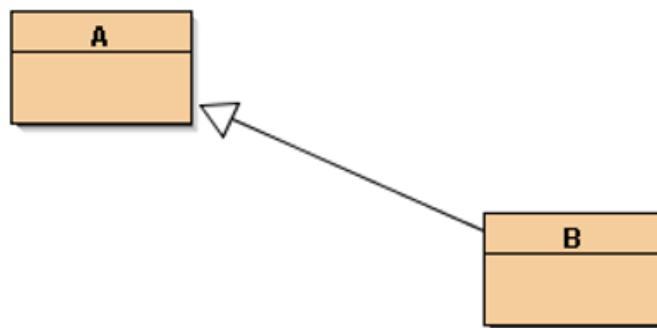


Apunte 08 - Herencia y Polimorfismo en Java

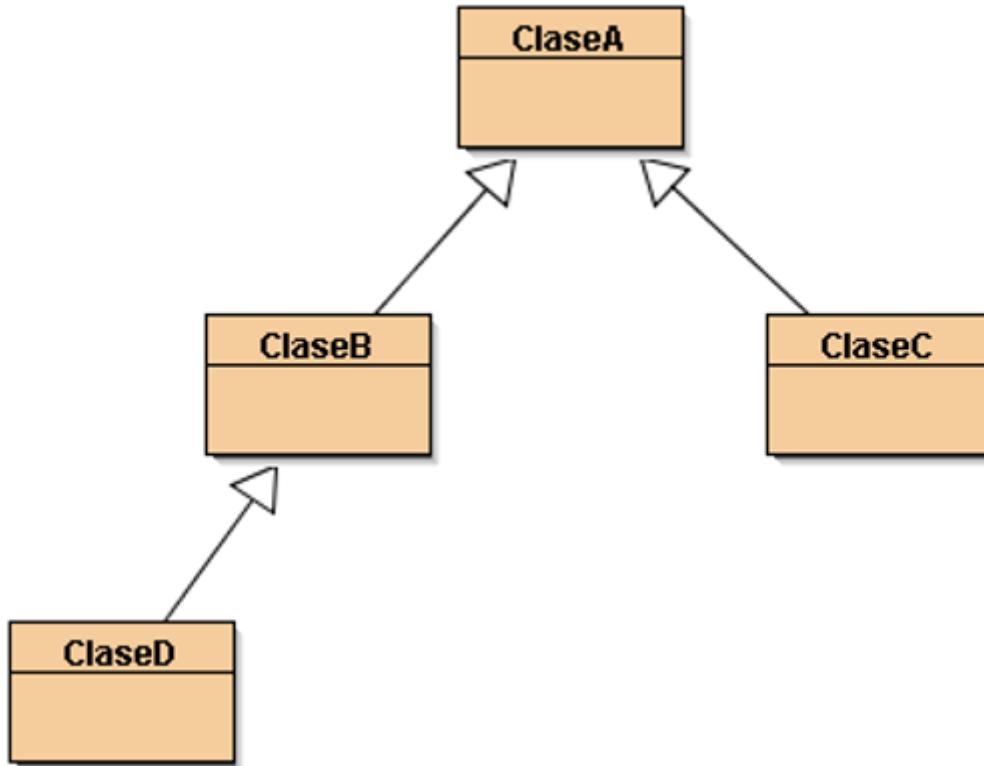
Herencia en Java

Conceptos de Herencia

En Programación Orientada a Objetos se llama herencia al mecanismo por el cual se puede definir una nueva clase B en términos de otra clase A ya definida, pero de forma que la clase B obtiene todos los miembros definidos en la clase A sin necesidad de hacer una redeclaración explícita. El sólo hecho de indicar que la clase B hereda (o deriva) desde la clase A, hace que la clase B incluya todos los miembros de A como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo al calificador de acceso [public, private, protected, "default"] que esos miembros tengan en A). Cuando la clase B hereda de la clase A, se dice que hay una relación de herencia entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o clase derivada) que sería B en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es A en nuestro caso):



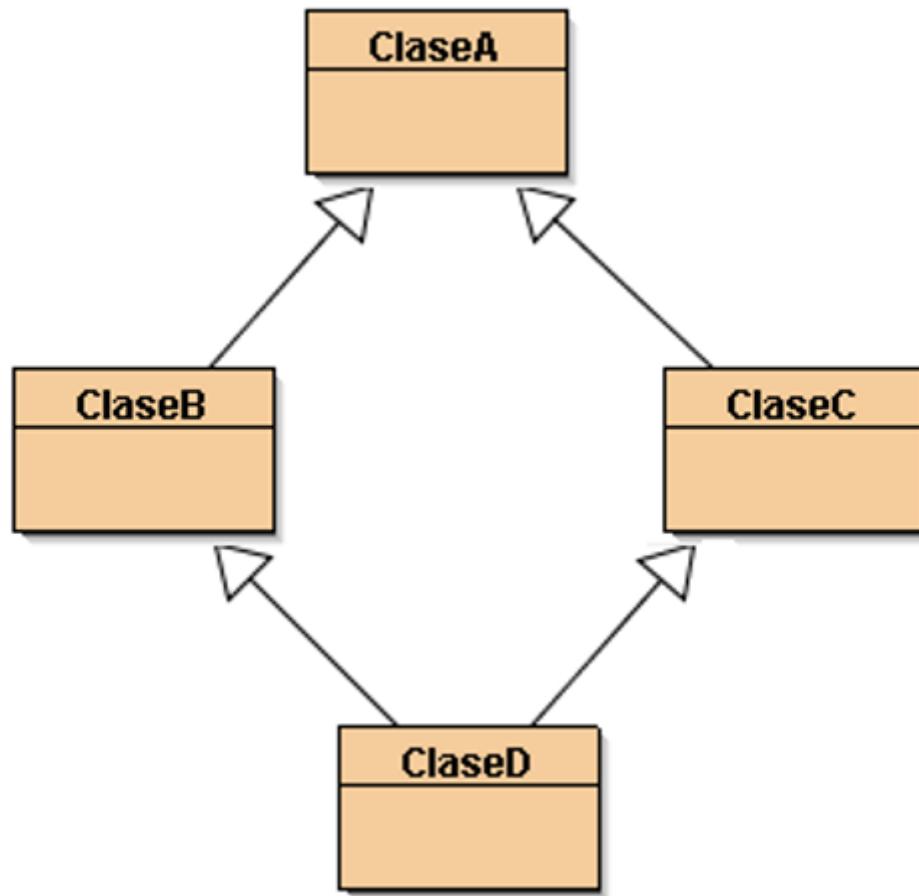
La clase desde la cual se hereda, se llama super clase, y las clases que heredan desde otra se llaman subclases o clases derivadas: de hecho, la herencia también se conoce como derivación de clases. Una jerarquía de clases es un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como clase base de la jerarquía. La idea es que la clase base reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una pequeña jerarquía de clases:



En esta jerarquía, la clase base es ClaseA. Las clases ClaseB y ClaseC son derivadas directas de ClaseA. Note que ClaseD deriva en forma directa desde ClaseB, pero en forma indirecta también deriva desde ClaseA, por lo tanto todos los elementos definidos en ClaseA también estarán contenidos en ClaseD. Siguiendo con el ejemplo, ClaseB es super clase de ClaseD, y ClaseA es super clase de ClaseB y ClaseC.

En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

- **Herencia simple:** Si se siguen reglas de herencia simple, entonces una clase puede tener una y sólo una super clase directa. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen herencia simple. La clase ClaseD tiene una sola super clase directa que es ClaseB. No hay problema en que a su vez esta última derive a su vez desde otra clase, como ClaseA en este caso. El hecho es que en herencia simple, a nivel de gráfico UML, sólo puede existir una flecha que parta desde la clase derivada hacia alguna super clase.
- **Herencia múltiple:** Si se siguen reglas de herencia múltiple, entonces una clase puede tener tantas super clases directas como se desee. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay herencia múltiple: note que ClaseD deriva en forma directa desde las clases ClaseB y ClaseC, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las clases ClaseB y ClaseC contra ClaseA es de herencia simple... tanto ClaseB como ClaseC tienen una y sólo una super clase directa: ClaseA.



No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación de clases de interface (que oportunamente discutiremos). El caso es que Java NO SOPORTA herencia múltiple: una jerarquía de clases como la mostrada en el gráfico anterior no es posible de ser definida en Java. Un ejemplo de un lenguaje que SÍ SOPORTA herencia múltiple es C++.

Sabemos que una relación de uso implica que un objeto de una clase usa a un objeto de otra. Pero una relación de herencia implica que un objeto b de una clase B, es a su vez un objeto de otra clase A. Cuando se diseñan clases, una pregunta que puede orientar a definir si debe usarse o no herencia entre las clases A y B, siendo que A está ya definida, es la siguiente: ¿se puede decir que los objetos de la clase B son objetos de la clase A? Por ejemplo, si tenemos diseñada una clase Persona, y queremos diseñar una clase Cliente (para un banco), la pregunta muestra en forma clara la relación: ¿Un Cliente es una Persona? Como la respuesta es sí, se podría hacer que la clase Cliente herede de Persona. Pero con respecto a las Cuentas... ¿un Cliente es una Cuenta? ó ¿un Cliente usa una Cuenta? Si bien para algunos bancos podría ser cierta la primera pregunta (🤔), en general es cierta la segunda, y por lo tanto se esperaría que un objeto de la clase Cliente tenga un atributo que sea una referencia a un objeto de la clase Cuenta, y no que Cliente herede de Cuenta....

