Introducción – Qué es un paradigma de programación?

Un paradigma de programación provee (y determina) la visión y métodos de un programador en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas (con la solución de múltiples "problemas" se construye una aplicación).

Los lenguajes de programación son basados en uno o más paradigmas Por ejemplo: Smalltalk y Java son lenguajes basados en el paradigma orientado a objetos. El lenguaje de programación Scheme, en cambio, soporta sólo programación funcional. En cambio Python, soporta múltiples paradigmas.

Clasificación por paradigmas de programación

Paradigma Imperativo: describe la programación como una secuencia instrucciones o comandos que cambian el estado de un programa. El código máquina en general está basado en el paradigma imperativo. Su contrario es el paradigma declarativo. En este paradigma se incluye el paradigma procedimental (procedural) entre otros.
Paradigma Declarativo: No se basa en el cómo se hace algo (cómo se logra un objetivo paso a paso), sino que describe (declara) cómo es algo. En otras palabras, se enfoca en describir las propiedades de la solución buscada, dejando indeterminado el algoritmo (conjunto de instrucciones) usado para encontrar esa solución. Es más complicado de implementar que el paradigma imperativo, tiene desventajas en la eficiencia, pero ventajas en la solución de determinados problemas.
Paradigma Estructurado: la programación se divide en bloques (procedimientos y funciones) que pueden o no comunicarse entre sí. Además la programación se controla con secuencia, selección e iteración. Permite reutilizar código programado y otorga una mejor compresión de la programación. Es contrario al paradigma inestructurado, de poco uso, que no tiene ninguna estructura, es simplemente un "bloque", como por ejemplo, los archivos batch (.bat).
Paradigma Orientado a Objetos: está basado en la idea de encapsular estado y operaciones en objetos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes (programación orientada a mensajes). Se puede incluir -aunque no formalmente- dentro de este paradigma, el paradigma basado en objetos, que además posee herencia y subtipos entre objetos. Ej.: Simula, Smalltalk, C++, Java, Visual Basic .NET, etc.
Su principal ventaja es la reutilización de códigos y su facilidad para pensar soluciones a determinados problemas.
Paradigma Funcional: este paradigma concibe a la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos (como lo hace el paradigma procedimental). Permite resolver ciertos problemas de forma elegante y los lenguajes puramente funcionales evitan los efectos secundarios comunes en otro tipo de programaciones.
Paradigma lógico: se basa en la definición de reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas. Ej.: prolog.

Otros paradigmas y subparadigmas son: paradigma orientado al sujeto, paradigma reflectante, programación basada en reglas, paradigma basado en restricciones, programación basada en prototipos, paradigma orientado a aspectos, etc.

Nota del Coordinador: La cátedra de PPR, ha actualizado contenidos permanentemente. Los contenidos que se dictan reflejan en lo mas posible la programación que se usa aquí y ahora. Un primer objetivo de esta permanente actualización es lograr que sea lo mas mínimo posible el "gap" entre lo que se usa actualmente y lo que se enseña.

Unidad I – Programación Orientada a Objetos avanzada

Introducción a colecciones . . . 3
Interfaces de colecciones . . . 4
Interfaces de colección e implementaciones separadas . 5
Colecciones y las interfaces de iteración en la librería Java 5
Colecciones concretas 8
Listas enlazadas (LinkedList) . . . 8
public class LinkedListTest . . 12
public class SortedList extends LinkedList . 13
Array de listas (ArrayList) . . . 16
Acceso a los elementos de un ArrayList . . 17
Inserción y eliminación de elementos intermedios 18
Una implementación de ArrItems extendiendo ArrayList 21
Conjuntos de hash 22
Usando class HashSet . . . 23
Arboles 25
Árboles rojo-negro . . . 25
Comparación de objetos . . 26
Class TreeSet 28
Arbol TreeSet ordenado según interfaz Comparable . 28
Arbol TreeSet ordenado según objeto Comparator 29
Conjuntos de bits . . . 31
public class CribaEras extends BitSet . . 31
Herencia y Polimorfismo en Java. Caso concreto. . 33
public class Progression . . . 33
class ArithProgression extends Progression . 34
class GeomProgression extends Progression . . 35
class FibonacciProgression extends Progression . 35
class Tester 36

Autor. Ing. Tymoschuk, Jorge

Introducción a colecciones

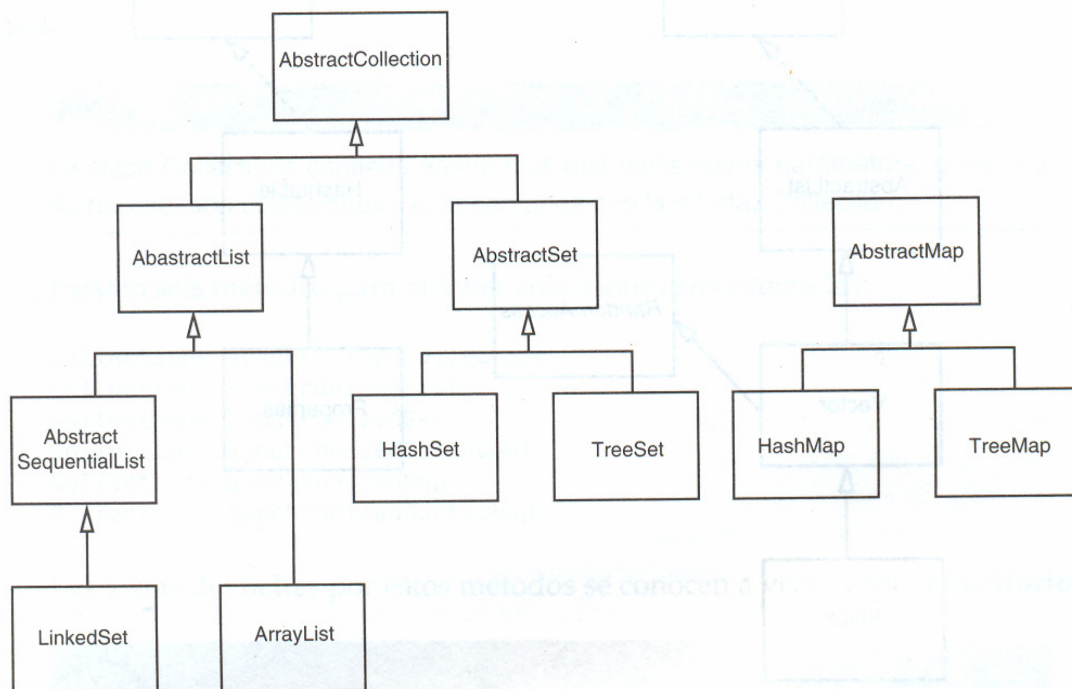
Antes de que apareciera la versión 2 de la plataforma Java, la librería estándar sólo soportaba un pequeño conjunto de clases para las estructuras de datos más habituales: Vector, Stack, Hashtable, BitSet y la interfaz Enumeration, que ofrecía un mecanismo abstracto para obtener elementos dentro de un contenedor arbitrario.

Con la aparición de la plataforma Java 2, los diseñadores sintieron que había llegado el momento de trabajar con un conjunto de estructuras de datos con todas las de la ley. En esta unidad trataremos del diseño básico de la estructura de colecciones Java, mostrándo cómo ponerlas a trabajar. Veremos cómo la tecnología Java puede ayudarle a llevar a cabo la estructuración tradicional de datos necesaria para la **programación profesional;** no se pretende cubrir en profundidad la totalidad de las colecciones que implementa Java, pero veremos ejemplos concretos de varias de las mas usuales.

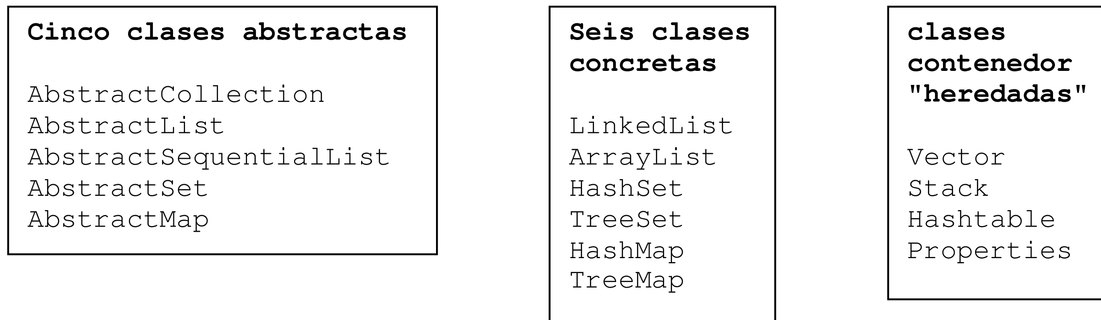
La estructura de las colecciones

Una estructura (framework) es un conjunto de clases que forman la base para una serie de funcionalidades avanzadas. Dichas estructuras pueden contener superclases con mecanismos, políticas y funcionalidades útiles. El usuario de una de estas estructuras compone subclases para heredar las funcionalidades sin que sea necesario reinventar los mecanismos básicos. Por ejemplo, Swing (Que utilizaremos en la siguiente unidad) es una estructura para interfaces de usuario.

La librería de colecciones Java está organizada en una estructura, que mostramos en la figura siguiente. Es una serie de clases abstractas e interfaces.



