Ficha 6

Patrón Iterator – Generics – Paquetes de Clases

1.] Recorrido de una lista. El patrón Iterator (implementación liviana).

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 SimpleList: tal método estaría suponiendo que la lista tiene objetos de la clase Inversion y eso sería falso en general: la clase *TSBSimpleList* fue planteada para suponer un info polimófico. Ese método sólo sería utilizable si efectivamente se crea una instancia de la clase *TSBSimpleList* en la que se inserten objetos que representen cuentas de Inversion:

```
public class TSBSimpleList
{
	// atributos y definiciones de la clase...

	public float saldoPromedio()
	{
		float a = 0;
		int c = 0;
		TSBNode p = frente;
		while(p!=null)
		{
			Inversion x = (Inversion) p.getInfo();
			a += x.getSaldo();
			c++;
			p = p.getNext();
		}
		float prom = 0;
		if (c!=0) prom = a / c;
		return prom;
	}

	// resto de los métodos de la clase...
}
```

En el esquema anterior de la clase *TSBSimpleList* se incluyó un método que calcula ese promedio. Puede verse claramente el problema: la línea remarcada en azul supone que la referencia *p* está apuntando a un nodo con un *info* de tipo *Inversion*, y hace un casting de esa referencia. Pero si la lista no contiene objetos de tipo *Inversion*, ese casting provocará una excepción de la forma *ClassCastException*. Por lo tanto ese método no puede ser usado en un contexto polimórfico y no debería estar en la clase *TSBSimpleList*.

Otra solución podría ser que el método se programe *fuera* de la clase *TSBSimpleList*, y desde allí recorrer la lista y obtener el promedio (por ejemplo, desde la clase *Principal* que

contiene al *main()*). Pero entonces tendremos otro (grave) problema: la clase donde se programe el método debería tener acceso al atributo *frente* de la lista, pero ese atributo es privado en la clase *TSBSimpleList*. Se podría agregar un método *getFrente()* en la clase *TSBSimpleList* que retorne la dirección del primer nodo de la lista, y luego usar el método *getNext()* de la clase *TSBNode* para seguir el recorrido desde la clase *Principal*. En el siguiente esquema suponemos que en la lista se insertarán objetos de la clase *Inversion*:

```
public class Principal
{
	private static TSBSimpleList lista;

	// más elementos de la clase Principal aquí….

	public static float saldoPromedio()
	{
		float a = 0;
		int c = 0;
		TSBNode p = lista.getFrente();
		while(p!=null)
		{
			Inversion x = (Inversion) p.getInfo();
			a += x.getSaldo();
			c++;
			p = p.getNext();
		}
		float prom = 0;
		if (c!=0) prom = a / c;
		return prom;
	}

	// resto de los métodos de la clase Principal...
}
```

Podría pensarse que de esta forma está terminado el problema, pero no es así: técnicamente, se sigue violando el *Principio de Ocultamiento* en la clase *Principal*, pues esta clase *tiene que saber que una lista de la clase TSBSimpleList está compuesta por objetos TSBNode*, y que esos *TSBNode* proveen una forma indirecta de recorrer la lista mediante *getNext()*... El programador de la clase *Principal* debería saber demasiados detalles respecto de la estructura interna de la clase *TSBSimpleList* y su clase asociada *TSBNode*, cuando en realidad este programador ni siquiera debería sospechar que existe una clase *TSBNode*. El recorrido de la lista desde una clase *externa* a *TSBSimpleList* debería ser más transparente y automático, sin implicar al programador en el conocimiento de su estructura interna.

¿Cómo hacer el recorrido de una estructura de datos genérica sin exponer ni tener que conocer la forma interna de la misma? Este es un problema clásico de la POO y así como éste existen muchos otros problemas de aparición recurrente en programación. Cuando se enfrenta un problema de aparición frecuente hay dos vías: o bien el programador "inventa la pólvora de nuevo" cada vez que el problema reaparece y da una solución forzada al mismo todas la veces; o bien (si es posible hacerlo) recurre a un *patrón de diseño*: esto es, una solución planteada por expertos, bien estudiada, documentada y clasificada del problema. Si existe un *patrón de diseño* para el problema que se enfrenta, usar ese patrón es normalmente una solución mejor que "reinventar la pólvora".

No es objetivo de esta asignatura hacer un estudio profundo de todos los patrones de diseño que existen, pero sí usar los patrones que sean relevantes a sus contenidos cuando aparecen situaciones problemáticas clásicas. El estudio profundo de los patrones se hace (como el alumno ya sabrá) en otra asignatura de la carrera.

Los patrones de diseño que se han propuesto, estudiado y clasificado tienen un nombre genérico que ayuda a los programadores a comunicar sus ideas con una sóla palabra o frase corta. Así, podemos decir que la solución general al problema del recorrido de una lista genérica es propuesta desde un patrón llamado *Iterator* (en castellano: *iterador*). La idea del *patrón Iterator* es crear una *clase separada* (llamada en general *clase iteradora*) de la clase cuyos objetos se quieren recorrer, de forma que la clase iteradora sea la que recorra la estructura. La clase iteradora debería proveer un pequeño conjunto de métodos genéricos tales como *next()* (para retornar el siguiente objeto en la estructura que se recorre) o *hasNext()* (para determinar si en la estructura que se recorre quedan objetos sin acceder).

En el modelo *TSB-SimpleList04* mostramos una *implementación liviana* de la idea del *patrón Iterator*. En lugar de una clase iteradora separada de la clase *TSBSimpleList*, para simplificar esta primera aproximación incluimos los métodos de iteración dentro de la misma clase *TSBSimpleList*. Es decir: la clase *TSBSimpleList* implementa *ella misma* el *patrón Iterator* (aunque, como se dijo, en forma liviana) proveyendo de métodos para facilitar su recorrido externo sin violar el encapsulamiento.