En el modelo Herencia entregado como anexo a esta ficha, asumimos una jerarquía sencilla de clases que representan cuentas bancarias. La clase base Cuenta contiene sólo un número de cuenta y un saldo, más los métodos para manejar esos atributos. El tipo de cada cuenta es lo que definirá a cada subclase, pues cada una tiene comportamiento diferente de acuerdo a su tipo. Por ahora asumimos que los métodos depositar() y retirar() serán iguales para todas las clases derivadas.

Forma de declaración y uso de constructores en una jerarquía

Para indicar que una clase hereda de otra, se usa en Java la palabra reservada extends al declarar la clase derivada, nombrando luego de ella a la super clase de la misma. Esto hace que todo el contenido de la clase base esté presente también en la derivada, sin tener que volver a definirlo. Como el lenguaje Java no soporta herencia múltiple, al declarar una clase derivada y luego de la palabra extends no puede escribirse más que el nombre una única clase. Las siguientes declaraciones, tomadas del modelo Herencia, muestran la forma de declarar que las clases Inversion y Corriente derivan de la clase Cuenta:

```
// File Inversion.java
public class Inversion extends Cuenta
{
    // contenido de la clase Inversion
}

// File Corriente.java
public class Corriente extends Cuenta
{
    // contenido de la clase Corriente
}
```

En Java, si al declarar una clase no se indica si la misma deriva de otra (o sea, no se escribe extends), entonces el lenguaje asume que esa clase hereda desde la clase Object, que es la base de toda la jerarquía de clases de Java (predefinidas o no: incluso las clases del programador derivan desde Object en Java). En otras palabras, si al definir, por ejemplo, una clase Persona no se indica de quien hereda:

```
public class Persona
```

entonces el compilador Java reemplaza la declaración anterior por la siguiente en el bytecode:

```
public class Persona extends Object
```

Este es el motivo por el cual el método `toString()` (por ejemplo) está presente en una clase aún cuando la misma no lo redefina: lo está heredando desde `Object`. La clase `Object` provee una serie de métodos que serán comunes a todo objeto. Algunos de esos métodos se usan tal cual vienen desde `Object`, sin redefinir, pero otros (`toString()`, por ejemplo) deberían ser redefinidos por cada clase para adecuar su funcionamiento al conjunto de atributos de la clase. La clase `Object` provee los siguientes métodos (los nombramos sólo a los efectos documentales): `toString()` – `finalize()` – `getClass()` – `clone()` – `equals()` – `hashCode()` – `wait()` – `notify()` – `notifyAll()`. Y todos estos métodos están entonces presentes en toda clase Java predefinida o definida por el programador.

Cuando hay herencia (y por lo tanto una jerarquía de clases), cobra nueva importancia el constructor por defecto o constructor nulo, que es el que no toma parámetro alguno. Por lo tanto veamos las reglas que sigue el compilador Java en cuanto a ese constructor particular:

- Si un programador no incluye ningún constructor en una clase, el compilador incluirá un constructor nulo en el código de byte de la clase (archivo con extensión .class), y por ello cualquier creación de instancias invocando al mismo, compilará.
- Si el programador incluye algún constructor que NO sea el nulo, entonces el compilador no incluirá el constructor nulo en el .class, y cualquier intento de crear una instancia sin enviar parámetros a su constructor no compilará.

Y atención a los constructores en un contexto de herencia: al invocar un constructor de una clase derivada, Java espera que ese constructor invoque a su vez a algún constructor de la super clase. Esta invocación debería ser incluida en la primera línea del bloque de acciones del constructor de la derivada. La palabra reservada super puede usarse para invocar explícitamente a un constructor de la super clase. Si el constructor de la clase derivada no incluye una llamada explícita a algún constructor de la super clase, entonces Java impone una llamada al constructor por default de la super clase. Si ese constructor no estuviera en la super clase y no hubiera sido insertado automáticamente por el compilador, se provocará un error de compilación. Por ese motivo la inclusión del constructor default o nulo es recomendable, aunque pueda parecer innecesaria.

Veamos los siguientes ejemplos, tomados del modelo Herencia:

```
public class Corriente extends Cuenta {  
    public Corriente() {  
    }  
  
    public Corriente(int num, float sal, boolean des) {  
        super(num, sal);  
        descubierto = des;  
    }  
    ...  
}
```

El primer constructor mostrado es el constructor default de la clase Corriente (que sabemos que deriva de la clase Cuenta). Ese constructor no incluye ninguna instrucción explícita y por lo tanto tampoco incluye una llamada explícita a un constructor de la clase Cuenta. En este caso, el compilador reemplaza esa definición por esta otra cuando genera el archivo Corriente.class:

```
public Corriente() {  
    super(); // invocación explícita al constructor default de Cuenta  
}
```

Note el uso de la palabra reservada super para hacer que un constructor de una clase derivada pueda invocar a un constructor de la super clase. La definición aquí mostrada puede ser indistintamente usada por el

programador, en lugar de la originalmente mostrada: son equivalentes.

El segundo constructor mostrado para la clase Corriente asigna el parámetro des en el atributo descubierto, pero antes de eso invoca en forma explícita al constructor de la clase Cuenta que recibe dos parámetros. En este caso el programador impone una llamada al constructor que necesita. Note que se pasan parámetros a super para elegir la sobrecarga correcta del constructor a invocar. También note que si la llamada explícita al constructor no se hubiera hecho, el compilador invocaría de nuevo al constructor default de la clase Cuenta. En ese sentido, la siguiente definición:

```
public Corriente(int num, float sal, boolean des) {  
    descubierto = des;  
}
```

equivale por completo a esta otra (y es lo que entendería el compilador):

```
public Corriente(int num, float sal, boolean des) {  
    super();  
    descubierto = des;  
}
```

En el caso del modelo Herencia, la clase Inversion hereda desde Cuenta todos sus atributos y métodos, agrega nuevos métodos (actualizar()), y redefine algunos otros (toString()). La clase derivada podrá redefinir un método heredado si el planteo del método como está en la super clase no tuviera en cuenta el comportamiento y los atributos específicos de la derivada: el método toString() tal y como se hereda desde Cuenta, retorna una cadena que no incluye a la tasa de interés, y eso resulta incompleto para la clase Inversion. Por otra parte, los métodos getNumero(), setNumero(), getSaldo() y setSaldo() sirven tanto a la super clase como a la derivada, y por lo tanto no se redefinen: simplemente se usan. Si la clase derivada incluye métodos propios (caso: actualizar()), entonces esos métodos no pueden ser usados por la super clase: sólo existen en la derivada. Atención a las diferencias:

- **Redefinir un método** es la acción que tiene lugar en una clase derivada, para volver a definir un método heredado desde la super clase pero tal que su especificación no es adecuada a la derivada. Para redefinir un método, la clase derivada debe volver a escribir su cabecera pero TAL CUAL COMO FIGURA DECLARADA EN LA SUPER CLASE, excluyendo eventualmente el calificador de acceso. Al redefinir un método, en la clase derivada valdrá entonces el redefinido, y no el heredado.
- **Sobrecargar un método** es la acción que tiene lugar en una clase cualquiera, para agregar distintas versiones del mismo método en esa clase (caso: los constructores). Para sobrecargar un método, debe mantenerse el mismo nombre pero debe modificarse la forma de la lista de parámetros del mismo. No se toma como sobrecarga la diferenciación en el tipo de salida del método.

Observar el efecto de los calificadores de acceso cuando hay herencia: si en la super clase los atributos o métodos son privados, entonces son inaccesibles incluso para las clases derivadas. Ver el método actualizar() de la clase Inversión, en el cual debimos usar getSaldo() y depositar() para acceder al atributo saldo. Si hubiésemos nombrado directamente a ese atributo en algún método de la clase Inversión, hubiéramos

provocado un error de compilación. En cambio, si en la super clase los atributos fueran `protected`, serían accesibles para las clases derivadas como si fueran públicos, aunque no lo serían para clases que no deriven de esa super clase (hay otras reglas para el alcance de estos calificadores, que oportunamente discutiremos).

Una vez creada la jerarquía, podemos crear instancias de cualquiera de las clases en ella. Al invocar métodos, Java seguirá la pista del método invocado hasta la definición hecha en la clase del objeto, siempre y cuando ese método aparezca definido en la base de la jerarquía. Un objeto de una clase más arriba en la jerarquía, no puede invocar un método cuya primera definición aparezca en alguna clase más abajo. Pero lo contrario es válido: un objeto de una clase más abajo, puede invocar métodos definidos más arriba (el método `main()` de la clase Principal en el modelo Herencia muestra varios ejemplos de los dicho, y el alumno puede probar distintas combinaciones para observar sus efectos).

Polimorfismo en Java

Conceptos de Polimorfismo

Una referencia definida a la clase base de una jerarquía, sirve para referenciar objetos de cualquier clase de esa jerarquía. Esta propiedad se conoce como polimorfismo: una referencia que apunta a un objeto cuya forma es diferente a la que se esperaría por la declaración de la referencia.

