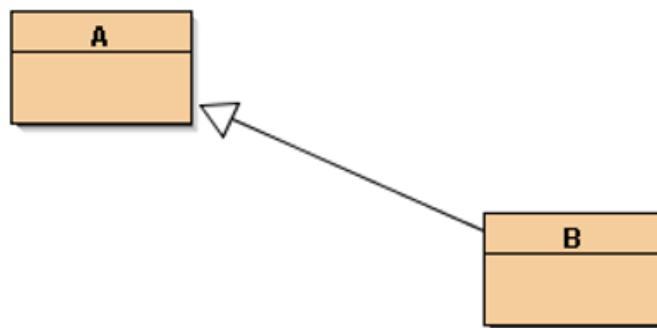


Apunte 08 - Herencia y Polimorfismo en Java

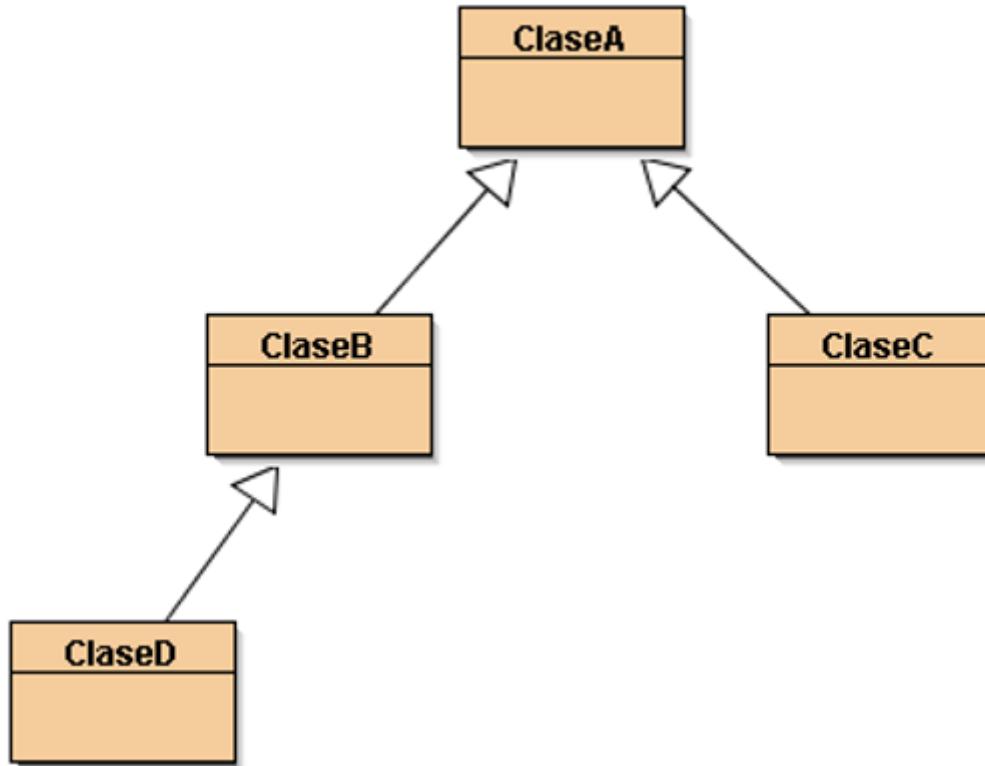
Herencia en Java

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):



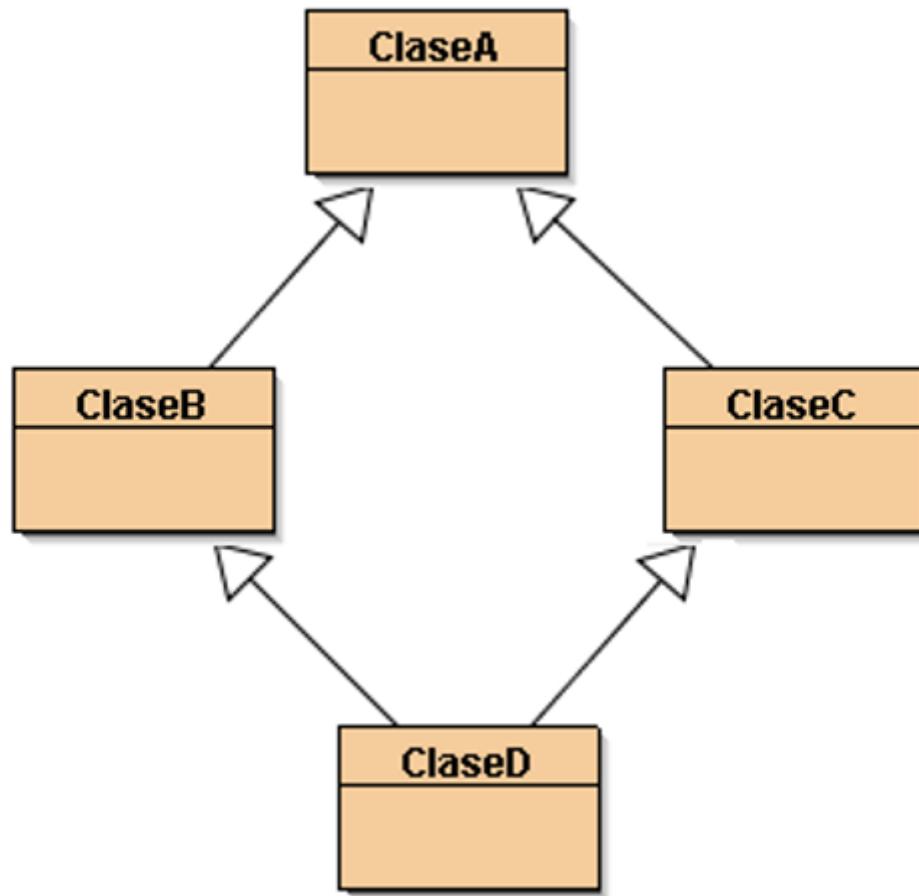
La clase desde la cual se hereda, se llama super clase, y las clases que heredan desde otra se llaman subclases o clases derivadas: de hecho, la herencia también se conoce como derivación de clases. Una jerarquía de clases es un conjunto de clases relacionadas por herencia. La clase en la cual nace la 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 deseé. 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

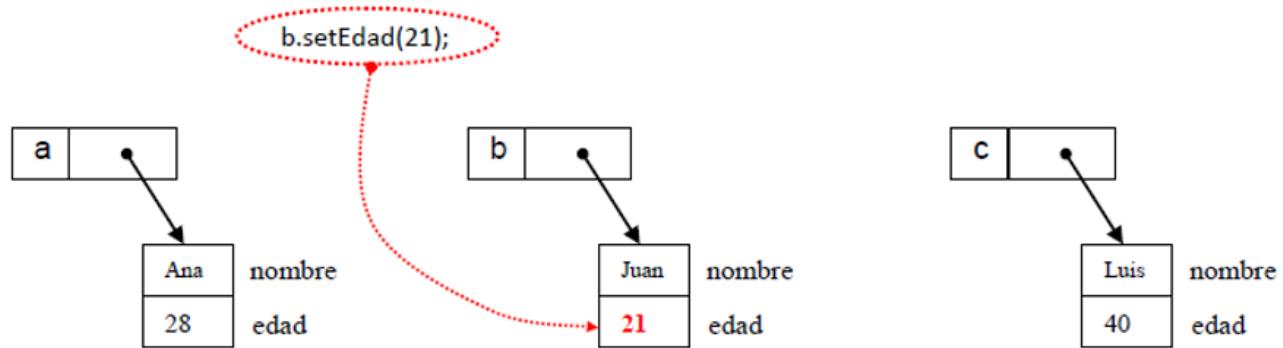
Sabemos que los distintos miembros (atributos y métodos) de una clase pueden marcarse como public o private (entre otros estados) según se desee imponer con más o menos fuerza el Principio de Ocultamiento. Sin embargo, existen otras palabras reservadas que pueden usarse para indicar la forma de acceder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

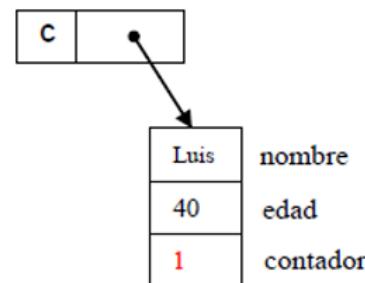
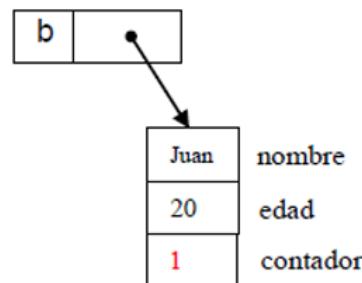
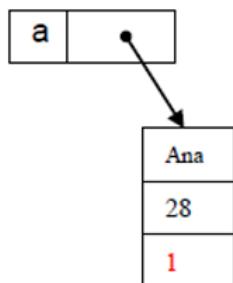
```

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);

