Ficha 4

Polimorfismo – Clases Abstractas - Interfaces

1.] Introducción a la gestión de proyectos con Apache Maven.

A medida que un proyecto de software crece, en general se va aumentando la cantidad de archivos de código que lo componen y en consecuencia el proceso de compilar dicho código manualmente se vuelve mas y mas engorroso. Para solucionar este problema existen herramientas que automatizan el proceso de construcción del software a partir del código fuente (y por eso se llaman *herramientas de automatización*). Estas herramientas permiten compilar el programa a partir del código fuente, administrar dependencias externas e inclusive empaquetar el programa en un un instalador.

Si bien es cierto que la mayoría de los programadores usan entornos de desarrollo que hacen esta tarea, muchas veces es necesario poder realizar una compilación en forma automática, por ejemplo en proyectos que usan integración continua para ejecutar tests de calidad con cada cambio que se realiza al programa. Además, *el uso de una herramienta de automatización permite no depender de un entorno de desarrollo (IDE) en particular, dando la libertad a que cada programador haga uso del entorno que más le guste*. Dentro de las *herramientas de automatización* utilizadas en proyectos Java se encuentran por ejemplo: *Apache Ant*, *Apache Maven* y *Gradle*.

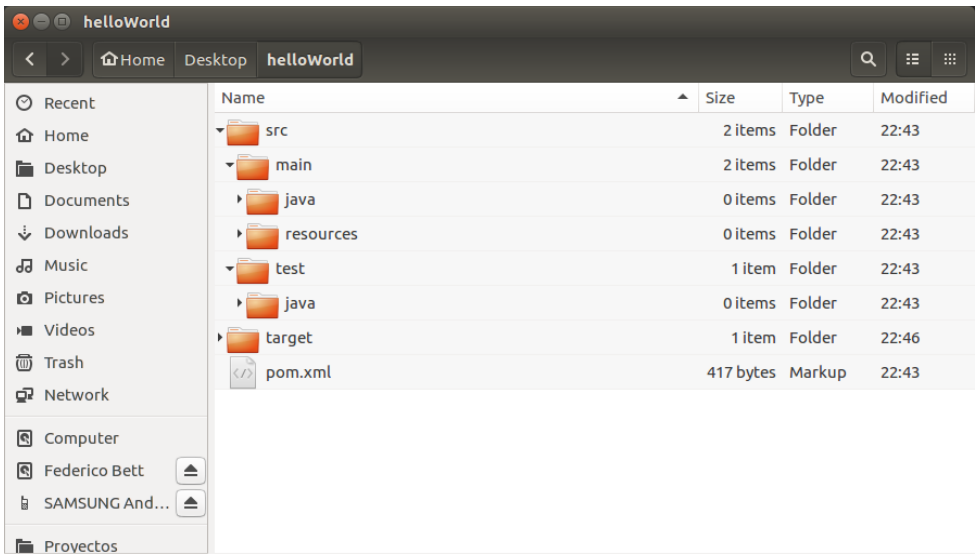
La que nos interesa y aplicaremos en el curso es ***Apache Maven***. Se trata de una herramienta de automatización de construcción de proyectos para Java, de tipo *open source*, mantenida por la *fundación Apache* y usada ampliamente por la comunidad Java. Tanto el código fuente, como los binarios, disponibles para Windows, Linux y MacOS, se pueden descargar desde la URL: https://maven.apache.org/.

Maven cumple principalmente dos funciones: describir el software y su proceso de construcción, y describir las dependencias de un programa automatizando su obtención. Para ello utiliza un archivo XML, llamado *pom.xml*, que se ubica en la raíz del proyecto y donde se detalla la configuración. Uno de los principios que tiene *Maven*, es que *la convención es preferible por sobre la configuración*, lo que quiere decir que si se respetan las convenciones que tiene Maven, la configuración a realizar es mínima y sólo es necesario detallar explícitamente las desviaciones de la convención.

Una vez definido el contenido del archivo *pom.xml*, *Maven* provee una *interfaz de línea de comandos* que permite compilar, limpiar, empaquetar y ejecutar los *unit tests* entre otras cosas. Todo se realiza a través del comando *mvn* al cual se le pasan parámetros de acuerdo a la acción que uno quiera realizar.