Dada la jerarquía de clases que representan cuentas bancarias en el modelo Polimorfismo que se entrega junto a esta ficha, entonces una variable de la forma `Cuenta x`; podrá contener una referencia a objetos tanto de la clase `Cuenta`, como de la clase `Inversion` o de la clase `Corriente`. Note que no es válida la inversa: la variable `Inversion a`; sólo podrá referir a objetos de la clase `Inversion` (o de cualquier otra clase que derive de ella). Las variables para las cuales es válido el polimorfismo, se llaman referencias polimórficas. La variable `x` citada más arriba, es una referencia polimórfica. Los siguientes ejemplos, tomados del modelo Polimorfismo, muestran referencias polimórficas apuntando a objetos de diversas clases (recuerde que se definió la clase `Cuenta` como base de una jerarquía, y que `Corriente` e `Inversion` derivan de `Cuenta`):

```
Cuenta a = new Cuenta(1, 1000);
Cuenta b = new Inversion(2, 2000, 2.31f);
Cuenta c = new Corriente(3, 1500, true);

System.out.println("\nValores originales: ");
System.out.println("Cuenta a: " + a.toString()); // toString() de Cuenta
System.out.println("Cuenta b: " + b.toString()); // toString() de
Inversion
System.out.println("Cuenta c: " + c.toString()); // toString() de
Corriente
```

Lo importante es que cuando desde una referencia polimórfica se invoque a un método, la Máquina Virtual Java (JVM) no tendrá problemas para invocar a la versión correcta del mismo, siempre y cuando la primera definición de ese método aparezca en la clase base de la jerarquía analizada. Por ejemplo: una variable de la forma `Cuenta a`; podrá contener la dirección de un objeto de la clase `Inversion`. Cuando se haga `a.toString()` Java invocará a la versión definida en la clase `Inversion`, sin problemas. En el ejemplo anterior, cada llamada a `toString()` activa una versión diferente de ese método, de acuerdo al tipo del objeto al que realmente apunta cada referencia.

Sin embargo, si se desea invocar un método cuya primera definición aparece en la propia subclase `Inversion`, el programa no compilará. Esto se debe a que el compilador busca el método en la clase base y acepta el código si lo encuentra, dejando a la JVM el enlace con la versión correcta del método en tiempo de ejecución. Pero si el compilador no encuentra el método en la clase base, no aceptará el código, aún cuando en la práctica es la JVM la que resuelve el enlace. Para evitar este problema, la referencia debe recibir un casting explícito a la clase correcta. El siguiente ejemplo ilustra estos conceptos (tomado del método `main()` del modelo Polimorfismo):

```
Cuenta b = new Inversion( 2, 2000, 2.31f );

    b.setNumero( 5 ); // ok... setNumero() está definido en la clase base
(Cuenta)
    b.actualizar(); // no compila: el método actualizar() no está
definido en la clase base.

Inversion x = (Inversion) b; // hacemos casting explícito, con lo que x
apunta al mismo objeto que b...
x.actualizar(); // ahora sí... la referencia x es de tipo Inversion y no
hay problema con actualizar()
```

Por lo tanto, si hay polimorfismo hay que tener cuidado en cómo actuar en cada caso. Sea la variable `Cuenta a = new Inversion();` apuntando polimórficamente a un objeto. Entonces considere estas situaciones:

- Si se desea desde la referencia a invocar a un método definido en la clase `Cuenta` (base de la jerarquía), no habrá problemas, cualquiera sea el tipo "real" del objeto apuntado por `a`. Ejemplo: `a.getNumero()` ó `a.retirar(x)`;
- Si se desea desde `a` invocar a un método definido por primera vez en la clase "real" del objeto apuntado por `a`, deberá moldear (casting explícito) la referencia antes de poder hacerlo, para evitar el error de compilación:

```
Inversion x = (Inversion) a;
x.actualizar(); // está definido sólo en la clase Inversion, y no en la
base...
```

Finalmente, observe que en algunas situaciones posiblemente se querrá determinar la clase a la que pertenece el objeto apuntado por una referencia polimórfica. En ese caso, tenemos dos vías:

- Usar el operador `instanceof`. La siguiente condición determina si el objeto referido por `a` es de la clase `Corriente`:

```
// notar que aquí Corriente NO es un String, sino el nombre de la clase...
if (a instanceof Corriente)
```

- Usar el método getClass() que viene heredado desde Object. Este método devuelve un objeto de la clase Class (la cual está definida en java.lang). Los objetos de la clase Class representan a las clases de los objetos de la aplicación en curso. Si tenemos dos referencias (polimórficas o no) a y b, la siguiente condición determina si los objetos apuntados son de la misma clase "real":

```
if ( a.getClass( ) == b.getClass( ) )
```

Arreglos de objetos de clases diferentes

Está claro que el máximo nivel de polimorfismo en Java, se logra definiendo una referencia a Object: Como esa clase es la base de la jerarquía de todas las clases en Java, incluidas las del programador, una referencia a Object podrá apuntar a objetos de cualquier clase en un programa. Lo siguiente es válido:

```
Object x = new Inversion();
System.out.println( x.toString() ); // invoca al método toString() de la
clase Inversion

Object y = "casa"; // que sería lo mismo que Object y = new String("casa");
System.out.println( y ); // invoca al método toString() de la clase String
```

El polimorfismo permite definir estructuras de datos genéricas, esto es, estructuras que son capaces de contener objetos de distintos tipos, pero de la misma jerarquía. Un caso simple se muestra en el modelo Polimorfismo, en el método main() de la clase Principal: las líneas que siguen (tomadas de ese modelo) definen una referencia a un arreglo de objetos de clase Cuenta. Como cada casilla de ese arreglo es una referencia a una Cuenta, entonces cada casilla es una referencia polimórfica y se puede apuntar a objetos Cuenta, Inversion o Corriente. Se crean algunos objetos y se asignan en las casillas del arreglo:

```
Cuenta v[ ] = new Cuenta[ 4 ]; // un arreglo de referencias
polimórficas.
// ahora llenamos el arreglo con objetos de clases distintas... pero
compatibles.
v[0]= new Inversion(1, 3500, 1.23f);
v[1]= new Corriente(2, 500, false);
v[2]= new Cuenta(3, 700);
v[3]= new Inversion(4, 1500, 2.1f);
```

El procesamiento de un arreglo de referencias polimórficas puede hacerse sin mayores problemas. El siguiente ciclo, recorre el arreglo e invoca al método retirar() para cada objeto. Como ese método está definido en la clase base (Cuenta) y cada derivada lo hereda o lo redefine, entonces cada invocación funciona sin problemas y la JVM invoca siempre a la versión correcta del método:

```

for(int i=0; i<4; i++) {
    v[ i ].retirar(1000);
}

```

El siguiente segmento pretende procesar sólo las cuentas del arreglo que sean de Inversion, y a esas cuentas les actualizamos el saldo mediante la suma de intereses. La clase Inversion cuenta con el método actualizar(), pero ese método sólo está definido en esa clase: no existe en la clase Cuenta, que es la base de la jerarquía. El operador instanceof se usa para chequear si el objeto analizado es de la clase Inversion, y luego se hace casting explícito para obtener una referencia de tipo Inversion y poder acceder al método actualizar() sin error de compilación:

```

for(i=0; i<4; i++) {
    if(v[i] instanceof Inversion) {
        Inversion inv = (Inversion) v[ i ];
        inv.actualizar(); //actualizar() está definido sólo en
la clase Inversion
    }
}

```

Y el siguiente segmento muestra en consola los datos de las cuentas que sean de la misma clase que la que está en la primera casilla del arreglo. Para eso se usa el método getClass():

```

System.out.println("\nObjetos de la misma clase que el primero");
Cuenta este = v[ 0 ];
for(i=0; i<4; i++) {
    if ( v[i].getClass() == este.getClass() ) {
        System.out.println( "v[" + i + "]: " + v[ i ].toString() );
    }
}

```

Referencias y tipos primitivos

El lenguaje Java provee una serie de clases que permiten representar valores de tipos primitivos como objetos. De esta forma, incluso los valores de tipos primitivos pueden usarse en jerarquías de clases y entrar en el polimorfismo: veremos que esa capacidad resulta muy útil y necesaria para manejar objetos de clases que representan estructuras de datos ya implementadas en Java, en el package java.util. Esas clases se implementan definiendo simplemente un atributo del tipo al que se desea representar, y dotando a la clase de métodos para manejar el valor almacenado. Debido a que ese valor aparece como envuelto en la clase, esas clases se llaman *clases de envoltorio* (*wrapper classes*), y existe una por cada tipo primitivo:

<u>Tipo primitivo</u>	<u>Wrapper class</u>
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Todas estas clases proveen métodos para convertir el valor contenido en un String y viceversa, además de métodos para acceder al valor representado. Notar que estas clases están marcadas ellas mismas como final (no pueden ser derivadas) y sus atributos también son final (el valor de los atributos no puede modificarse una vez creado el objeto). Hemos usado diversos métodos de estas clases cuando se necesitó convertir cadenas que contenían números hacia valores int o float (por ejemplo, al usar los métodos de la clase Scanner para cargar números por teclado, se realizaba la conversión de String a números mediante los métodos Integer.parseInt() o Float.parseFloat()). Aunque esto lo hace Java y ya nos brinda métodos que devuelven tipos primitivos, si lo quisiéramos hacer nosotros mismos a partir del método next, esto sería obligatorio).