```

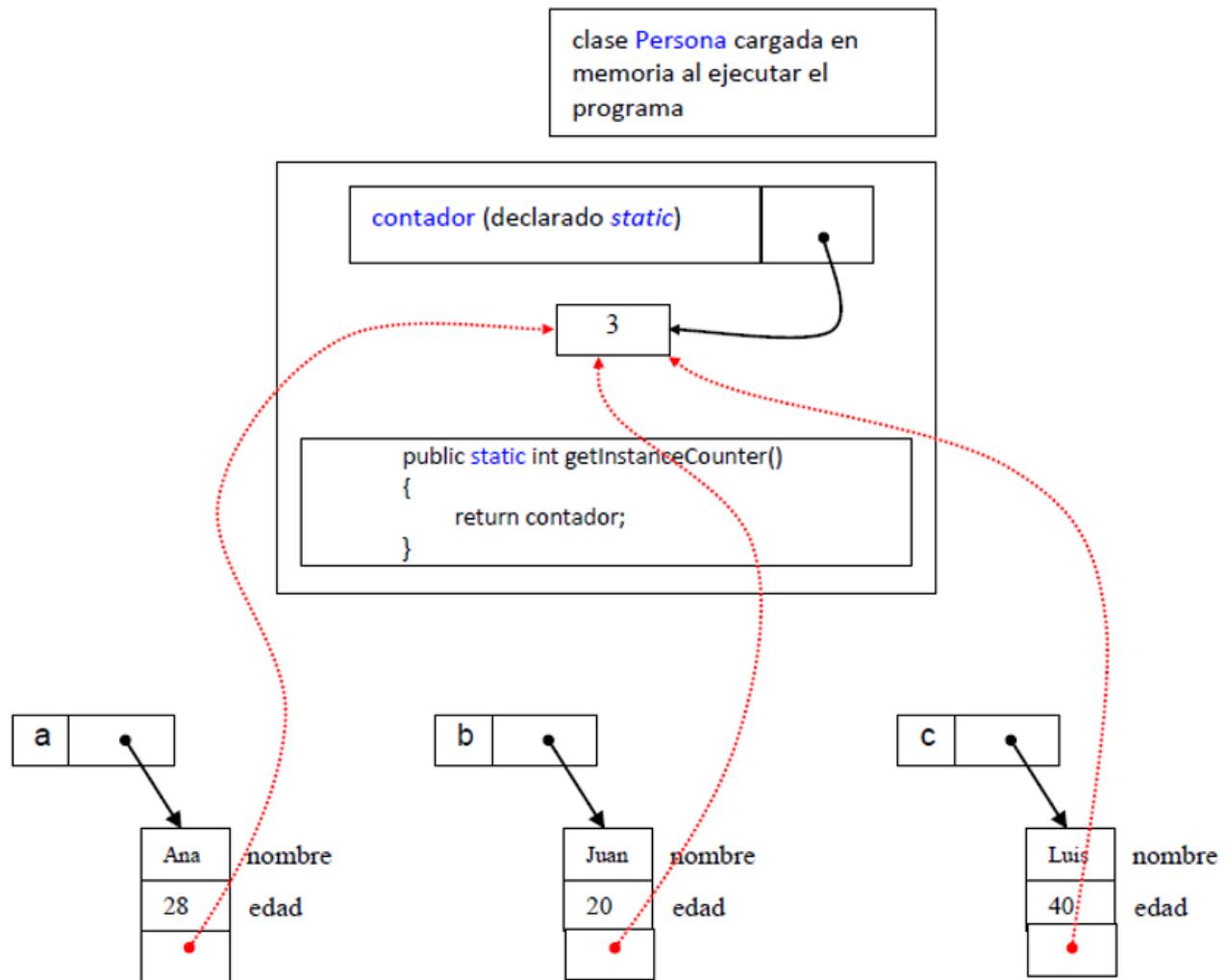


La solución sería hacer que el contador sea único: no que cada instancia tenga su propia copia, sino que exista una copia única de ese contador y que todas las instancias hagan referencia a esa única copia. Esto puede lograrse si el contador se declara static:

```
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 returnedo:

- $r == 0 \Rightarrow$ 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.

# Apunte 09 - Iteradores y Colecciones en Java

---

## Introducción

A continuación vamos a comenzar el proceso de analizar las alternativas de implementación de estructuras de datos en java al nivel que requerimos en Backend de Aplicaciones para comprender el funcionamiento de las colecciones nativas de Java y poder aprovechar su potencia pero sin profundizar en los conceptos reales de la programación de estructuras de datos no nativas que sería material para otra asignatura completa.

En pocas palabras, el objetivo de programar una estructura de datos es independizar el concepto que muestra la interfaz de los objetos de dicha estructura del almacenamiento subyacente y real de los datos. Es decir que vamos a encapsular dentro de una clase un esquema de almacenamiento de datos de forma tal que el usuario de las instancias de la clase no se tenga que preocupar por dicho almacenamiento pero que pueda utilizar el objeto como si fuera el concepto que implementa.

Para esto vamos a implementar una lista sobre un vector o array en Java, como ya hemos discutido cuando analizamos arrays o vectores estos tiene ciertas particularidades que los hacen hasta cierto punto estáticos ya que una vez creados no pueden cambiar su tamaño y agregan la particularidad de requerir todos los elementos del mismo tipo. Una lista en cambio es una estructura de datos que agrega elementos a medida que se requiere y no tiene un tamaño fijo, mantiene el concepto de orden de inserción y el de estructura lineal, es decir que un elemento le sigue a otro y cada elemento tiene 0 o solo 1 elemento siguiente o anterior.

La estructura de datos natural para implementar este concepto son las listas vinculadas, pero no corresponde a esta asignatura el análisis del desarrollo de las listas vinculadas (fuera que luego las usemos en Java), por lo que vamos a implementar el concepto de la lista pero utilizando como repositorio de datos interno los vectores que ya conocemos.

## Lista sobre vectores o Arrays

Vamos entonces con la clase que mencionamos:

© ListaArrayBasica	
(m) <code>ListaArrayBasica (int )</code>	
(m) <code>ListaArrayBasica ()</code>	
(f) <code>elementos</code>	Object []
(f) <code>capacidadInicial</code>	int
(f) <code>count</code>	int
(f) <code>actual</code>	int
(m) <code>insertar (int , Object )</code>	void
(m) <code>get(int )</code>	Object
(m) <code>iniciarIterador ()</code>	void
(m) <code>hayMas ()</code>	boolean
(m) <code>siguiente ()</code>	void
(m) <code>contains (Object )</code>	boolean
(m) <code>agregar (Object )</code>	boolean
(m) <code>clear ()</code>	void
(m) <code>asegurarCapacidad (int )</code>	void
(m) <code>toString ()</code>	String
(m) <code>getActual ()</code>	Object
(m) <code>set(int , Object )</code>	Object
(m) <code>quitar (int )</code>	Object
(m) <code>size ()</code>	int
(m) <code>isEmpty ()</code>	boolean

En este caso el problema fundamental radica en ocultar las desventajas de los vectores al usuario de la lista, es decir el usuario de la lista debe poder utilizar la misma como si fuera una lista de Python o Javascript que no tiene limitación en cuanto a la cantidad, y un detalle extra que vamos a notar es que deberemos trabajar en todos los casos en base a Object para lograr que nuestra ListaArray pueda contener objetos del tipo que haga falta.

En primer lugar analicemos atributos:

```
9 public class ListaArrayBasica {
10 // el arreglo que contendrá los elementos...
11 private Object[] elementos;
12
13 // el tamaño inicial del arreglo...
14 private int capacidadInicial;
15
16 // la cantidad de casillas realmente usadas...
17 private int count;
18
19 // índice de la posición actual del iterador
20 private int actual;
21
22
```

El primer atributo que vemos es el vector de elementos donde vamos a almacenar los objetos que sean agregados a la lista. Este vector es lo que hemos denominado almacenamiento subyacente y el usuario no tendrá contacto con él.

El segundo parámetro que vemos es la capacidad inicial de dicho vector y nos servirá para volver al estado inicial la lista usando el método `clear()`

El siguiente es el parámetro que indica la cantidad de posiciones del vector utilizadas actualmente y aquí vamos a aclarar una situación. Evidentemente no podemos crear el vector vacío y agrandarlo cada vez que el usuario desee agregar un nuevo elemento porque eso provocaría un importante overhead de trabajo al copiar cada vez que recreamos el vector los elementos del vector viejo al vector nuevo, esto se debe a que los vectores en java no permiten de manera natural redimensionar su contenido. Retomaremos este tema más adelante.

Definimos 2 constructores:

```

22
23 > /** Crea una lista con capacidad inicial de 10 casilleros, pero ninguno ...*/
24 new *
25
26 public ListaArrayBasica() {
27
28 this(capacidad: 10);
29 }
30
31
32 > /** Crea una lista con capacidadInicial casilleros de capacidad, pero ninguno ...*/
33 new *
34
35 public ListaArrayBasica(int capacidad) {
36 if (capacidad <= 0) {
37 capacidad = 10;
38 }
39 elementos = new Object[capacidad];
40 capacidadInicial = capacidad;
41 count = 0;
42 }
43
44
45
46
47

```

El primero no recibe parámetros y el segundo recibe la capacidad inicial, este último está orientado a que si sabemos que vamos a manejar gran cantidad de datos, creamos la lista ya con la cantidad aproximada para evitar o reducir la cantidad de veces que hará falta redimensionar el vector subyacente.

Notar también que ambos constructores además del vector y la capacidad inicial, establecen el valor de count en cero lo que indica que la lista inicia vacía.

Ahora bien vamos con el comportamiento fundamental que implica agregar un elemento, este comportamiento es homólogo al append de Python que tantas veces se usa en AED.

```

47
48 > /** Añade al final de la lista el objeto e. La inserción será rechazada si la ...*/
49 new *
50
51 public boolean agregar(Object e) {
52 if (e == null) return false;
53
54 if (count == elementos.length) this.asegurarCapacidad(capacidadMinima: elementos.length * 2);
55
56 elementos[count] = e;
57 count++;
58 return true;
59 }
60
61
62
63
64

```

El método valida que el objeto a agregar no sea null y luego, si el mundo fuera ideal y los vectores infinitos, se limita a agregar el elemento en la posición que indique count (como el primer subíndice siempre será 0, la cantidad siempre indicará el próximo subíndice a ser utilizado, esta realidad merece una revisión por parte del alumno en caso de que no esté clara). Eso producen las líneas 60 y 61.

Pero, como el mundo nunca es ideal y los vectores en Java no son infinitos, la línea 58 es la que determina si en el vector subyacente aún hay espacio para agregar el elemento o no, cuando la cantidad actual de elementos del vector es igual al tamaño del vector subyacente quiere decir que el vector subyacente está completamente ocupado y por lo tanto que no hay espacio para agregar un nuevo elemento. En este caso será necesario redimensionar el vector y esa es la función que cumple el método asegurarCapacidad:

```
126 private void asegurarCapacidad(int capacidadMinima) {
127 if (capacidadMinima == elementos.length) return;
128 if (capacidadMinima < count) return;
129
130 Object[] temp = new Object[capacidadMinima];
131 System.arraycopy(elementos, srcPos: 0, temp, destPos: 0, count);
132 elementos = temp;
133 }
```

Este método, luego de algunas validaciones, crea un nuevo vector más grande que el anterior, notar que no agrega una sola posición sino que agrega una cantidad de elementos libres para no tener que volver a ejecutar este método ante cada nuevo elemento atravesado el umbral de capacidad inicial.

Una vez creado el vector la sentencia `System.arraycopy(elementos, 0, temp, 0, count);` copia el contenido del vector actual al primer segmento del nuevo vector quedando así nuevos elementos no utilizados. Finalmente se reemplaza el vector subyacente con el nuevo vector y al volver al método agregar este entiendo que el vector tiene elementos libres y agrega el nuevo elemento como si nada hubiera pasado.

A partir de aquí la clase implementa varios métodos para dar soporte a distintas acciones en la lista y las herramientas al usuario para poder usarla, no vamos a describir cada uno de los métodos queda para el alumno revisar estos y consultar en clase si queda alguna duda al respecto.

## Probando la Lista sobre Array básica

En la clase `PruebaListaArray` hay bastante código probando la Lista descrita en el punto anterior donde se puede observar el uso de los comportamientos que hemos mencionado y también el del resto de los comportamientos desarrollados en la clase.

Ahora bien, en muchos problemas será necesario tomar una lista y recorrerla para obtener algún resultado relevante. Por ejemplo, si una aplicación crea una lista de cuentas de Inversion se podría esperar que en algún momento se nos pida el saldo promedio entre todas las cuentas que están en la lista. El método que recorra la lista, tome y acumule el saldo de cada cuenta y finalmente retorne el promedio, no debería formar parte de la clase `ListaArray`, tal método estaría suponiendo que la lista tiene objetos de la clase `Inversion` y eso sería falso en general, la clase `ListaArray` fue planteada para suponer elementos polimóficos. Ese método sólo sería utilizable si efectivamente se crea una instancia de la clase `ListaArray` en la que se inserten objetos que representen Cuentas de Inversion.

Si bien como sabemos que en esta implementación existe un vector subyacente la opción de hacer el recorrido con un `for` usando el método `size()` que retorna la cantidad y el método `get(i)` que devuelve el elemento en la posición `i` podríamos lograr un recorrido de la lista desde afuera del objeto. En este ejemplo de la clase `PruebaListaArray`, recorremos una lista en la que hemos agregado cadenas con nombres:

```
61 System.out.println("Contenido usando get: ");
62 for (int i = 0; i < la.size(); i++)
63 {
64 String x = (String) la.get(i);
65 System.out.println(x);
66 }
67 }
```

sin embargo, nos estamos basando en que conocemos la estructura de almacenamiento y que también conocemos que soporta acceso directo con lo que usar `get(i)` no supone un overhead de trabajo, cosa que no pasa con otras estructuras de datos de soporte subyacente. Por lo tanto será necesario encontrar un esquema que permita recorrer el contenido de la lista sin tener ningún conocimiento de su estructura o almacenamiento interno y esa situación se resuelve utilizando el patrón iterador.

En una primera versión absolutamente simplificada y trivial del patrón pensemos en cuáles son los comportamientos obligatorios que requeriríamos al intentar agregarla a nuestra lista la capacidad de iteración:

- Un comportamiento para reiniciar el iterador que permita resetear este al primer elemento de la lista.
- Un comportamiento que permita saber si hay más elementos de forma de poder saber si debemos seguir recorriendo o debemos detener el recorrido porque se terminó el contenido de la lista.
- Un comportamiento que permita obtener el elemento actual del recorrido.
- Un comportamiento que permita avanzar al elemento siguiente en el recorrido.

Si nuestra clase soporta esos comportamiento podríamos pensar en un recorrido de la lista de esta forma:

```
67
68 System.out.println("Contenido usando iterador: ");
69 la.iniciarIterador();
70 while(la.hayMas())
71 {
72 String x = (String) la.getActual();
73 System.out.println(x);
74 la.siguiente();
75 }
76 }
```

notar que en este caso no tenemos conocimiento de la estructura interna de la lista, de hecho podría estar implementada incluso sobre una estructura no lineal y de todas maneras nosotros podemos recorrer todos sus elementos de forma lineal sin preocuparnos por su estructura interna.

Primera versión trivial del iterador