Para utilizar *Maven* en forma mínima no hace falta demasiado trabajo. *Siempre y cuando se respete la estructura de carpetas que espera Maven*, un par de líneas de configuración son suficientes para poder compilar un conjunto de archivos fuente.

Parte de las convenciones que propone *Maven* es la definición de una estructura de carpetas donde uno debería ubicar los archivos Java. Si bien es posible usar la estructura que uno quiera, si no se sigue la convención es necesario detallar las ubicaciones de los archivos de código en el archivo *pom.xml*. La convención de *Maven* es la siguiente:



Los archivos de código fuente se van a ubicar en la carpeta *src/main/java*, y los archivos de recursos (o sea archivos de configuración, o imágenes) se deberían ubicar en la carpeta *src/main/resources*. Las carpetas equivalentes ubicadas en *src/test* están destinadas al código y recursos de los *tests unitarios*.

El contenido del archivo *pom.xml* es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>TSB</groupId>
    <artifactId>helloWorld</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Las cinco primeras líneas son parte del esquema mínimo necesario y en general son siempre iguales. Las siguientes tres líneas describen el proyecto a construir. En ellas se detalla lo siguiente:

- **groupId**: Este valor representa el nombre de la organización que está creando el software, en general tiene una forma similar a una url, pero con los componentes en el orden contrario, como por ejemplo org.apache.
- **artifactId**: Este valor es el identificador del software dentro de la organización.

- **version**: El número de versión del software que se está construyendo. Generalmente se usan entre dos y cuatro números separados por un punto. Si la versión incluye el sufijo SNAPSHOT significa que es una versión de desarrollo y se espera que no sea estable.

Una vez que se tiene el archivo *pom.xml* y el código se puede usar la interfaz de línea de comandos para poder realizar las acciones provistas por *Maven*. Como se dijo anteriormente el nombre del ejecutable que provee esta funcionalidad es *mvn* y para poder usarlo tiene que estar incluido en la variable de entorno *PATH*, de forma similar a lo que se necesita para ejecutar Java. Algunas de las acciones provistas por *Maven* son:

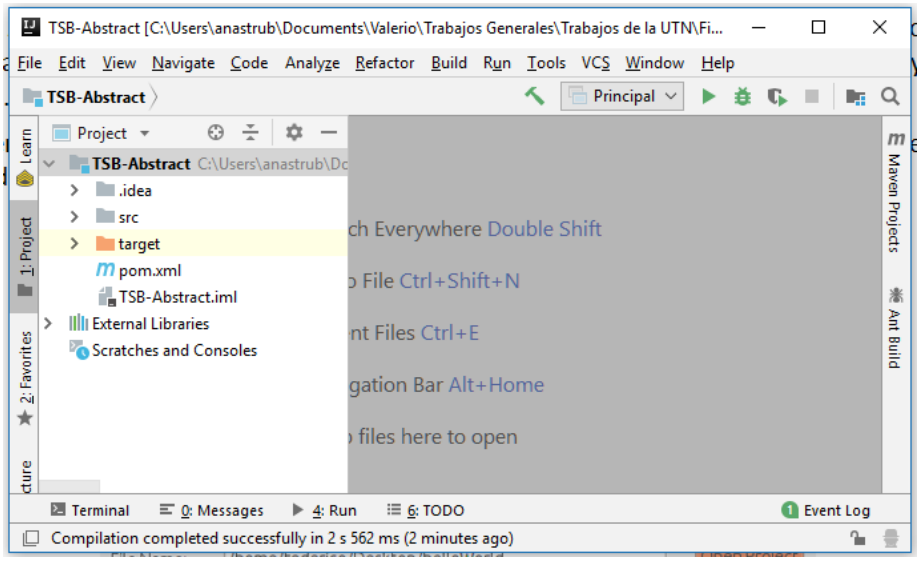
- ✓ **compile**: compila todo el código del proyecto. En general, todos los archivos de compilados generados por maven son ubicados dentro de una carpeta llamada target
- ✓ **clean**: borra todos los archivos generados en el proceso de compilación, y elimina la carpeta target
- ✓ **test**: Compila y ejecuta los unit test del proyecto.
- ✓ **package**: compila y empaqueta el código en un archivo .jar, el archivo generado se va a ubicar en la carpeta target.