En el siguiente esquema, se crean objetos de algunas de estas clases primitivas, y se obtienen luego los valores contenidos para pasarlo a variables primitivas comunes. Note que cada clase de envoltorio provee al menos un método de la forma intValue(), floatValue(), charValue(), etc, que permite recuperar en forma de valor primitivo el valor envuelto por el objeto:

```
Integer i1 = new Integer(23);
Float f1 = new Float(2.34f);
Character c1 = new Character('$');
int i2 = i1.intValue();
float f2 = f1.floatValue();
char c2 = c1.charValue();
```

Note, sin embargo, que esta forma de pasar de un objeto wrapper a una variable primitiva y viceversa, era la única forma de hacerlo hasta la versión 1.4 (incluida) del lenguaje. Pero a partir de la versión 5.0 esto sigue siendo válido (lo anterior obviamente compilará en las nuevas versiones del lenguaje), pero también está disponible la posibilidad de pasar en forma directa entre los objetos wrapper y las variables primitivas: esta característica se conoce como auto-boxing, y es por supuesto mucho más cómoda. Lo que sigue es equivalente al esquema anterior, pero con auto-boxing, y compilará si usted dispone de un compilador desde la versión 5.0 o posterior de Java:

```
// asignación de primitivos en objetos wrapper, con auto-boxing...
Integer i1 = 23;
Float f1 = 2.34f;
```

```
Character c1 = '$';  
  
// extracción del valor contenido en el objeto...  
int i2 = i1;  
float f2 = f1;  
char c2 = c1;
```

Observe que en las tres primeras líneas se están creando tres objetos de las clases Integer, Float y Character, pero el compilador ya no exige un new explícito: ese new es insertado en forma automática por el compilador, y el valor primitivo que está a la derecha del operador de asignación es automáticamente enviado al constructor de la clase wrapper que esté participando en forma implícita. En esto, el compilador realiza el mismo servicio que ya realiza cuando se asigna directamente una cadena sobre una referencia de tipo String (y los programadores, agradecidos... 😊)

Modificadores static, final y abstract

Modificador **static** - Miembros de clase vs miembros de instancia

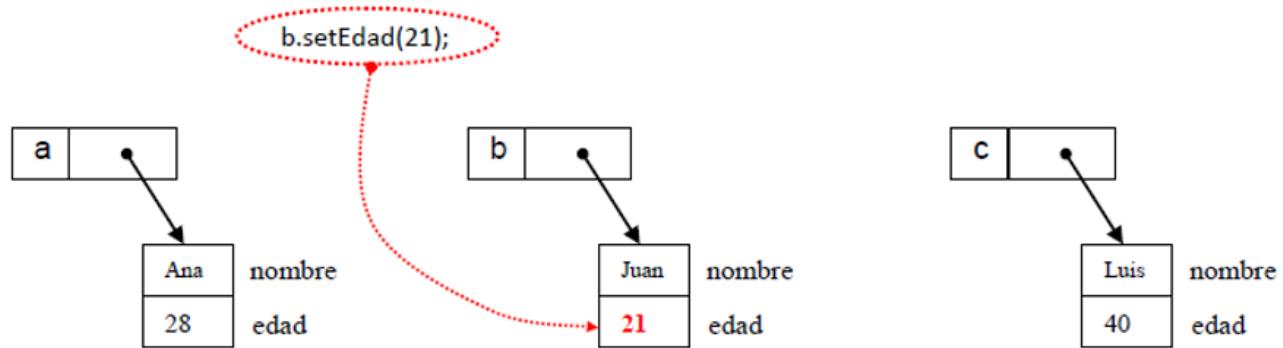
Sabemos que los distintos miembros (atributos y métodos) de una clase pueden marcarse como public o private (entre otros estados) según se desee imponer con más o menos fuerza el Principio de Ocultamiento. Sin embargo, existen otras palabras reservadas que pueden usarse para indicar la forma de acceder o compartir un miembro de la clase. Dos de ellas son las palabras static y final.

Si se tiene una clase cualquiera, cada vez que se crea un objeto mediante el operador new ese objeto tendrá dentro de sí una copia propia de todos los atributos de la clase. Cada objeto tendrá su propio juego de atributos y si un objeto modifica el valor de uno de sus atributos, eso no cambiará los valores de ese atributo en el resto de los objetos. Sea por ejemplo la clase Persona que usamos como modelo:

```
1 class Persona
2 {
3     private String nombre;
4     private int edad;
5
6     public Persona(String nom, int ed)
7     {
8         nombre = nom;
9         edad = ed;
10    }
11    public String getNombre()
12    {
13        return nombre;
14    }
15    public int getEdad()
16    {
17        return edad;
18    }
19    public void setNombre(String nom)
20    {
21        nombre = nom;
22    }
23    public void setEdad(int ed)
24    {
25        edad = ed;
26    }
27    public String toString()
28    {
29        return "\n\tNombre: " + nombre + "\tEdad: " + edad;
30    }
31 }
```

La clase tiene dos únicos atributos, declarados privados. Si ahora creamos algunos objetos, el gráfico que sigue muestra que cada uno tendrá acceso a su propio conjunto de atributos, que son una copia local de los que se declaran en la clase. Por lo tanto, si el objeto b invoca al método setEdad() y cambia su edad por otro valor, ese cambio sólo ocurrirá en el atributo edad contenido en b: nada ocurrirá con la edad de a o la de c:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
```



Ahora bien: si un miembro (atributo o método) se marca como estático (static), eso lo convierte en un miembro compartido por todas las instancias de la clase. Los miembros estáticos de una clase se cargan en memoria antes que se cree cualquier instancia de la clase, y todas las instancias que se creen luego, acceden exactamente al mismo elemento: no hay una copia de un atributo estático en cada instancia, sino que todas las instancias "ven" la misma variable (cada instancia posee un puntero a la única copia que hay de ese miembro). Esto puede parecer difícil de captar, o incluso puede parecer poco útil. Sin embargo, hay ciertas circunstancias en las que esta característica es muy valiosa.

Por ejemplo, supongamos la misma clase Persona ya vista, pero ahora supongamos que se requiere poder llevar la cuenta de cuántas instancias de la clase se han ido creando (por caso, supongamos que se ha organizado un evento social en el cual sólo hay lugar para cierto número de Personas, y se quiere llevar la cuenta de cuantas Personas ya han entrado). Se podría llevar la cuenta con un contador implementado desde el método `main()`, y cada vez que se invoque a `new` para crear una Persona, incrementar el contador. Sin embargo, esto resultaría algo molesto para quien está haciendo el programa, pues debería llevar control permanente sobre ese contador en todos y cada uno de los métodos donde se pudiera invocar a `new`.

Una mejor solución es hacer que la propia clase Persona lleve la cuenta de la cantidad de instancias que van creando de ella. En principio, el contador de instancias debería ser ahora un atributo de la clase, y también en principio ese contador debería incrementarse dentro del constructor (o constructores) de la clase. Un método como `getInstanceCounter()` podría usarse para retornar el valor de ese contador cada vez que se necesite saber la cantidad de instancias creadas:

```

1  class Persona
2  {
3      private String nombre;
4      private int edad;
5      // el contador de instancias de la clase
6      private int contador = 0;
7
8      public Persona(String nom, int ed)
9      {
10         nombre = nom;
11         edad = ed;
12         contador++;
13     }
14
15     public int getInstanceCounter()
16     {
17         return contador;
18     }
19
20     public String getNombre()
21     {
22         return nombre;
23     }
24
25     public int getEdad()
26     {
27         return edad;
28     }
29
30     public void setNombre(String nom)
31     {
32         nombre = nom;
33     }

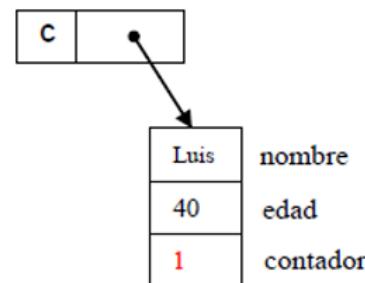
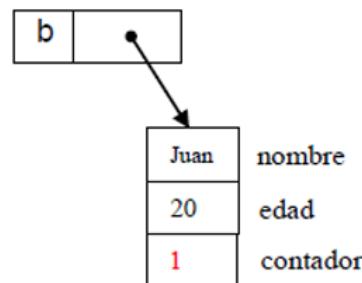
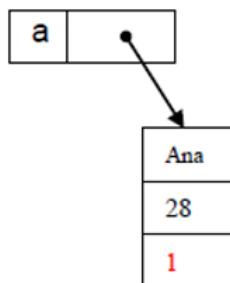
```

Puede verse fácilmente que esta solución es tristemente incorrecta: cada instancia que se cree tendrá su propio contador, que será puesto en cero al crear la instancia e incrementado a uno cuando se active el constructor. En todas las instancias de la clase, habrá un contador distinto y el valor de cada uno de ellos será exactamente igual a uno..