Para cumplir con el patrón, hemos incorporado un atributo adicional a la clase *TSBSimpleList*: la referencia *actual* apuntará al último nodo que haya sido procesado si se está recorriendo la lista desde el exterior:

```
public class TSBSimpleList
{
	private Node frente;
	private Node actual;

	public SimpleList
	{
		frente = null;
		actual = null;
	}

	// resto de la clase aquí...
}
```

El valor inicial de *actual* es *null*. La idea es que si desde el exterior se quiere recorrer la lista, se debe invocar al método *next()* (que se incorporó a la clase) y este método retornará el *info* del nodo *siguiente* al que apunta *actual* en ese momento. Si *actual* es ya el último nodo de la lista, o la lista está vacía, el método *next()* lanzará una excepción de la clase *NoSuchElementException*. Para evitar esa excepción, se debería invocar al método *hasNext()* *antes* de invocar a *next()*: el método *hasNext()* retorna *true* si la lista tiene algún elemento que todavía no haya sido alcanzado por *next()*. Cada vez que se invoque a *next()*, el puntero *actual* se moverá al siguiente nodo de la lista, por lo cual es de esperar que en algún momento actual llegará al último nodo: de allí en adelante, cualquier invocación a *next()* provocará una excepción y una llamada a *hasNext()* retornará *false*. Cuando eso ocurra, la lista se habrá terminado de recorrer. Si el programador necesita volver a recorrer la lista o en la mitad de un recorrido necesita volver al principio, entonces deberá invocar al método

startIterator(), el cual reinicia el recorrido del mecanismo del iterador liviano que hemos implementado: este método simplemente vuelve a poner *null* en la referencia *actual*. Los métodos citados se implementaron así:

public void starIterator()

```
public void startIterator()
{
   actual = null;
}
```

public boolean hasNext()

```
public boolean hasNext()
{
   if ( frente == null ) return false;
   if ( actual != null && actual.getNext() == null ) return false;
   return true;
}
```

public Comparable next()

```
public Comparable next()
{
    if(!hasNext()) throw new NoSuchElementException("No quedan elementos");

    if(actual == null) actual = frente;
    else actual = actual.getNext();
    return actual.getInfo();
}
```

La clase *Principal* del modelo *TSB-SimpleList04* muestra un *main()* en el cual se crea una lista de objetos de cuentas de *Inversion*, y procede a recorrer esa lista dos veces mediante el mecanismo de iteración propuesto: en el primer recorrido calcula el saldo promedio, y en el segundo determina cuántas cuentas tienen saldo mayor al promedio:

```
public class Principal
{
    public static void main(String args[])
    {
        // creamos una lista para guardar cuentas de Inversion
        TSBSimpleList c = new TSBSimpleList();
        c.addInOrder( new Inversion (101, 2000, 2.1f) );
        c.addInOrder( new Inversion (212, 1000, 1.2f) );
        c.addInOrder( new Inversion (511, 3000, 3) );
        System.out.println( "\nLista de Cuentas: " + c );

        // ahora recorremos la lista para calcular el saldo promedio...
        // ...usamos nuestra implementación liviana del patrón Iterator
        float a = 0;
        int b = 0;
        c.startIterator();  // inicializamos el mecanismo de recorrido
        while( c.hasNext() )
        {
            Inversion x = (Inversion) c.next();
            a += x.getSaldo();
            b++;
        }
        float prom = 0;
        if( b != 0 ) prom = a / b;
```

```
        System.out.println("Saldo Promedio: " + prom);

        // recorrer la lista otra vez: reinicializar el iterador...
        c.startIterator();
        int s = 0;
        while( c.hasNext() )
        {
            Inversion y = (Inversion) c.next();
            if ( y.getSaldo() > prom ) s++;
        }
        System.out.println("Cantidad con saldo mayor al promedio: " + s);
    }
}
```

2.] El patrón Iterator (implementación detallada).

En la sección anterior hemos mostrado la forma de implementar el *patrón Iterator* en forma muy elemental para permitir el recorrido de una lista sin develar su estructura interna. Para ello, la clase *TSBSimpleList* incluía algunos métodos simples para iniciar el iterador, comprobar si existe un elemento sin visitar y recuperar el siguiente elemento que no haya sido visitado.

La ventaja de nuestra implementación básica o liviana es la sencillez: sin demasiado trabajo, la propia clase *TSBSimpleList* provee métodos para facilitar el recorrido. Sin embargo, esta implementación liviana tiene al menos un problema evidente: *no se pueden implementar múltiples recorridos de una misma lista en un mismo proceso*. Si se requiere avanzar en la lista con dos o más iteradores en paralelo, la implementación liviana que hemos sugerido no permite hacerlo.

El *patrón Iterator* es en realidad más complejo justamente para poder enfrentar esta clase de situaciones. La idea general es que si se necesita recorrer una colección (por ejemplo una lista) sin develar su estructura interna, se le pida a esa colección que cree y provea un *objeto separado* (llamado justamente *objeto iterador*) tal que ese objeto provea los métodos para recorrerla (y no que sea la propia colección la que implemente los recorridos). De esta forma, *se pueden crear tantos objetos iteradores como recorridos se necesiten hacer*, y luego pedir a cada iterador que se encargue de avanzar en la colección. Además, cada clase que implemente un proceso de iteración puede plantear el recorrido en otras formas (y no necesariamente "hacia adelante"), lo cual constituye una ventaja adicional.

El *patrón Iterator* formalmente puede implementarse de distintas formas, pero todas finalmente coinciden en los elementos esenciales. Básicamente, y para facilitar el trabajo en contextos polimórficos, se requiere primero de una *clase de interface* que enumere los métodos que debería implementar una clase iteradora (por ejemplo, los métodos *hasNext()* y *next()* entre otros). Así, toda clase que se diseñe para controlar el recorrido de una colección, tendrá las mismas responsabilidades que las otras y los métodos a usar serán generales. Si bien el programador puede diseñar su propia clase de interface para enumerar estos métodos, lo cierto es que Java ya provee algunas de estas interfaces ya listas para importar e implementar. Entre esas interfaces nativas, citamos dos:

- *java.util.Iterator*: Esta interface requiere la implementación de solo tres métodos: *hasNext()* (para chequear si se ha llegado al final de la colección recorrida), *next()* (para recuperar el siguiente objeto) y *remove()* (para eliminar el objeto actualmente referenciado por el iterador). El constructor de una clase que implemente *Iterator*, debería iniciar el iterador para que comience el recorrido desde donde el programador haya dispuesto (por ejemplo, el frente de una lista). La idea es que un objeto *Iterator* permita recorrer en una sola dirección una colección, de tal forma que al llegar al final de la misma deba crearse otro *Iterator* para volver a recorrerla.