Un ejemplo de compilación usando *Maven*:

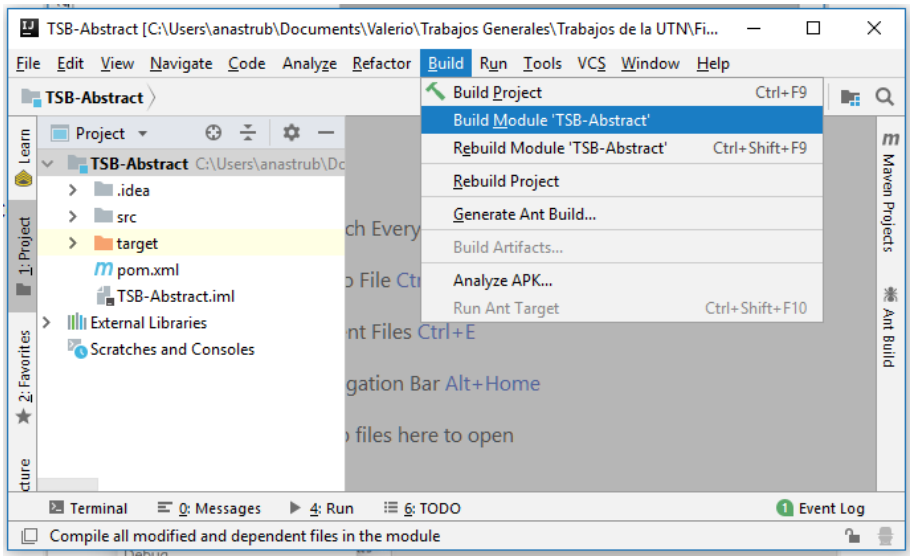
```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building helloWorld 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @
helloWorld ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered
resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.2:compile (default-compile) @ helloWorld
---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding UTF-8,
i.e. build is platform dependent!
[INFO] Compiling 1 source file to
/home/federico/Desktop/helloWorld/target/classes
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 2.310 s
[INFO] Finished at: 2017-08-14T07:43:28-03:00
[INFO] Final Memory: 13M/199M
[INFO] ------------------------------------------------------------
```

Si bien *Maven* principalmente se usa desde la línea de comandos, la mayoría de los entornos de desarrollo permiten abrir el archivo *pom.xml* directamente, como si fuera un proyecto del IDE. Luego se podrán ejecutar las acciones de *Maven* desde el propio entorno.

Por ejemplo para el caso de IntelliJ, se puede abrir el *pom.xml* como un proyecto (nótese el ícono de *Maven* a la derecha de la ventana del IDE). En otras palabras, un proyecto Maven puede abrirse con cualquier IDE que tenga instalados los plugins de Maven o que sea directamente compatible con Maven (este último es el caso de IntelliJ):



Y las acciones se pueden ejecutar desde un menú contextual del proyecto:



La compilación o ejecución del proyecto se pueden realizar normalmente como si fuera un proyecto Java normal.

2.] Polimorfismo: introducción y conceptos básicos.

Una referencia definida para apuntar a objetos de la clase base de una jerarquía, sirve para referenciar objetos de cualquier clase de esa jerarquía. Y en particular, una referencia que se

defina para apuntar a objetos de cualquier clase que tenga derivadas, podrá usarse para apuntar a objetos de esa clase o de cualquiera de sus derivadas.

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 *TSB [Polimorfismo]*, 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 definida como *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 *TSB [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("Valores originales: ");
System.out.println("a: " + a.toString());  // toString() de Cuenta
System.out.println("b: " + b.toString());  // toString() de Inversion
System.out.println("c: " + c.toString());  // toString() de Corriente
```

Cuando desde una referencia polimórfica se invoque a un método, la Máquina Virtual Java (JVM) no tendrá problemas para invocar la versión correcta del mismo, siempre y cuando la primera definición de ese método aparezca en la clase base de la jerarquía analizada. 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 modelos *TSB [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: actualizar() no está definido en la clase base

Inversion x = (Inversion) b;  // casting explícito: x apunta al mismo objeto que b
x.actualizar();  // ahora sí: x es de tipo Inversion
```

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:

- Si se desea desde *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 hacer casting explícito con la referencia antes de poder hacerlo, para evitar el error de compilación:

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

Finalmente, si en algún momento queremos determinar la clase a la que pertenece el objeto apuntado por una referencia polimórfica, 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())
```

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 a Inversion.toString()

Object y = "casa"; // es lo mismo que Object y = new String("casa");
System.out.println(y); // invoca a String.toString()
```

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 *TSB [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];  // arreglo de referencias polimórficas.
```

```
// 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 tipo *Inversion*, y a esas cuentas actualizarles 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(); // 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("Objetos de la misma clase que el primero");
Cuenta este = v[0];
for(int i=0; i<4; i++)
{
   if(v[i].getClass() == este.getClass())
   {
      System.out.println("v[" + i + "]: " + v[ i ].toString());
   }
}
```

3.] Clases Abstractas.

En muchos casos una clase se diseña pero no para ser instanciada, sino para permitir el agrupamiento de características comunes, facilitar le herencia, y posibilitar el polimorfismo.

Por ejemplo, tomando el caso del modelo *TSB [Abstract]* en el que 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 permita instanciar clases que cumplan esos requisitos de abstracción, declarando a dichas clases como *abstractas* (mediante la palabra reservada ***abstract***) 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(); // esto sí compila... si no, perdemos el
polimorfismo...
```

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

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

Por contraposición, las clases "no abstractas", se dicen *clases concretas*, y pueden ser instanciadas 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 (modelo *TSB [Abstract]*), hemos marcado 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);
```

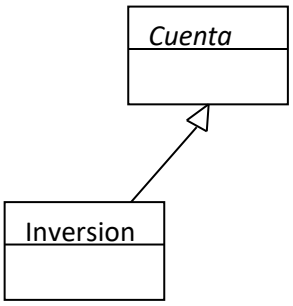
Esto significa que la clase *Cuenta* obliga a sus derivadas a contar con ese método, y también las obliga a redefinirlo (si esas derivadas no son abstractas a su vez). La clase *Cuenta* no necesita saber cómo se implementa la operación de retiro de fondos, pues esa operación es muy particular de cada 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.

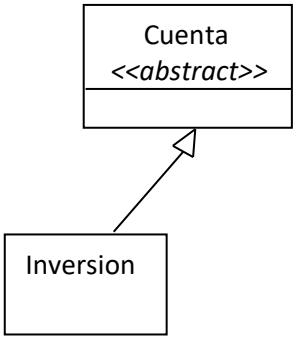
Notar que la clase *Inversion* hereda de *Cuenta*, que es abstracta y provee un método abstracto *retirar()*. La clase *Inversion* debe redefinir ese método. Al hacerlo, la palabra *abstract* no debe volver a escribirse (pues el método de otro modo volvería a marcarse abstracto, obligando a la clase 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 UML hay dos formas de representar una clase abstracta: la primera *es poner en cursiva el nombre de la clase* en el gráfico que la representa:



Y la otra es usar un *estereotipo*: se agrega la palabra *abstract* en el ícono de la clase, encerrada entre los símbolos << y >>:



4.] Clases de Interface.

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 asignadas en cada casillero las direcciones de los objetos requeridos, el arreglo puede procesarse en forma normal, recorriéndolo 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 retornado:

```
r == 0     los objetos a y b eran iguales
r > 0      el objeto invocante era mayor al parámetro: a > b
r < 0      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 (aunque a partir de la versión Java 8 pueden contener también métodos concretos si se declaran precedidos de la palabra reservada *default*).

Una interface 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 derivan: *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:

```
public class Cliente implements Comparable
```

Una clase en Java sólo puede derivar desde una única clase (herencia simple), pero puede implementar 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. Asi, 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 aplicarse polimorfismo entre clases que originalmente no forman parte de la misma jerarquía de herencia, a condición 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).

En el proyecto *TSB [Comparable]* mostramos una clase *Array* que implementa en forma sencilla las ideas que acabamos de exponer. La clase tiene un atributo *v* que se declara como una referencia a un arreglo que a su vez contiene referencias de tipo *Comparable*. Los dos constructores de la clase crean el arreglo invocando a *new*, y luego se ofrecen un par de método *get()* y *set()* que permiten, respectivamente, recuperar el objeto en la posición *i*, y asignar un objeto en la posición *i*. La clase finalmente posee dos métodos *sort()* y *toString()* para ordenar el arreglo de menor a mayor y obtener la conversión a *String* del vector.