```

Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);

```

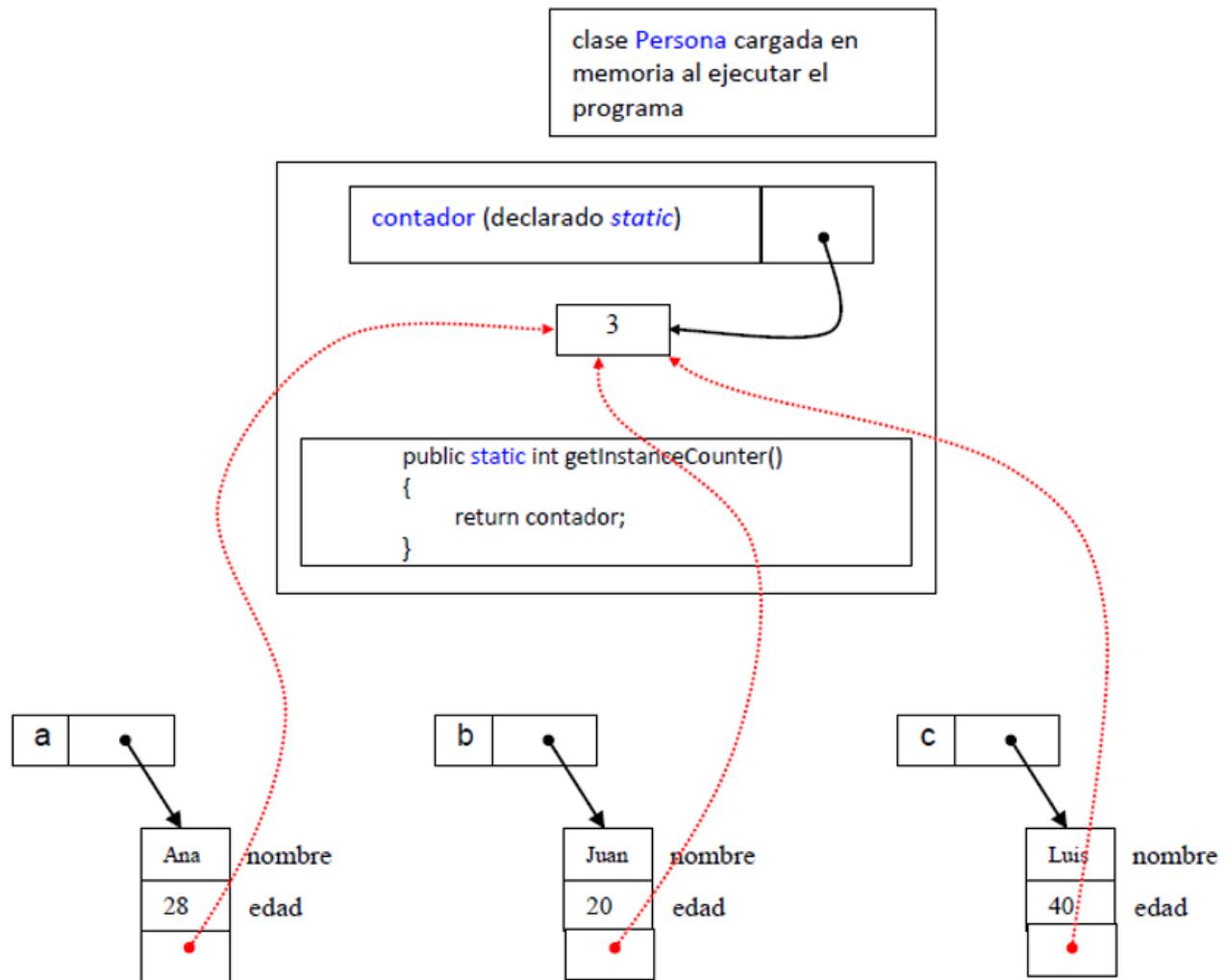


La solución sería hacer que el contador sea único: no que cada instancia tenga su propia copia, sino que exista una copia única de ese contador y que todas las instancias hagan referencia a esa única copia. Esto puede lograrse si el contador se declara static:

```
1  class Persona
2  {
3      private String nombre;
4      private int edad;
5      // el contador de instancias de la clase
6      private static int contador = 0;
7
8      public Persona(String nom, int ed)
9      {
10         nombre = nom;
11         edad = ed;
12         contador++;
13     }
14
15     public static int getInstanceCounter()
16     {
17         return contador;
18     }
19
20     public String getNombre()
21     {
22         return nombre;
23     }
24
25     public int getEdad()
26     {
27         return edad;
28     }
29
30     public void setNombre(String nom)
31     {
32         nombre = nom;
33     }
34 }
```

Como dijimos, si un atributo se declara static existe una copia única del mismo, y cada instancia tendrá un puntero a esa copia única. Cuando el programa comienza a ejecutarse, la clase se carga en memoria antes que se cree instancia alguna y junto con ella se cargan los elementos que se hayan declarado static dentro de ella. En ese momento la única copia del atributo se inicializa en cero. Cuando luego se crean las instancias, cada una de ellas hace referencia a la única copia del atributo mediante un puntero interno. Por lo tanto, luego de crear tres instancias el constructor se habrá invocado tres veces, y el único contador quedará valiendo tres:

```
Persona a, b, c;
int x = Persona.getInstanceCounter();
```



Como se ve, la única copia del atributo `contador` está en memoria junto con la clase, y no dentro de cada instancia. Por eso se suele decir que si un miembro (en este caso un atributo) es estático (declarado como `static`) entonces le pertenece a la clase y no a las instancias. Y también por eso, los atributos estáticos suelen designarse como atributos de clase. Si un atributo no es `static`, entonces cada instancia tendrá su propia copia: puede decirse que son las instancias las dueñas de esos atributos y eso hace que se los llame también atributos de instancia. En la clase `Persona`, los atributos `nombre` y `edad` son atributos de instancia (no son `static`) y el atributo `contador` es un atributo de clase (es `static`).

Notemos que también un método puede declararse `static`, y el efecto técnicamente es el mismo: si el método se declara `static` existe una única copia del mismo (que le pertenece a la clase) y se carga en memoria al cargarse la clase, del mismo modo que los atributos `static`. Si el método NO se declara `static`, entonces cada instancia posee su propia versión del método. Este hecho no parece tener mayor relevancia: al fin y al cabo que el método le pertenezca a la clase o a las instancias no cambia el proceso definido dentro de él. Sin embargo, hay una sutil (y útil) diferencia práctica: si un método es `static`, el mismo existe en memoria aunque no se haya creado aún ninguna instancia, y puede entonces ser invocado sin necesidad de crear instancias... Simplemente, se usa el nombre de la clase para invocar al método, en lugar de una referencia a una instancia. Notemos el método `getInstanceCounter()` de la clase `Persona`: está declarado `static` y por lo tanto podemos invocarlo de la siguiente forma:

```
int x = Persona.getInstanceCounter();
```

Esto tiene sentido: es la clase la que está llevando la cuenta de la cantidad de instancias creadas, y conceptualmente es válido que sea la propia clase la que llame al método... No obstante, tenga en cuenta que un método static también puede ser invocado sin problemas por una instancia de la clase, en forma normal. Lo siguiente es válido:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
int x = a.getInstanceCounter();
System.out.println("Cantidad de instancias creadas:" + x);
```

y se mostrará en consola un mensaje indicando que la cantidad de instancias creadas es 3. El mismo resultado se obtendría si en vez de la referencia a, se usara b o c para invocar a getInstanceCounter(), o si se invocara el método con el nombre de la clase en lugar de una referencia (puede ver esto ejecutando el modelo Static que acompaña a esta ficha).

Por cierto, si un método NO es static entonces sólo existe dentro de una instancia, y por lo tanto no puede ser invocado mientras esa instancia no se cree. En otras palabras, sólo se puede invocar mediante una referencia a un objeto y no mediante el nombre de la clase. En la clase Persona, todos los métodos (salvo getInstanceCounter()) son no – static, y cualquier intento de invocarlos mediante el nombre de la clase producirá un error de compilación. Observe el siguiente ejemplo:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
int y = b.getEdad(); // correcto...

int z = Persona.getEdad(); // no compila: el método no es static...
```

También vale decir que si un método es static, entonces es un método de clase (como getInstanceCounter() en la clase Persona). Y si un método no es static, entonces es un método de instancia (como el resto de los métodos de la clase Persona).

Un análisis detallado del alcance del calificador static en una clase, muestra que si un método es static, entonces los atributos que ese método acceda dentro de la misma clase también deben ser static, y si el método invoca a otros métodos de la misma clase esos métodos deben ser a su vez static... De otro modo, no compilará: el método estaría accediendo a elementos que no existen (o podrían no existir) al momento de la invocación. Por ejemplo, el método getInstanceCounter() no podría acceder al atributo edad o al atributo nombre, y tampoco podría invocar a ninguno de los otros métodos de la clase Persona:

```
// el método es de clase...
public static int getInstanceCounter()
{
    int x = edad + 1; // no compila: edad es atributo de instancia...
    String n = getNombre(); // no compila: getNombre() es método de instancia...
    return contador;
}
```

Notar por último, que el método main() (cuando está presente) se define como static: debe poder invocarse desde el sistema operativo o desde el entorno de programación que se esté usando, sin necesidad de crear una instancia de la clase que lo contiene (¡¡¡...!!!). Cuando se pide la ejecución de una aplicación, la máquina virtual Java debe saber cuál de todas las clases contiene al método main(), y esa máquina virtual se encarga de ejecutar el método. Como main() es estático, no puede invocar otros métodos de su misma clase que a su vez no sean estáticos, ni acceder a atributos de esa clase que no sean estáticos. Ese es el motivo por el cual hasta ahora, los programas realizados debían declarar static a los atributos y métodos de la clase Principal (o como fuese que se haya llamado la clase del main()) si esos atributos y métodos iban a ser accedidos desde main()...

Modificador **abstract**

En muchos casos una clase se diseña pero no para crear instancias de esta, sino para permitir el agrupamiento de características comunes, facilitar la herencia, y posibilitar el polimorfismo. Por ejemplo volvemos a presentar una jerarquía de clases que representan cuentas bancarias, no se esperaría que en una aplicación se creen instancias de la clase Cuenta, simplemente porque dichas instancias en la práctica serían incompletas. La clase Cuenta no representa cuentas concretas con datos aplicables a la práctica, sino cuentas abstractas con la esencia básica de lo que una Cuenta "es" y "hace".

Se puede indicar al compilador Java que no permite crear instancias de clases que cumplan esos requisitos de abstracción, declarando a dichas clases como abstractas. La clase Cuenta podría serlo:

```
public abstract class Cuenta {
    ...
}
```

Las clases así declaradas se dicen clases abstractas y el intento de instanciarlas provoca un error de compilación:

```
Cuenta x = new Cuenta();      // no compila...
Cuenta y = new Inversion();   // pero esto sí compila.... si no, el
polimorfismo se nos va!!!