- *java.util.ListIterator*: Esta interface *deriva de la interface Iterator*, por lo que contiene los tres métodos ya indicados por *Iterator*, pero además enumera un conjunto amplio de otros métodos, que permiten recorrer la colección "hacia adelante" y "hacia atrás", además de prever métodos no sólo para remover el elemento actual, sino también para insertar un nuevo elemento en la posición actual.

Está claro que si se desea implementar una clase iteradora para una lista, la interface *ListIterator* sería la adecuada para proveer exhasutivamente el conjunto de métodos a usar para un recorrido. Sin embargo, considerando que nuestra clase *TSBSimpleList* no prevé enlaces que permitan recorridos "hacia atrás" y que además buscamos mostrar en forma rápida la manera de implementar el patrón, es que nos concentraremos en la interface *Iterator* para mayor sencillez.

Con estas ideas en mente, lo primero entonces sería diseñar la clase iteradora que implemente *Iterator*. Esa clase debería contener una referencia al nodo actualmente accedido por el iterador, y otra referencia que mantenga la dirección del nodo anterior al actual, para facilitar luego la tarea de remoción del nodo actual. El modelo *TSB-SimpleList05* contiene todos los elementos que mostraremos de aquí en adelante y hasta el final de la esta sección. Una clase iteradora para nuestra clase *TSBSimpleList* podría tener el siguiente aspecto:

```
import java.util.Iterator;
public class TSBSimpleListIterator implements Iterator
{
    private TSBNode actual; // dirección del nodo que toca procesar.
    private TSBNode previo; // dirección del nodo anterior al actual.

    public TSBSimpleListIterator()
    {
        actual = null;
        previo = null;
    }

   /**
    * Indica si queda algun objeto en el recorrido del iterador.          *
    * @return true si queda algun objeto en el recorrido - false si no
    * quedan objetos.
    */
   public boolean hasNext()
   {
        if (frente == null)
        {
            return false;
        }
        if (actual != null && actual.getNext() == null)
        {
            return false;
        }
        return true;
```

```
    }

    /**
     * Retorna el siguiente objeto en el recorrido del iterador.          *
     * @return el siguiente objeto en el recorrido.
     * @throws NoSuchElementException si la lista esta vacia o en la lista
     * no quedan elementos por recorrer.
     */
    public Comparable next()
    {
       if (!hasNext())
       {
           throw new NoSuchElementException("No quedan elementos");
       }

       if (actual == null)
       {
           actual = frente;
       }
       else
       {
           previo = actual;
           actual = actual.getNext();
       }
       return actual.getInfo();
    }

    /**
     * Elimina el objeto actual del iterador. El iterador queda ubicado en el
     * elemento siguiente al eliminado.
     */
    public void remove()
    {
       TSBNode aux = actual;
       if (actual != null)
       {
           if (previo == null)
           {
               frente = actual.getNext();
           }
           else
           {
               previo.setNext(actual.getNext());
           }
           actual = actual.getNext();
           aux.setNext(null);
       }
       else
           throw new NoSuchElementException();
    }
}
```

El constructor de la clase *TSBSimpleListIterator* sólo pone en null las referencias *actual* y *previo* (lo cual equivale a "iniciar el iterador desde el principio") El método *hasNext()* retorna *true* si la lista contiene al menos un nodo que aún no haya sido alcanzado por la referencia *actual* (*false* en caso contrario) El método *next()* avanza el puntero *actual* hacia el siguiente nodo y retorna el info del mismo (note que si no existe un nodo sucesor al actual y se invoca a *next()*, se producirá una excepción de la clase *NoSuchElementException*) Y el método *remove()* ellimina el nodo apuntado por actual en ese momento, avanzando el iterador al nodo siguiente (o a *null* si el nodo removido era el úlitmo de la lista). Esos son los tres métodos exigidos por la *interface Iterator*, y salvo por el constructor, son los únicos métodos que la clase *TSBSimpleListIterator* provee.

Así planteada la clase iteradora, surge un inconveniente: como puede observarse, para poder recorrer una lista representada por una instancia de la clase *TSBSimpleList*, la clase

iteradora *TSBSimpleListIterator* necesita acceder al valor del puntero *frente*, que está declarado como *private* en la clase *TSBSimpleList*, ya que sólo de esa forma el iterador sabrá dónde comienza la lista a recorrer. Y si *frente* es *private*, entonces todos los métodos de la clase *TSBSimpleListIterator* provocarán error de compilación al intentar accederlo. Para evitar este problema, una solución simple y elegante consiste en declarar a la clase *TSBSimpleListIterator* como *interna* de la clase *TSBSimpleList*.

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. 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 *TSB-SimpleList05* la clase *TSBSimpleList* declara como *clase interna* a la clase *TSBSimpleListIterator* que hemos mostrado más arriba, de acuerdo al siguiente esquema:

```
import java.util.Iterator;
public class TSBSimpleList
{
      private TSBNode frente;
      private int cantidad;

      // resto de la clase TSBSimpleList aquí…

      // clase interna para implementar el iterador…
      private class TSBSimpleListIterator implements Iterator
      {
          private TSBNode actual;
          private TSBNode previo;

          public TSBSimpleListIterator()
          {
          // contenido del método aquí…
          }

         public boolean hasNext()
         {
          // contenido del método aquí…
         }

         public Comparable next()
         {
          // contenido del método aquí…
         }

         public void remove()
         {
          // contenido del método aquí…
         }
      }
}
```

De esta forma, la clase interna *TSBSimpleListIterator* no tendrá problemas en acceder al atributo *frente* de la clase *TSBSimpleList*, y sus métodos harán su trabajo sin restricciones. Sólo nos queda un detalle: la clase *TSBSimpleList* ahora debe proveer un método que

permita crear y retornar un objeto de la clase *TSBSimpleListIterator*: esta última no sólo es interna sino también *private* en la clase *TSBSimpleList*, y por lo tanto su propia existencia es invisible para métodos de otras clases.