El método *sort()* es el que realmente aprovecha la presencia del método *compareTo()* en cada objeto almacenado en el vector. En el mismo modelo *TSB [Comparable]*, la clase *Cuenta* (abstracta y base de una jerarquía de clases que modelan cuentas bancarias), implementa la interface *Comparable*, y se desarrolla dentro de ella el método *compareTo()*:

```
// método compareTo() de la clase Cuenta...
public int compareTo (Object x)
{
   Cuenta p = (Cuenta) x;
   return numero - p.numero;
}
```

La idea es que el método retorne 0 si los objetos comparados son iguales, o retorne un número negativo si el objeto que invocó al método es "menor" que el que entró como parámetro, o un número positivo si el primero es "mayor" que el segundo. Eso podría hacerse con un par de condiciones, pero es claro que si se hace la resta entre los valores que se quiere comparar, los números obtenidos serán exactamente los que se esperaría obtener. Si al comparar dos objetos de la clase *Cuenta* se quisiera hacer esa comparación usando los números de las cuentas, entonces la resta entre ambos números se ajustaría lo esperado por *compareTo()*. Note que a la referencia *x* que entra como parámetro se le aplica un operador de *casting explícito* para obtener una referencia *p* de tipo *Cuenta*, ya que de otra forma la expresión *x.numero* no compilará: *x* es de tipo *Object* pero el atributo *numero* no existe en esa clase…

De la clase *Cuenta* derivan las clases *Corriente* e *Inversión*, y como a ambas el método definido en *Cuenta* les sirve, ambas usan ese mismo. La clase *Cliente* (también provista en el modelo) implementa a su vez *Comparable* y provee su propia versión del método *compareTo()*:

```
// método compareTo() de la clase Cliente...
public int compareTo(Object x)
{
   Cliente c = (Cliente) x;
   return this.getDni() - c.getDni();
}
```

Las dos clases puede ser usadas para definir objetos que luego puedan ser incluidos dentro de uno de nuestros arreglos.

Notar que si la idea fuera que en la clase *Cuenta* el método *compareTo()* sea a su vez abstracto, dejando la implementación del mismo para las clases derivadas, no habría problema porque la clase *Cuenta* es abstracta. Si una clase es abstracta no está obligada a implementar los métodos de una interface que implemente. Lo único que debemos hacer, es que la clase *Cuenta* no nombre siquiera al método, y lo "acepte" tal como viene de *Comparable*. Si este es el caso, la situación sería la siguiente:

- Una interface es en esencia una clase abstracta con todos sus métodos también abstractos (el hecho de que la versión Java 8 permita incluir métodos concretos en una interface no modifica este análisis).
- La clase *Cuenta* implementa *Comparable*, por lo tanto debería incluir y redefinir el método *compareTo()*.
- Pero la clase *Cuenta* es también abstracta, y si hereda un método abstracto (desde una clase o por implementar una interface) puede simplemente no redefinirlo y ni siquiera nombrarlo.

- Por lo tanto, si en la clase *Cuenta* se desea dejar sin implementar el método *compareTo()* y mantenerlo abstracto para obligar a cada clase derivada a redefinirlo, simplemente nos limitaríamos a no nombrarlo...

Por otra parte, notemos que muchas clases predefinidas del lenguaje Java implementan la interface *Comparable*. La clase *String* es una de ellas: esta clase provee una versión del método *compareTo()* que realiza una comparación lexicográfica entre dos cadenas, de forma que siempre es posible preguntar si una cadena es mayor, igual o menor que otra. Intuitivamente, se entiende que una cadena *cad1* es menor que otra *cad2*, si *cad1* aparece antes que *cad2* en un diccionario. Y eso es lo que testea el método *compareTo()* de la clase *String*:

```
String cad1, cad2;

// suponemos que se asignan valores en estas variables
cad1 = ...
cad2 = ...

int r = cad1.compareTo(cad2);
if ( r == 0) // son iguales...
if ( r < 0 )  // cad1 es menor que cad2...
if ( r > 0 )  // cad1 es mayor que cad2...
```

Por lo tanto, podemos crear instancias de nuestra clase *Array* y almacenar en ella objetos de la clase *String* o de cualquier otra clase de Java que haya implementado *Comparable*. Mostramos un ejemplo en el método *main()* de la clase *Principal* del modelo *TSB [Comparable]*.