Estructura de clases de colecciones. Está formada por:



Cinco clases abstractas

AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractMap

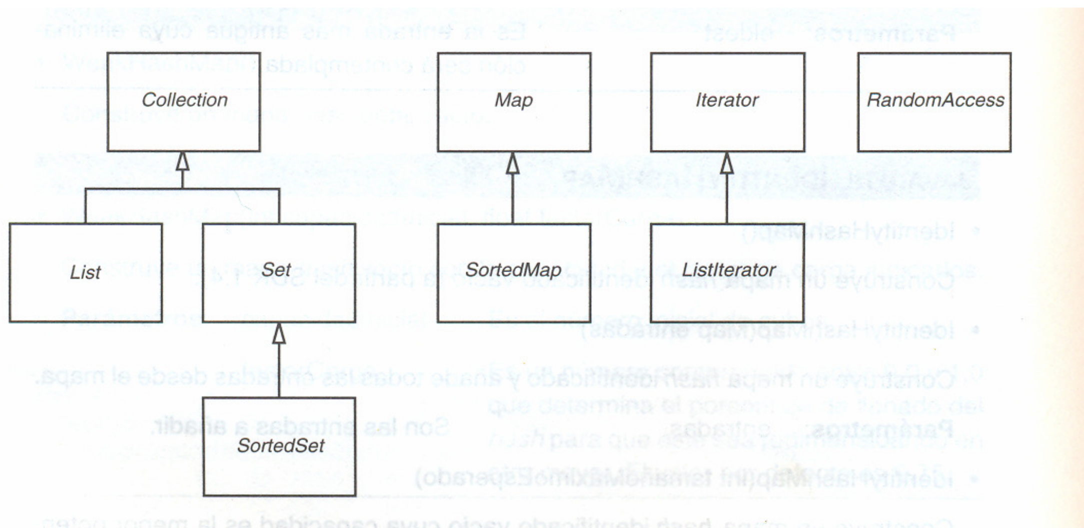
Seis clases concretas

LinkedList
ArrayList
HashSet
TreeSet
HashMap
TreeMap

clases contenedor "heredadas"

Vector
Stack
Hashtable
Properties

Estas clases implementan la siguiente **estructura de interfaces**



Ahora veamos como se usan.

Interfaces de colección e implementaciones separadas

Al igual que ocurre con las modernas estructuras de datos, Java separa las interfaces y las implementaciones. Vamos a echar un vistazo a esta separación con una conocida estructura de datos, la cola. La librería Java no soporta colas, pero esto no es impedimento para que resulte un buen ejemplo introductorio a los conceptos básicos.

NOTA: Ya hemos visto colas implementadas en una estructura de nodos vinculados en AED. (en esa ocasión hacíamos todo el trabajo de programación). Ahora le adelantamos que si necesita una cola, basta con usar la clase LinkedList que veremos más tarde en este capítulo.

Una interfaz de cola especifica que los elementos se añaden por la parte final de la misma, se eliminan por la cabecera y permite determinar cuántos elementos existen dentro de ella. Puede utilizar una cola cuando necesite una colección de objetos que se deban recuperar de la forma "primero en entrar, primero en salir" (“first in, first out")

Si hubiera una interfaz de cola en la librería de colecciones, podría parecer:

```
interface Queue{
      void add(Object obj);
      Object remove();
      int size();
}
```

La interfaz no indica nada acerca de la implementación de la cola. Clásicamente existen dos de estas implementaciones, una usa un "array circular" y otra que utiliza una lista de nodos vinculados.

```
class ColaEnArrayCircular implements Queue{
…   …
}

class ColaConNodosVinculados implements Queue{
…   …
}
```

Cuando use una cola en un programa, no necesitará saber cuál de estas implementaciones se ha utilizado después que instanciamos el objeto. Ejemplos:

```
Queue colin = new ColaEnArrayCircular[100];
colin.add(new Customer("Harry"));
```

Si optásemos por la otra implementación:

```
Queue colin = new ColaConNodosVinculados();
colin.add(new Customer("Harry"));
```

¿Qué puede llevarle a elegir entre una u otra implementación? La interfaz no aclara nada acerca de la eficacia de cada una de ellas. Un array circular es algo más eficaz que la lista enlazada, por lo que suele ser preferible su uso. Sin embargo, el array circular es una colección limitada; es decir, tiene una capacidad finita. Si no sabe con exactitud el número máximo de objetos que necesitará usar, mejor una lista enlazada.

Este ejemplo sirve para ilustrar otro tema en el diseño de una librería de clases de colecciones. Hablando estrictamente, en una colección limitada, la interfaz del método add debe indicar que el método puede fallar:

```
class ColaEnArrayCircular implements Queue{
     public void add(Object obj) throws CollectionFullException
     …
}
```

Y aquí comienzan los problemas, ya que la clase **ColaEnArrayCircular** no puede implementar la interfaz Queue, puesto que ***no se puede añadir especificadores de excepción cuando se sobrescribe un método***. ¿Debería tener dos interfaces, BoundedQueue y Queue? ¿O debería lanzar el método add una excepción no comprobada? Existen ventajas e inconvenientes en ambas implementaciones. Y son precisamente este tipo de situaciones las que hacen muy difícil diseñar de forma coherente una librería de clases de colecciones.

Resumiendo, la librería Java no dispone de una clase específica para las colas. Hemos utilizado este ejemplo para ilustrar la diferencia entre interfaz e implementación, ya que una cola tiene una sola interfaz con dos implementaciones distintas.

Colecciones y las interfaces de iteración en la librería Java

La interfaz fundamental de las clases de colección en la librería Java es Collection. Dicha interfaz tiene dos métodos fundamentales:

- **boolean add(Object obj)**
- **Iterator iterator()**

Además de estos dos, existen otros métodos que veremos más adelante.

El método add añade un objeto a la colección, y devuelve true si dicho objeto cambia la colección, o false en caso contrario. Por ejemplo, si intenta añadir un objeto a un conjunto y dicho objeto ya está presente en él, se rechaza la petición, ya que el conjunto repetidos.

El método iterator devuelve un objeto que implementa la interfaz Iterator. Puede utilizar un objeto iterador (iterator) para recorrer elementos del contenedor uno a uno.

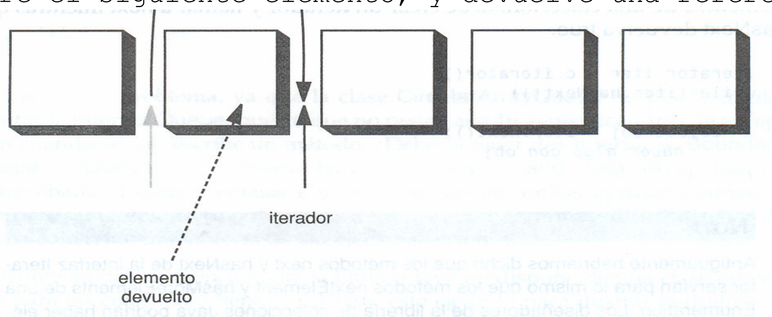
La interfaz Iterator dispone de tres métodos fundamentales:

- Object next()
- boolean hasNext()
- void remove()

Las llamadas repetidas a next le permiten recorrer los elementos de una colección uno a uno. Sin embargo, cuando se alcanza el final de la colección, next lanza una NoSuchElementException. Por consiguiente, es necesario invocar el método hasNext() antes de llamar a next(). Este método devuelve true si el objeto iterador aún dispone de más elementos que visitar. Si quiere inspeccionar todos los elementos de una colección, debe crear un iterador, llamar a next() mientras hasNext() devuelva true.

```
Iterator iter = c.iterator();
while (iter.hasNext()){
      Object obj = iter.next();
      // hacer algo con obj
}
```

Por ultimo, el método remove() elimina el elemento que ha sido devuelto por la última llamada a next.
Conceptualmente puede resultar extraño que el método remove forme parte de la interfaz Iterator, pero es muy funcional. Es sencillo eliminar un elemento si se sabe donde está, y como el iterador conoce las posiciones en la colección, es conveniente. Si visitó un elemento que no le gustaba, puede eliminarlo fácilmente.
La iteración recorre elementos. Cuando llame a next(), el iterador salta sobre el siguiente elemento, y devuelve una referencia al anterior.



Debe ser cuidadoso cuando use el método remove(). La llamada a este método elimina el elemento que fue devuelto por la última llamada a next(). Esto parece lógico si quiere eliminar un valor particular y a lo sumo necesita comprobar primero ese elemento antes de decidir si debe ser borrado o no. Pero si desea realizar la eliminación del elemento sobre el cual está posicionado, primero deberá avanzar al siguiente. Por ejemplo, aquí tiene la forma de eliminar el primer elemento de una colección.

```
Iterator it = c.iterator(); // Nos posicionamos al inicio
it.next(); // Avanzamos al siguiente
it.remove(); // ahora eliminamos el primero
```

Existe una dependencia entre las llamadas a los métodos next() y remove(). Es ilegal realizar una llamada a remove() si no ha existido una previa a next(). Si lo intenta, obtendrá una IllegalStateException.

Si quiere borrar dos elementos adyacentes:

```
it. remove () ;
it.next() ;
it. remove();
```

Ya que la colección y las interfaces iterador son genéricas, puede escribir métodos que operen sobre cualquier tipo de colección. Por ejemplo, aquí tiene un método print genérico que muestra todos los elementos de una colección.

```
public static void print(Collection e){
      System.out.print(“[ “);
      Iterator iter = c.iterator();
      while (iter.hasNext()
            System.out.print(iter.next() + “ “);
      System.out.println("]");
}
```

Un método que copia todos los elementos de una colección a otra podría ser:

```
public static boolean addAll(Collection desde, Collection hasta){
      Iterator iter = desde.Iterator();
      boolean modified = false;
      while (iter.hasNext()
            if (hasta.add(iter.next()))
                  modified = true;
      return modified;
}
```

Recuerde que la llamada al método add() devuelve true si la incorporación del elemento modifica la colección. Es posible implementar estos métodos para colecciones arbitrarias porque tanto la interfaz Collection como Iterator proporcionan los métodos fundamentales add() y next().

Los diseñadores de la librería Java vieron que algunos de estos métodos eran tan útiles que decidieron hacerlos accesibles. De esta forma, los usuarios no tendrían que reinventar la rueda. addAll() es uno de estos métodos.