```

231 /**
232 * Metodo que inicia la posibilidad de iteración a partir del primer elemento del vector
233 */
234 public void iniciarIterador() {
235 actual = 0;
236 }
237
238 public boolean hayMas() {
239 if(isEmpty()) { return false; }
240 if(actual >= size() - 1) { return false; }
241 return true;
242 }
243
244 public Object getActual() {
245 if (actual >= count) {
246 throw new NoSuchElementException("next(): no quedan elementos por recorrer...");
247 }
248 return elementos[actual];
249 }
250
251 public void siguiente() {
252 if (!hayMas()) {
253 throw new NoSuchElementException("next(): no quedan elementos por recorrer...");
254 }
255 actual++;
256 }
257

```

Como vemos no hay mayor complejidad en la implementación del iterador directamente escrito como métodos de la clase, cabe aclarar que estamos usando aquí el último de los atributos que no explicamos al principio, el atributo actual contiene el índice del elemento actual en la iteración.

Por ello el método `iniciarIterador` pone el valor 0 en el atributo actual para que apunte al primer elemento de la lista. Luego el método `getActual()` retorna el elemento que está en la posición indicada por el atributo actual. Los métodos `hayMas()` y `siguiente()` también se apoyan en dicho atributo para determinar si hay más contenido por recorrer y avanzar al siguiente elemento.

Esta podría ser una solución pero tiene un problema particular y es que si lo implementamos de esta forma solo podemos plantear un recorrido por vez, ya que si estamos en medio de un recorrido y queremos volver a iniciar por ejemplo para realizar una búsqueda implicaría modificar el recorrido actual y ese problema hace que la solución no sea viable.

¿Cómo resolvemos esto entonces?, la respuesta a esta pregunta es **otro** objeto capada de llevar a cabo la iteración de la lista, pero, si programamos otra clase para que itere la lista esta nueva clase tendría que conocer los detalles internos de la clase `ListaArray` y eso rompería el encapsulamiento, la solución a esta situación está dada, en Java, por las clases internas.

## Clases internas en Java

Una clase interna es una clase que se define dentro del ámbito de otra clase, como un miembro más. Así, una clase que contenga una o más clases internas tendrá miembros que serán o bien atributos propios, o bien métodos propios, o bien clases internas. No hay demasiada novedad en una clase interna, de hecho es en todo similar a una clase común, sin embargo, bastará con saber que si una clase B se declara como interna de otra clase A, entonces los métodos de la clase B pueden acceder en forma directa a todos los atributos y métodos de A (sin importar si se declararon `private` o no). Esto tiene sentido si se considera que la clase B es

miembro de A, y por lo tanto tiene acceso a los otros miembros de A (de la misma forma que los métodos de A tienen acceso a los atributos de la misma clase A).

Por lo tanto, y aplicando lo dicho, en el modelo la clase [ListaArrayMejorada](#) declara como clase interna a la clase [IteradorLineal](#) de acuerdo al siguiente esquema:

```

236 public class IteradorLineal {
237
238 // índice de la posición actual del iterador
239 private int actual;
240 public IteradorLineal() {
241
242 actual = 0;
243 }
244
245 public boolean hayMas() {
246 if(isEmpty()) { return false; }
247 if(actual >= size() - 1) { return false; }
248 return true;
249 }
250
251 public Object getActual() {
252 if (actual >= count) {
253 throw new NoSuchElementException("next(): no quedan elementos por recorrer...");
254 }
255 return elementos[actual];
256 }
257
258 public void siguiente() {
259 if (!hayMas()) {
260 throw new NoSuchElementException("next(): no quedan elementos por recorrer...");
261 }
262 actual++;
263 }
264 }
265 }
```

Ahora tenemos el iterador de forma tal que permite generar tantos objetos iteradores independientes como haga falta y que además pueden acceder a los atributos de la clase [ListaArray](#) sin romper el encapsulamiento. el problema es que vamos a necesitar también agregar en la clase lista la capacidad de obtener un iterador de esta para poder utilizarlo, esa función la cumple el método:

```

227 public IteradorLineal iterador() {
228 return new IteradorLineal();
229 }
230 }
```

Que devuelve una instancia de [IteradorLineal](#) para la clase [ListaArray](#) y a partir del cual podemos realizar el mismo recorrido que hicimos con el iterador básico con mínimas diferencias, el recorrido quedará como sigue:

```
37 System.out.println("Contenido usando iterador: ");
38 ListaArrayMejorada.IteradorLineal it = la.iterador();
39 while(it.hayMas())
40 {
41 String x = (String) it.getActual();
42 System.out.println(x);
43 it.siguiente();
44 }
45 }
```

La diferencia principal radica en que lo que antes hacíamos con la referencia a la clase `ListaArray` ahora lo hacemos con la referencia de tipo `IteradorLineal` pero el concepto de mantiene de la misma forma.

## Clases anónimas en java - clases inline

En muchos casos en Java las clases sólo se la implementa para disponer del método que permita realizar cierta tarea necesaria y obligada por una interfaz o una clase abstracta.

Estas clases que se implementan con este objetivo único en muchos casos sólo serán usadas una única vez, para crear un único objeto. Se trata de una clase "de un solo uso", y sin embargo al declararla se le coloca un nombre y se obliga con ello a la máquina virtual no sólo a cargarla en memoria cuando el programa arranca, sino también a registrarla y darle soporte en tiempo de ejecución, por si en otro momento surge otro new para crear nuevas instancias. En casos así, no parece siquiera necesario que la clase lleve un nombre y sobrecargar de trabajo a la máquina virtual. Se puede recurrir entonces a una clase anónima.

Una clase anónima es una clase interna sin nombre, que se crea, se compila y se carga en memoria en el momento que se necesita, con el objetivo de contener, por lo general, un único método simple o lo conjunto acotado de métodos sencillos. El compilador genera un .class para la clase anónima con un nombre compuesto por el nombre de la clase outer, el signo \$, y un número de orden que será correlativo por orden de aparición de cada clase anónima.

La restricción para crear una clase anónima es la existencia de una clase abstracta de la que nuestra clase anónima va a derivar o de una interfaz que nuestra clase anónima va a implementar y la forma de crearla es utilizando dicha clase Abstracta o Interfaz como si fuera concreta pero agregando inmediatamente las llaves y la declaración del cuerpo de la clase, por esto también se las denomina clases inline.

## Ejemplo de Uso

Supongamos que tienes la siguiente interfaz:

```
interface Procesador {
 void procesar();
}
```

Si deseas crear una instancia única de esta interfaz y definir su método procesar(), puedes usar una clase anónima:

```
...
Procesador miProcesador = new Procesador() {
 @Override
 public void procesar() {
 System.out.println("Procesamiento en clase anónima.");
 }
};

...
```

En resumen, las clases anónimas son una herramienta valiosa para crear instancias de clases que implementan interfaces o extienden clases abstractas en el lugar donde se necesitan. Son particularmente útiles para implementaciones temporales y reducen la complejidad del código al evitar la creación de clases separadas para casos simples. Sin embargo, cuando se trata de implementaciones más largas o reutilizables, es posible que sea más apropiado usar clases internas o las expresiones lambda que veremos más adelante.

## Clases Parametrizadas en Java - Generics

En nuestra última versión de la clase ArrayList, tenemos la situación que todos los datos almacenados en la lista son de tipo Object y por lo tanto si bien puedo almacenar cualquier cosa necesito hacer casting luego para poder usar los objetos no tengo forma de controlar que se agreguen objetos de un solo tipo a la lista. Si bien podríamos programar un control usando el método getClass() y controlando si el objeto que se inserta es de la misma clase que el que figura primero en la lista, y de ser así, la inserción se acepta. Incorporando esa idea en todo método de inserción o en todo método que necesite comparar un objeto tomado como parámetro con el contenido de la lista, aseguramos la homogeneidad. Sin embargo, esto no resuelve el casting necesario a los objetos obtenidos de la lista.

A partir de la versión 1.5 del lenguaje Java o simplemente de Java 5 se incorpora un mecanismo de parametrización de clases con un tipo específico, designado como Generics y que aquí designaremos como clases parametrizadas. A partir de Java 1.5, las colecciones o estructuras de datos predefinidas del lenguaje (como LinkedList, ArrayList o HashMap, por ejemplo) pueden definirse en forma clásica, o pueden definirse también con un tipo como parámetro, donde podrán insertarse en la colección objetos de esti tipo:

```
ArrayList a = new ArrayList();
ArrayList<String> b = new ArrayList<>();
```

Nota: en la primera declaración y creación creamos una lista sobre un arreglo que permite cualquier tipo de objetos, en la segunda, en cambio, creamos la misma lista sobre un arreglo pero de forma que solo podemos operar con Objetos de tipo String.

En las definiciones anteriores, la lista a admite que se inserten en ella objetos de cualquier clase, en forma heterogénea (y esto es lo único que se podía hacer hasta la versión 1.5):

```
a.add("casa");
a.add(new Integer(5));
```

```
a.add(new Cuenta());
```

Pero la lista b sólo admitirá inserción de objetos de la clase String: cualquier intento de invocar a add() para agregar en b un objeto de otra clase, no compilará:

```
b.add("casa"); // ok...
b.add(new Integer(5)); // no compila...
b.add(new Cuenta()); // no compila...
b.add("perro"); // ok...
```

Se dice que la clase ArrayList admite otra clase como parámetro, o que está parametrizada. Y el mecanismo que permite hacer eso es el que se designa como generics desde Java 5 en adelante. Todas las clases del paquete java.util que representan colecciones o estructuras de datos, están parametrizadas de la misma forma. Note que es el propio compilador el que se encarga de rechazar una inserción heterogénea si la instancia se creó parametrizada, con lo cual la forma de controlar la homogeneidad es más clara que la que usamos en nuestra implementación. En nuestro caso, el control se hace en tiempo de ejecución y no hay forma de declarar una variable (de tipo SimpleList, por ejemplo) de forma que quede claro en la propia declaración cuál es la clase de objeto que queremos poder añadir a la lista, debemos esperar a la primera inserción para que de allí en adelante el resto de las inserciones se ajusten al primer objeto.

El mecanismo de parametrización de clases está disponible también para las clases del programador. El núcleo del lenguaje Java ha sido modificado a partir de Java 5 para que los programadores puedan incorporar generics en sus propias clases. Por lo tanto, podemos intentar modificar nuestro planteo de [ListaArray](#), para que pueda parametrizarse y dejar en manos del compilador el control de tipo de los objetos que agregamos a la lista.

El primer cambio que vamos a notar es en la declaración de la clase puesto que debemos declarar la definición del parámetro de tipo. A continuación veamos nuestra [ListaArrayMejorada](#) pero ahora incluyendo generics para validación del tipo:

```
9 public class ListaArrayMejorada<E>{
10 // el arreglo que contendrá los elementos...
11 private Object[] elementos;
12
13 // el tamaño inicial del arreglo...
14 private final int capacidadInicial;
15
16 // la cantidad de casillas realmente usadas...
17 private int count;
18
```

como podemos observar estamos agregando la letra E entre < > para terminar el nombre del parámetro que determinará luego el tipo para el resto de código de la clase.

Un elemento particular es que no estamos modificando el tipo del vector y esto tiene que ver con que Java no permite la creación ni manipulación de vectores raw de tipos parametrizados por lo que vamos a mantener el tipo del vector como Object y vamos llevar los cast a los accesos a los elementos de dicho vector.

Luego cada vez que hacemos un método público que va a requerir un objeto desde fuera de la clase vamos a usar `E` para hacer referencia al tipo:

```

43 > /** Añade al final de la lista el objeto e. La inserción será rechazada si la ...*/
51 > public boolean agregar(E e) {...}
60
61 > /** Añade el objeto e en la posición index de la lista . La inserción será ...*/
72 > public void insertar(int index, E e) {...}
86
87 > /** Retorna el objeto contenido en la casilla index. Si el valor de index no ...*/
96 > public E get(int index) {...}
102
103 > /** Remueve de la lista el elemento contenido en la posición index. Los ...*/
114 > public E quitar(int index) {...}
129
130 > /** Reemplaza el objeto en la posición index por el referido por element, y ...*/
141 > public E set(int index, E element) {...}
149

```

Como se puede observar cada vez que se hace referencia al tipo se utiliza el parámetro `E`, en este sentido no hay demasiado trabajo extra que hacer en el caso de los métodos que agregan pero cuando en cambio queremos obtener un elemento tenemos la situación en la que deberemos hacer casting para avisarle a Java que estamos seguros de la conversión de tipo, por ejemplo en el caso del método `get()`:

```

87 > /** Retorna el objeto contenido en la casilla index. Si el valor de index no ...*/
96 > public E get(int index) {
97 if (index < 0 || index >= count) {
98 throw new IndexOutOfBoundsException("get(): indice fuera de rango...");
99 }
100 return (E) elementos[index];
101 }

```

Así planteada, la clase ListaArrayMejorada sigue siendo genérica y controla homogeneidad, pero surge un nuevo problema, al definir los elementos del vector de tipo Object y en esencia el compilador entenderá cualquier cosa de tipo `E` como si fuera Object generaremos que cualquier acción que necesitemos llevar a cabo sobre estos elementos se reducirán a los métodos de la clase Object. Como vimos en el apartado de Polimorfismo, el tipo de la referencia define la interfaz de la instancia.

Entonces, supongamos que quisiéramos poder comparar los elementos del vector para ordenarlos de mayor a menor primero deberíamos cambiar el vector pero luego además deberíamos indicar que `E` ya no puede ser cualquier object sino cualquier objeto que implemente la interfaz comparable, para ello, al declarar el parámetro de clase `E`, indicar que esa clase deriva de (o implementa a) Comparable:

```

public class ListaArray<E extends Comparable> {
 ...
}

```

Un detalle a ver es que usamos la palabra extends más allá de que Comparable es una clase, bueno pues, dentro del operador rombo < > la palabra extends representa una restricción y no un hecho del compilador.

Existen muchas particularidades más sobre este tema y su estudio requiere revisar varios aspectos específicos algunos de los cuales difieren de una versión a otra, pero como hemos dicho no estamos en un curso de Java así que dejamos esa investigación para el alumno y nos limitamos a revisar los elementos requeridos.

Agregaremos la definición de los métodos a medida que sean necesarios si lo son.

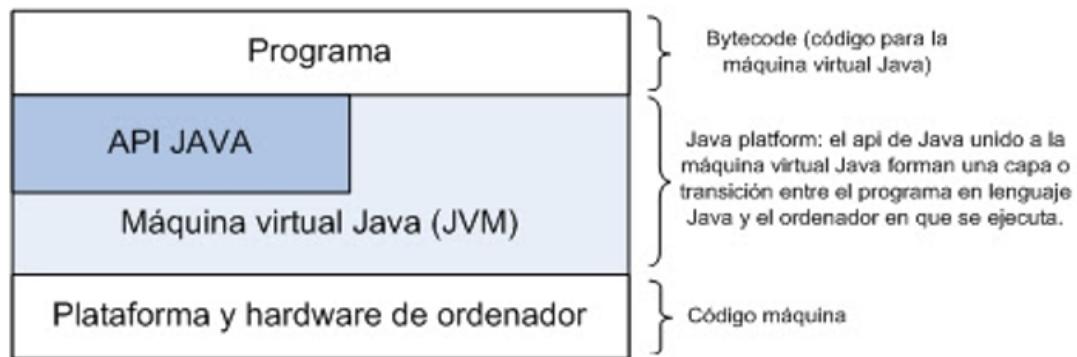
## Framework de Colecciones de Java

### Una breve referencia al conjunto de Apis de Java

La API Java es una interfaz de programación de aplicaciones (API, por sus siglas del inglés: Application Programming Interface) provista por los creadores del lenguaje de programación Java, que da a los programadores los medios para desarrollar aplicaciones Java.

Como el lenguaje Java es un lenguaje orientado a objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

Los siguientes esquemas, parte de la documentación de Java, nos dan una idea de cómo funciona el sistema Java y de qué se instala cuando instalamos Java (el paquete JDK) en nuestro ordenador.



		Java Language																	
		java	javac	javadoc	apt	jar	javap	JPDA	JConsole	Java VisualVM									
JDK	Tools & Tool APIs	Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI									
	RIAs	Java Web Start					Applet / Java Plug-in												
	User Interface Toolkits	AWT			Swing			Java 2D											
	Integration Libraries	Accessibility	Drag n Drop		Input Methods		Image I/O	Print Service	Sound										
	JRE	IDL	JDBC		JNDI		RMI		RMI-IIOP										
	Other Base Libraries	Beans	Intl Support		Input/Output		JMX		JNI										
	lang and util Base Libraries	Networking	Override Mechanism		Security		Serialization		Extension Mechanism										
		lang and util	Collections		Concurrency Utilities		JAR		Logging										
		Preferences API	Ref Objects		Reflection		Regular Expressions		Versioning										
		Java Virtual Machine							Zip										
		Java Hotspot Client and Server VM																	

La manera que java utiliza para organizar el mundo de clases que implementan las APIs de los esquemas anteriores son las librerías y los paquetes.

## Paquetes de clases

Tanto las clases nativas del lenguaje Java como las que pueda definir un programador en un proyecto propio pueden agruparse de acuerdo a objetivos y funcionalidades comunes, y formar colecciones de clases que en Java se llaman paquetes (o packages). Un package es lo que comúnmente se designaría como una librería de clases en otros lenguajes o en otros contextos. En principio, el único criterio para decidir si dos clases deben ir juntas en un package es la funcionalidad común: si ambas clases sirven para tareas y objetivos similares, es común pensar en agruparlas en el mismo package junto a otras clases que también tengan ese objetivo. Así, por ejemplo, todas las clases que originalmente se pensaron para el diseño de interfaces visuales de usuario, se agruparon en el package `java.awt` y todas las clases nativas pensadas para permitir entrada o salida hacia dispositivos externos se agruparon en el package `java.io`.

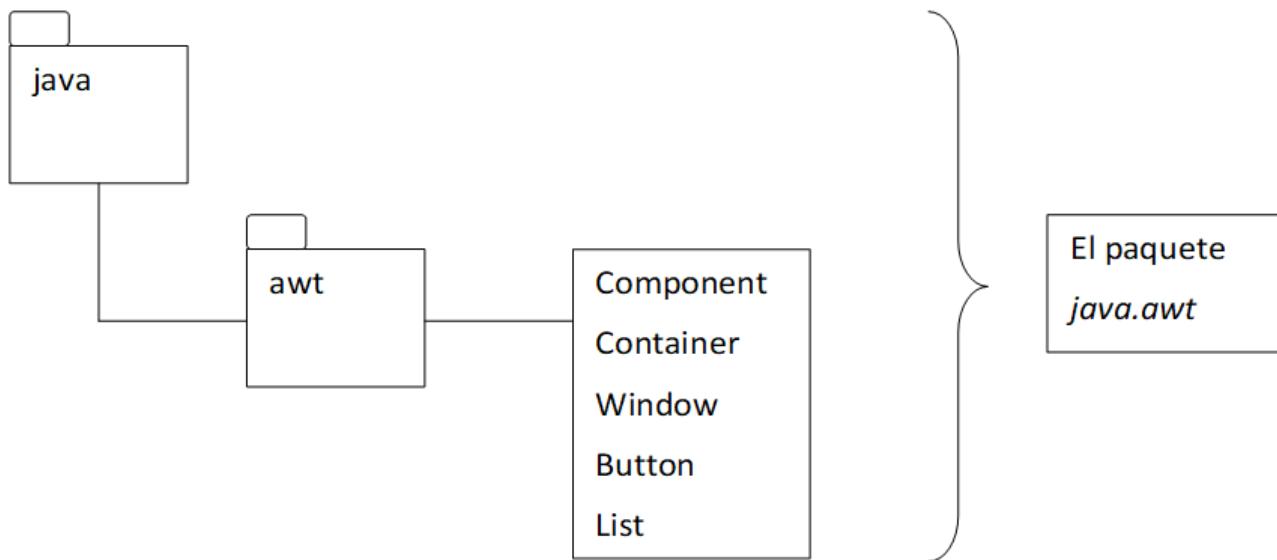
No es obligatorio que las clases de un package tengan entre ellas relación de herencia. De hecho, podrían no tener ninguna relación entre ellas, más que la nombrada de la funcionalidad común. Por otra parte, tampoco hay nada que prohíba que una clase de un package derive de otra ubicada en el mismo package o en otro package diferente. Sólo deberán considerarse los alcances del calificador `protected` cuando eso ocurra, pues en caso de clases ubicadas en distintos packages la visibilidad de miembros marcados `protected` será diferente. Remarcamos: un package es una colección de clases que normalmente tienen funcionalidad común, y se agrupan para lograr ciertas ventajas de organización y transparencia en el uso de nombres, que no se tendrían si todas las clases se mantuvieran separadas.

Al hacer que las clases de un proyecto se agrupen en paquetes, el proyecto se organiza y administra mejor. Las clases orientadas a la gestión de la interfaz de usuario pueden ir juntas en un package, mientras que aquellas que representan excepciones propias podrían ir en otro. Las clases específicas del modelo o del problema pueden agruparse en un tercer package y aquellas orientadas a la gestión de entrada o salida hacia archivos o bases de datos pueden ir en un cuarto package. Resultará más sencillo rehusar y distribuir esas clases, documentarlas y mantenerlas.

Por otra parte, el uso de packages permite solucionar potenciales problemas en cuanto a nombres repetidos: suponga que se está desarrollando un proyecto para gestión académica, y se está usando una clase propia

llamada Materia. Si luego se incorpora al proyecto el desarrollo de otro equipo de programadores, y en ese proyecto también hay una clase Materia (o varias clases con el mismo nombre que las del proyecto propio), no habría forma de hacer que el compilador pueda distinguir esas clases a menos que se las haga pertenecer a paquetes diferentes. Un package actúa como un espacio de nombres: el nombre o identificador del package al cual pertenece una clase forma parte del nombre de la clase del mismo modo que la ruta de directorios o carpetas en la cual está grabado un archivo forma parte del nombre físico de ese archivo. Así como no puede haber dos archivos con el mismo nombre en la misma ruta de directorios, no puede haber dos clases con el mismo nombre en el mismo package. Pero puede haber clases en diferentes packages, que tengan el mismo nombre.

Físicamente, un package no es otra cosa que una carpeta o directorio, que tiene un nombre que lo identifica y una ruta física para llegar hasta él desde una carpeta o directorio origen. El nombre completo de un package se forma escribiendo el nombre de todas las carpetas desde la carpeta origen hasta la carpeta del package, separando esos nombres con un punto. Así, cuando hablamos del package `java.awt` (nos pusimos nostálgicos, Abstract Windowing Toolkit - antiguo paquete Java de clases para implementación de ventanas), estamos diciendo que existe una carpeta `java`, la cual a su vez contiene una carpeta `awt`, y en esa carpeta se guardan las clases que conocemos para la gestión de interfaces del AWT:



Tenga en cuenta que un package finalmente es la forma en que se agrupan las clases frente al intérprete o máquina virtual Java (JVM), y que desde esta perspectiva finalmente lo que un package contiene son los archivos compilados de las clases (o sea, los archivos .class). Por razones prácticas obvias, se distribuyen en paquetes también los archivos fuente de un proyecto, pero estrictamente hablando un package es la forma en que se organizan los archivos .class de un proyecto para favorecer el trabajo de la JVM a la hora de encontrar, cargar e interpretar esos archivos para proceder a su ejecución.

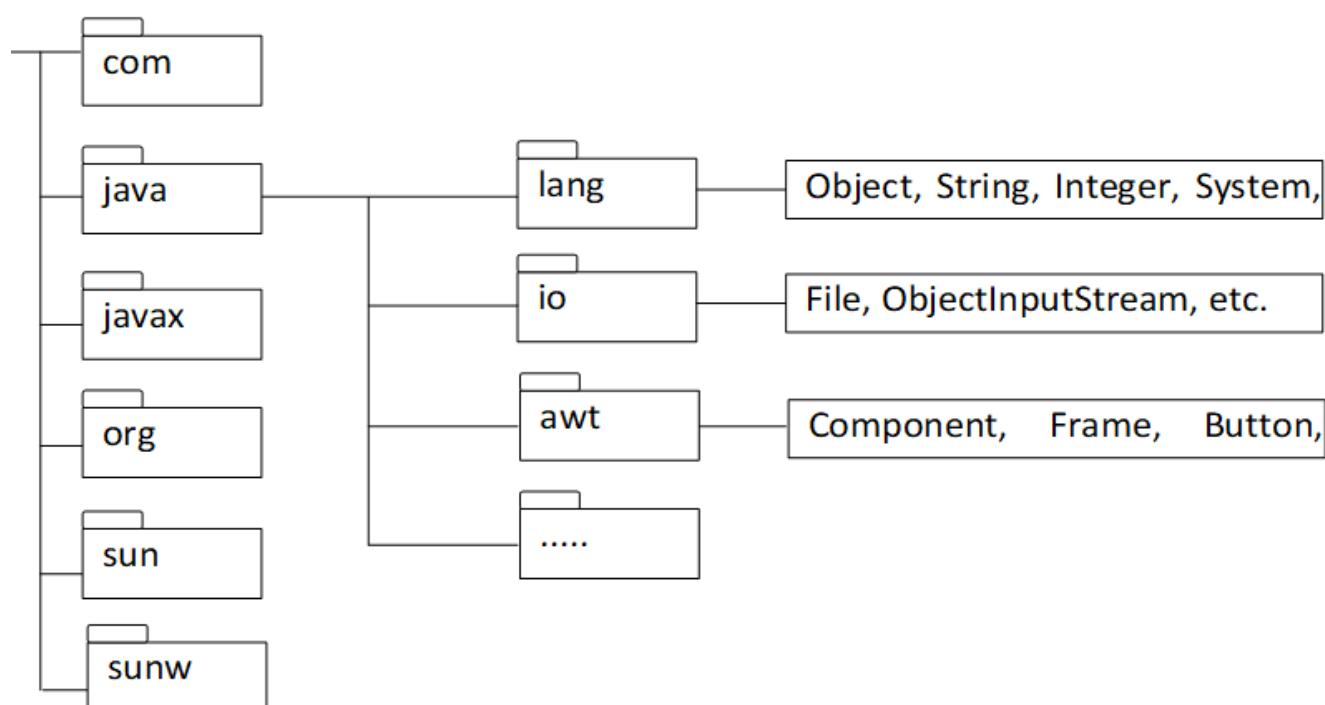
### **Clases nativas de Java - El uso de `import`**

Todas las clases nativas de Java vienen en packages predefinidos, y todos esos packages a su vez vienen comprimidos y distribuidos en un archivo llamado `.jar` que forma parte de la librería de Java. Cuando se ejecuta un programa, la JVM busca dentro de `rt.jar` los archivos compilados de las clases nativas que se estén usando en ese programa, y desde allí los extrae, los carga, y los ejecuta. Dentro de ese archivo, están contenidas las carpetas origen de todos los packages que hemos estado usando hasta aquí, que son, entre otras, las carpetas `java` y `javax`. Prácticamente todos los paquetes y clases de la distribución estándar de Java

(o JSE) vienen en la carpeta java de rt.jar. Y también sabemos que la carpeta javax contiene al package swing que incluye casi todas las clases de Swing para el desarrollo de interfaces avanzadas de usuario.

El gráfico de la página siguiente (ver Figura 1) muestra un esquema del contenido de rt.jar. Recuerde que ese archivo contiene los archivos de código de byte o .class de las clases nativas de Java. No obstante, también existe un archivo src.zip en el directorio donde se instala el JDK (que suele ser c:\Program Files\java\open-jdk-17 o similar) que contiene la misma estructura de paquetes ya vista en rt.jar, pero ahora conteniendo los archivos fuente (archivos .java) de todas las clases Java.

Para que una clase pueda acceder a otras clases que se encuentran en un package distinto, se usa la instrucción import al comienzo del archivo de la clase. Con esa instrucción, lo que se está haciendo es indicarle a la JVM en qué lugar se encuentran las clases que se están usando. Por ejemplo, si una clase necesita acceder a diversas clases del AWT, podría hacerlo así: import java.awt.\*; public class Ejemplo { private Button b1, b2; private Label l1, l2, .... }



De esta forma, la JVM sabrá dónde ubicar el código compilado de las clases Button y Label y de cualquier otra clase que esté en java.awt. Otra manera (más clara y específica, pero más larga...) consiste en usar el nombre de la clase completo dentro de la instrucción import, y repetir esta técnica con cada clase que se desee usar:

```

import java.awt.Button;
import java.awt.Label;

public class Ejemplo
{
 private Button b1, b2;
 private Label l1, l2,
 ...
}

```

Cuando una clase se nombra anteponiendo el nombre completo del package al nombre de la clase, se dice que el nombre de la clase está completamente calificado. Se puede prescindir de la instrucción import, si se está dispuesto a calificar completamente el nombre de cada clase que vaya a usar en cada lugar donde la misma se use:

```
public class Ejemplo
{
 private java.awt.Button b1, b2;
 private java.awt.Label l1, l2,
 public Ejemplo()
 {
 b1 = new java.awt.Button("Ok");
 l1 = new java.awt.Label("Ejemplo");
 }
 ...
}
```

Esto último será necesario si en un mismo programa se usan dos clases con el mismo nombre pero provenientes de packages diferentes: la única forma de distinguirlas, será calificando completamente el nombre de ambas...

Notemos que el único package que no necesita una instrucción import explícita para poder acceder a sus clases, es el package `java.lang` que contiene las clases básicas del lenguaje (como `Object`, `String`, `StringBuffer`, `Integer`, `Float`, `System`, `Math`, etc.) El package `java.lang` es importado automáticamente al cargar la clase, de modo la JVM siempre sabrá dónde buscar las clases básicas de Java. Todo otro package, requerirá la instrucción import o la calificación completa del nombre de sus clases.

Por otra parte, un programador puede dividir las clases de sus proyectos en packages propios, en forma simple. Para empezar, al comienzo del archivo fuente de la clase se debe incluir la instrucción `package`, seguida del nombre del paquete (o carpeta) al cual pertenecerá la clase. Sólo puede haber una instrucción `package` en una clase, y debe estar al comienzo del archivo fuente. Si luego necesita usar clases de otros packages, use la instrucción import tantas veces como requiera, pero debajo de la instrucción `package`. El siguiente código muestra la forma de indicar que la clase `Ejemplo` pertenece al package `prueba` (y al no indicarse la ruta de carpetas para llegar a `prueba`, se entenderá que esa carpeta está en el mismo directorio que el proyecto):

```
package utnfc.isi.back.ejemplo;

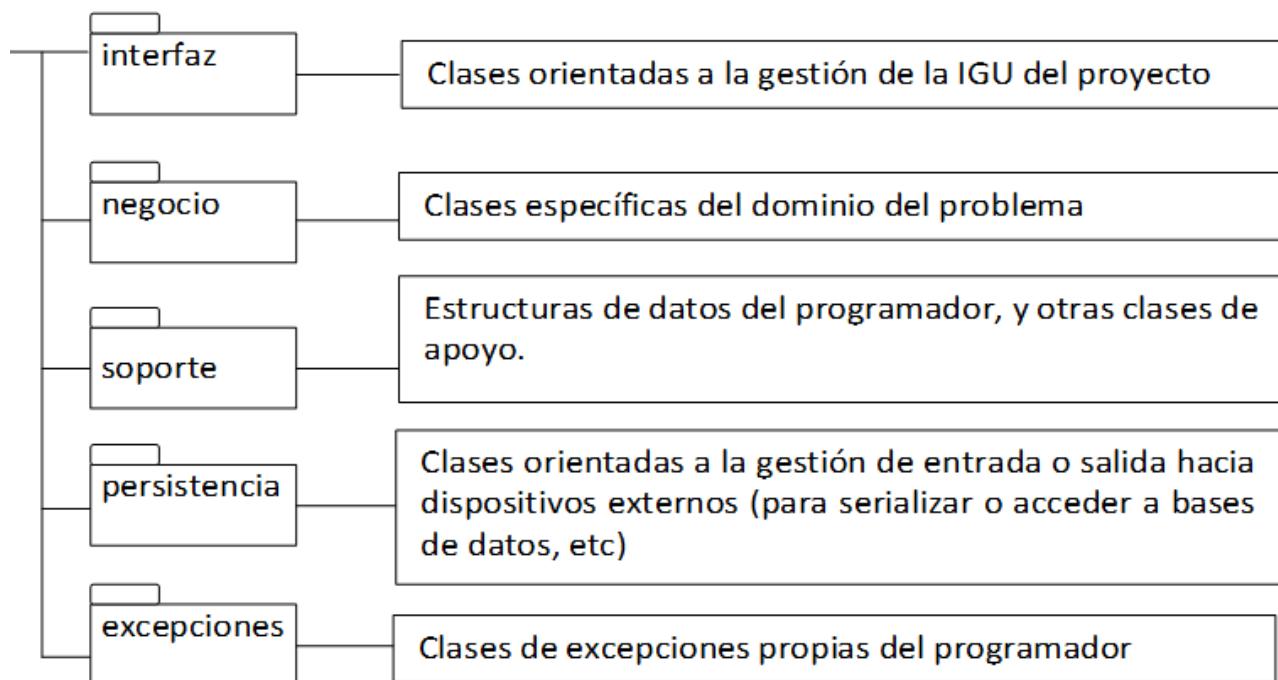
import java.awt.*;
import java.io.*;

