Contenido

[PROGRAMANDO EN JAVA 2](#_Toc450137351)

[Clases 2](#_Toc450137352)

[Objetos 4](#_Toc450137353)

[Constructores 6](#_Toc450137354)

[Sobrecarga de métodos 7](#_Toc450137355)

[ENCAPSULAMIENTO 8](#_Toc450137356)

[Protección de datos 9](#_Toc450137357)

[Métodos getter() y setter() 9](#_Toc450137358)

[Facilidad en el mantenimiento de una clase 11](#_Toc450137359)

[Javabeans 12](#_Toc450137360)

[HERENCIA 13](#_Toc450137361)

[Nomenclatura y reglas 13](#_Toc450137362)

[Creación de herencia en Java 14](#_Toc450137363)

[Sobreescritura de métodos 18](#_Toc450137364)

[ANEXO 20](#_Toc450137365)

# PROGRAMANDO EN JAVA

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:

**package** ar.edu.unlam.basica2.clase2;

**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.basica2.clase2.Alumno es una clase diferente a ar.edu.unlam.basica2.clase3.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). Como podemos observar, en Java tenemos los mismos (mejor dicho casi los mismos) tipos primitivos que en C, diferenciándose cada uno, casi exclusivamente del tamaño que ocupa en la memoria.

|  |  |
| --- | --- |
| **Tipo primitivo** | **Tamaño** |
| byte | 8 bits |
| short | 16 bits |
| int | 32 bits |
| long | 64 bits |
| char | 16 bits |
| float | 32 bits |
| double | 64 bits |
| boolean | depende de la máquina virtual |

**public** **void** setEdad(int valor): Como vemos, la declaración de un método tiene una forma muy similar a la declaración de una función en C. Una vez más aparece 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, 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 C, por ejemplo un entero, el compilador hace varias cosas:

1. Guarda el nombre de la variable, desde donde se va a acceder al valor de la misma
2. Reserva una 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 la misma
4. Le asigna un valor por defecto. Para los enteros por ejemplo 0

**int** edad;

**double** nota;



nota edad

edad edad

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:

1. Reservar un espacio en memoria para almacenar dicha variable
2. 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
3. [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();

## 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 la que utilizábamos cuando invocábamos a una función en C, o 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).



juann edad

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 la misma, 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 ***{***

**publicint**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 equalsy 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:

**superclase/ clase base**

**subclase/ clase derivada**

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).