La interfaz Collection declara algunos métodos útiles que implementan las clases más utilizadas. Entre ellos están:

```
int size()      //   Devuelve la cantidad de elementos de la colección
boolean isEmpty()    //   Devuelve verdadero si la colección está vacia
boolean contains(Object obj) // Dev. verdadero si obj existe en la col.
boolean containsAll(Collection e) // Verdadero si col. E está contenida
boolean equals(Object otra)     // Verdadero si las col. son iguales
boolean addAll(Collection otra) // Verd. si la colección invocante cambia
                                // por incorporación de elems. de otra.
boolean remove(Object obj)      // Verd. Si el objeto obj es removido
boolean removeAll(Collection otra) // Verd. Si la colección invocante
                                // cambia por remoción de eles. de otra
void clear()               //Elimina todos los elems. de col. invocante
boolean retainAll(Collection otra) // Elimina todos los elementos de la
                           // col. inv. que no pertenezcan a otra. Verda-
                           // dero si hay cambios.
Object[] toArray()   // Retorna un array con los objetos de la colección
```

Por supuesto, es bastante molesto si cada clase que implementa la interfaz Collection tenga que sustituir tantos métodos. Para facilitar en parte las implementaciones, tenemos una clase, la AbstractCollection que deja abstractos los métodos fundamentales (*como add() e iterator()*) pero implementa la rutina de los demás métodos en función de ellos. Por ejemplo:

```
public class AbstractCollection implements Collection{
     public abstract boolean add(Object obj);   //   Método abstracto
     public boolean addAll(Collection desde){   //   Método definido
          Iterator iter = desde.iterator();
          boolean modified = false;
          while (iter.hasNext())
               if (add(iter.next()))
                    modified = true;
          return modified;
     }
}
```

Una clase colección concreta puede ahora heredar la clase AbstractCollection. Ésta es la forma de suministrar a una clase colección concreta un método add, que debe definirse en la clase concreta. El método addAll está definido en la superclase AbstractCollection, pero, si la subclase dispone de una forma más eficiente de implementar addAll, es libre de redefinir el método heredado.

Éste es un buen diseño para la estructura de una clase. Los usuarios de las clases colección disponen de un amplio conjunto de métodos en la interfaz genérica, aunque los implementadores de las actuales estructuras de datos no tienen la obligación de la implementación de todos los métodos.

Colecciones concretas

Listas enlazadas

Una lista enlazada guarda cada objeto en un nodo enlace separado. Cada uno de estos enlaces también almacena una referencia al siguiente elemento de la secuencia. Ya hemos visto listas lineales implementadas

mediante estructuras de nodos vinculados, en la unidad III – Tipos de datos abstractos, en la asignatura AED – Algoritmos y Estructuras de Datos. Allí hacíamos todo el trabajo; Por ello es probable que no tenga muy buenos recuerdos acerca de lo intrincado que resultaba la eliminación o adición de elementos a la lista. En Java, todas las listas enlazadas son doblemente enlazadas; es decir, cada nodo guarda la referencia a su predecesor y siguiente.

Le alegrará saber que la librería de colecciones Java ofrece una clase LinkedList lista para usar. LinkedList implementa la interfaz Collection, y puede utilizar los métodos tradicionales para recorrer la estructura. El siguiente ejemplo muestra los tres elementos incluidos en una lista, y después los elimina.

```
LinkedList equipo = new LinkedList();
equipo.add("Angelina");
equipo.add("Pablito");
equipo.add("Carlitos”);
Iterator iter = equipo.iterator();
for (int i = 0; i < 3; i++){
     System.out.println(iter.next());//   Visitamos
     iter.remove();                  //   Eliminamos
}
```

Sin embargo, existe una importante diferencia entre las listas enlazadas y las colecciones genéricas. Una lista enlazada puede ser una colección ordenada donde importa la posición de los objetos. El método add() de LinkedList añade el objeto al final de la lista. Pero es muy probable que alguna vez necesite insertar un elemento en mitad de la misma, o dependiendo de algún ordenamiento. Este método add dependiente de la posición es responsabilidad de un iterador, ya que éstos mantienen posiciones en las colecciones.

El uso de iteradores sólo tiene sentido en colecciones que deben mantener un orden natural. Por ejemplo, el tipo de datos conjunto (set) que veremos mas adelante no impone ningún orden a sus elementos. Por consiguiente, no es necesario el método add de la interfaz Iterator. En cambio, la librería de colecciones dispone de una subinterfaz ListIterator que contiene un método add:

```
interface Listlterator extends Iterator{
     void add(Object);
     …
}
```

A diferencia del método add de Collection, éste no devuelve un valor boolean (se asume que esta operación add() siempre modifica la lista).

Además, la interfaz Listlterator dispone de dos métodos que se pueden emplear para recorrer las listas hacia atrás.

- Object previous()
- boolean hasPrevious()

Al igual que ocurre con el método next(), previous() devuelve el objeto que ha sido saltado (dejado atrás).

Si queremos implementar un objeto tipo ListIterator de la clase LinkedList que vimos antes:

```
ListIterator iter = equipo.ListIterator();
```

El método add añade un nuevo elemento antes de la posición que marca el iterador. Si al código anterior agregamos:

```
iter. next () ;
iter.add("Julieta");
```

Nos debería quedar una lista: Angelina, Julieta, Pablito, Carlitos. Lo verificaremos en la demo de LinkedListTest.

Si realiza sucesivas llamadas a add(), los elementos son insertados en el orden en el que los vaya suministrando, y siempre en la posición anterior a la que marca en ese momento el iterador.

Cuando realice operaciones add con un iterador devuelto por ListIterator y que apunte al comienzo de la lista enlazada, el nuevo elemento añadido pasa a ser la cabecera de la lista.

Cuando a vez que el iterador alcanza el último elemento (es decir, cuando hasNext() devuelve false), el elemento añadido pasa a ser la cola de la lista. Si la lista enlazada tiene n elementos, existen n+1 lugares posibles para añadir nuevos elementos. Dichos lugares se corresponden con las n+1 posibles posiciones del iterador:

- Antes del primero
- n-1 posiciones intermedias
- Después del último

Total: n+1 lugares posibles.

Nota: remove() siempre elimina el elemento **anterior** al que estamos referenciando; **anterior** depende del recorrido que estamos llevando a cabo:

- Recorriendo con next(), eliminamos el nodo a izquierda.
- Recorriendo con previus(), eliminamos el nodo a derecha.

A diferencia del método add(), que sólo depende de la posición del iterador, remove() considera el estado de dicho iterador.

Para concluir, existe un método set() que sustituye el último elemento devuelto por una llamada a next() o a previous() por uno nuevo. Por ejemplo, el siguiente fragmento de código reemplaza "Angelina" por "Jorgelina"

```
ListIterator iter = equipo.listIterator();
Object oldValue = iter.next();  //   devuelve   el   primer   elemento
iter.set("Jorgelina");                // y le da nuevo valor
```

Si estuviéramos trabajando en programación concurrente, (Unidad III) podríamos tener más problemas. Por ejemplo, si un iterador recorre una colección mientras otro la modifica, se producirán situaciones confusas. Suponga que un iterador apunta a un elemento que otro iterador acaba de eliminar. El primero es ahora inválido y no podrá utilizarse. Los iteradores de una lista enlazada que será usada en ese entorno tienen que

ser diseñados para detectar modificaciones de este tipo. Si uno de estos iteradores detecta que la colección ha sido modificada por otro iterador o por un método de la propia colección, debe lanzar una ConcurrentModificationException.

Por ejemplo, considere el siguiente código:

```
LinkedList list = . . .;
ListIterator iter1 = list.listIterator(); //   Iter1 referencia 1er nodo
ListIterator iter2 = list.listIterator();  //   iter2   referencia   1er
nodo
iter1.next();                              //   iter1   referencia   2do
nodo
iter1.remove();                            //   removemos 1er nodo
iter2.next();   // lanza ConcurrentModificationException
```

La llamada a iter2.next() lanza una ConcurrentModificationException, ya que iter2 detecta que la lista se ha modificado externamente. (No existe próximo de un nodo removido)

Para evitar excepciones por modificaciones concurrentes, siga esta sencilla regla: puede enlazar tantos iteradores como quiera a un contenedor, pero haciendo que sólo uno de ellos pueda modificar el contenedor.

La detección de modificaciones concurrentes se consigue de una forma muy sencilla. El contenedor sigue la pista del número de operaciones "mutantes" (como la adición y la eliminación de elementos). Cada iterador mantiene un contador con el número de operaciones mutantes de las que es responsable. Al comienzo de cada método iterador se comprueba si su contador de mutaciones es igual al de la colección. En caso negativo, lanza una ConcurrentModificationException.

Ésta es una excelente comprobación y una importante mejora sobre los iteradores no seguros del entorno STL de C++.

NOTA: Existe una criteriosa excepción en la detección de una modificación concurrente. La lista enlazada sólo sigue la pista de las **modificaciones estructurales** de la misma, como puede ser una inserción o una eliminación de enlaces. El método set(), que modifica valores contenidos en los nodos, no se considera como una modificación estructural. Por tanto, puede añadir a una lista tantos de estos métodos como quiera y que todos ellos llamen a set() para cambiar los contenidos de los nodos existentes. Esta característica es necesaria para varios algoritmos de la clase Collections que veremos más adelante en esta unidad.

Como ya vio en la sección anterior, existen otros muchos métodos interesantes declarados en la interfaz Collection que sirven para la operativa con listas. La mayor parte de ellos se encuentran implementados en la superclase AbstractColIection de la clase LinkedList. Por ejemplo, el método toString() invoca a toString() en todos los elementos y genera una única cadena con la forma [A, B, C]. Esto resulta útil para la depuración. Puede usar el método contains() para comprobar si un elemento está presente en una lista enlazada. Por ejemplo, la llamada equipo.contains("Harry") devuelve true si la lista contiene una cadena que sea igual a la String "Harry". Sin embargo, no hay ningún método que devuelva un iterador para esa posición. Si desea hacer alguna operación con el elemento más allá que la de saber si existe, tendrá que programar usted mismo un bucle.