El método de *TSBSimpleList* que cree y retorne el objeto iterador podría ser cualquiera, pero en un contexto polimórfico se esperaría que los métodos que implementan el patrón sean estandarízados en forma general y conocida por todo programador. Para lograr esto, la clase *TSBSimpleList* debería entonces implementar una clase de interface que enuncie ese método para crear el iterador, de forma que toda clase que represente una colección que pueda recorrerse con iteradores, invoque de la misma forma al método que los crea. En Java esa clase de interface ya está prevista y definida en forma nativa: se trata de *java.lang.Iterable*, la cual sólo define un único método llamado *iterator()*. El objetivo de este método, es simplemente crear y retornar el iterador deseado. La versión final de la clase *TSBSimpleList* (tomada del modelo *TSB-SimpleList05*) quedaría entonces así:

```
import java.util.Iterator;
public class TSBSimpleList implements Iterable
{
      private TSBNode frente;
      private int cantidad;

      public Iterator iterator()
      {
        return new TSBSimpleListIterator();
      }

      // resto de la clase TSBSimpleList aquí…

      // clase interna para implementar el iterador…
      private class TSBSimpleListIterator implements Iterator
      {
          private TSBNode actual;
          private TSBNode previo;

          public TSBSimpleListIterator()
          {
          // contenido del método aquí…
          }

         public boolean hasNext()
         {
          // contenido del método aquí…
         }

         public Comparable next()
         {
          // contenido del método aquí…
         }

         public void remove()
         {
          // contenido del método aquí…
         }
      }
}
```

Con todo lo expuesto, la implementación queda finalizada y sólo resta mostrar ejemplo de uso y aplicación. La clase *Principal* del modelo *TSB-SimpleList05* contiene un método *main()* con esos ejemplos. Lo primero que se muestra en ese *main()* es la creación de una lista *c* con algunos objetos de la clase *Inversion*:

```
TSBSimpleList c = new TSBSimpleList();
c.addInOrder( new Inversion (101, 2000, 2.1f) );
c.addInOrder( new Inversion (212, 1000, 1.2f) );
c.addInOrder( new Inversion (511, 3000, 3) );
System.out.println( "\nLista de Cuentas: " + c );
```

Luego se muestra un primer recorrido de esa lista *c* usando el iterador que acabamos de implementar:

```
float a = 0;
int b = 0;
Iterator it = c.iterator();  // inicializamos el mecanismo de recorrido
while( it.hasNext() )
{
    Inversion x = (Inversion) it.next();
    a += x.getSaldo();
    b++;
}
float prom = 0;
if( b != 0 ) prom = a / b;
System.out.println("Saldo Promedio: " + prom);
```

Observe que en la tercera línea del esquema anterior *se crea un objeto iterador invocando al método iterator()* que la lista provee (por haber implementado *Iterable*), y el ciclo *while* que comienza en la cuarta línea usa el iterador (llamado *it* en el ejemplo) para recorrer la lista: el método *hasNext()* permite controlar que no se llegue al final de la lista, y el método *next()* accede al siguiente elemento.

En el mismo *main()* se recorre nuevamente la misma lista *c*, *creando un nuevo iterador it2* y usando ahora un ciclo *for*, más compacto:

```
int s = 0;
for(Iterator it2 = c.iterator(); it2.hasNext();  )
{
    Inversion y = (Inversion) it2.next();
    if ( y.getSaldo() > prom ) s++;
}
System.out.println("Cantidad de cuentas con saldo mayor al promedio: " + s);
```

3.] Clases Parametrizadas (Generics).

En nuestra última versión de la clase *TSBSimpleList*, uno de los agregados finales fue el control de homogeneidad. Usando el método *getClass()* controlábamos 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, asguramos la homogeneidad.

Sin embargo, notemos que a partir de la versión 1.5 del lenguaje Java se incorpora un mecanismo de control automático del tipo los objetos en una colección, 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 *Vector*, por ejemplo) pueden definirse en forma clásica, o pueden definirse también con un tipo base, cuyos objetos son los únicos que podrán insertarse en la colección:

```
LinkedList a = new LinkedList();   // definición y creación clásica: heterogénea.
LinkedLis<String> b = new LinkedList<String>();   // generics: homogénea.
```

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 *LinkedList 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 1.5 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 *TSBSimpleList*, 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 la versión 1.5 para que los programadores puedan incorporar *generics* en sus propias clases. Por lo tanto, podemos intentar modificar nuestro planteo de *TSBSimpleList*, para que pueda parametrizarse y dejar en manos del compilador el control de homogeneidad...

En principio, es suficiente con declarar las clases *TSBNode* y *TSBSimpleList* de la siguiente forma (ver modelo *TSB-SimpleList-GenericsA*):

```
public class TSBNode <E>
{

       private E info;
       private TSBNode next;

       public TSBNode (E x, TSBNode p)
       {
              info = x;
              next = p;
       }
       ….
}

public class TSBSimpleList <E>
{

      private TSBNode <E> frente;

      public void addFirst(E x)
      {
           ...
```

```
        }

        ...
}
```

Al hacer lo anterior, ambas clases se parametrizan. La expresion *<E> representa una clase* (***no*** una variable) que se envía como parámetro a las clases *TSBNode* y *TSBSimpleList* al crear un objeto con *new*. De aquí en más, se podrán hacer creaciones de objetos de la forma:

```
TSBNode < String > p = new TSBNode<String>("casa", null);
TSBSimpleList <Cuenta> lis = new TSBSimpleList<Cuenta>();
```

Pero debe notarse que lo anterior (a partir de Java 7) es equivalente a las siguientes instrucciones simplificadas:

```
TSBNode <String> p = new TSBNode<>("casa", null);
TSBSimpleList < Cuenta > lis = new TSBSimpleList<>();
```

En el primer caso, se está creando un objeto *p* de la clase *TSBNode*, pero específicamente para contener una cadena en su *info*: en el nodo *p* no podrá almacenarse otra cosa que un *String*, y cualquier intento de hacer otra cosa no compilará. En la segunda linea se está creando un objeto *lis* de la clase *TSBSimpleList*, pero en esa lista sólo se podrán almacenar objetos de la clase *Cuenta*.