Cuenta z; // ok... esto es una referencia polimórfica!!!
z = new Cuenta(); // no compila...
```

Nota: Observe que el error no es declarar una referencia a una clase abstracta, sino usar new para crear un objeto de una clase abstracta.

Por contraposición, las clases "no abstractas", se dicen clases concretas, y pueden crearse instancias normalmente, incluso guardando polimórficamente la dirección en una referencia a una clase abstracta (ver ejemplo anterior). En nuestro caso, las clases Inversion y Corriente son concretas.

Ahora bien, es posible que al definir una clase abstracta nos encontremos con la necesidad de incluir métodos en ella para facilitar luego el polimorfismo. Pero muchos de esos métodos podrían no tener mucho sentido para la clase en cuestión, sino sólo para sus derivadas.

En ese caso, tales métodos pueden marcarse ellos mismos como *abstractos*, agregando la palabra `abstract` en su cabecera, pero sin incluir su bloque de acciones. La definición del bloque de esos métodos se deja para las derivadas. Si una clase tiene un método abstracto, ella misma se convierte en abstracta, y por lo tanto debe definirse como tal (de otra forma, no compila...) Y si una clase se deriva de una clase abstracta que contiene un método abstracto, la derivada está obligada por el compilador a redefinirlo o a definirse a sí misma como abstracta.

En nuestro caso, marcamos como abstracta a la clase Cuenta, y al método retirar() incluido en ella también. La implementación de ese método se dejó para cada derivada. En la clase Cuenta, el método está definido así (note la palabra `abstract`, la ausencia del bloque de acciones, y el punto y coma al final de la cabecera del método):

```
public abstract void retirar (float imp);
```

Esto significa que la clase Cuenta obliga a sus derivadas a contar con ese método, y también las obliga a redefinirlo (si esas derivadas no son abstractas a su vez). La clase Cuenta no necesita saber cómo se implementa la operación de retiro de fondos, pues esa operación es muy particular de cada clase derivada. Pero el diseñador del sistema puede haber notado que la operación `retirar()` es vital en una jerarquía de clases que representan cuentas, y necesita que esa operación esté disponible para toda clase de la jerarquía y de esta forma activar el proceso en forma polimórfica.

Notar que la clase Inversion hereda de Cuenta, que es abstracta y provee un método abstracto `retirar()`. La clase Inversion debe redefinir ese método. Al hacerlo, la palabra `abstract` no debe volver a escribirse (pues el método de otro modo volvería a marcarse abstracto, obligando a la clase a marcarse ella misma como tal..) Como ahora la clase Cuenta es abstracta, cualquier intento de crear una instancia de esa clase, provocará un error de compilación. Sin embargo, se puede seguir aplicando el polimorfismo sin problemas. En la clase Inversion, el método está declarado así:

```
...
public void retirar (float imp) {
    float s = getSaldo();
    if (s - imp >= 0) {
        setSaldo(s - imp);
    }
}
```

...

En conclusión el modificador **abstract** se puede utilizar tanto en la definición de clases como en la definición de métodos y si quisieramos leer su significado traducido a lenguaje coloquial no sería el mismo en cada caso. En el caso del modificador **abstract** en las clases podría leerse como *debe ser derivada*, de hecho en otro lenguaje de programación este modificador se escribe como *MustInherit* que es precisamente esto en inglés. Pero para el caso de los métodos el significado es sutilmente diferente, los métodos se heredan si o sí en Java y por lo tanto el modificador **abstract** en un método significa o puede leerse como *debe ser redefinido*, del mismo modo en el lenguaje de programación que mencionamos a este modificador le corresponde la palabra reservada *MustOverride* con dicho significado del inglés.

Modificador **final** - Declaración y uso de constantes

El modificador **final** puede ser utilizado en los mismos lugares que el modificador **abstract**, sin embargo, su significado es directamente lo opuesto al modificador **abstract** es decir, si agregamos **final** a la definición de una clase, estaremos diciendo que dicha clase *no puede ser derivada*, es decir, no podré declarar otra clase que herede de una clase marcada como **final**.

Luego en el caso de utilizar el modificador **final** en un métodos, nuevamente estaremos diciendo lo opuesto que para **abstract**, diremos que dicho método *no puede ser redefinido*, y si intentamos redefinirlo provocaremos un error de compilación.

Ahora bien, en el caso del modificador final, también puede ser utilizado para marcar atributos, parámetros e incluso variables locales, y sobre este uso existen una serie de buenas prácticas acerca de los niveles de optimización de código, pero como ya dijimos no estamos en un curso del lenguaje Java así que nos limitaremos a definir su comportamiento.

Un atributo puede ser marcado también como final, lo cual en esencia lo convierte en una constante. Una vez asignado un valor inicial a una constante, el mismo no puede ser modificado durante el resto del programa (cualquier intento de hacerlo provocará un error de compilación). Cualquier intento de modificar el valor de ese atributo luego de haberle dado su valor inicial, provoca un error de compilación.

Además, los calificadores **final** y **static** pueden combinarse: un atributo marcado con ambos calificadores será una constante (por ser **final**), y también será compartida la misma copia de esa constante por todas las instancias de la clase (por ser **static**). En casos como estos, en que el atributo es compartido y constante, se estila también declararlo publica en lugar de private al fin y al cabo, el Principio de Ocultamiento busca evitar que se manipule de manera incorrecta el valor de un atributo de la clase, pero eso no podrá ocurrir si ese atributo es único para todas las instancias y marcado como constante... no hay nada que cada instancia pueda hacer con ese atributo salvo devolver su valor, y por lo tanto la práctica de declararlo public brinda flexibilidad para acceder a valores constantes de uso general que se almacenan en alguna clase, a manera de constantes globales para todo el programa.

Hay ciertas reglas que conviene enumerar si se trabaja con atributos estáticos y/o finales para evitar errores de compilación (puede usar la clase Aerolinea del modelo para probar las distintas combinaciones que se sugieren a continuación):

- Un atributo declarado **static** (pero sólo **static** y sin que haya combinación con **final**) puede ser asignado en el momento de la declaración, o dentro de un constructor de la clase, o en cualquier método de la

clase. Y una vez asignado, su valor puede ser cambiado desde cualquier otro método (a fin de cuentas, si el atributo sólo es static no es una constante: es sólo una variable de uso compartido y puede ser inicializada o cambiar su valor en cualquier parte).

- Un atributo declarado final (pero sólo final y sin que haya combinación con static) puede ser asignado en el momento de la declaración, o dentro de un constructor de la clase (pero no en ambos lugares: sólo en uno de ellos). Luego de realizada la asignación, el valor asignado no puede cambiarse (el compilador lanzará un error si se intenta).
- Si un atributo se declara static final, debe ser asignado al declararse (y no ya dentro de un constructor o en otro método de la clase). Si se intenta hacer la inicialización en otro lugar, se provocará un error de compilación. Esto se debe a que dicho atributo será cargado y mantenido inmodificable antes que cualquier instancia sea creada, y por lo tanto no se puede esperar a que alguna de ellas invoque a un constructor para asignarle un valor definitivo.

Finalmente, si agregamos el modificador `final` a un parámetro de un método o a una variable local nuevamente estamos indicando que una vez establecido un valor en dicho identificador ese identificador ya sea parámetro o variable local no podrá ser modificado provocando en el caso de hacerlo un error de compilación.

Polimorfismo Aplicado - Interfaces o clases de Interfaz en Java

Hemos visto que el polimorfismo permite el diseño de estructuras de datos genéricas, capaces de contener objetos de distintas clases siempre que pertenezcan a clases de la misma jerarquía. También hemos sugerido que en Java, el máximo nivel de polimorfismo se lograría declarando referencias a la clase `Object`, ya que con esas referencias se podría apuntar y manejar objetos de cualquier clase (sea nativa de Java o diseñada por el programador)

Supongamos que se desea programar una clase que contenga un vector o arreglo tan genérico (tan polimórfico) como fuera posible (es decir, supongamos que se desea poder almacenar objetos de cualquier clase en ese vector). Parece obvio que la forma de hacerlo sería declarar y crear ese arreglo de forma que contenga referencias de tipo `Object` en cada casillero:

```
Object v[ ] = new Object[10];
```

El arreglo `v` creado en la instrucción anterior será claramente capaz de contener hasta diez referencias a objetos de cualquier clase, los cuales deberán ser creados y asignados oportunamente en forma similar a como se muestra en el ejemplo siguiente:

```
v[0] = new Inversion();
v[1] = "Una cadena";
v[2] = new Estudiante();
// resto de las asignaciones aquí...
```

Una vez creado el arreglo y asignados en cada casillero las direcciones de los objetos requeridos, el arreglo puede procesarse en forma normal, recorriendo con los consabidos ciclos for de avance secuencial sobre su rango de índices, e invocando métodos que estén definidos en la clase Object o bien identificando el tipo real del objeto y usando casting descendente (downcasting) si se desea rescatar objetos de una clase en particular:

```
// mostrar el contenido completo: invocación a toString()...
for(int i = 0; i < v.length; i++) {
    System.out.println(v[i].toString());
}

// Acumular el saldo de los objetos que sean de tipo Inversion...
float ac = 0;
for(int i = 0; i < v.length; i++) {
    // identificación del tipo del objeto...
    if(v[i] instanceof Inversion) {
        // downcasting...
        Inversion x = (Inversion) v[i];
        ac += x.getSaldo();
    }
}
```

Pero supongamos ahora que se desea operar con el contenido de cada objeto, por ejemplo para poder comparar sus valores (por caso, para mantener ordenado el arreglo). Entonces, todo objeto almacenado en los casilleros del vector debería proveer un método que permita compararlo con otro objeto que sea de su misma clase. Tal método podría llamarse compareTo() y podría retornar un valor numérico que indique el resultado de la comparación. Si a y b son dos objetos cualesquiera pero de la misma clase, y esa clase tiene implementado ese método, su uso podría ser el siguiente:

```
int r = a.compareTo( b );
```

y aceptar la siguiente convención en cuanto al resultado returned:

- $r == 0 \Rightarrow$ los objetos a y b eran iguales
- $r > 0 \Rightarrow$ el objeto invocante era mayor al parámetro: $a > b$
- $r < 0 \Rightarrow$ el objeto invocante era menor al parámetro: $a < b$

¿Dónde incluir tal método? Si todos los objetos a incluir en nuestro arreglo deben tenerlo, entonces el método debería definirse en Object, pero esa clase no lo contiene y el programador no puede modificarla para que lo incluya...

Además, la operación de comparar si un objeto es “mayor” o “menor” que otro, podría no tener sentido conceptual para ciertas clases. Por ejemplo, los número complejos no admiten relación de orden: puede compararse si un complejo es igual a otro, pero no tiene sentido preguntar si uno de ellos es menor que otro. Básicamente, ese es el motivo por el cual compareTo() no viene predefinido en Object y sí viene predefinido

el método equals(), que puede ser redefinido por el programador para que verifique si dos objetos son iguales (lo cual es siempre válido).

Otra idea sería definir una nueva clase (que podría ser abstracta) que contenga a ese método como abstracto, y luego hacer que todas las clases que se desee almacenar en nuestro arreglo hereden de ella, para forzarlas a implementar el método... Pero algunas de esas clases podrían estar ya heredando de otra... y Java no admite herencia múltiple.

La solución a este tipo de problemas son las clases de interface (o simplemente interfaces). Muy esencialmente una clase de interface es una clase abstracta que sólo provee métodos abstractos (no puede contener atributos, a menos que esos atributos sean definidos como constantes). Para simplificar la sintaxis y no entrar en problemas de violación de herencia múltiple, Java usa la palabra reservada interface en lugar de abstract class, y automáticamente asume como public abstract a todo método definido en ella. Una interface que solucione nuestro problema podría verse así:

```
public interface Comparable {  
    int compareTo (Object x);  
}
```

Se estila designar a las interfaces usando nombres terminados en "able" que sugieran que un objeto de una clase que implemente sus métodos adquiere esa propiedad. Por ejemplo, los siguientes podrían ser nombres efectivos: Comparable, Ejecutable, Visualizable, etc. Los objetos que implementen el método compareTo() serán entonces "Comparables".

En el caso particular de la interface Comparable, no es necesario que el programador la defina, pues en Java ya viene definida en forma nativa e incluida en el paquete java.lang, que se carga automáticamente y está disponible para el programador. En otras palabras, si usted desea que una clase asuma la propiedad de ser Comparable, simplemente declare que la misma implementa a Comparable, y luego agregue dentro de ella su definición para el método compareTo(). No defina la interface en sí misma, porque ya viene definida en Java.

Las clases de interface en general no se heredan: se implementan. Para indicar que una clase implementa una interface, al definir esa clase debe usarse la palabra implements en lugar de extends, y luego el nombre de la interface implementada. En este contexto, si una clase implementa una interface, esa clase debe redefinir todos los métodos que esa interface declara (de otro modo, la clase no compilará). Si la clase Cliente implementa la interface Comparable, su declaración es:

```
// File Cliente.java  
public class Cliente implements Comparable {  
    ...  
}
```

```
// File Corriente.java  
public class Corriente extends Cuenta implements Comparable {
```

```
...  
}
```

Una clase en Java sólo puede indicar herencia desde una única clase, pero puede indicar implementación de tantas interfaces como se desee. Si una clase B hereda de otra A, y al mismo tiempo implementa las interfaces I1 e I2, la definición de la clase B se vería así: `public class B extends A implements I1, I2`

Las clases de interface también posibilitan polimorfismo a partir de ellas. Así, todas las instancias de las clases que implementen la interface Comparable, pueden ser apuntadas por una referencia polimórfica:

```
Comparable c; // referencia polimórfica  
c = new Cliente(); // ok!!!
```

La ventaja de esto último es que puede lanzarse polimorfismo entre clases que originalmente no forman parte de la misma jerarquía de herencia, a condición de que todas esas clases implementen la misma interface. En ese sentido, la implementación de interfaces aparece como más amplia que la herencia múltiple, aunque conceptualmente más simple.

En nuestro arreglo polimórfico, entonces, deberíamos hacer que cada casilla apunte a objetos de clases que hayan implementado Comparable y no a Object, si es que queremos contar con la posibilidad de comparar esos objetos. De este modo, se podrá aprovechar la presencia de ese método para implementar otras operaciones que requieran comparar objetos (y no sólo el ordenamiento).

Clases String y StringBuilder

El lenguaje Java permite manejar cadenas de caracteres, a través de variables de la clase String. En muchos programas se requerirá almacenar nombres de personas, ciudades, descripciones de artículos, y otros datos o resultados que esencialmente consisten de varios caracteres formando una hilera. En este caso, la hilera misma es el dato o el resultado y no cada uno de los caracteres que la forman.

Las cadenas de caracteres no forman un tipo de datos básico o primitivo del lenguaje, debido a que una cadena es un elemento sustancialmente más complejo que un simple valor numérico, un `char` suelto o un valor booleano: podemos ver fácilmente que una cadena es un conjunto de varios valores `char`, que tiene cierta longitud, y otras muchas propiedades. Los tipos primitivos representan valores simples o únicos, que se implantan directamente en memoria a través de una variable que contiene a ese único valor.

Las cadenas de caracteres en Java se representan como valores de un tipo especial de datos llamado String. En principio, manejar un String es simple, y el procedimiento no difiere mucho de la forma de manejar primitivos: se declaran variables String, y se asignan valores en ellas en forma normal mediante el operador de asignación.

En nuestros primeros programas usaremos cadenas en forma básica. Pero en algún momento será necesario realizar ciertas operaciones muy comunes con cadenas, tales como las comparaciones de cadenas, que no pueden plantearse directamente con los operadores de comparación comunes (estos operadores como `<`, `>`, `<=`, etc., sólo son válidos, en general, para comparar valores primitivos).

Comparación de cadenas

La comparación de cadenas de caracteres es una operación muy común en programación. Muchas veces es necesario poder determinar si una cadena es igual a otra, o poder determinar cuál de dos cadenas es menor o mayor que la otra. En el contexto de un lenguaje de programación, se dice que una cadena es menor (o mayor, según el caso), cuando dicha cadena es alfabéticamente menor (o mayor) que la otra. Expresado con un ejemplo más cotidiano, decimos que una cadena es menor que otra de acuerdo al orden en que aparecerían ubicadas en un diccionario. Así, la cadena "anillo" es alfabéticamente menor que la cadena "casa". Notar que la longitud de las cadenas no tiene, en principio, incidencia alguna en la determinación del orden entre ellas.

Si se observa, la determinación del orden entre dos cadenas se hace comparando los caracteres de inicio de ambas. Será menor la cadena cuyo carácter de inicio sea menor (en el caso del ejemplo, la cadena "anillo" comienza con 'a', que es alfabéticamente menor que 'c', que es el carácter de inicio de la cadena "casa"). Si dos cadenas comienzan con los mismos caracteres, entonces se continúa comparando los que siguen a la derecha, y así sucesivamente hasta encontrar un par de caracteres distintos. Así, la cadena "anillo" es alfabéticamente mayor que la palabra "anillado". Por otra parte, y como es obvio, dos cadenas serán iguales si coinciden en todos sus caracteres. Este principio de ordenamiento de cadenas (que es el que se sigue en el ordenamiento de palabras en un diccionario, por ejemplo), se denomina principio de ordenación lexicográfica.

No se alarme el lector: en todos los lenguajes de programación modernos, la comparación de cadenas de caracteres se realiza a través de operadores y/o instrucciones o métodos propios del lenguaje. No es necesario desarrollar un programa para aplicar el principio lexicográfico. En el lenguaje Java, la comparación de cadenas se realiza a través del método `compareTo()` que viene incluido en la clase `String` (del mismo modo que la clase `Consola` contiene los métodos `readInt()`, `readDouble()` y `readFloat()`).

El método `compareTo()` trabaja con dos `Strings`, las compara lexicográficamente, y retorna un valor de tipo `int`, que indica el resultado de la comparación. Supongamos que queremos comparar dos cadenas guardadas en dos variables. La forma de hacerlo sería:

```
2  import java.util.Scanner;
3
4  public class Principal {
5
6      public static void main(String args[]) {
7          String a, b;
8          int r;
9          Scanner miEscaner = new Scanner(System.in);
10
11         System.out.print("\nIngrese primer String: ");
12         a = miEscaner.nextLine();
13         System.out.print("\nIngrese segundo String: ");
14         b = miEscaner.nextLine();
15         //compara los dos string
16         r = a.compareTo(b);
17
18         if (r == 0)
19             System.out.println("\n" + a + " y " + b + " son iguales");
20         else
21             if (r > 0)
22                 System.out.println(a + " es mayor que " + b);
23             else System.out.println(a + " es menor que " + b);
24     }
25 }
```

Nota: La clase String implementa **Comparable** por lo que provee el método `compareTo` que ya analizamos. Así, si en el ejemplo anterior la variable `a` se carga con la cadena "anillo" y `b` se carga con "casa", saldría en la consola el mensaje: "anillo es menor que casa".

La clase String provee aún otro método para comparar cadenas, que es útil cuando sólo se desea determinar si dos cadenas son iguales o no, sin importar el orden lexicográfico entre ambas. Se trata del método `equals()`, el cual es invocado por una de las cadenas y toma a la otra como parámetro. Retorna un boolean: true si las cadenas eran iguales, o false en caso contrario:

```
1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[]) {
6          String a, b;
7          int r;
8          Scanner miEscaner = new Scanner(System.in);
9
10         System.out.print("\nIngrese primer String: ");
11         a = miEscaner.nextLine();
12         System.out.print("\nIngrese segundo String: ");
13         b = miEscaner.nextLine();
14         //compara los dos string
15         if (a.equals(b) == true)
16             System.out.println(a + " y " + b + " son iguales");
17         else
18             System.out.println(a + " y " + b + " son diferentes");
19     }
20 }
```

Concatenación de cadenas

La clase String está marcada final, con lo cual no puede ser derivada. Pero un detalle interesante es que sus atributos principales también están marcados final, por lo cual son constantes: una vez que se asigna una cadena a un objeto String, ese valor no puede ser modificado. El siguiente es un extracto de la declaración de la clase String:

```
public final class String implements Serializable, Comparable, ...
{
    private final char value[]; // aquí se guarda la cadena...
    private final int offset;
    private final int count;
    ...
}
```

El lenguaje Java hace esto por razones de eficiencia: si se supone que el contenido de un String no puede cambiar, gestionará ese String de forma de hacer más rápido su acceso y menos costoso su mantenimiento en memoria (esto forma parte de la optimización en el uso de cadenas que citamos en el apartado anterior). Si se observa la documentación de la clase String, no se encontrará ningún método para cambiar el valor de ningún carácter contenido en la cadena.

Notar que el contenido de un String no puede modificarse, pero sí puede cambiarse el valor de la referencia. Supongamos las siguientes definiciones:

```
String a;
a = "casa"; //no podemos cambiar ninguna letra por otra...
a = "caza"; // pero sí podemos cambiar la referencia...
```

La primera asignación mostrada crea un objeto de la clase String, y lo apunta con la referencia a. La segunda asignación crea otro objeto de la clase, lo apunta con a, y al hacer eso deja des-referenciado al primer objeto... El contenido del objeto no puede modificarse, pero la referencia puede cambiarse. Si se desea crear un String por concatenación reiterada de otras cadenas, eso resulta ineficiente por el hecho de crear un nuevo objeto cada vez, reasignar la nueva referencia, y dejar disponible el viejo objeto para el garbage collector.

El fragmento siguiente carga los datos de n productos y sus importes, los acumula en una variable string para mostrarlos todos juntos al final:

```
1  import java.util.Scanner;  
2  
3  public class Principal {  
4  
5      public static void main(String args[]) {  
6          Scanner miEscaner = new Scanner(System.in);  
7          String nombre;  
8          float precio;  
9          int n;  
10         String res = "";  
11  
12         System.out.print("\nIngrese la cantidad de productos: ");  
13         n = miEscaner.nextInt();  
14  
15         for (int i = 0; i < n ; i++)  
16         {  
17             System.out.print("\nIngrese el nombre: ");  
18             nombre = miEscaner.nextLine();  
19             System.out.print("\nIngrese el precio: ");  
20             precio = miEscaner.nextFloat();  
21  
22             res += "\n"+ "Nombre: "+nombre+ ", Precio: "+precio;  
23         }  
24  
25         System.out.print("\nLos productos cargados son: ");  
26     }  
27 }
```

En cada vuelta del ciclo, se crea un objeto String, se suma al contenido anterior de res, y se reasigna el nuevo objeto en res, perdiendo la referencia al anterior. Esto efectivamente concatena todos los strings, pero lo hace de forma ineficiente: se pierde mucho tiempo en crear objetos nuevos, desreferenciarlos y dejarlos para el garbage collector.

Para evitar los problemas de eficiencia que genera el estatus final de los atributos de la clase String, existe también la clase StringBuilder que permite definir objetos que representan cadenas cuyos contenidos pueden modificarse sin tener que cambiar las referencias, en base a métodos que acceden al contenido y pueden agregar caracteres, concatenar, etc. El tamaño de un StringBuilder se va ajustando a las necesidades de la cadena que se está almacenando.

La clase StringBuilder implementa el método `toString()` de forma que al invocarlo, se retorna un objeto de la clase String con la cadena contenida en el StringBuilder original. El método `append()` de la clase StringBuilder, permite añadir al final de una cadena contenida en un objeto StringBuilder, otra cadena tomada como parámetro. En el programa anterior, lo reescribimos usando la clase StringBuffer en lugar de String para concatenar:

```

1  import java.util.Scanner;
2
3  public class Principal {
4
5      public static void main(String args[]) {
6          Scanner miEscaner = new Scanner(System.in);
7          String nombre;
8          float precio;
9          int n;
10         StringBuilder res = new StringBuilder("");
11
12         System.out.print("\nIngrese la cantidad de productos: ");
13         n = miEscaner.nextInt();
14
15         for (int i = 0; i < n ; i++)
16         {
17             System.out.print("\nIngrese el nombre: ");
18             nombre = miEscaner.nextLine();
19             System.out.print("\nIngrese el precio: ");
20             precio = miEscaner.nextFloat();
21
22             res.append("\n"+ "Nombre: "+nombre+ ", Precio: "+precio);
23         }
24
25         System.out.println("\nLos productos cargados son: "+res);
26     }
27 }
```

Planteado de esta forma, el resultado obtenido es el mismo, pero el proceso es más eficiente en cuanto al tiempo empleado y al uso de la memoria. Java posee también la clase StringBuffer idéntica a StringBuilder, pero que posee sus métodos sincronizados, por lo cual se la podemos usar de manera segura en un ambiente de multihilos, tema que no se tratará en este curso.

🔗 Anexo I: Uso de Lombok en Jerarquías de Herencia

Cuando trabajamos con herencia en Java y queremos aprovechar Lombok para reducir el código boilerplate, es importante tener en cuenta cómo se comportan las anotaciones como `@ToString`, `@EqualsAndHashCode`, `@Getter` y `@Setter` en relación a la superclase.

Por defecto, Lombok **NO incluye automáticamente los campos heredados** en los métodos `toString()`, `equals()` ni `hashCode()` que genera. Para incluirlos, debemos especificarlo con los atributos `callSuper = true`.

Ejemplo: `@ToString(callSuper = true)`

```

@Data
public class Cuenta {
    private int numero;
    private float saldo;
}

```java
@ToString(callSuper = true)

```

```
@EqualsAndHashCode(callSuper = true)
@Data
public class Inversion extends Cuenta {
 private float tasaInteres;
}
```

En este ejemplo:

- `@Data` genera los métodos `toString()`, `equals()` y `hashCode()`.
- `@ToString(callSuper = true)` agrega al `toString()` de `Inversion` los campos heredados de `Cuenta`.
- `@EqualsAndHashCode(callSuper = true)` hace lo mismo para `equals()` y `hashCode()`.

⚠ Si no se incluye `callSuper = true`, sólo se consideran los campos de la subclase.

## 💡 Recomendaciones

- Usar `callSuper = true` cuando la superclase tenga campos relevantes para la identidad o representación textual del objeto.
- Evitar `callSuper = true` si la superclase tiene muchos campos no significativos o si `equals()` debe estar desacoplado del padre.