La librería también ofrece algunos métodos que, desde un punto de vista teórico, son algo ambiguos. Las listas enlazadas no soportan acceso directo. Si desea procesar el elemento n, tendrá que empezar desde el comienzo de la lista y saltar los n-1 primeros. Este proceso no se puede acotar. Por esta razón, los programadores no suelen usar listas enlazadas en aquellos programas en los que se debe acceder a los elementos por un valor entero. Si UD. necesita acceso directo use la colección ArrayList.

```
import java.util.*;
//   Algunas operaciones con listas enlazadas.
public class LinkedListTest{
   public static void main(String[] args){
      List a = new LinkedList();
      a.add("a-Angela");
      a.add("a-Carl");
      a.add("a-Erica");
      System.out.println("Lista a contiene:");
      System.out.println(a);

      List b = new LinkedList();
      b.add("b-Bob");
      b.add("b-Doug");
      b.add("b-Frances");
      b.add("b-Gloria");

      System.out.println("Lista b contiene:");
      System.out.println(b);

      // intercale palabras de b en a

      ListIterator aIter = a.listIterator();
      Iterator bIter = b.iterator();

      while (bIter.hasNext())
      {
         if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }
      System.out.println("\n Luego de intercalacion");
      System.out.println("lista a contiene:" );
      System.out.println(a);

      // elimine las palabras pares de b

      bIter = b.iterator();
      while (bIter.hasNext()){
         bIter.next(); // salte un elemento
         if (bIter.hasNext())
         {
            bIter.next(); // salte el siguiente
            bIter.remove(); // y remuevalo
         }
      }
      System.out.println("\n Luego de eliminacion palabras pares");
      System.out.println("lista b contiene:" );
      System.out.println(b);    //  Veamos como está b
```

```
      a.removeAll(b);
      System.out.println("\n Luego de eliminar todas las palabras");
      System.out.println("de la lista b contenidas en lista a queda en a"
);
      System.out.println(a);
   }
}
```

```
run:
Lista a contiene:
[a-Angela, a-Carl, a-Erica]
Lista b contiene:
[b-Bob, b-Doug, b-Frances, b-Gloria]

 Luego de intercalación
lista a contiene:
[a-Angela, b-Bob, a-Carl, b-Doug, a-Erica, b-Frances, b-Gloria]

 Luego de eliminación palabras pares
lista b contiene:
[b-Bob, b-Frances]

 Luego de eliminar todas las palabras
de la lista b contenidas en lista a queda en a
[a-Angela, a-Carl, b-Doug, a-Erica, b-Gloria]
BUILD SUCCESSFUL (total time: 1 second)
```

A continuación presentamos a **public class SortedList** que contiene 5 demos generando listas, invirtiéndolas, verificando existencia de elementos, y finalmente generando una lista ordenada; (mucho mas sencillo que como lo hacíamos en AED, tan de artesanos, aunque este autor extraña que no exista una forma de insertar ordenado, un addSorted() y ya…)

```
package ArrayItems;
import java.util.*;
//   Generando listas ordenadas y otros. Author Tymos

public class SortedList extends LinkedList{
    ArrayItems arrayItems;
    SortedList list01;
    SortedList list02;

    public SortedList(){            //  Primer constructor
        super();                    //  Genera una lista vacía
    }

    public SortedList(ArrayItems array){ //  Segundo constructor
        super(array);  //  Generamos coleccion SortedList
    }                  //  usando colección ArrayItems


      public void demo01(){// Genera lista con = secuencia de arrayItems
         System.out.println("Demo01 - genero list01 a partir de
                                                         arrayItems");
         arrayItems = new ArrayItems(8,'R');//arrayItems con 8 items
         System.out.println("objeto arrayItems contiene");
```

```
        System.out.println(arrayItems);
        list01 = new SortedList(arrayItems);
        System.out.println("lista list01 contiene");
        System.out.println(list01);
    }


     public void demo02(){   //  Genera list02 invirtiendo list01
        System.out.println("Demo02 -  genera list02 como list01
                                                     invertida");
        list02 = new SortedList(); //   lista vacía
        Iterator list01It = list01.iterator();// lista creada en demo01
        while (list01It.hasNext()){   //  mientras tengamos en list01
            list02.addFirst(list01It.next());
        }
        System.out.println("lista list02  contiene");
        System.out.println(list02);
    }


     public void demo03(){//Verifica existen elementos en ambas listas
        System.out.println("Demo03 - Verifica si todas de list01 existen
                                                          en list02");
        Iterator list01It = list01.iterator();//lista creada en demo01
        boolean todos = true;
        while (list01It.hasNext())   //  mientras tengamos en list01
            if (!list02.contains(list01It.next()))
                todos = false;
        if(todos)
            System.out.println("list02  contiene todos de list01");
        else
            System.out.println("list02  no contiene todos de list01");
    }


     public void demo04(){//Remuevo ultimo de list02, luego idem demo03
        System.out.println("Demo04 - remuevo ultimo de list02, luego idem
                                                          demo03");
        Iterator list01It = list01.iterator();//lista creada en demo01
        boolean todos = true;
        list02.removeLast();
        while (list01It.hasNext())   //  mientras tengamos en list01
            if (!list02.contains(list01It.next()))
                todos = false;
        if(todos)
            System.out.println("list02  contiene todos de list01");
        else
            System.out.println("list02  no contiene todos de list01");
    }

     public void demo05(){   //  Genera list02 ordenada desde list01
        System.out.println("Demo05 - genero list02 ordenada desde
                                                          list01");
        list02 = new SortedList(); //   list02 vacía
        Item item01;    // para items de list01
        Item item02;    // para items de list02
        boolean inserto;//Un flag para verificar inserciones intermedias
```

```
        Iterator list01It = list01.iterator();// Un iterador para list01
        while (list01It.hasNext()){   //  recorriendo list01
            inserto = false;    //  A priori, suponemos que no las hay
            item01 = (Item)(list01It.next());   // item de list01

            ListIterator list02It = list02.listIterator();//y p/ list02
            while (list02It.hasNext()){   //  recorriendo list02
                item02 = (Item)(list02It.next()); // item de list02
                if(item02.esMayor(item01)){
                    System.out.println("Insercion intermedia " +
                                            item01.getCodigo());
                    list02It.previous();  //  Retrocedo al anterior
                    list02It.add(item01);   //  inserto un intermedio
                    System.out.println(list02);
                    inserto = true;
                    break;
                }
            }   //  while (list02It.hasNext())
            if(!inserto){ // insercion en los extremos de la lista
                System.out.println("Insercion en extremos "+
                                            item01.getCodigo());
                list02It.add(item01);   //  luego inserto
                System.out.println(list02);
            }
        }   //  while (list01It.hasNext())
        System.out.println("lista list02  contiene");
        System.out.println(list02);
    }

     public static void main(String[] args){
       SortedList list = new SortedList();
       list.demo01();
       list.demo02();
       list.demo03();
       list.demo04();
       list.demo05();
   }
}
```

Demo01 – genero list01 a partir de arrayItems

```
objeto arrayItems contiene
[4 - 25.340279, 7 - 11.706503, 8 - 62.82343, 13 - 53.154453, 6 -
20.911102, 9 - 60.64827]
lista list01 contiene
[4 - 25.340279, 7 - 11.706503, 8 - 62.82343, 13 - 53.154453, 6 -
20.911102, 9 - 60.64827]
```

Demo02 – genera list02 como list01 invertida

```
lista list02  contiene
[9 - 60.64827, 6 - 20.911102, 13 - 53.154453, 8 - 62.82343, 7 -
11.706503, 4 - 25.340279]
```

```
Demo03 – Verifica si todas de list01 existen en list02
list02  contiene todos de list01
```

```
Demo04 – remuevo ultimo de list02, luego idem demo03
list02  no contiene todos de list01
```

```
Demo05 – genero list02 ordenada desde list01
Insercion en extremos 4
[4 - 25.340279]
Insercion en extremos 7
[4 - 25.340279, 7 - 11.706503]
Insercion en extremos 8
[4 - 25.340279, 7 - 11.706503, 8 - 62.82343]
Insercion en extremos 13
[4 - 25.340279, 7 - 11.706503, 8 - 62.82343, 13 - 53.154453]
Insercion intermedia 6
[4 - 25.340279, 6 - 20.911102, 7 - 11.706503, 8 - 62.82343, 13 - 53.154453]
Insercion intermedia 9
[4 - 25.340279, 6 - 20.911102, 7 - 11.706503, 8 - 62.82343, 9 - 60.64827, 13 -
53.154453]
lista list02  contiene
[4 - 25.340279, 6 - 20.911102, 7 - 11.706503, 8 - 62.82343, 9 - 60.64827, 13 -
53.154453]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Array de listas

Un **ArrayList** encapsula un array Object[] reubicable dinámicamente.

NOTA: Si UD. es un experto en programación Java, seguro que habrá usado la clase Vector siempre que haya necesitado un array dinámico. (Si no lo ha usado, no se preocupe, ahora le enseñaremos a gestionar arrays dinámicos.) ¿Y por qué hablamos de usar un **ArrayList** en lugar de un **Vector**? Por una simple razón: todos los métodos de la clase Vector están sincronizados(Esto es tema de la Unidad III). Resulta seguro acceder a un objeto Vector desde varios threads. Pero si accede a uno de estos objetos desde un único thread, el programa desperdiciará tiempo por causa de la sincronización. En contraste, los métodos de ArrayList **no están sincronizados**. Recomendación: usar un **ArrayList** en lugar de un **Vector** siempre que no necesite sincronización.

En muchos lenguajes de programación, (todavía muy usados) es necesario fijar el tamaño de todos los arrays antes de la compilación. Algunos ofrecen algo más: se puede fijar el tamaño en tiempo de ejecución. Pero una vez fijado, el tamaño es inamovible. Y que hago cuando este tamaño es muy variable de una sesión a otra?

La forma más sencilla de tratar esto en Java es usar una clase de Java que se comporta exactamente igual que un array pero que permite la expansión y reducción del mismo en tiempo de ejecución. Justamente, la clase ArrayList. En ella un arreglo puede crecer o disminuir de tamaño sin escribir ninguna línea de código para que esto ocurra.

Existen mas diferencias entre un array “clásico” y un ArrayList. El primero es una característica del propio lenguaje lava, y hay un array de tipo T[] para cada tipo de elemento T. Por otro parte, ArrayList es una clase de biblioteca, definida en el paquete java.util. Es un tipo "en el tamaño encaja todo" que alberga elementos de tipo Object. (Al ser Object sus elementos, deberá realizar un moldeado siempre que quiera extraer un elemento de él. Esto no es ningún problema).

Se usa su método add para añadir nuevos elementos. Por ejemplo, ésta es la forma de crear un ArrayList y de rellenado con objetos empleado:

```
ArrayList equipo = new ArrayList();
equipo.add(new Empleado( . . ));
equipo.add(new Empleado( . . ));
```

La clase ArrayList gestiona un array interno de referencias Object. Este array no tiene límites. Ésta es la forma “mágica” en la que trabaja ArrayList: si llama a add() y el array interno está lleno, el ArrayList crea otro array más grande y copia automáticamente los elementos del primero en el segundo.

Si ya sabe, o tiene una idea aproximada, del número de elementos a almacenar, puede invocar el método ensureCapacity antes de rellenar el array de listas:

```
equipo.ensureCapacity(100);
```

Esto permite la ubicación de un array interno de 100 objetos. A continuación, puede seguir llamando a add(), y la reubicación se llevará a cabo sin ningún coste.

También se puede pasar un tamaño inicial al constructor de ArrayList:

El método equipo.size() devuelve el número real de objetos en ArrayList.

Esto es equivalente a a.length() para un array definido, por ejemplo, int a[100]

Una vez que esté razonablemente seguro del tamaño de su ArrayList, puede llamar al método trimToSize. Dicho método ajusta el tamaño del bloque de memoria para que use exactamente el espacio necesario para contener el número actual de elementos. El recolector de basura reclamará el exceso de memoria.

Acceso a los elementos de un ArrayList.

En lugar de usar la cómoda sintaxis [] para acceder, o cambiar, los elementos de un array, en ArrayList debe utilizar los métodos get() y set()

Por ejemplo, para asignar valor al elemento i, debe usar:

```
equipo.set (i, “Antonio”); // equivalente a
```

```
a[i] = “Antonio”; para un array a “clásico”.
```

Obtener un elemento requiere moldear dicho valor al tipo que necesite:

```
Empleado e = (Employee)equipo.get(i);
```

ArrayList, al igual que los arrays, empiezan a contabilizar sus elementos a partir de cero.

Al ser los elementos de tipo Object, ArrayList puede contener objetos de diversas clases sin ningún problema.

ArrayList es inherentemente inseguro. Es posible añadir por accidente un elemento de un tipo erróneo a los mismos:

```
Date birthday = . . .;
equipo.set(i, birthday);
```

El compilador no se quejará. Es perfectamente posible intanciar un objeto Date a Object, pero cuando este elemento sea recuperado posteriormente, si no comprobamos tipos es muy posible que sea moldeado a Empleado. Esto naturalmente es totalmente incorrecto y seguramente disparará una excepción. Dicha situación es posible porque un ArrayList almacena valores de tipo Object.

En muy raras ocasiones, los ArrList son usados para mantener colecciones heterogéneas, es decir, objetos de distintas clases.
Cuando se recupera una entrada, se debe comprobar el tipo de cada objeto, tal y como se hace en este fragmento de código:

```
ArrayList list;
list.add (new Empleado(. . .));
list.add(new Date(. . .));

Object obj = list.get(n);
if (obj instanceof Empleado){
     Empleado e = (Empleado)obj;
```

Esta forma de trabajar tiene puntos a favor y en contra.

- **A favor**: Es interesante poder procesar de una colección heterogénea; inclusive, a la hora de recuperar y procesar sus elementos, si ellos pertenecen a una estructura jerárquica podemos dejar que el polimorfismo se encargue de invocar el comportamiento adecuado.
- **En contra**: si los objetos pertenecen a clases sin ninguna relación, puede ser laborioso tener que identificar (como en el ejemplo anterior) cada posible clase para poder procesarla.

Inserción y eliminación de elementos intermedios en un ArrList

En un ArrList los elementos se pueden insertar en cualquier punto de la estructura:

```
equipo.add(n,e);
```

El elemento que se encuentra en la posición n, y los que están por encima de él, se desplazan hacia arriba para hacer hueco a la nueva entrada. Si, una vez efectuada la inserción, el nuevo tamaño del array de listas excede la capacidad, la estructura será reubicada de nuevo en memoria para almacenar su array de almacenamiento.

De forma análoga, es posible eliminar un elemento situado en el interior del array de listas:

```
Empleado e = (Empleado)equipo.remove(n);
```

Los elementos situados por encima se desplazan hacia abajo, y el tamaño del array se reduce una unidad.

La inserción y eliminación de elementos no es demasiado eficiente. Sobre todo si el ArrayList es grande. El criterio podría ser:

- La gran mayoría de las operaciones son accesos directos. Debería andar bien la colección implementada sobre un ArrayList.
- El grueso de las operaciones son inserción y eliminación. Considere el uso de una colección LinkedList.

El comportamiento de esta clase es bastante amplio, un resumen:

Constructores:

ArrayList() Construye un objeto vacío
ArrayList(Colección c) Construye el objeto a partir de la colección c
ArrayList(int capacidadInicial) Construye un objeto con esa capacidad

Métodos

add(int ind, Object elem) **i**nserta elem en la posición ind
add(Object elem) agrega elem al final de la colección
addAll(Colección c) agrega la colección c al final
addAll(int ind, Colección c) inserta la colección a partir de ind
clear() excluye todos sus elementos
clone() retorna una copia del objeto ArrayList
contains(Object elem) retorna verdadero si elem existe
ensureCapacity(int minCap) mod. capacidad del ArrayList si es menor
get(int ind) retorna el elemento de orden ind
indexOff(Objet elem) retorna el índice del 1er objeto igual a elem
isEmpty() retorna verdadero si ArrayList está vacia
…
Hay 8 métodos más, y otros 25 heredados de varias clases e interfaces.

Como podría ser **una implementación de la clase ArrItems** (Array de ítems) usando ArrayList, que vimos en Algoritmos implementada en la forma "clásica" de array?

La clase Item de Algoritmos

```
class Item {    // Una clase de claves asociadas a un valor ...
      protected int codigo;
      protected float valor;
      public Item(){};          //    Constructor sin argumentos

      public    Item(int cod, float val){ //  Constructor de la clase
              codigo=cod;
              valor=val;
      }
      public    String    toString(){ //   Exhibimos
            String aux = "";
            aux+=codigo+" - "+valor;
            return aux;
      }          //    Exhibimos
      public  void leerCodigo(){
            System.out.print("Código? ");
```

```
            codigo = In.readInt();
      }
      public     int   getCodigo(){return codigo;}
      public     float getValor(){return valor;}
      public     Item  getItem(){return this;}
      public     void  setCodigo(int cod){codigo=cod;}
      public     void  setValor(float val){valor=val;}
      public     boolean esMayor(Item item){
            return(codigo > item.codigo?true:false);}
      public     boolean esMenor(Item item){
            return(codigo < item.codigo?true:false);}
      public     void  intercambio(Item item){
            Item burb= new Item(item.codigo,item.valor
            item.codigo = this.codigo;
            item.valor  = this.valor;
            this.codigo = burb.codigo;
            this.valor  = burb.valor;
      }
}
```

La clase ArrItems

```
class ArrItems{             //   Una clase de implementación de un
      protected int talle; //  Tamaño del array de objetos Item
      protected Item[] item;   // array de items, comportamiento mínimo
      public ArrItems(int tam, char tipo) {// Constructor
            int  auxCod = 0;  // Una variable auxiliar
            talle=tam;      // inicializamos talle (capacidad)
            item = new Item[talle];    // Generamos el array
            for(int i=0;i<talle;i++){  // la llenaremos de Item's,
                  switch(tipo){// segú tipo de llenado requerido
                        case 'A':  // los haremos en secuencia ascendente
                              auxCod = i;
                              break;
                        }
                        case 'D':{    // o descendente ...
                              auxCod = talle - i;
                              break;
                        }
                        case 'R':{    // o bien randomicamentente (Al azar)
                              auxCod = (int)(talle*Math.random());
                        }
                  }
                  item[i] = new Item();
                  item[i].setCodigo(auxCod);
                  item[i].setValor((float)(talle*Math.random()));
            }
            System.out.print(this);
      }
      public String toString(){
            int ctos = (talle < 10 ? talle : 10);
            String aux = " Primeros "+ctos+" de "+talle+"\n elementos
                          Item\n";
            for(int i=0;i<ctos;i++)
                  aux+=item[i].toString()+"\n";
            return aux;
      }
}
```

Una posible implementación de **ArrItems** extendiendo **ArrayList**:

```
package arritems01;
import java.util.ArrayList;
public class ArrayItems extends ArrayList{
    public ArrayItems(int cap, char tipo){
      super(5); //  Construimos objeto ArrayList con capacidad de 5
      int auxCod = 0;
     for(int i=0; i<cap-2; i++){ // incluimos solo 3 items
        switch(tipo){     // dependiendo del tipo de llenado
           case 'A':{    // los haremos en secuencia ascendente
                   auxCod = i*3;
                   break;
           }
           case 'D':{ // o descendente ...
                   auxCod = cap - i;
                   break;
           }
           case 'R':{ // o bien randomicamentente (Al azar)
                   auxCod = (int)(cap*2*Math.random());
           }
        }   //  swith
            add(new Item(auxCod, (float)(cap*10*Math.random())));
     }   //  for
    }   //  constructor ArrayItems

    public void demo(){
        System.out.println("Objeto array recién construido");
        System.out.println(this);

        System.out.println("Insertamos item en posición 1");
        add(1, new Item(1, (float)10.3));
        System.out.println(this);

        System.out.println("Agregamos 2 items al final");
        System.out.println("(Incremento capacidad automático)");
        add(new Item(10, (float)100));
        add(new Item(10, (float)100));
        System.out.println(this);

        System.out.println("Removemos el 3er elemento");
        remove(2);
        System.out.println(this);

        System.out.println("Insertamos String en posición 2");
        add(2, new String("Heterogeneo"));
        System.out.println(this);
        System.out.println("Demo terminado!!!");
    }

    public static void main(String[] args) {
        ArrayItems array = new ArrayItems(5, 'A');
        array.demo();
    }
}
```

```
run:
Objeto array recién construido
[0 - 36.433395, 3 - 1.6486598, 6 - 38.06825]
Insertamos item en posición 1
[0 - 36.433395, 1 - 10.3, 3 - 1.6486598, 6 - 38.06825]
Agregamos 2 items al final
(Incremento capacidad automático)
[0 - 36.433395, 1 - 10.3, 3 - 1.6486598, 6 - 38.06825, 10 - 100.0, 10 - 100.0]
Removemos el 3er elemento
[0 - 36.433395, 1 - 10.3, 6 - 38.06825, 10 - 100.0, 10 - 100.0]
Insertamos String en posición 2
[0 - 36.433395, 1 - 10.3, Heterogeneo, 6 - 38.06825, 10 - 100.0, 10 - 100.0]
Demo terminado!!!
BUILD SUCCESSFUL (total time: 1 second)
```

Conjuntos de hash

Las listas enlazadas y los arrays le permiten indicar en qué orden desea recuperar los elementos. Sin embargo, si está buscando un elemento en particular y no recuerda su posición, deberá visitar todos hasta encontrar el que quiere. Esto puede llevar mucho tiempo si la colección contiene muchos elementos. Si no le preocupa el orden de los elementos, existen otras estructuras de datos que le permitirán encontrar elementos mucho más deprisa. El inconveniente es que estas estructuras no le permiten tener control sobre el orden en el cual aparecen los elementos. Las estructuras de datos organizan los elementos en el orden que más les conviene para sus propósitos.

Una buena estructura de datos para localizar objetos de una forma rápida son los arrays asociativos, o tablas hash. Un hash procesa un entero, llamado código hash, para cada objeto. Veremos su procesamiento un poco mas adelante. Lo que importa por ahora es que los códigos hash puede ser procesados rápidamente, y que dicho procesamiento sólo depende del estado del objeto a recuperar, y no del resto de objetos almacenados en el hash.

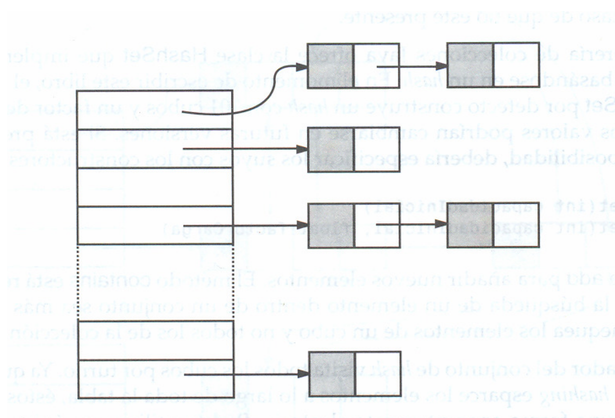
Un hash es un array de listas enlazadas. Cada lista recibe el nombre de cubo, o bucket. Para localizar la ubicación de un objeto en la tabla, se procesa su código hash.

Ejemplo de un algoritmo de randomización:

Indice(del cubo) = resto de la división del código hash por la cantidad de cubos.

Si código hash = 345 y cantidad de cubos = 101, tenemos:

Indice cubo = 345 % 101 = 42



Si desea más control sobre el comportamiento de un hash, puede suministrar el número inicial de cubos. Dicho valor indica cuántos cubos contendrá el hash.

Cada “cubo” es en realidad una lista que contiene los elementos cuyo código hash randomizado determinó su índice en el array. Si el algoritmo es adecuado, los cubos contendrán cantidades de elementos parecidas.

Si conoce aproximadamente cuántos elementos puede llegar a tener la tabla, debería establecer el valor inicial de cubos en aproximadamente el necesario para almacenar el 150% del número de elementos esperados. Es usual dimensionar el hash con un valor inicial para prevenir el apiñamiento de claves. Por ejemplo, si necesita almacenar alrededor de 100 entradas, establezca un tamaño inicial en 150. Esto lo lleva a prever una capacidad inicial 50% mayor (mas cubos)

Si no sabe cuantos elementos tendrá que almacenar, puede que su tabla hash se llene. Esto ocurre cuando la cantidad de elementos almacenados supera la prevista en un coeficiente llamado **factor de carga**. Este coeficiente tiene un valor por defecto = 0.75, que puede modificarse en la construcción. Si esto ocurre, Java **redimensiona automáticamente** el hash pasando a tener el doble de cubos. Para la mayoría de aplicaciones, es razonable dejar el factor de carga en 0,75.

Las tablas hash pueden usarse para implementar varias estructuras de datos importantes. La más sencilla de ellas es el tipo conjunto, o set. Un conjunto es una colección de elementos sin duplicaciones. El método add de un conjunto intenta primero localizar el objeto a añadir dentro de la estructura y sólo lo añade en el caso de que no esté presente.

La librería de colecciones Java ofrece la clase **HashSet** que implementa un conjunto basándose en un hash. Tiene varios constructores.

```
HashSet();
HashSet(int capacidadInicial);
HashSet(int capacidadInicial, float factorCarga);
HashSet(Collection); //   Constructor copia
```

El método contains() sólo se chequea los elementos de un cubo correspondiente.

El iterador del conjunto de hash visita todos los cubos por turno. Ya que la operación de hashing esparce los elementos a lo largo de toda la tabla, éstos son visitados de una forma aparentemente aleatoria.

El programa lee todas las palabras desde la corriente de entrada y las añade al hash. Informa cantidad total de palabras leídas y añadidas al hash. Usa un iterador para mostrar el conjunto.

```
import java.util.*;
import java.io.*;
//   Este programa utiliza el comportamiento de un conjunto (HashSet)
// para mostrar las palabras introducidas desde System.in, ignorando
// repeticiones
public class SetTest{
   public static void main(String[] args){
      Set palabras = new HashSet();
```

```
      // una referencia de Set puede ser instanciada con objetos
      // HashSet, LinkedHashSet or TreeSet
      int leidas    = 0;
      int lineas    = 0;
      try{
         BufferedReader in = new
                BufferedReader(new InputStreamReader(System.in));
         String line;
         System.out.println("Introduzca texto desde System.in");
         while (lineas < 2){
             line = in.readLine();
             lineas++;
             System.out.println(line);
             StringTokenizer tokenizer = new StringTokenizer(line);
             while (tokenizer.hasMoreTokens()){
                String palabra = tokenizer.nextToken();
                palabras.add(palabra);
                leidas++;
            }   //  while
         }  //  while
      } //  try
      catch (IOException e){
         System.out.println("Error " + e);
      }

      Iterator iter = palabras.iterator();
      System.out.println("Iteramos ...");
      int contPalLin = 0;
      while (iter.hasNext()){
        System.out.print(iter.next()+", ");
        if(++contPalLin>5){System.out.println();contPalLin = 0;}
      }
        System.out.println("\nTotal palabras leidas   "+leidas);
        System.out.println("Total palabras distint. "+palabras.size());
   }
}
```

