD_DE_CLASES_BRINDADAS. Es importante observar que para llamar a la variable no se utiliza la palabra reservada *this* sino a través del nombre de la clase, por los motivos que se explicaron en párrafos anteriores.

```
public class Alumno {

	private static int CANTIDAD_DE_CLASES_BRINDADAS = 0;
	public static final int CANTIDAD_DIAS_DE_CLASES = 20;

public static void habilitarDiaDeClase(){
		Alumno.CANTIDAD_DE_CLASES_BRINDADAS++;
}
```

Capítulo 3: Estructuras de control

Decisiones

Es muy habitual, al escribir un programa en cualquier lenguaje de programación, tener que evaluar distintas alternativas para decidir qué decisión se debe tomar.

En Java contamos con dos construcciones para atender esta necesidad:

- If / Else
- Switch

if / else

```
if(condicion) {
      sentenciasParaResultadoVerdadero;
}
} else {
      sentenciasParaResultadoFalso;
}
```

Donde *condicion* es el resultado de una evaluación booleana. Es decir, cualquier expresión que dé como resultado un valor booleano. Esto es:

a. Variables de tipo boolean
b. El resultado de una comparación (==;!=; >; <; >=; <=)
c. El resultado de varias comparaciones unidas por un operador relacional booleano (&&; ||; !).

sentenciasParaResultadoVerdadero son el conjunto de instrucciones que se deben ejecutar para el caso que el resultado de la evaluación de la *condición* sea verdadero (true).

sentenciasParaResultadoFalso son el conjunto de instrucciones que se deben ejecutar para el caso que la evaluación de la *condición* sea falsa (false).

A modo de ejemplo podemos revisar el método setCanal de la clase Televisor, donde sabemos que sólo se aceptan aquellos canales entre 0 y 200.

```
public void setCanal(int canal){
      if (canal>0 && canal<200){
            this.canal = canal;
      }
      else{
            this.canal = 0;
      }
}
```

Es importante señalar que la cláusula else es opcional, dado que van a existir numerosas situaciones donde no sea necesario ejecutar instrucciones cuando la condición no se cumpla. Para el mismo ejemplo utilizado quizás sea preferible omitir la instrucción por el lado falso y directamente dejar el valor de canal que tenía el objeto antes de invocar al método setCanal.

```
public void setCanal(int canal){
	if (canal>0 && canal<200){
		this.canal = canal;
	}
}
```

if / else if

Existen situaciones donde quizás sea necesario anidar “condicionales”, este concepto, también conocido como “if anidados”, se trata de aquellos casos donde sea necesario evaluar una condición luego de haber evaluado una condición previa.

Tomemos como ejemplo la clase Cuenta, haciendo referencia a una cuenta bancaria y particularmente aquella conocida como CuentaCorriente. Recordemos que las cuentas corrientes tienen un valor de sobregiro para aquellos casos donde el importe que se desea retirar sea mayor al saldo actual de la cuenta. Tomando como base esta característica,el método extraerDinero puede tomar la siguiente forma:

```
public double extraerDinero(double importeARetirar){
		double importeRetirado;

		if(importeARetirar <= this.saldo){
			importeRetirado = importeARetirar;
			this.saldo -= importeARetirar;
		}
		else if(importeARetirar <= (this.saldo +
this.sobregiro)){
			sobregiro -= (importeARetirar - this.saldo);
			this.saldo = 0.0;
			importeRetirado = importeARetirar;
		}
		else{
			importeRetirado = 0.0;
		}
		return importeRetirado;
	}
}
```

Switch… case

El Switch es una construcción que nos permite tomar más de dos caminos a la hora de evaluar una expresión.

```
switch(expresion) {
 case VALOR1:
      sentencias;
      break;
 case VALOR2:
      sentencias;
 case VALORN:
      sentencias;
      break;
 default:
      sentencias;

}
```

Donde:

- *expresion* es lo que se desea evaluar. Puede ser el resultado de una operación o simplemente una variable. Es importante tener en cuenta que "*expresión*" siempre debe representar un valor de tipo byte, short, int o char
- *VALORX* representa cada uno de los valores posibles que puede tomar "*expresion*". Estos valores no son variables, y si bien se pueden utilizar "literales", se recomienda siempre utilizar constantes para representar los mismos.
- *sentencias* es el conjunto de instrucciones que se deben ejecutar en cada bloque.