Note que sigue siendo válido crear objetos sin parametrizar. Esto es así pues se debió mantener compatibilidad con las versiones anteriores del lenguaje. Aún si las clases *TSBNode* y *TSBSimpleList* están parametrizadas, se puede seguir creando objetos en la forma:

```
TSBNode p = new TSBNode();
TSBSimpleList lis = new TSBSimpleList();
```

Y la diferencia es simple: en el nodo creado, el *info* puede contener cualquier clase de objeto, y la lista admitirá de manera heterogénea cualquier objeto para ser insertado en ella.

En principio, y como consecuencia entonces de todo lo anterior, la clase *TSBNode* debe modificar su contenido, de forma que en todo método donde hasta ahora se trabajaba con *Comparable*, diga ahora *E*:

```
public class TSBNode <E>
{

   private E info;
   private TSBNode <E> next;

   public TSBNode ( )
   {
   }

   public TSBNode (E x, TSBNode p)
   {
     info = x;
     next = p;
   }

   public TSBNode <E> getNext()
```

```
    {
      return next;
    }

    public void setNext(TSBNode <E> p)
    {
      next = p;
    }

    public E getInfo()
    {
      return info;
    }

    public void setInfo(E p)
    {
      info = p;
    }
    ....
}
```

La idea es que la clase que sea que se haya enviado como parámetro a la clase *TSBNode*, será entendida en lugar de *E* cada vez que se instancie *TSBNode*, y el propio compilador controlará la compatibilidad de la clase del objeto enviado con la verdadera clase *E* que se informó al declarar la variable. En la clase *TSBSimpleList* corresponde un cambio similar, pero ahora ya no será necesario hacer el control de homogeneidad que hacíamos antes con *getClass()*: el método *isHomogeneus()* ya no hace falta en la clase. Como ejemplo, mostramos la forma en que quedó el método *addFirst()* y el resto puede verse en el modelo *TSB-SimpleListGenericsA* (en ese modelo, por las razones que se explican un poco más abajo, no están incluidos los métodos que requieren el uso de *compareTo()* y por razones de simplificación tampoco se incluyó el mecanismo *iterador*):

```
public void addFirst(E x)
{
      if (x == null) return;

      TSBNode<E> p = new TSBNode<>(x, frente);
      frente = p;
}
```

Así planteada, la clase *TSBSimpleList* sigue siendo genérica y controla homogeneidad, pero surge un nuevo problema: hemos renunciado a que el *info* de cada *TSBNode* sea *Comparable*... y eso implica que ya no podríamos usar métodos como *addInOrder()*, *contains()* o *search()* (aunque estos dos últimos podrían incluirse igual si usamos *equals()* en lugar de *compareTo()*) Si queremos mantener nuestra idea original que en la lista sólo puedan incluirse objetos de clases que hayan implementado *Comparable*, podemos hacerlo en base a un pequeño cambio: al declarar el parámetro de clase *E*, indicar que esa clase deriva de (o implementa a) *Comparable*:

```
public class SimpleList <E extends Comparable>
```

Aquí la palabra *extends* se usa indistintamente para señalar que la clase *E* debe *derivar de* o *implementar a* la clase que se nombre en segundo lugar. Esta notación no ha incorporado la palabra *implements* si la segunda clase es una interface: debe usar *extends* en ambos casos.

Si tanto la clase *TSBNode* como la clase *TSBSimpleList* se declaran de esta forma, entonces ambas clases sólo aceptarán objetos de clases que hayan implementado *Comparable*, y no hay más cambios significativos que hacer en ese sentido. Nuestra clase *TSBSimpleList* puede incorporar ahora los métodos *addInOrder()*, *contains()*, *search()*, etc. (ver modelo *TSB-SimpleList-GenericsB*):

```
public class TSBSimpleList<E extends Comparable>
{

      private TSBNode<E> frente;
      …
      public void addInOrder(E x)
      {
            if ( x == null ) return;
            TSBNode<E> nuevo = new TSBNode<>(x, null);
            TSBNode<E> p = frente, q = null;
            while(p != null && x.compareTo(p.getInfo()) >= 0)
            {
                q = p;
                p = p.getNext();
            }
            nuevo.setNext( p );
            if( q != null ) q.setNext( nuevo );
            else frente = nuevo;
      }
      ...
}
```

Note además, que la propia interface *Comparable* está parametrizada a partir de la versión 1.5 del lenguaje. Hasta esta versión, al implementar el método *compareTo()* se debía hacer una operación de casting explícito para convertir el parámetro tomado a una referencia del tipo del objeto que invocó al método y evitar una excepción de conversión de tipos imposible. Por ejemplo, la forma típica en que la clase *Cliente* puede implementar la interface *Comparable* sería la siguiente (tomada del modelo *TSB-SimpleList-GenericsA*):

```
public class Cliente implements Comparable
{

    private int dni;
    private String nombre;

    // resto de la clase aquí…

    public int compareTo(Object x)
    {
       Cliente c = (Cliente) x;
       return this.getDni() - c.getDni();
    }
}
```

Sin embargo, a partir de la versión 1.5 *la interface* ***Comparable*** *puede ser parametrizada*, de forma de indicarle al método *compareTo()* de qué clase se espera que sea el objeto que entrará como parámetro explícito, y de esta forma evitar la necesidad del casting explícito. Una forma de hacerlo puede verse a continuación en la misma clase *Cliente*, pero tomada ahora del modelo *TSB-SimpleList-GenericsB*:

```
public class Cliente implements Comparable<Cliente>
{

    private int dni;
```

```
    private String nombre;

    // resto de la clase aquí…

    public int compareTo(Cliente x)
    {
       return this.getDni() - x.getDni();
    }
}
```

En el modelo *TSB-SimpleList-GenericsB* se han parametrizado las clases *TSBNode* y *TSBSimpleList*, y además las clases *Cuenta* y *Cliente* se plantearon de forma que implementan *Comparable* pero con parametrización, para evitar el casting explícito.