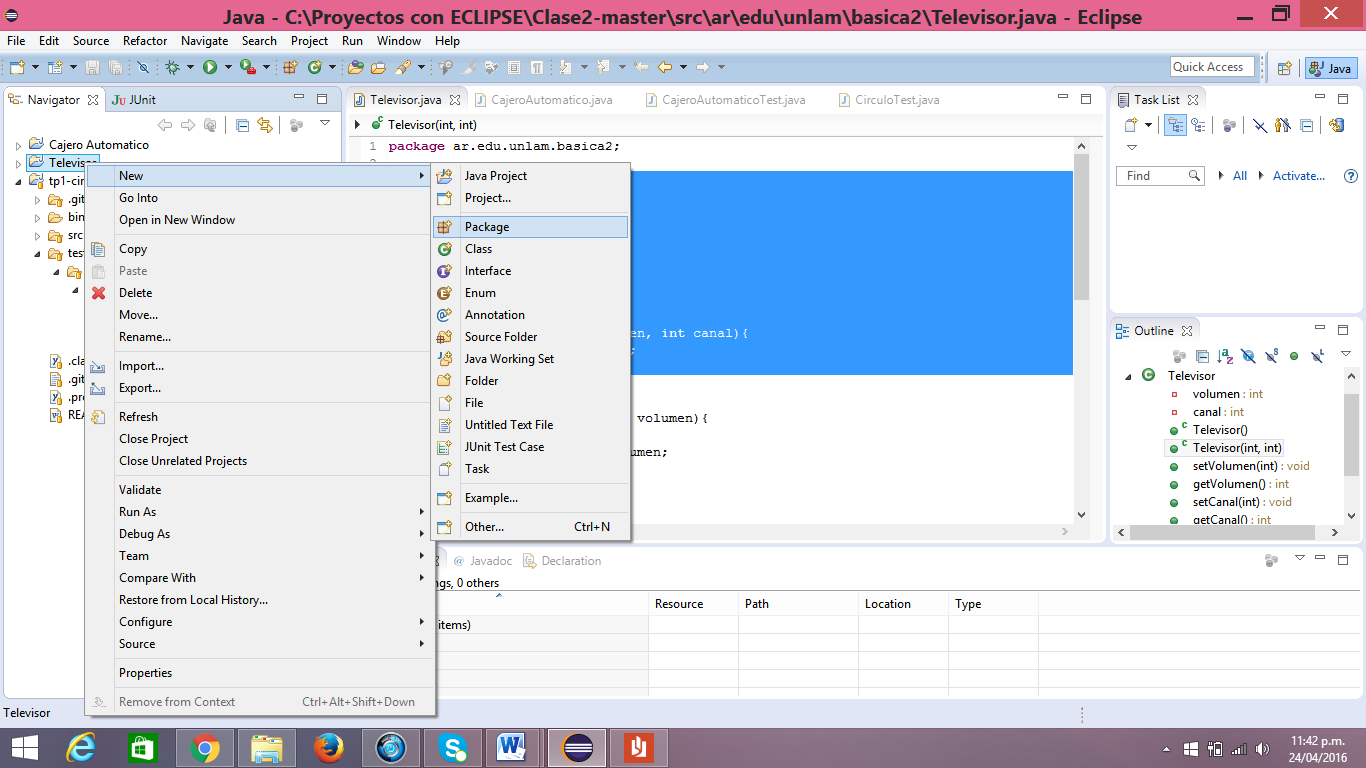
## ANEXO

EMPAQUETADO DE CLASES

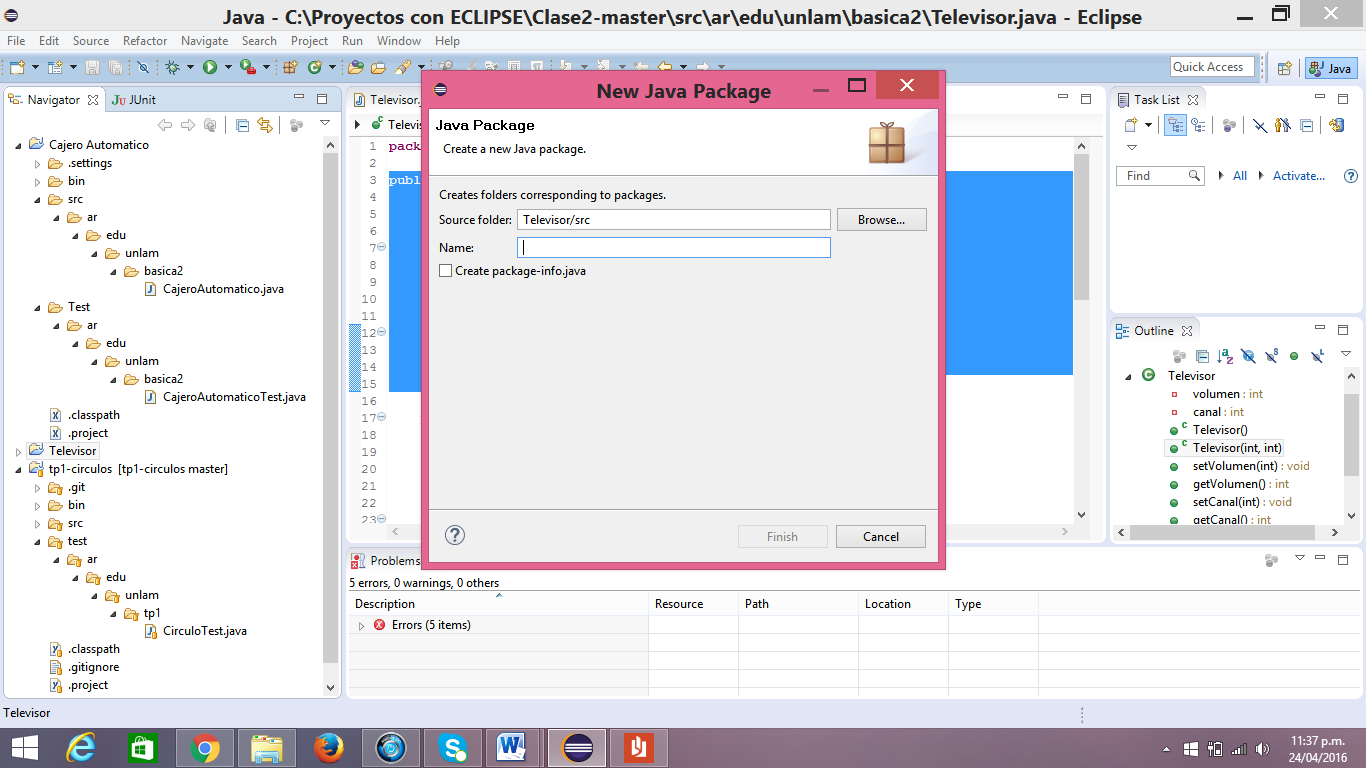
El empaquetado de clases es una práctica recomendable ya que permite el acceso de una clase a otras que se encuentran en un mismo paquete, sin necesidad de usar la cláusula *import*. Para crear una estructura de paquetes se debe proceder de la misma manera que al crear un árbol de directorio dado que un paquete no es más que un directorio en el que se encuentran los archivos .class, (las clases compiladas) dentro del directorio de trabajo. Ese directorio de trabajo corresponde a la variable classpath que se ha configurado durante la instalación del programa.