Notar que de forma opcional se incluyó la palabra reservada break. Esto se debe a que, de no incluirla, el compilador, una vez que entre en un case determinado, seguirá ejecutando las sentencias de los "case" siguientes. La inclusión o no de esta sentencia dependerá del objetivo que se busque en cada caso.

Por último, hay que mencionar que la cláusula "default" es aquella en la que se ingresa cuando la "expresion" no coincida con ninguno de los "case" definidos previamente, o bien no se haya incluido un break en los case anteriores donde se pudiera haber ingresado.

Como ejemplo de esta construcción imagine el método

```
public String extraerDinero(char letra){
      String resultado;

      switch(letra) {
            case a:
            case e:
            case i:
            case o:
            case u:
                  resultado = "Vocal"
```

```
                    break;

            default:
                    resultado = “Consonante”
        }

        return resultado;
}
```

Iteraciones

For

Como todos sabemos la programación tiene como uno de sus objetivos principales automatizar operaciones repetitivas de forma que se simplifiquen las tareas. En ese sentido es el for la estructura de control por excelencia para lograr este objetivo.

La idea del for es poder ejecutar un conjunto de instrucciones una cantidad de veces determinada.

```
for(expresion_inicial; condicion_corte; expresión_incremento) {
      sentencias;
}
```

Donde:

expresion_inicial representa el estado inicial en donde comienza el ciclo.

condicion_corte es la condición que se debe cumplir para finalizar la iteración.

expresión_incremento es la operación que se realiza en cada ciclo.

Sentencias son el conjunto de instrucciones que se deben ejecutar en cada ciclo.

A modo de ejemplo podemos considerar el método potencia de la clase Calculadora:

```
public int potencia(int base, int exponente){
      int resultado = 1;
      for(int = 0; i<exponente; i++) {
            resultado *= base;
      }

      return resultado;
}
```

While

En términos generales el while cumple un objetivo similar al for, en el sentido que permite también ejecutar sentencias una cantidad de veces.

La diferencia principal entre for y while es que para este último no conocemos la cantidad de repeticiones que se van a ejecutar, y en general dentro de las sentencias a ejecutar existirá una condición que determine el fin del ciclo.

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

Donde:

expresion: Es la expresión que se evaluará cada vez que se cumpla un ciclo del while. Siempre que expresión dé como resultado true, se ejecutará un nuevo ciclo.

sentencias: Es el conjunto de instrucciones que se ejecutarán en cada iteración dentro del while. Es importante tener presente que casi siempre va a ser necesario que alguna de estas instrucciones, tengan un efecto sobre "expresion", de manera que transforme el resultado de su evaluación. Esto es así porque en caso contrario nunca terminaríamos de iterar dentro del ciclo while.

Para el caso del while utilicemos el ejemplo del ascensor, donde el método subir, debe hacerlo siempre y cuando el piso actual sea menor que el piso deseado:

```
public void subir(int pisoDeseado){

        while(pisoActual<pisoDeseado) {

                pisoActual++;

        }

}
```

Do… While

La única diferencia que existe entre el "do while" y el "while", es que la condición se evalúa luego de ejecutar la sentencias. Esto significa que en el "do while" siempre voy a ejecutar al menos una vez las instrucciones dentro del bloque.

En la gran mayoría de casos, seguramente podríamos utilizar uno u otro indistintamente, pero usaremos el "do while" solo en aquellos casos donde no tenga sentido evaluar las expresión de condición antes de ejecutar el primer ciclo.