Para terminar, observe que a partir de Java 1.5 también están parametrizadas las interfaces *Iterator* e *Iterable* en forma similar a *Comparable*: la interface *Iterator* puede recibir como parámetro al declararse la clase a la que pertenecen los objetos de la lista sobre la que se itera, y que es también el tipo de objeto que debe retornar el método *next()*, y la interface *Iterable* puede parametrizarse en forma similar para indicarle a la colección subyacente el tipo de objetos que retornarán sus iteradores.

En el modelo *TSB-SimpleList-GenericsB* la clase *TSBSimpleList* declara una clase interna llamada *TSBSimpleListIterator*, para permitir el recorrido de la lista. A su vez la clase *TSBSimpleListIterator* implementa la interface *Iterator*, parametrizándola en el momento de declarar esa implementación:

```
import java.util.Iterator;
public class TSBSimpleList <E extends Comparable> implements Iterable<E>
{

      private TSBNode<E> frente;
      private int cantidad;

      // resto de la clase aquí…

      public Iterator<E> iterator()
      {
           return new SimpleListIterator();
      }

      private class TSBSimpleListIterator implements Iterator <E>
      {

          private TSBNode<E> actual; // el nodo actual…
          private TSBNode<E> previo; // el nodo anterior al actual...

          public TSBSimpleListIterator()
          {
             actual = null;
             previo = null;
          }

         public boolean hasNext()
         {
              if (frente == null)
              {
                  return false;
              }
              if (actual != null && actual.getNext() == null)
              {
                  return false;
              }
              return true;
         }
```

```
            public E next()
            {
               if (!hasNext())
               {
                   throw new NoSuchElementException("No quedan elementos");
               }

               if (actual == null)
               {
                   actual = frente;
               }
               else
               {
                   previo = actual;
                   actual = actual.getNext();
               }
               return actual.getInfo();
            }

            public void remove()
            {
               TSBNode <E> aux = actual;
               if (actual != null)
               {
                   if (previo == null)
                   {
                       frente = actual.getNext();
                   }
                   else
                   {
                       previo.setNext(actual.getNext());
                   }
                   actual = actual.getNext();
                   aux.setNext(null);
               }
               else
                   throw new NoSuchElementException();
            }
        }
    }
```

Como vimos, el hecho de que la clase *TSBSimpleList* implemente *Iterable* permite que luego esa lista pueda ser recorrida en forma explícita por medio de un iterador que implemente *Iterator*, pero ahora también puede ser recorrida en forma implícita por medio del *ciclo for – each*, como se muestra en el *main()* de la clase *Principal* del modelo *TSB-SimpleList-GenericsB*:

```
        // una lista de numeros...
        TSBSimpleList <Integer> lista = new TSBSimpleList<>();
        lista.addInOrder(1);
        lista.addInOrder(2);
        lista.addInOrder(3);
        lista.addInOrder(4);
        lista.addInOrder(5);

        // mostrar la lista a partir del iterador...
        System.out.println();
        Iterator <Integer> i = lista.iterator();
        while(i.hasNext())
        {
            System.out.println("Objeto recuperado: " + i.next() );
        }

        // suma de la lista de numeros... usando el ciclo for each...
        int suma = 0;
        for(int item : lista)
        {
```

```
        suma += item; // aca hace auto-deboxing...
    }

    System.out.println("\nSuma: " + suma);
```

4.] Paquetes de clases: conceptos introductivos.

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, 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*.

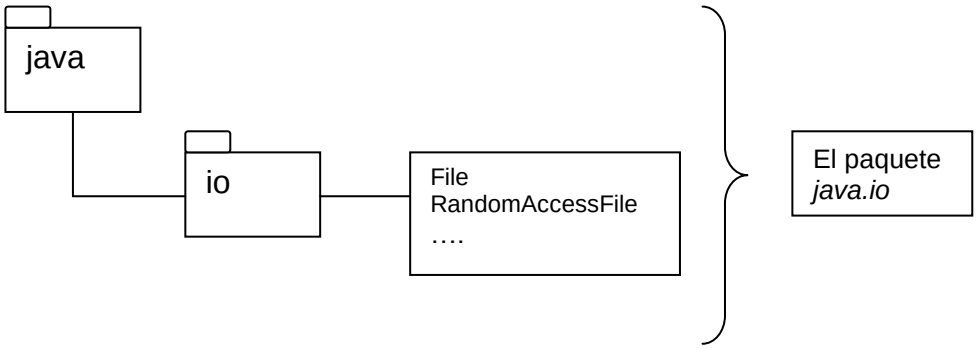
No es obligatorio que las clases de un package tengan entre ellas relación de herencia. De hecho, podrían no tener ninguna relación entre ellas, más que la nombrada de la funcionalidad común. Por otra parte, tampoco hay nada que prohiba que una clase de un package derive de otra ubicada en el mismo package 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 puede 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 reusar 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 hecer 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 el mismo package. Pero sí 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 los 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.io*, estamos diciendo que existe una carpeta *java*, la cual a su vez contiene una carpeta *io*, y en esa carpeta se guardan las clases para la gestión de archivos de Java:

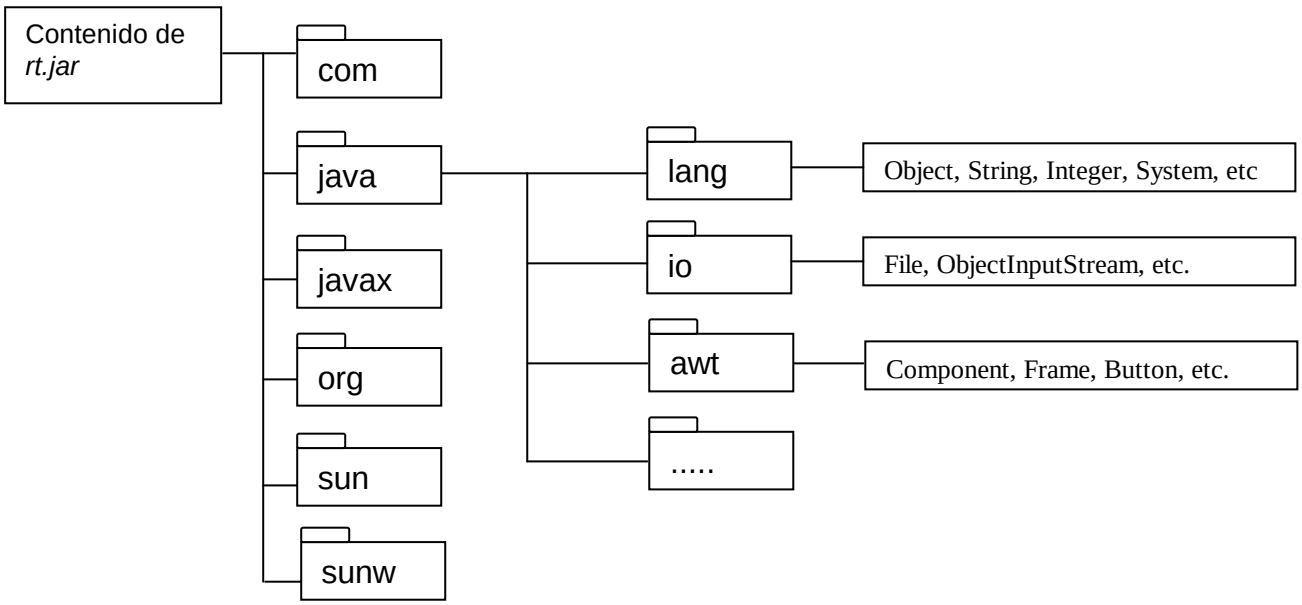


Tenga en cuenta que un package finalmente es la forma en que se agrupan las clases frente al intérprete o máquina virtual Java (JVM), y que desde esta perspectiva finalmente lo que un package contiene son *los archivos compilados de las clases (o sea, los archivos .class)* Por razones prácticas obvias, 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.

5.] Clases nativas de Java. Uso de import y package.

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 *rt.jar* (ubicado en la carpeta *jdk1.6.0\jre\lib* o similar, donde se instaló el *JDK* en su computadora). 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*. La carpeta *javax* contiene al package *swing* que incluye casi todas las clases de *Swing* para el desarrollo de interfaces avanzadas de usuario (entre ellas, *JOptionPane*).

El gráfico siguiente 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 *Program Files\jdk1.6.0* 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 paquete *java.io*, podría hacerlo así:

```
import java.io.*;
public class Ejemplo
{
    private File f1;
    private RandomAccessFile r1;
            ….
}
```

De esta forma, la JVM sabrá dónde ubicar el código compilado de las clases *File* y *RandomAccessFile* y de cualquier otra clase que esté en *java.io*. 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.io.File;
import java.io.RandomAccessFile;
public class Ejemplo
{
      private File  f1;
      private RandomAccessFile r1;
      ...
}
```

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.io.File f1;
    private java.io.RandomAccessFile r1;
```

```
        public Ejemplo()
        {
            f1 = new java.io.File("prueba.txt");
            r1 = new java.io.RandomAccessFile(f1, "r");
        }
        ...
}
```

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 donde 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 llegara a prueba, se entenderá que esa carpeta está en el mismo directorio que el proyecto):