public class Ejemplo
{
 private Button b1, b2;
 private Label l1, l2,
 ...
}
```

```
}
```

Por supuesto, deberá crear las carpetas que correspondan a cada package que quiera disponer en su proyecto. Eso puede hacerlo mediante el sistema operativo, o puede confiar en el IDE de desarrollo que esté usando para programar (BlueJ, NetBeans, Eclipse, etc.) Todos ellos pueden crear las carpetas de cada proyecto, en forma muy sencilla, y luego el programador puede agregar clases a uno u otro package. El propio IDE se encargará de colocar la instrucción package al inicio del archivo de la clase.

En general, el programador es dueño de decidir qué packages crear, con qué nombres designarlos y qué clases agrupar en ellos. Sólo debe tener en cuenta una pequeña convención para el nombre de un package: ese nombre debería designarse comenzando con una letra minúscula. En general, a partir de la versión 1.4 de Java no se espera que una clase esté fuera de todo package (o sea,, toda clase debería explícitamente ubicarse dentro de un package). Si no se usa la instrucción package para una clase, se asume que la misma pertenece al "package default" (que no es otra cosa que el directorio del proyecto que se está desarrollando), pero como dijimos, debería evitar esa situación. Una distribución de paquetes muy útil, aunque cada programador puede diseñar su propia distribución (y según los patrones de diseño que aplique tendrá eventualmente varias combinaciones), es la siguiente:



## Empaquetado y distribución de artefactos de software

Cuando se termina de desarrollar una aplicación o un paquete de clases para ser usado por otros programadores, es buena idea pensar en comprimir y empaquetar esa aplicación, para luego distribuir el archivo empaquetado con más sencillez y poder usarlo como un todo en otras aplicaciones. Piense en lo que hace Java con el ya citado archivo rt.jar: prácticamente todas las clases nativas del lenguaje vienen distribuidas y organizadas adecuadamente en ese archivo, el cual se usa simplemente como fuente de una gran librería de clases.

En el directorio bin del JDK existe un programa, llamado jar (que sería jar.exe en la plataforma Windows) cuyo objetivo es producir un archivo comprimido y empaquetado con todo el contenido de un proyecto. La aplicación jar.exe toma la carpeta que se le indique, la cual debe contener un proyecto Java ya compilado, y crea un único archivo de salida con extensión .jar (que es abreviatura de Java Archive). Esencialmente, un archivo con extensión .jar no es otra cosa que un archivo comprimido con formato .zip, y por lo tanto puede ser luego abierto con cualquier compresor que reconozca ese formato (WinRAR por ejemplo). Un detalle adicional a considerar, es que un archivo .jar puede contener una aplicación Java lista para ejecutarse (o sea, una aplicación que contiene un método main() en alguna de sus clases) o puede tratarse de una simple librería de clases distribuidas entre uno o mas packages pero sin método main() disponible. Si el .jar contiene un main(), el archivo .jar se conoce como un "jar ejecutable" y su principal característica es que el programa contenido en él, podrá ejecutarse directamente haciendo doble click sobre él desde el sistema operativo (por cierto, lo que realmente ocurrirá al hacer ese doble click, es que el sistema operativo transferirá la responsabilidad de ejecutar el contenido del archivo a la JVM que esté instalada en ese computador...).

Se puede crear un archivo .jar desde la línea de órdenes del sistema operativo, llamando directamente a jar.exe y pasándole los parámetros y modificadores que requiera, o se puede crear un .jar desde dentro de un IDE de desarrollo: la mayor parte de estos IDE permiten crear archivos .jar con mucha sencillez, ocultando los detalles de creación al desarrollador. Si está trabajando con BlueJ, por ejemplo, asegúrese de tener compilado el proyecto, luego seleccione la opción "Project" del menú principal, y dentro del menú que aparezca seleccione a su vez la opción "Create Jar File". En la ventana de diálogo que sigue, deberá ajustar algunas opciones de configuración (como el nombre de la clase que contiene al main() si es que hay alguna, o indicar si desea incluir los fuentes en el archivo a crear) y luego presionar Ok. A continuación se le pedirá el nombre del archivo .jar que quiere crear y la carpeta donde debe crearse, y finalmente el archivo .jar se creará en esa carpeta.

Si está trabajando con NetBeans, para crear el archivo .jar de su proyecto deberá hacer click derecho con el mouse al proyecto. Luego seleccionar la opción "Properties" en el menú emergente. En la ventana de configuración que aparece, seleccione el ítem "Build" y dentro de él elija el ítem "Packaging". Verá a la derecha de la ventana unas pocas opciones de configuración y algunos elementos que le indican donde será ubicado el archivo y cómo se llamará (en este caso, note que el archivo será colocado en el directorio dist, que será creado dentro del directorio del proyecto, y que el archivo se llamará igual que el proyecto pero con extensión .jar) Entre las pocas opciones de configuración que vea, simplemente asegúrese de tener tildada la casilla que dice "Build Jar after compiling". Eso es todo. De aquí en adelante, cada vez que el proyecto se compile, será creado el archivo .jar y será guardado en el directorio dist dentro del mismo proyecto. Si ya existía el archivo, será regenerado. Simplemente, asegúrese que una vez que haya terminado de desarrollar su aplicación haga una última "gran compilación" para que se genere el .jar definitivo y correcto.

Si damos un pequeño vistazo al contenido de un archivo .jar, notaremos que el mismo empaqueta la misma estructura de carpetas en la cual se distribuyeron las clases originales y dentro de esas carpetas aparecen los archivos .class (podrían aparecer también los .java si no los excluyó al crear el archivo .jar) Note que figura una carpeta llamada META-INF y que dentro de ella hay un archivo MANIFEST.MF. Ese archivo es un simple archivo de texto, que contiene unas pocas líneas que indican a la JVM algunos elementos para interpretar el contenido del .jar que se está abriendo. Sólo es interesante remarcar que si el archivo es un jar ejecutable, entonces el archivo MANIFEST.MF contendrá una línea comenzando con la expresión Main-Class: en la cual estará indicado el nombre de la clase del proyecto que contiene al método main() que debe usarse para ejecutar el programa. Si esa línea no viene acompañada de ningún nombre de clase, entonces el .jar no es un

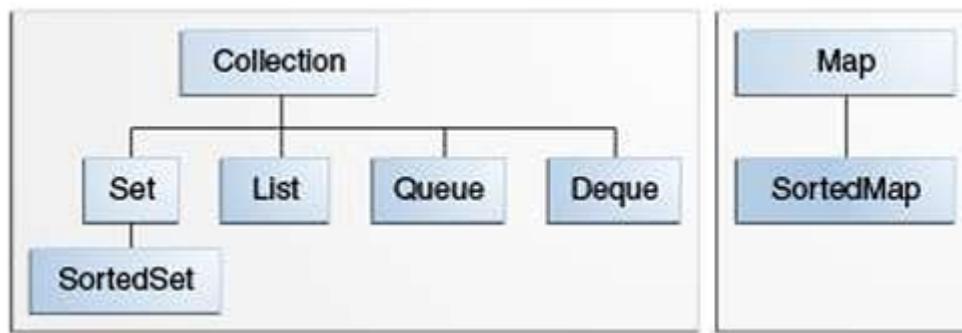
ejecutable y al hacer doble click sobre él probablemente se ejecute la máquina virtual pero provoque un error indicando que no existe un main class en el archivo MANIFEST.mf.

## Api de colecciones

El lenguaje Java dispone de un complejo esquema de jerarquías de clases e interfaces nativas ya previstas para llevar adelante esta estrategia, en el marco de trabajo definido por el package java.util que ya hemos citado.

En general, para referirse a cada estructura de datos provista por ese paquete, se usa el nombre Colección (o Collection). Los objetos Collection representan estructuras de datos capaces de contener objetos (y no tipos primitivos). En ese sentido, un objeto Collection difiere de los arreglos comunes en cuanto a que estos últimos pueden contener objetos, pero también tipos primitivos. Sin embargo, un arreglo común tiene tamaño fijo, en tanto que un objeto Collection puede crecer a medida que lo requiere el programa. Por otra parte, si se desea almacenar valores de tipos primitivos en un objeto Collection, se pueden usar las clases de envoltorio (wrapper classes: Integer, Character, Double, etc.) sin problemas (y a partir de las últimas versiones de Java está también disponible el autoboxing, que facilita el trabajo del programador: si se desea almacenar valores de tipos primitivos en una colección, simplemente se invoca a los métodos de esa colección para hacerlo y el mecanismo de autoboxing de encarga de convertir esos valores primitivos a objetos wrapper).

El paquete java.util provee una jerarquía amplia y completa de interfaces y clases que representan a cada estructura de datos básica. Existen dos clases de interface que dan origen a sendas jerarquías para representar estructuras de datos: Collection y Map. La novedad es que estas jerarquías comienzan con una jerarquía de interfaces: desde Collection y Map se definen nuevas interfaces que derivan de Collection o Map, y a partir de ellas vienen las clases para cada estructura. Como vemos, se comienza con clases de interfaces que heredan entre ellas. Un esquema de las dos jerarquías de interfaces es el siguiente:



Mostramos aquí un breve resumen del fundamento de cada una de ellas:

- Collection: es la raíz de una de las jerarquías de colecciones. Una colección representa un grupo de objetos que se conocen como sus elementos. La interface Collection es el denominador común que muchas de las colecciones implementan y se usa cuando se desea el máximo grado de generalidad en aplicaciones polimórficas. Algunos tipos de colecciones permiten elementos repetidos y otros no. Algunas colecciones son ordenadas y otras no. La plataforma Java no provee clases concretas que implementen directamente a Collection, pero sí provee varias que implementan a sus sub-interfaces específicas Set, List, Queue y Deque.
- Set: una colección que no contiene elementos repetidos. Esta interface modela la abstracción matemática del conjunto y se usa para representar conjuntos de diversos tipos de objetos.

- List: una colección o secuencia ordenada (en el sentido de preservar del orden de inserción). Puede contener elementos repetidos. Normalmente el programador que usa un objeto List tiene control preciso de en qué lugar de la lista se insertan los elementos y puede acceder a ellos mediante sus índices o posiciones.
- Queue: una colección usada para soportar y almacenar múltiples elementos antes de su procesamiento. Una Queue típicamente (pero no necesariamente) mantiene sus elementos en orden FIFO (first-in, first-out). No obstante, puede mantenerse el orden de acuerdo a algún criterio y obtener una cola de prioridad. Sea cual sea el orden de almacenamiento usado, el primer elemento de una Queue es el elemento que debería removido al invocar al método remove() o al método poll(). En una Queue de tipo FIFO, todo nuevo elemento se inserta al final de la estructura. Otros tipos de Queues deben especificar sus propias reglas.
- Deque: una colección usada para soportar y almacenar múltiples elementos antes de su procesamiento. Una Deque puede ser usada tanto en forma FIFO (first-in, first-out) como en forma LIFO (last-in, first-out). En una deque todo nuevo elemento puede ser insertado, recuperado y eliminado desde cualquiera de sus dos extremos.
- Map: un objeto que puede mapear ciertas claves hacia ciertos valores. Un Map no puede contener claves duplicadas; y cada clave puede mapear a lo sumo a un valor. Las tablas hash son posiblemente las formas más conocidas de implementación de un Map.
- SortedSet: un Set que mantiene sus elementos en orden ascendente. Es simplemente la versión ordenada de la interface Set. Provee algunas operaciones adicionales para aprovechar la ventaja de tener sus elementos ordenados.
- SortedMap: un Map que mantiene los elementos mapeados en él en orden ascendente de sus claves. Es simplemente la versión ordenada de la interface Map.

Estas jerarquías de interfaces brindan el contexto general y polimórfico básico para la implementación de las estructuras de datos nativas ya provistas por Java, y para el diseño de nuevas estructuras abstractas que pudiera necesitar el programador.

Para aprovechar al máximo este contexto de trabajo, es bueno hacer un paréntesis y poner en claro algunas ideas referidas a las estructuras de datos que por el momento nos ocupan: las listas. Estrictamente hablando, una lista es una estructura de datos lineal (cada elemento tiene o puede tener un único elemento sucesor y un único elemento antecesor) que además es adaptativa: a medida que necesita (o deja de necesitar) memoria, puede crecer o decrecer en su tamaño, y de tal forma además que un nuevo elemento puede ser insertado en cualquier lugar de la lista (y no necesariamente al principio o al final). Esta capacidad adaptativa es la que distingue a las listas de los arreglos. En muchos de los primeros lenguajes (o incluso en versiones antiguas de lenguajes modernos) los arreglos se definían como colecciones lineales esencialmente no adaptativas (no podían crecer ni decrecer en tamaño) pero que tenían la ventaja del acceso directo a sus componentes en tiempo constante. Esto se debe a que un arreglo se aloja completo en un bloque contiguo de memoria, y conocida la dirección del primer elemento es simple y directo calcular la dirección de cualquier otro.

Con el tiempo, y a partir de la flexibilidad para el manejo de memoria de lenguajes como C o C++ (en los que se programan muchos de los lenguajes modernos...) todos los lenguajes comenzaron a brindar capacidad adaptativa a sus arreglos (ya sea en forma directa o facilitando funciones de librería o clases especiales para hacerlo), llegando a un estado en el que la línea que separa a los arreglos de las listas se ha vuelto difusa: baste recordar (por ejemplo) que en el lenguaje Python directamente no se habla ya de arreglos sino de listas, y que esas listas en Python son en todo equivalentes a las java.util.ArrayList de Java, o a nuestras ArrayList,

estructuras lineales adaptativas, que se alojan en un bloque contiguo de memoria y disponen por tanto de acceso directo en tiempo constante.

De acuerdo a la forma en que una lista se implemente, se podrá contar con ciertas ventajas y desventajas. Esencialmente, hay dos formas básicas de implementar una lista:

- Implementación encadenada (o ligada): Cada elemento de la lista se almacena en un objeto separado designado como nodo. Cada nodo contiene un campo o slot para almacenar al elemento que contiene, y también incluye uno o dos campos más para almacenar las direcciones de su nodo sucesor y de su nodo antecesores. Si sólo se almacena la dirección de uno de ellos, la lista se dice simplemente vinculada, y si se almacenan los dos, se dice doblemente vinculada. La dirección del primero de los nodos se almacena en una variable separada (y también la del último si la lista es doblemente vinculada). Cuando la lista necesita crecer, se crea un nuevo nodo y se engancha al mismo en el lugar requerido, cambiando los valores de los enlaces al sucesor y/o antecesor. Si la lista necesita decrecer, se sueltan los enlaces que apuntan al nodo que se desea eliminar y se los ajusta para que apunten al nuevo sucesor y/o antecesor. Puede verse que de esta forma, la memoria ocupada por la lista no constituye un bloque contiguo (cada nodo se crea por separado y se aloja en el lugar que esté libre en ese momento) y esto lleva a que el acceso a cada nodo sólo pueda hacerse mediante un recorrido secuencial comenzando desde el principio (o desde el final) y siguiendo los enlaces al sucesor o antecesor. En otras palabras: el acceso a un elemento individual de una lista ligada requiere tiempo lineal ( $O(n)$ ) en el peor caso. Un ejemplo de este tipo de implementación es la clase `java.util.LinkedList` de Java (doblemente vinculada).
- Implementación sobre un arreglo de soporte: Toda la lista se almacena sobre un arreglo de forma de ir ocupando ese arreglo de izquierda a derecha, sin saltar casillas. El arreglo de soporte puede contener más casillas que las que realmente está usando la lista en un momento dado, pero como la memoria ocupada es contigua se puede acceder en forma directa y en tiempo constante ( $O(1)$ ) a cada elemento a través del índice que el mismo tiene en el arreglo. Si la lista tiene que crecer y el arreglo de soporte no tiene lugar libre, se crea un nuevo arreglo más grande, se copian los elementos del viejo arreglo al nuevo y se desechará el viejo, y se procede a la inversa cuando el arreglo debe acortarse. Obviamente, los dos ejemplos más directos que podemos citar son la clase `java.util.ArrayList` de Java, y nuestra `ArrayList`.

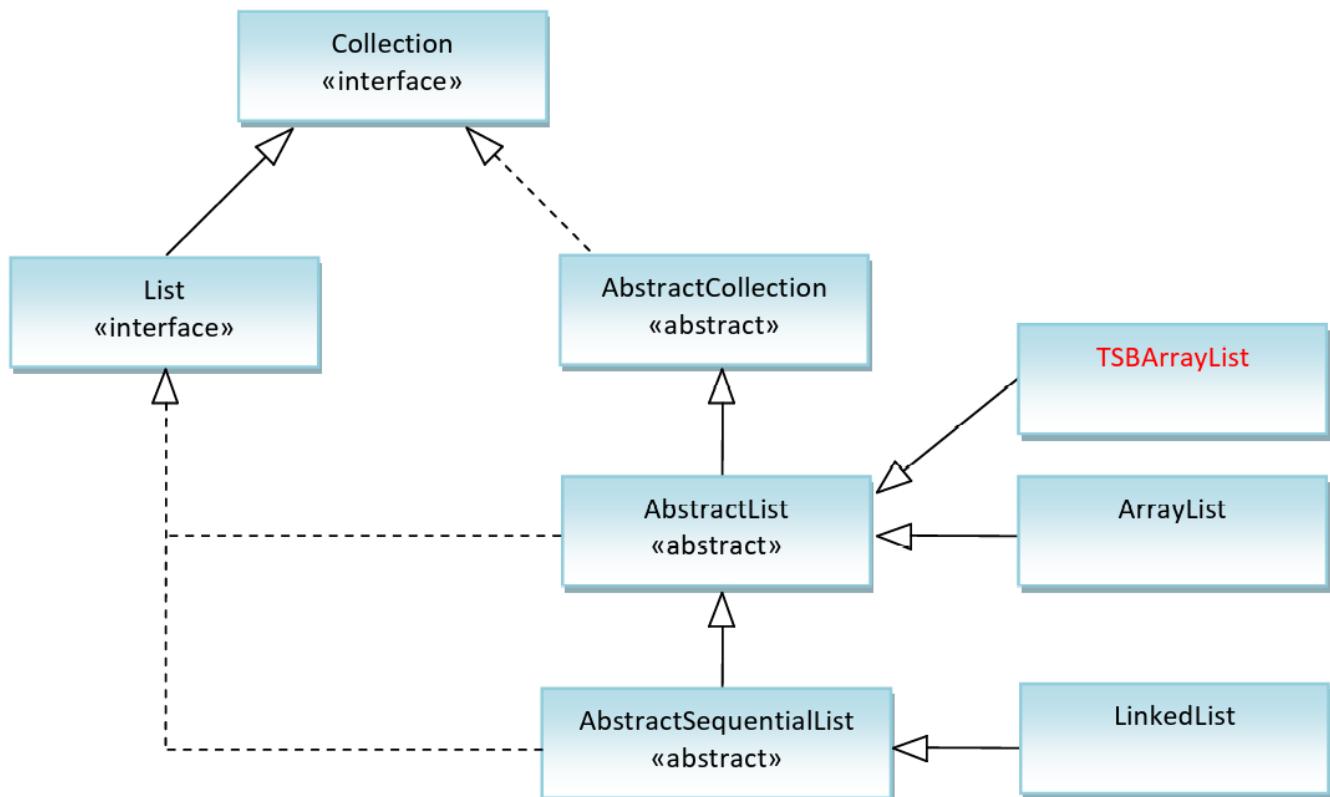
## Interface List<E> extends Collection<E>

Tipo de salida	Método – Descripción básica
boolean	<b>add(E e)</b> Agrega el elemento especificado al final de esta lista (operación opcional).
void	<b>add(int index, E element)</b> Inserta el elemento especificado en la posición especificada en esta lista. (operación opcional).
boolean	<b>addAll(Collection&lt;? extends E&gt; c)</b> Agrega todos los elementos de la colección especificada, al final de esta lista, el orden en que son retornados por el iterador de la colección (operación opcional).
boolean	<b>addAll(int index, Collection&lt;? extends E&gt; c)</b> Inserta en esta lista todos los elementos de la colección especificada, a partir de la posición especificada (operación opcional).
void	<b>clear()</b> Elimina todo el contenido de esta lista (operación opcional).
boolean	<b>contains(Object o)</b> Retorna true si esta lista contiene al elemento especificado.
boolean	<b>containsAll(Collection&lt;?&gt; c)</b> Retorna true si esta lista contiene a todos los elementos de la colección especificada.
boolean	<b>equals(Object o)</b> Compara el objeto especificado, con esta lista, para determinar si son iguales.
E	<b>get(int index)</b> Retorna el elemento en la posición especificada de esta lista.
int	<b>hashCode()</b>

	Retorna un valor de hash code para esta lista.
int	<b>indexOf(Object o)</b> Retorna el índice de la primera ocurrencia del elemento especificado en esta lista, o el valor -1 si la lista no contiene a ese elemento.
boolean	<b>isEmpty()</b> Retorna true si la lista no contiene elementos.
Iterator<E>	<b>iterator()</b> Retorna un iterador sobre los elementos de esta lista, en una secuencia apropiada.
int	<b>lastIndexOf(Object o)</b> Retorna el índice de la última ocurrencia del elemento especificado en esta lista, o el valor -1 si la lista no contiene a ese elemento.
ListIterator<E>	<b>listIterator()</b> Retorna un iterador de lista sobre los elementos de esta lista, en una secuencia apropiada.
ListIterator<E>	<b>listIterator(int index)</b> Retorna un iterador de lista sobre los elementos de esta lista, en una secuencia apropiada, comenzando desde la posición especificada.
E	<b>remove(int index)</b> Elimina el elemento ubicado en la posición especificada en esta lista (operación opcional).
boolean	<b>remove(Object o)</b> Elimina la primera ocurrencia del elemento especificado en esta lista, si está presente (operación opcional).
boolean	<b>removeAll(Collection&lt;?&gt; c)</b> Elimina de esta lista todos los elementos que están contenidos en la colección especificada (operación opcional).
default void	<b>replaceAll(UnaryOperator&lt;E&gt; operator)</b> Reemplaza cada elemento de esta lista con el resultado de aplicar el operador especificado en ese elemento.
boolean	<b>retainAll(Collection&lt;?&gt; c)</b> Retiene en esta lista sólo los elementos que están contenidos en la colección especificada (operación opcional).
E	<b>set(int index, E element)</b> Reemplaza el elemento ubicado en la posición especificada, con el elemento especificado (operación opcional).
int	<b>size()</b> Retorna el número de elementos de esta lista.
default void	<b>sort(Comparator&lt;? super E&gt; c)</b> Ordena esta lista de acuerdo al orden inducido por el <b>Comparator</b> especificado.
default Spliterator<E>	<b>spliterator()</b> Crea un <b>Spliterator</b> sobre los elementos de esta lista.
List<E>	<b>subList(int fromIndex, int toIndex)</b> Retorna una vista de una porción de esta lista entre el índice <i>fromIndex</i> , inclusive, y

	el índice <code>toIndex</code> , exclusive.
<code>Object[]</code>	<code>toArray()</code> Retorna un arreglo conteniendo todos los elementos de esta lista en una secuencia apropiada (desde el primero hasta el último).
<code>&lt;T&gt; T[]</code>	<code>toArray(T[] a)</code> Retorna un arreglo conteniendo todos los elementos de esta lista en una secuencia apropiada (desde el primero hasta el último); el tipo de tiempo de ejecución del arreglo returned es el del arreglo especificado.

Para ahorrar algo de trabajo al programador, Java provee ciertas clases abstractas que ya implementan la interface List y proveen una implementación adecuada de la mayor parte de estos métodos (y de otros que a su vez provienen de la interface Collection). Dos de esas clases son muy útiles: AbstractList y AbstractSequentialList (ver diagrama de clases siguiente):



La clase abstracta AbstractCollection provee una implementación esencial de los métodos de la interface Collection, y de ella deriva a su vez la clase AbstractList que es super clase de la clase AbstractSequentialList. Y las dos últimas implementan la interface List. La clase abstracta AbstractList provee una implementación básica (a modo de plantilla) de la interface List para minimizar el esfuerzo requerido para implementar esa interface cuando se diseñan estructuras lineales basadas en almacenamiento de acceso directo (como un arreglo, o una lista soportada sobre un arreglo). Para la implementación de estructuras lineales basadas en acceso secuencial (como una lista ligada), debería usarse como base la clase AbstractSequentialList en lugar de AbstractList. De hecho, en la implementación nativa de Java para su clase `java.util.ArrayList`, la misma deriva de AbstractList, mientras que `java.util.LinkedList` deriva de AbstractSequentialList (ver diagrama de clases anterior).

Por lo tanto, si un programador desea implementar alguna clase que permita modelar una lista basada en un soporte de acceso directo, puede emular lo que hace Java con su propia clase `java.util.ArrayList`, derivarla de la clase AbstractList, la cual le facilita mucho trabajo ya desarrollado. Y si desea implementar una clase para

modelar una lista basada en la idea de lista ligada, debería pues derivarla desde `AbstractSequentialList` y aprovechar lo que esa clase ya provee. En nuestro caso, la clase `ArrayList` se ha hecho derivar desde `AbstractList`. Lo queda es analizar las convenciones y recomendaciones que la propia documentación provee para cada una de estas clases cuando se hace un trabajo similar al que vamos a intentar. Veremos que en esencia no es demasiado trabajo:

***Si se deriva desde `AbstractList` (para una lista con soporte de acceso directo):***

1. Si la lista es inmodificable, el programador sólo necesita derivar esta clase y proveer implementaciones para los métodos `get(int)` y `size()`.
2. Si la lista es modificable (esto es, algún elemento pre-existente puede cambiar por otro, pero sin cambiar el tamaño de la lista), el programador debe adicionalmente sobre-escribir el método `set(int, E)` (el cual en caso contrario lanzará una excepción de la clase `UnsupportedOperationException` al ser invocado).
3. Si además el tamaño de la lista puede cambiar (ya sea que aumente o disminuya), entonces el programador debe adicionalmente sobre-escribir los métodos `add(int, E)` y `remove(int)`.
4. El programador debería también proveer una implementación del constructor sin parámetros, y también del "constructor collection" (es decir, del constructor que toma como parámetro a otra colección y crea la lista copiando el contenido de esa otra colección).
5. A diferencia de otras implementaciones de colecciones abstractas, en este caso el programador no está obligado a implementar una clase iteradora ni tampoco a dar una implementación de los métodos `iterator()` y `listIterator()`: la clase `AbstractList` ya provee una implementación apropiada basada en el uso de los métodos de acceso directo `get(int)`, `set(int, E)`, `add(int, E)` y `remove(int)`.
6. Finalmente un detalle técnico: la clase `AbstractList` dispone de un atributo protegido de tipo `int`, llamado `modCount`. Ese atributo se usa para llevar la cuenta de la cantidad operaciones realizadas sobre la lista que modificaron su estructura, es decir, operaciones que cambiaron el tamaño de la lista. Usando ese atributo, las clases iteradoras pueden inferir si la lista ha cambiado mientras se está realizando concurrentemente un recorrido con algún iterador, y en ese caso, interrumpir de inmediato el proceso iterador lanzando una excepción de la clase `ConcurrentModificationException`. Si esto se aplica, se dice que la clase sigue una estrategia de fail-fast iterator (o iterador de respuesta rápida a fallos). No hay que alarmarse por este detalle: si el programador desea una implementación fail-fast iterator, lo único que debe hacer es sumar 1 al atributo `modCount` en cada método que efectivamente lleve a cabo una modificación estructural de la lista (operación que implique un cambio de tamaño). El control de la falla en un contexto concurrente y el lanzamiento de la excepción, es llevado a cabo por los métodos ya implementados en la clase `AbstractList`. Y si el programador no desea una implementación fail-fast iterator, simplemente puede ignorar el atributo `modCount` y dejar que cada iterador haga lo que pueda cuando recorra una lista y esta pueda cambiar durante el recorrido (en este caso, no hay seguridad de obtener resultados correctos, e incluso podría ocurrir de todos modos una excepción si el iterador llega a un punto (por ejemplo) en el que no pueda recuperar el próximo elemento).

***Si se deriva desde `AbstractSequentialList` (para una lista ligada):***

1. Para implementar una lista ligada, el programador sólo necesita derivar esta clase y proveer implementaciones para los métodos `listIterator()` y `size()` (y obviamente deberá proveer una clase iteradora que implemente `ListIterator`). Si la lista es inmodificable, el programador sólo necesita implementar los métodos `hasNext()`, `next()`, `hasPrevious()`, `previous()` e `index()` de la clase iteradora que programe.

2. Si la lista es modificable (esto es, algún elemento pre-existente puede ser reemplazado por otro pero sin cambiar el tamaño de la lista) entonces el programador debería adicionalmente implementar el método set() de la clase iteradora para listIterator().
3. Si la lista es de tamaño variable, entonces el programador debería adicionalmente implementar los métodos remove() y add() de la clase iteradora para listIterator().
4. El programador debería también proveer una implementación del constructor sin parámetros, y también del "constructor collection" (es decir, del constructor que toma como parámetro a otra colección y crea la lista copiando el contenido de esa otra colección).
5. También aquí es válido considerar las recomendaciones respecto al atributo modCount heredado desde AbstractList (ver para ello el punto f de las recomendaciones para implementar una lista derivando desde AbstractList.)

Tanto si se deriva desde AbstractList como si se lo hace desde AbstractSequentialList, la documentación para cada uno de los métodos no abstractos en cada clase describe su implementación en detalle. Y por supuesto, cada uno de esos métodos puede ser sobre escrito si la colección que está siendo diseñada admite implementaciones más eficientes.

Llegamos al fin a poder intentar nuestra propia colección en base al planteo inicial de ListaArray. Con estos elementos ya presentados, sólo nos queda ver en detalle la implementación final de nuestra clase ArrayList, que se define esencialmente así:

```
public class ArrayList<E> extends AbstractList<E>
 implements List<E>, RandomAccess, Cloneable,
Serializable {
 ...
}
```

Note que los atributos de la clase son los mismos que ya hemos incluido en la versión original de la clase, con las mismas especificaciones y usos.

La interface RandomAccess es una interface vacía o de marcado, que sirve para que una clase notifique que admite operaciones de acceso directo a sus elementos (normalmente de tiempo constante). De esta forma, un algoritmo genérico (que tome un parámetro polimórfico) puede chequear si una instancia pertenece a una clase que haya implementado RandomAccess y modificar el comportamiento de tal algoritmo para aplicar técnicas más veloces que aprovechen el acceso directo. Por el momento, es suficiente con saber que el programador sólo debe indicar que la clase implementa RandomAccess. Y algo similar sucede con la interface Serializable (ya conocida) que también es una interface vacía cuya implementación se usa para notificar que la clase admite la serialización de sus instancias.

La interface Cloneable también es una interface vacía, pero en este caso hay algún trabajo extra para realizar: La implementación de esta interface se usa para notificar que la clase admite la operación de clonar sus instancias invocando al método clone() (que NO viene especificado en la interface Cloneable, sino en la clase Object, por lo que toda clase dispone de la implementación esencial del método). La idea es la siguiente: si una instancia de una clase X invoca a clone() pero la clase X no implementó Cloneable, se producirá una

excepción de la clase `CloneNotSupportedException`. La clase X debe implementar `Cloneable` si se tiene pensado invocar a `clone()` con sus instancias.

Si se usa directamente el método `clone()` heredado desde `Object`, este método creará y retornará una copia superficial del objeto: todos sus atributos de tipo primitivo serán efectivamente replicados y copiados en el nuevo objeto, pero para los atributos que sean referencias a otros objetos, este método replicará las referencias (sin copiar los objetos en sí mismos). Por lo tanto, posiblemente el programador querrá implementar su propia versión de `clone()` en las clases que implementen `Cloneable`. En ese caso, lo aconsejable es que en la nueva versión del método se comience invocando al método `clone()` de la super clase (y si todas las clases siguen esta convención, finalmente todas invocarán a `Object.clone()`). Con esto se garantiza que se creará un objeto de la clase correcta, con sus atributos primitivos correctamente replicados, quedando para el programador la tarea de modificar en la copia clonada los atributos que sean referencias. En nuestra clase `ArrayList` el método `clone()` se redefinió así:

```
public Object clone() throws CloneNotSupportedException {
 ArrayList<?> temp = (ArrayList<?>) super.clone();
 temp.items = new Object[count];
 System.arraycopy(this.items, 0, temp.items, 0, count);
 // fail-fast iterator para el nuevo objeto...
 // modCount se hereda desde AbstractList y es protected...
 temp.modCount = 0;
 return temp;
}
```

El método define en su cabecera que puede lanzar una excepción de la clase `CloneNotSupportedException` (lo cual teóricamente ocurrirá si se invocase a este método sin que la clase `ArrayList` implemente `Cloneable`... pero en la práctica eso no pasará, ya que efectivamente nuestra clase hizo esa implementación).

El método comienza creando el nuevo objeto (que finalmente será el que se retorne), pero invocando a `super.clone()` para hacer esa creación. Con esto, el nuevo objeto `temp` tendrá todos sus atributos con los mismos valores que el objeto actual (entre ellos, `initial_capacity` y `count`). Note que `clone()` retorna una referencia de la clase `Object`, y por eso se aplica una operación de casting explícito para convertir el tipo del resultado a `ArrayList`. Para evitar que la referencia `temp.items` apunte al mismo arreglo referenciado por `this.item` (que es el efecto de la copia superficial), se crea nuevamente y por separado el arreglo `temp.items` y se usa luego `System.arraycopy()` para copiar las direcciones contenidas en `this.items` (y en este momento, note que `clone()` replicará la lista, pero no los objetos contenidos en ella). Finalmente, se pone a cero el valor del atributo `temp.modCount` (heredado desde `AbstractList`) para activar desde cero el mecanismo fail-fast iterator en el objeto recién creado (y evitar que ese atributo tenga en `temp` el mismo valor que tenía en `this...`). Cuando todo esto ha sido concluido, el método termina retornando el nuevo objeto.

Como la clase `ArrayList` deriva de `AbstractList`, entonces la mayor parte del trabajo de implementar los métodos de la interface `List` está ya realizada (aunque sea a un nivel esencial). El contenido completo de la clase es el que puede consultarse en el ejemplo que acompaña el presente apunte.

## El patrón Iterador en Java

Una revisión interesante a realizar es la implementación que tiene java del patrón iterador, dicha implementación se basa en las interfaces `Iterable<T>` e `Iterator<T>`. La interfaz `Iterable` obliga a la clase que la implementa a implementar el método:

```
public Iterator<T> iterator() {
}
```

de forma tal que dicho método devuelva una instancia de iterador que será implementado en una clase interna, sin embargo, vale la pena observar que el uso de la interfaz `Iterator`, provoca un desacople entre el objeto iterador específico de la clase y quién requiera utilizarlo, es decir, quien requiera un iterador no necesita saber si está iterando un `ArrayList` de Backend o un `ArrayList` de Java o un `LinkedList` u otra clase `Iterable`, esto potencia y flexibiliza el modelo.

Sin embargo, para mostrar la implementación del Iterador como Java lo esperaría, hemos agregado también en el ejemplo que acompaña este apunte, entre las clases parametrizadas, la clase `ListaArrayIterable`, que implementa la interfaz `Iterable` y declara la clase interna `ListaArrayIterator` que implementa la interfaz `Iterator`.

En este caso la implementación cambia sutilmente respecto de la implementación de iterador que analizamos previamente, puesto que si bien mantiene el constructor como creador e iniciador del iterador como es de esperarse, y también el método `hasNext()` que determina si aún quedan elementos por recorrer. Agrega un solo método extra que cumple las funciones de `getActual()` y `siguiente()` y en este orden preciso, este método es el método `next()` que al mismo tiempo mueve el iterador al siguiente elemento y retorna el elemento actual.

Finalmente, el método `remove()` debe eliminar el último elemento que haya sido alcanzado por `next()` (es decir, el elemento que actualmente esté en la casilla `current`), pero de forma que `current` quede apuntando al elemento que estaba a la izquierda de este (es decir, `current` debe quedar valiendo el índice de la casilla anterior). Por lo tanto, primero debe chequear que `next()` haya sido invocado (lo cual hace con la condición de la primera línea), lanzando una excepción en caso contrario e interrumpiendo la ejecución del método. Si todo estaba bien con `next()`, entonces para llevar a cabo efectivamente el borrado simplemente se invoca al método `remove(index)` que ya está implementado en la propia clase `ListaArrayIterable` (y que ya hemos mostrado): al invocarlo, se le pasa como parámetro el valor `current`, y el método hace el resto, incluyendo la disminución del tamaño del arreglo si fuese necesario y la operación de restar uno al valor de `count` para reflejar el hecho de que la lista ha perdido un componente. Note que para invocar al método `remove(index)` de la clase `ListaArrayIterable`, el método `remove()` de la clase `ListaArrayIterator` accede al mismo de la forma siguiente:

```
ListaArrayIterable.this.remove(current)
```

Esto es: si desde el interior de un método que pertenece a una clase interna (como el método `remove()` de la clase `ListaArrayIterator`) se quiere acceder a un método o atributo de su clase contenedora (como el método

remove(index) de la clase ListaIterable) pero usando la referencia this, entonces el método de la clase interna debe anteponer el nombre de la clase contenedora antes de acceder a this (pues de otro modo, estará accediendo al puntero this de la propia clase interna...).

El método remove() finaliza volviendo a false el valor del atributo next\_ok, la idea es que si remove() fue invocado es porque next() había sido invocado a su vez, pero una vez eliminado el elemento actual, next() debe volver a invocarse para poder activar nuevamente a remove(). Si remove() pone false en next\_ok, entonces dos invocaciones seguidas a remove() (sin invocar a next() entre ambas) provocarán una excepción.

## Anexo I: Comparación de recorridos sobre colecciones

### Enfoque evolutivo

Una vez construida una lista con objetos, es habitual necesitar recorrerla para inspeccionar, transformar o acumular datos. En esta sección final, ilustramos cómo ha evolucionado el enfoque de recorrido sobre estructuras como List, desde los métodos más acoplados con la estructura interna, hasta formas más genéricas y desacopladas que Java promueve a partir de sus interfaces Iterable e Iterator.

 **1. Recorrido con subíndices (for tradicional)** Este enfoque solo es viable si la estructura interna de la lista soporta acceso aleatorio, como sucede en las listas sobre arrays (como ArrayList). Se accede a cada elemento por posición:

```
...
List<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Luis");
nombres.add("Carla");

for (int i = 0; i < nombres.size(); i++) {
 System.out.println(nombres.get(i));
}
...
```

 En este caso:

- `size()` indica cuántos elementos tiene la lista.
- `get(i)` permite acceder a cada elemento directamente.
- Requiere que el almacenamiento subyacente sea indexado o permita el acceso por subíndice.

Cabe mencionar que, más allá de la necesidad de conocer la estructura interna de la colección para realizar este recorrido, en el caso de los ArrayList que soportan acceso directo este tipo de acceso no provoca overhead, pero si estuviéramos hablando de una lista vinculada, este recorrido sería

**Orden( $n^2$ )**

 **2. Recorrido usando Iterator (while + patrón iterador)** Este enfoque sigue el patrón iterador tal como lo vimos en este apunte, y funciona sobre cualquier implementación de `Iterable`, sin importar cómo esté implementado internamente.

```
...
Iterator<String> it = nombres.iterator();
while (it.hasNext()) {
 String nombre = it.next();
 System.out.println(nombre);
}
...
```

### 💡 En este caso:

- Se desacopla completamente del tipo de colección (ArrayList, LinkedList, etc.).
- Es útil incluso para estructuras sin acceso directo, ya que delega la responsabilidad de obtener el siguiente elemento a la estructura.
- Refuerza el patrón de diseño Iterator explicado en este material.
- Permite eliminar elementos durante el recorrido (`it.remove()`), con control de fail-fast.

### ☒ 3. Recorrido con for-each (forma moderna y recomendada)

El `for-each` es azúcar sintáctico provisto por Java para recorrer cualquier colección que implemente `Iterable`. Bajo el capó utiliza un `Iterator`, pero sin necesidad de declararlo explícitamente.

```
...
for (String nombre : nombres) {
 System.out.println(nombre);
}
...
```

### 💡 Características:

- Desacopla aún más al programador del mecanismo de iteración.
- Funciona con cualquier clase que implemente `Iterable<E>`.
- Es la forma recomendada para recorrer colecciones en la mayoría de los casos simples.
- No permite eliminar elementos durante el recorrido (no expone `remove()` del Iterator).

### ☑ Conclusión

A medida que avanzamos en abstracción, pasamos de depender del almacenamiento interno (índices) a usar interfaces (`Iterable`, `Iterator`), y finalmente llegamos a formas sintéticas (for-each) que logran mayor claridad, flexibilidad y mantenibilidad.

Este recorrido gradual ilustra cómo la evolución del lenguaje Java acompaña los principios de diseño orientado a objetos: ocultamiento de implementación, uso de interfaces, y separación entre qué se hace y cómo se hace.