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

Por ejemplo, imaginemos un método que tenga que tomar un número aleatorio par:

```
public int buscarUnPar(int pisoDeseado){
      int numeroAleatorio;
      do {
            numeroAleatorio = Math.random() * 10;
      } while(numeroAleatorio%2<>0);

      return numeroAleatorio;

}
```

Sentencias de salto

break; Rompe el bucle más cercano inmediatamente

continue; Continúa con la siguiente iteración del bucle, salteando las sentencias hasta el final del bloque

Si bien estas son sentencias válidas dentro del lenguaje de programación Java, su uso no se recomienda dado que resulta difícil de seguir la lógica desarrollada en el algoritmo utilizado para resolver el problema por el hecho de saltar de un lugar a otro. Sólo se avalará el uso de break para la estructura switch dado que es la única forma de lograr el funcionamiento deseado en dicha estructura.

Capítulo 4: Uso de clases existentes

Arrays Unidimensionales

En Java, un arreglo o *array* es un grupo de variables (llamadas elementos o componentes) que contienen valores, todos del mismo tipo. Los arreglos son objetos (pertenecen al grupo de tipos objeto), por lo que se consideran como tipos de referencia. Cada uno de los elementos del *array* tiene asignado un índice numérico según su posición, siendo 0 el índice del primero. La sintaxis para declarar un *array* es la siguiente:

```
int variable_array[]; o bien int[] vatiable_array;
```

Ejemplos:

```
int[ ] k;
String [ ] p;
char cads[ ];
```

Los arrays pueden declararse como atributos de una clase o como variables locales, al ser una variable de tipo objeto se inicializa implícitamente con el valor *null*. En caso en que el array se encuentre dimensionado o sea que se indique la cantidad de posiciones con que cuenta el array, todos sus elementos son inicializados al valor por defecto del tipo correspondiente, independientemente de que el array corresponda a un atributo o a una variable local. Para dimensionar un array se utiliza la siguiente expresión:

```
variable_array = new tipo [tamaño];
```

Ejemplos:

```
K=new int[5];
Cads=new char[10];
String [] nombres= new String [10];
```

Existe una forma de declarar, dimensionar e inicializar un array en una misma sentencia. La siguiente instrucción crea un array de tres enteros y los inicializa en los valores indicados entre llaves:

```
int [ ]nums= {10,20,30};
```

Todos los objetos array exponen un atributo público ***length*** que permite conocer el tamaño del array. Este atributo resulta muy útil al querer recorrer el array. En el próximo ejemplo se puede observar cómo se utiliza *length* para recorrer un array y cargarlo con números enteros pares consecutivos, empezando por el 0.

```
int[ ] nums=new int [10];
for (int i=0; i<nums.length; i++){
      nums[i]     = i*2;
```

```
}
```

Arrays Bidimensionales

Los arrays pueden tener dos dimensiones, de forma que se pueda representar una estructura de tipo “Matriz”.

variable_array = new tipo [cantidadFilas] [cantidadColumnas]

Podríamos por un minuto imaginarnos una planilla de cálculo, de forma que representemos gráfica y sencillamente un array bidimensional. De igual manera que una planilla, los arrays bidimensionales tienen filas y columnas, siendo que las filas se identifican por el primer índice y las columnas por el segundo.

Imaginémonos ahora que queremos realizar un mapa de las mesas de un restaurant. Sabiendo que lo que nos interesa conocer es si la mesa está libre u ocupada, podríamos desarrollar una matriz donde se identifique la posición de cada mesa y almacene true si la mesa está ocupada o false si la mesa está libre.