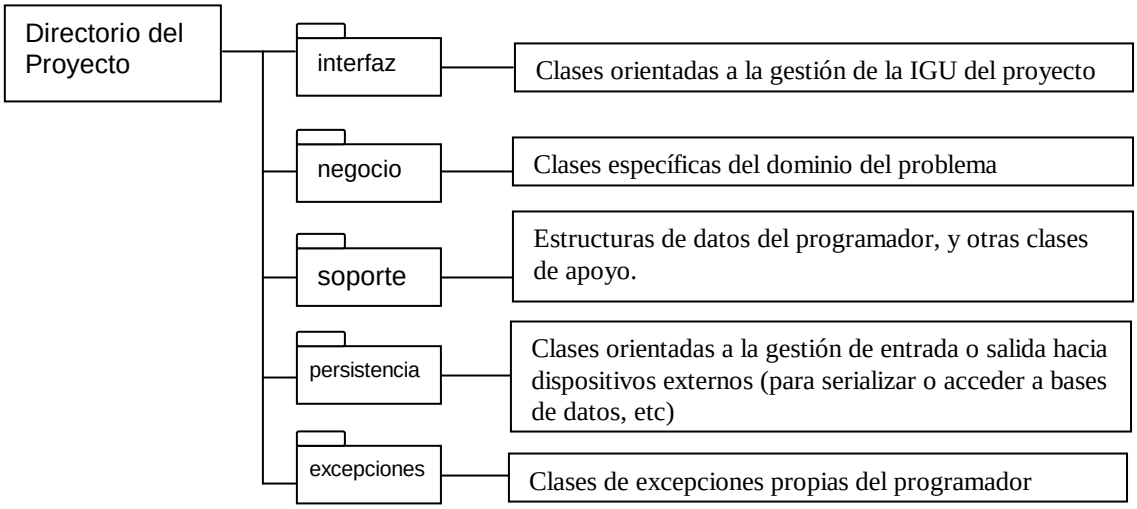
```
package prueba;
import java.awt.*;
import java.io.*;

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:



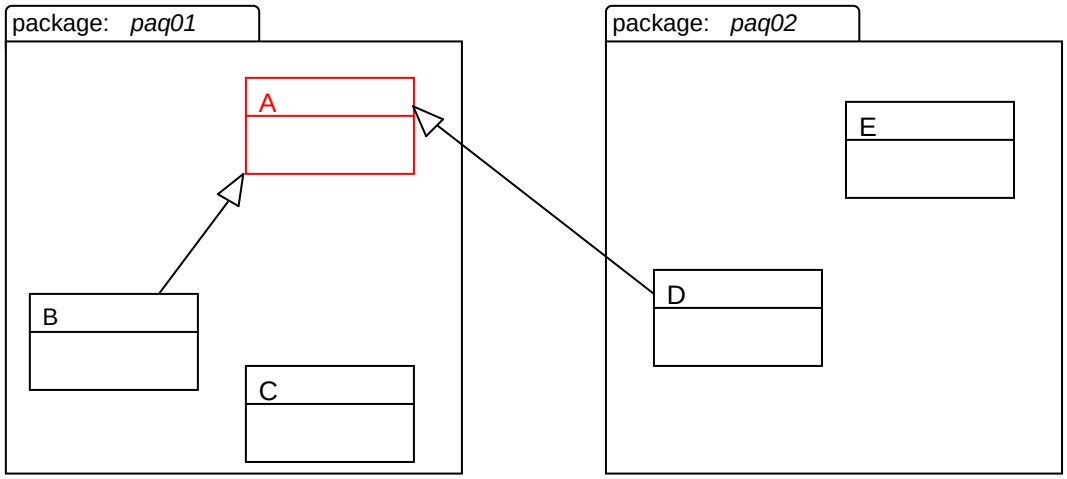
6.] Alcance de los calificadores public, private y protected.

Sabemos que los atributos y métodos de una clase pueden marcarse básicamente como *public* o *private*, para favorecer un mejor orden de acceso respecto del *Principio de Ocultamiento*. En esencia, si un miembro de una clase es *public*, ese miembro es accesible por cualquier método de esa clase o de cualquier otra clase. Pero si el miembro es *private*, sólo es accesible por métodos de esa misma clase, y no es visible ni accesible desde otras clases. Cuando se entra en contextos de herencia, suele usarse un tercer calificador de acceso designado como *protected*: si un miembro se marca como *protected*, en principio es accesible por esa misma clase y por cualquier clase derivada de ella. En otras palabras, un miembro *protected* es *público* para la clase y para sus derivadas, pero es *privado* para otras clases.

Sin embargo, existen sutilezas en cuanto al alcance de estos calificadores, sobre todo cuando se combina el efecto de la herencia con el uso de packages. Por lo pronto, considere que un atributo podría ser declarado *sin anteponer ningún calificador delante de él*, y eso en realidad implica *un cuarto nivel de acceso que no equivale estrictamente a ninguno de los otros tres*. Si un miembro no se marca con ningún calificador, se dice que su nivel de acceso es el *nivel por defecto o nivel default*. En el siguiente ejemplo, la clase *Prueba* tiene cuatro atributos, cada uno con un nivel de acceso difente:

```
public class Prueba
{
    private   int  a1;   // nivel de acceso: privado
    protected   int a2;  // nivel de acceso: protegido
    public   int a3;     // nivel de acceso: público
    int a4;              // nivel de acceso: default
    ...
}
```

Note que el atributo *a4* no tiene ningún calificador de acceso delante de él. Por lo tanto, su nivel de acceso es *default*, pero la palabra *default* ***NO ES una palabra reservada del lenguaje***. El significado preciso de cada calificador en un contexto amplio, en el cual hay clases que derivan entre ellas y pertenecen a paquetes eventualmente distintos, puede analizarse con más sencillez en una gráfica como la siguiente, y llevando todos los casos a una tabla. En el gráfico, se supone una *clase A* en el paquete *paq01*, la cual tiene atributos de distintos niveles de acceso (como la clase *Prueba* del ejemplo anterior). Suponemos además varias otras clases: alguna deriva de *A* y está en el mismo paquete *paq01* (como la *clase B*). Otra (como la *clase C*) no deriva de *A* pero está el mismo paquete *paq01*. Hay otra *clase D* que deriva de *A* pero está en el paquete *paq02*. Y finalmente tenemos la *clase E* que no deriva de *A* ni está en el mismo paquete que *A*. Nos preguntamos a qué atributos de *A* pueden acceder las otras clases del modelo, y eso se refleja en la tabla resumen de más abajo:



Nivel de acceso del miembro en *A* que se quiere acceder	**¿Puede acceder a ese miembro una clase con estas características?**			
	Derivada de *A* en el mismo package (*B*)	No derivada de *A*, pero en el mismo package (*C*)	Derivada de *A*, pero en un package distinto (*D*)	No derivada de *A*, y en un package distinto (*E*)
private	no	no	no	no
protected	sí	**sí**	sí	no
public	sí	sí	sí	sí
<default>	sí	**sí**	**no**	no

Observar que el acceso *default* entonces no es estrictamente equivalente a ninguno de los otros tres, aunque se parece mucho al acceso *protected*. Puede ver en acción estas restricciones en el modelo *TSBAccesos* que acompaña a esta Ficha. Note que si una clase tiene un miembro *default*, entonces toda clase que esté en el mismo package podrá acceder a ese miembro como si fuera público, sea o no derivada de la clase original. Pero si el miembro es *default* y se pretende accederlo desde una clase no derivada ubicada en *otro* package, el compilador no lo permitirá. El acceso *default* existe para permitir que clases ubicadas en el mismo package puedan operar en forma colaborativa sin tantas restricciones de ocultamiento: a fin de cuentas, si forman parte del mismo paquete es porque fueron pensadas para compartir funcionalidad y objetivos, así que en determinadas situaciones es aceptable que permitan acceder a sus atributos y miembros como si fueran públicos. El acceso *default* también suele llamarse *acceso a nivel de package*.

7.] Empaquetado y distribución de una aplicación: archivos jar.

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 *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 "*Buid 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 le 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 metodo *main()* que debe usarse para ejecutar el programa. Si esa línea no viene acompañada de ningún nombre de clase, entonces la JVM considera que el *.jar* no es un ejecutable y al hacer doble click sobre él no ocurrirá nada en especial (posiblemente, sólo se abra el archivo mediante *WinRar*).