Para crear un paquete desde el entorno de trabajo Eclipse (Mars) es necesario cumplir los siguientes pasos:

1. Posicionados sobre el Proyecto Java recién creado, pulsar botón secundario del mouse y seleccionar New Package (paquete)



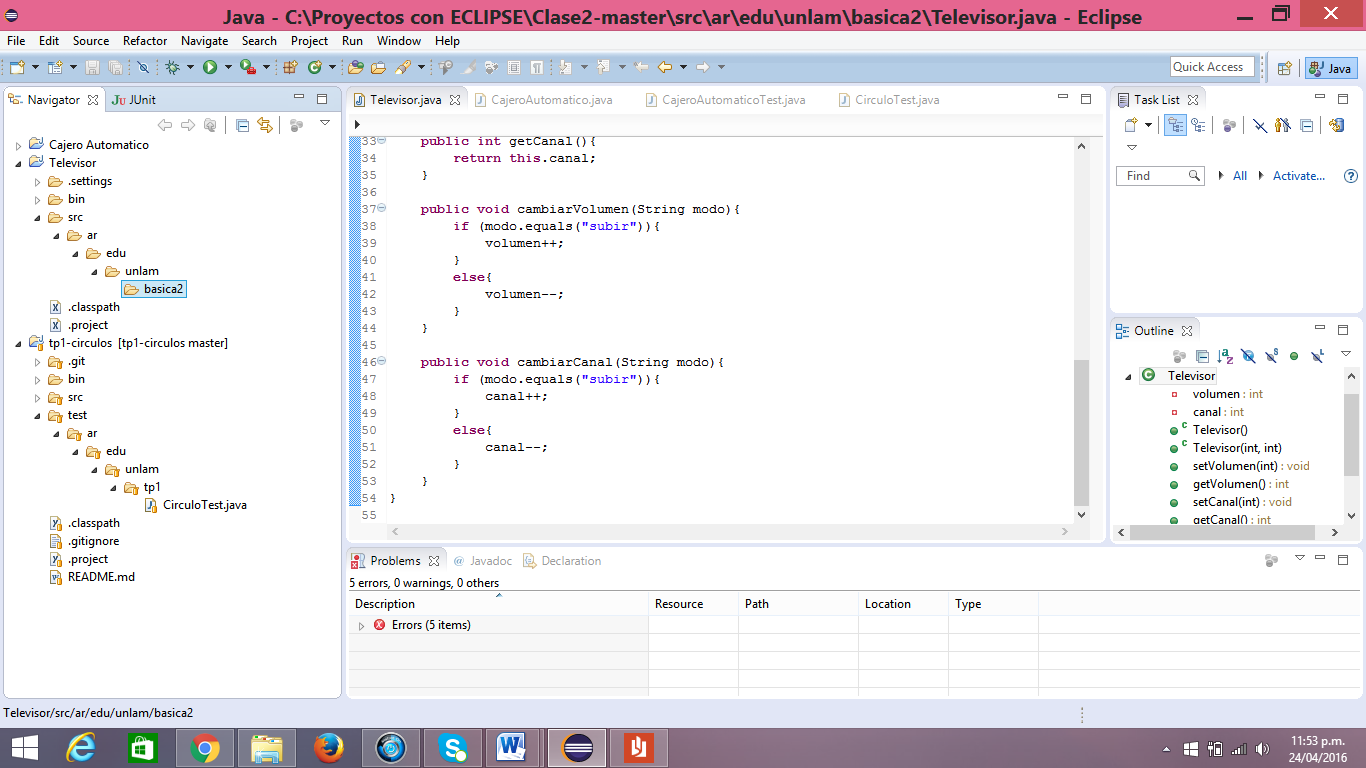
1. En la ventana que se despliega ingresar el path ( árbol de directorios donde se ubicarán las clases del proyecto, tanto los archivos .class, como los test de JUnit. En nuestro caso el path será **ar.edu.unlam.basica2.**



Aquí ingresar la dirección:

**ar.edu.unlam.basica2**

1. Una vez creado el paquete, posicionarse en la última carpeta y crear el archivo .class pulsando botón derecho y seleccionando New Class



Agregar aquí la nueva clase:

Proceder del mismo modo para crear una nueva *Source Folder* dentro del proyecto e incluir dentro de la misma (y en el mismo paquete) la clase para los test de JUnit.