Desarrollemos la clase completa que da solución a este problema:

```
public class Restaurant {

	private boolean mesas[][];

	public Restaurant(int cantidadFilas, int cantidadColumnas) {
		mesas = new boolean[cantidadFilas][cantidadColumnas];
	}

	public void inicializarMapaDeMesas() {
		for(int i=0; i<mesas.length; i++) {
			for(int j=0; j<mesas.length; j++) {
				mesas[i][j] = false;
			}
		}
	}

	public boolean estaLaMesaOcupada(int fila, int columna) {
		return mesas[fila][columna];
	}

	public void ocuparMesa(int fila, int columna) {
		mesas[fila][columna] = true;
	}
}
```

Clases de Envoltorio o Wrapper

Para cada uno de los tipos primitivos Java proporciona una clase que lo representa. A esas clases se las conoce como clases de envoltorio, y se diferencian en la nomenclatura de los tipos primitivos ya que estos se escriben en minúscula, en tanto que los nombres de las clases de envoltorio se inician con una letra mayúscula (como por convención se escriben todos los nombres de una clase). Veámoslo en un ejemplo:

```
private int edad = 10;

private Integer edad = new Integer(10);
```

Un objeto es distinto a un tipo primitivo, aunque "porten" la misma información. Es importante tener siempre presente que los objetos en Java tienen un tipo de tratamiento y los tipos primitivos, otro. ¿Para qué tener esa aparente duplicidad entre tipos primitivos y tipos envoltorio? Esto es una cuestión propia de la concepción del lenguaje de programación. Un tipo primitivo es un dato elemental y carece de métodos, mientras que un objeto es una entidad compleja y dispone de métodos. Por otro lado, de acuerdo con la especificación de Java, es posible que necesitemos utilizar dentro de un programa un objeto que "porte" como contenido un número entero. Desde el momento en que sea necesario un objeto habremos de pensar en un envoltorio, por ejemplo, *Integer.* Inicialmente puede costar distinguir cuándo usar un tipo primitivo y cuándo un envoltorio en situaciones en las que ambos sean válidos.

Las clases de envoltorio o envolventes en Java son las siguientes:

- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Double
- java.lang.Integer
- java.lang.Float
- java.lang.Long
- java.lang.Short

Es importante tener presente que como todas las clases, éstas tendrán un comportamiento a través de sus métodos que nos ayudarán a resolver un problema determinado. Un ejemplo claro para las clases de envoltorio es el método que permite convertir un String en el tipo primitivo que representa. Por ejemplo:

```
int i = Integer.parseInt("1000");
```

String

Todo lenguaje de programación requiere contar con algún tipo de dato que permita manipular cadenas de caracteres. Históricamente, éste tratamiento, por ejemplo en lenguajes como “c” se realizó a través de arrays de elementos char (char cadenaDeCaracters[]) .

En Java, si bien se podría tranquilamente reproducir la misma metodología utilizada en “c”, contamos con una clase específica, String.

La ventaja que tiene el uso de esta clase es poder contar con un comportamiento específico para este tipo de dato, que nos da la posibilidad de realizar operaciones y manipular las variables de una forma totalmente versátil.

Se recomienda al lector indagar particularmente en los siguientes métodos de la clase String para conocer sus posibilidades:

- charAt(int index);
- equals();
- equalsIgnoreCase(String anotherString)
- length();
- substring(beginIndex, endIndex);
- substring(beginIndex);
- startsWith(prefix);
- split(String regex)
- toLowerCase();
- toUpperCase();
- trim();

Math

Esta clase nos ofrece el comportamiento relacionado a un gran número de operaciones matemáticas y es, por ende, la clase que debemos utilizar para realizar este tipo de operaciones. Es importante mencionar que la totalidad de métodos y atributos de esta clase son estáticos, es decir que para utilizarlos, en lugar de crear objetos de dicha clase, lo que se hace es invocar los mismos en su forma estática, Math.atributoOMétodo .

Principalmente se utilizan los siguientes elementos de la interfaz pública de Math:

- abs(int a);
- abs(double a);
- abs(float a);
- abs(long a);

- E
- exp(double a);
- PI
- pow(double a, double b)
- random()
- round(float a);
- round(double a);
- sqrt(double a);

System

La clase System contiene muchos campos y métodos de uso cotidiano, y que vamos a utilizar habitualmente. Esta clase no puede ser instanciada, es decir la forma de utilizar sus métodos y campos es de manera estática.

Los campos principales que contiene esta clase son:

- PrintStream err: Flujo de salida por error estándar
- PrintStream in: Flujo de entrada estándar
- PrintStream out: Flujo de salida estándar

Estos campos (objetos), los utilizaremos cada vez que queramos manipula algún ingreso a nuestro programa (a través de in), egreso o salida (a través de out) o informar la ocurrencia de un error (a través de err).

Como métodos principales podemos mencionar los siguientes:

- currentTimeMilis();
- gc();
- getenv();
- getProperties();
- setErr(PrintStream err);
- setIn(InputStream in);
- setOut(OutputStream out);

Scanner

La clase Scanner nos permite manipular de forma simple cualquier ingreso (input) que queramos realizar en nuestro programa. Por default vamos a utilizar siempre el flujo de entrada

estándar (System.in), de manera que podamos realizar cualquier ingreso de datos a través del teclado de la computadora.

El uso de este será de la forma:

```
Scanner teclado = new Scanner(System.in);
int numeroEnteroAIngresar = teclado.nextInt();
```

De forma similar podremos realizar el ingreso de cualquier tipo de dato primitivo:

- next();
- nextBoolean();
- nextByte();
- nextDouble();
- nextFloat();
- nextInt();
- nextLine();
- nextLong();
- nextShort();

Tipos Enumerados

Los tipos numerados constituyen una nueva característica incluida en la versión 5 de J2SE, que permite definir nuevos tipos de datos cuyos posibles valores están limitados a un determinado conjunto dado, o sea a un conjunto de valores constantes. En versiones anteriores de Java, la forma de definir un determinado conjunto de valores mediante la utilización de constantes, como se muestra en este ejemplo del estado final de un alumno después de una cursada:

```
public class Estado {

	public static final int INSCRIPTO = 0;
	public static final int CURSANDO = 1;
	public static final int PROMOCIONADO = 2;
	public static final int PENDIENTE_DE_RENDICION_DEL_FINAL = 3;
	public static final int APROBADO = 4;
	public static final int AUSENTE = 5;
	public static final int DESAPROBADO = 6;
}
```

De esta forma, para referirse a alguno de los valores definidos en las constantes anteriores se utilizaría la expresión:

Nombre_clase.CONSTANTE, por ejemplo Estado.APROBADO

Este sistema es inseguro, ya que nada impide que se asigne a través de un método setter un nuevo valor de estado a la clase. Gracias a la incorporación de los tipos numerados, es posible

definir un conjunto de valores que puede almacenar una variable de este tipo, generando un error de compilación cualquier intento de querer asignar un valor no definido en la enumeración. Un tipo numerado se define según el siguiente formato :

[public] enum Nombre_tipo {VALOR1,VALOR2,…}

siendo Nombre_tipo el nombre que se va a signar al tipo numerado y VALORN los posibles valores que puede asumir. La declaración de tipo numerado corresponde a las clases nunca puede estar incluida dentro de un método. Para el ejemplo anterior, el tipo numerado correspondiente sería el siguiente:

```
public enum Estado{INSCRIPTO, CURSANDO, PROMOCIONADO,
PENDIENTE_DE_RENDICION_DEL_FINAL, APROBADO, AUSENTE, DESAPROBADO};
```

Una enumeración es un tipo especial de clase que hereda de **java.lang.Enum**. A diferencia de las clases estándar, una clase de enumeración no permite el uso del operador *new* para la creación de objetos, dado que cada uno de los valores de la enumeración representan uno de los posibles objetos de la clase, los que fueron creados en forma implícita al declarar la enumeración. Además de los métodos heredados de la clase Enum, todas las enumeraciones disponen del método estático values(), que devuelve un array con todos los objetos de la clase.