```
run:
Introduzca texto desde System.in
En 1551 Vieta introdujo las letras como notacion en lugar de los numeros.
La idea de los numeros como magnitudes discretas paso a segundo plano
Iteramos ...
La, Vieta, notacion, como, plano, magnitudes,
1551, en, discretas, los, introdujo, lugar,
de, numeros, a, paso, idea, En,
las, numeros., segundo, letras,
Total palabras leidas   25
Total palabras distint. 22
BUILD SUCCESSFUL (total time: 2 minutes 58 seconds)
```

Si lo desea, puede insertar cadenas dentro de hash porque la clase String dispone de un método hashCode que procesa el código hash de la cadena. Un código hash es un entero que, de algún modo, es derivado a partir de los caracteres de la cadena.

Árboles

La clase TreeSet es similar al hash pero con una mejora añadida. Un árbol, si así definido, es una colección ordenada; es decir, los elementos tienen en la colección un orden determinado. Cuando se recorra la colección, los valores son presentados en el orden correcto.

Tal y como sugiere el nombre de la clase, la ordenación es ejecutada por el propio árbol. La implementación actual usa un árbol rojo-negro (red-black tree). Cada vez que se añade un elemento a un árbol, éste se coloca en su posición correcta. Por consiguiente, el iterador siempre visita los elementos obteniendo el orden adecuado. A continuación describimos sucintamente este árbol. (Una descripción muy detallada puede encontrarla en Estructuras de Datos y Algoritmos en Java, Goodrich/Tamassia, editorial CECSA, cap. 9 - Arboles de búsqueda; asimismo allí puede encontrar descripciones en detalle de los árboles AVL y 2-4, citados mas adelante en el mismo libro).

Árboles rojo-negro

Aunque los árboles AVL y los árboles (2,4) tienen varias propiedades buenas, hay algunas aplicaciones de diccionario (Elementos con claves duplicadas) para las que no se adaptan bien. Por ejemplo, puede ser que los árboles AVL requieran muchas operaciones de reestructuración (rotaciones) después de la remoción de un elemento, y los árboles (2,4) pueden requerir muchas operaciones de fusión o de partición después de cada inserción o remoción. La estructura de datos que se describe en esta sección es el árbol rojo-negro, y no tiene estos inconvenientes porque sólo requiere hacer O(1) cambios estructurales después de una actualización para permanecer balanceado. *(Todo esto decimos para introducir la idea de que la clase Java TreeSet nos facilita una eficiente implementación de árbol de búsqueda binario balanceado en altura)*
Un árbol rojo-negro es un árbol de búsqueda binaria con los nodos "iluminados" de rojo y de negro en una forma que satisface los siguientes

requisitos:

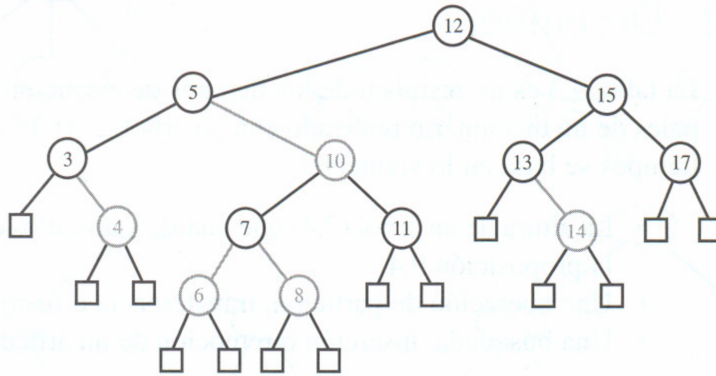
Propiedad de la raíz: La raíz es "negra".

Propiedad externa: Todo nodo externo es "negro". (Cuadrados en la figura)

Propiedad interna: Los hijos de un nodo "rojo" son "negros".

Propiedad de profundidad: Todos los nodos externos tienen la misma profundidad negra, que se define como la cantidad de antepasados negros menos uno.

En la figura a continuación se muestra un ejemplo de árbol rojo-negro.
Son "negros" los nodos 12, 5, 15, 3, 13, 17, 7 y 11
Son "rojos" los nodos 10, 4, 14, 6 y 8



Cada nodo externo de este árbol rojo-negro tiene 3 antepasados negros, y en consecuencia, su profundidad negra es 3.

Una convención podría ser que los datos están guardados en los nodos internos de un árbol rojo-negro, y que los nodos externos se encuentran vacíos y son de relleno. También se pueden codificar algoritmos construyendo este árbol sin nodos externos o reemplazándolos por una referencia a un único objeto NODO_NULO en los nodos internos.

Añadir un elemento a un árbol es más lento que hacerlo a un hash, pero es mucho más rápido que hacerlo en una lista enlazada. Si el árbol contiene n elementos, se necesita la media de 10g2 n comparaciones para localizar la posición del nuevo elemento. Por ejemplo, si el árbol contiene alrededor de 1.000 elementos, se necesitarían en media 10 comparaciones para insertar un nuevo elemento.

De esta forma, la adición de elementos a un TreeSet es algo más lenta que la adición a un HashSet (véase la tabla comparativa siguiente) pero, a cambio, el TreeSet ordena automáticamente los elementos, en la secuencia que UD desee.

Documento	*Número total de palabras*	*Número de palabras distintas*	*HashSet*	*TreeSet*
Alicia en el País de las Maravillas	28.195	5.909	5 segundos	7 segundos
El Conde de Monte Cristo	466.300	37.545	75 segundos	98 segundos

Constructores de la clase java.util.TreeSet

· TreeSet() Construye un árbol vacío.
· TreeSet(Collection elementos) Construye un árbol y añade todos los elementos de la colección.

Comparación de objetos

¿Cómo sabe un TreeSet la forma en la que desea que se ordenen los elementos? Por defecto, asume que usted inserta elementos que **implementan la interfaz Comparable.** Esta interfaz define un solo método:

```
int compareTo(Object otro)
```

La llamada a a.compareTo(b) debe devuelve:

- 0 si a y b son iguales,
- un valor negativo si a < b
- un valor positivo si a > b

Varias clases estándar de la plataforma Java implementan la interfaz Comparable. Un ejemplo es la clase String. Su método compareTo compara cadenas en orden alfabético (o lexicográfico).

Si inserta sus propios objetos, debe definir su propio orden implementando la interfaz Comparable. No existe una implementación por defecto para compareTo en la clase Object.

Para trabajar en la clase TreeSet con nuestros propios objetos, vamos a usar clases de objetos muy usados en varias unidades de la asignatura AED: Item, a la que incorporamos el método compareTo, y ArrItems, con pequeñas adecuaciones para que utilice ese método. Y aprovecharemos su comportamiento en todo lo posible.

Las clases, con las adecuaciones citadas:

```
class Item implements Comparable{// claves asociadas a un valor
      protected int codigo;
      protected float valor;

      public Item(){};          //    Constructor sin argumentos
      public Item(int cod, float val){ //  Constructor de la clase
              codigo=cod; valor=val;
      }
      public String toString(){ //     Exhibimos
            String aux = "";
            aux+=codigo+" - "+valor;
            return aux;
      }          //    Exhibimos
      public  void leerCodigo(){
            System.out.print("Código? ");
            codigo = In.readInt();
      }
      public     int   getCodigo(){return codigo;}
      public     float getValor(){return valor;}
      public     Item  getItem(){return this;}
      public     void  setCodigo(int cod){codigo=cod;}
      public     void  setValor(float val){valor=val;}

     public int compareTo(Object other){
            Item otro = (Item)other;
            return codigo - otro.codigo;
     }

      public void  intercambio(Item item){ // usando un Item burbuja
            // intanciamos burb con datos parametro
            Item burb= new Item(item.codigo,item.valor);

            // asignamos atributos del invocante al parametro
            item.codigo = this.codigo;
            item.valor  = this.valor;
```

```
            //asignamos atributos del objeto burb al invocante
            this.codigo = burb.codigo;
            this.valor  = burb.valor;
      }
}
class ArrItems{  //  Una clase de implementación de un array de items
      protected int talle; //  Tamaño del array de objetos Item
      protected Item[] item; // array de objetos item
      // Constructor de un array de Item's
      public ArrItems(int tam, char tipo) {
            int  auxCod = 0;// Una variable auxiliar
            talle=tam;      // inicializamos talle (tamaño)
            item = new Item[talle]; // Generamos el array
            for(int i=0;i<talle;i++){ // la llenaremos de Item's,
                  switch(tipo){ // dependiendo del llenado requerido
                        case 'A':{ // secuencia ascendente
                              auxCod = i; break;
                        }
                        case 'D':{ // o descendente ...
                              auxCod = talle - i; break;
                        }
                        case 'R':{ // o bien randomicamentente
                              auxCod = (int)(talle*Math.random());
                        }
                  }
                  item[i] = new Item(); item[i].setCodigo(auxCod);
                  item[i].setValor((float)(talle*Math.random()));
            }
            System.out.print(this);
      }
      public String toString(){
            int ctos = (talle < 10 ? talle : 10);
            String aux = " Primeros "+ctos+" de ";
                   aux+= talle+"\n elementos Item\n";
            for(int i=0;i<ctos;i++)
                  aux+=item[i].toString()+"\n";
            return aux;
      }
}
import java.util.*;
// Este programa genera un árbol rojo_negro ordenado por el código de item
public class TreeSetTest{
    public static void main(String[] args){
       TreeSetTest arbolTest = new TreeSetTest();
       arbolTest.demo();
    }
    public void demo(){
        TreeSet arbol = new TreeSet();
        ArrItems aItem = new ArrItems(20,'R');
        for(int i=0;i<aItem.talle;i++)
            arbol.add(aItem.item[i]);
        System.out.println("El arbol de items, ordenado por codigo");
        System.out.println(arbol);
    }
}
```

```
run:
 Primeros 10 de 20
 elementos Item
19 - 8.442203
12 - 6.7378287
3 - 6.5650682
0 - 0.4888661
9 - 6.533705
4 - 11.78439
16 - 0.3754599
13 - 10.487459
7 - 3.4245462
6 - 16.192942
El árbol de ítems, ordenado por código
[0 - 0.4888661, 1 - 17.911428, 3 - 6.5650682, 4 - 11.78439,
5 - 2.7127392, 6 - 16.192942, 7 - 3.4245462, 8 - 2.3552706,
9 - 6.533705, 12 - 6.7378287, 13 - 10.487459, 14 - 9.641634,
16 - 0.3754599, 19 - 8.442203]
BUILD SUCCESSFUL (total time: 1 second)
```

Sin embargo, usar la interfaz Comparable para definir el orden tiene limitaciones. Sólo puede implementar la interfaz una vez. Eso significa que los objetos que queremos organizar en el arbol ya vienen con su ordenamiento definido (por código en el ejemplo que vimos).

Pero, ¿qué podemos hacer si necesitamos ordenar un puñado de elementos numéricamente en una colección y por otro atributo en otra? Además, ¿qué se puede hacer si necesitamos ordenar los objetos de una clase cuyo creador no se molestó en implementar la interfaz Comparable?

Tenemos una solución flexible. Haremos que el árbol use métodos de comparación diferentes, según los objetos Comparator que pasemos al constructor TreeSet.

Arbol TreeSet ordenado según el objeto Comparator pasado al constructor

La interfaz Comparator tiene un solo método, con dos parámetros explícitos:

int compare(Object a, Object b)

Al igual que el método compareTo, compare devuelve un entero negativo si a está antes que b, cero si ambos son idénticos o un valor positivo en cualquier otro caso.

Para ordenar los elementos por un atributo determinado, definimos una clase que implementa la interfaz Comparator para ese atributo. Como nos interesa ordenar por dos atributos, necesitamos 2 clases:

```
import java.util.*;
//   Este programa genera un arbol rojo_negro ordenado por el código del
// objeto ítem y otro ordenado por valor.
public class TreeSetTest02{

    public static void main(String[] args){
       TreeSetTest02 arbolTest = new TreeSetTest02();
       arbolTest.demo();
    }
```

```
    public void demo(){
        ArrItems aItem = new ArrItems(20,'R');

        // Arbol ordenado por código de items
        ComparaCodigo compCod = new ComparaCodigo();
        TreeSet arbol = new TreeSet(compCod);
        for(int i=0;i<aItem.talle;i++)
            arbol.add(aItem.item[i]);
        System.out.println("El arbol de items, ordenado por codigo");
        System.out.println(arbol);

        // Arbol ordenado por valor de items
        ComparaValor compVal = new ComparaValor();
                arbol = new TreeSet(compVal);
        for(int i=0;i<aItem.talle;i++)
            arbol.add(aItem.item[i]);
        System.out.println("El arbol de items, ordenado por Valor");
        System.out.println(arbol);
    }
}

class ComparaCodigo implements Comparator{
    public int compare(Object a, Object b){
        Item itemA = (Item)a;
        Item itemB = (Item)b;
        int codA   = itemA.getCodigo();
        int codB   = itemB.getCodigo();
        return codA - codB;
    }
}

class ComparaValor implements Comparator{
    public int compare(Object a, Object b){
        Item itemA = (Item)a;
        Item itemB = (Item)b;
        int valA   = (int)itemA.getValor()*100;
        int valB   = (int)itemB.getValor()*100;
        return valA - valB;
    }
}
```

```
run:
 Primeros 10 de 20
 elementos Item
2 - 18.15176      15 - 11.350283    13 - 3.1841564    8 - 16.001719
18 - 3.0430632    15 - 5.5474677    12 - 17.207684    12 - 8.129453
7 - 4.601063      6 - 14.339405

El arbol de items, ordenado por codigo
[2 - 18.15176, 5 - 2.405572, 6 - 14.339405, 7 - 4.601063, 8 - 16.001719,
9 - 3.380474, 10 - 4.297023, 12 - 17.207684, 13 - 3.1841564, …(sigue)

El arbol de items, ordenado por Valor
[5 - 2.405572, 13 - 3.1841564, 7 - 4.601063, 15 - 5.5474677, 17 - 6.918947,
6 - 7.834509, 12 - 8.129453, 15 - 11.350283, 16 - 12.039538, …(sigue)
BUILD SUCCESSFUL (total time: 1 second)
```

Utilizando comparadores, podrá ordenar los elementos como desee.

Conjuntos de bits

La clase BitSet almacena secuencias de bits en un vector de bits. Es recomendable utilizar esta estructura siempre que necesite almacenar secuencias de bits (por ejemplo, indicadores) de un modo eficiente. Los bits se empaquetan en bytes, resulta mucho más eficaz emplear un conjunto de bits que un ArrayList de objetos Boolean.

La clase BitSet le proporciona una interfaz adecuada para la lectura, el almacenamiento y la reinicialización de bits individuales. El uso de esta interfaz evita el enmascaramiento, además de otras operaciones no significativas sobre bits, que serían necesarias si almacenara dichos bits en variables int o long.

Por ejemplo, dado un BitSet llamado bucketOfBits, la llamada:

bucketOfBits.get(i) // true si el bit(i) activo, y false en caso contrario

bucketOfBits.set(i) // activa el bit i

bucketOfBits.clear(i) // desactiva el bit i.

JAVA.UTlL.BITSET

* BitSet(int nbits) // Construye un conjunto de bits.
* int length() // Devuelve la "longitud lógica" del conjunto de bits
* boolean get(int bit) // obtiene un bit.
* void set(int bit) // Activa un bit.
* void clear(int bit) // Desactiva un bit.
* void and(BitSet conjunto) // AND entre este conjunto / invocante.
* void or(BitSet conjunto) // OR entre este conjunto / invocante.
* void xor(BitSet conjunto) // XOR entre este conjunto / invocante.
* void andNot(BitSet conjunto) // Borra todos los bits de este conjunto que están activos en el invocante.

Algoritmo "criba de Eratóstenes"

Como ejemplo del uso de los conjunto de bits, vamos a mostrarle la implementación del algoritmo llamado "criba de Eratóstenes" para buscar números primos. Un número primo es aquel que sólo es divisible por sí mismo y por la unidad (como el 1, e1 3 o el 5), y la criba de Eratóstenes fue el primer método desarrollado para enumerar dichas estructuras numéricas. Éste no es el mejor algoritmo para buscar números primos, pero su uso se ha popularizado para comprobar el rendimiento de un compilador o de Pcs(Benchmark).

Este programa cuenta todos los números primos contenidos entre 2 y 1.000.000 (en total, son 78.498, por lo que lo que mostraremos los 10 primeros.

La clave de todo el proceso estriba en recorrer un conjunto de bits compuesto por un millón de posiciones. En primer lugar, debemos activar todos esos bits. Después, desactivamos aquellos bits que son múltiplos de números que sabemos que son primos. Las posiciones de los bits que permanecen tras este proceso son, en sí mismos, los números primos.

```
import java.util.*;
// Este programa ejecuta la Criba de Erastótenes.
// para computar números hasta 1.000.000.
public class CribaEras extends BitSet{
    long start, end;
```

```
    int count, i;
    public CribaEras(int n){
        super(n);
        start = System.currentTimeMillis();
        count = 0;
        for (i = 2; i <= n; i++) set(i);
        i = 2;
        while (i * i <= n){
            if (get(i)){
                count++;
                int k = 2 * i;
                while (k <= n){
                    clear(k);
                    k += i;
                }   //  while
            }   // if
            i++;
        }   // while
        while (i <= n){
            if (get(i)) count++;
            i++;
        }
        end =  System.currentTimeMillis();
     }

    public String toString(){
        String aux = "Criba de Erastótenes 1..1000000\n";
        aux+= count + " primos\n";
        aux+= (end - start) + " milisegundos\n";
        aux+= "Primeros 10 primos \n";
        for(int i = 0, cont = 0; cont < 10 && i < 1000;i++)
           if (get(i)){
              aux+=i + ", ";
              cont++;
           }
        return aux;
    }

    public static void main(String[] s){
        CribaEras criba = new CribaEras(1000000);
        System.out.println(criba);
    }
}
```

```
run:
Criba de Erastótenes 1..1000000
78498 primos
161 milisegundos
Primeros 10 primos
2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
```

Herencia y Polimorfismo en Java. Ejemplos

Afirmando tópicos ya vistos en AED, sobre herencia y polimorfismo, se analizarán algunos ejemplos sencillos en Java: varias clases para recorrer e imprimir series numéricas. Una serie numérica es una sucesión de números en donde el valor de cada uno depende de uno o más de los valores anteriores. Por ejemplo, una serie aritmética determina el número siguiente sumando; así, la serie geométrica determina el número siguiente multiplicando. En cualquier caso, una serie requiere una forma de definir su primer valor y también una forma de identificar el valor actual.

Se comenzará definiendo una clase, Progression, que se ve en mas adelante, que define los campos "genéricos" y los métodos de una serie numérica. En especial, define los siguientes campos de dos enteros largos:

first: el primer valor de la serie;
cur: el valor actual de la serie;

y los tres métodos siguientes:

getFirstValue(): Restablecer la serie al primer valor, y mostrar ese valor.
getNextValueO: Mover la serie al siguiente valor, y mostrar ese valor.

printProgression(n): Reiniciar la serie e imprimir sus primeros *n* valores.

Se dice que el método printProgression no tiene salida, o resultado, porque no retorna valor alguno, mientras que los métodos getFirstValue y getNextValue si lo hacen.

La clase Progression también incluye un método ProgressionO, que es un método *constructor.* Recuérdese que los métodos de constructor establecen todas las variables de instancia en el momento en que se crea un objeto de esta clase. La clase Progression pretende ser superclase genérica, de la cual se heredan clases especializadas, por lo que este constructor es el programa que se incluirá en los constructores para cada caso que extienda la clase Progression.

```
public class Progression {
      protected long first;   // Primer valor de la serie
      protected long cur;           // Valor actual de la serie.
      Progression(){                // Constructor predeterminado
            cur = first = 0;
      }
      protected long getFirstValue(){ // reinicializar y regresar el primer 
valor
            cur = first;
            return cur;
      }
      protected long getNextValue(){      // Retona corriente previamente 
avanzado
            return ++cur;
      }
      public void printProgression(int n){ // Imprime n primeros valores
            System.out.print(getFirstValue());
            for (int i = 2; i <= n; i++){
                  System.out.print(" " + getNextValue());
            }
      }
}
```

Una clase de serie aritmética

A continuación se describirá la clase ArithProgression.

Esta clase define a una serie en la que cada valor se determina sumando inc, un incremento fijo, al valor anterior. Esto es, ArithProgression define a una serie aritmética.

La clase ArithProgression hereda los campos first y cur, y los métodos getFirstValueO e printProgression(n) de la clase Progression. Se agrega un campo nuevo, inc para guardar el incremento, y dos constructores para establecer el incremento. Por último, sobrepone el método GetNextValuerO para ajustarse a la forma de obtención de un siguiente valor para la serie aritmética.

En este caso lo que funciona es el **polimorfismo**. Cuando una referencia Progression apunta a un objeto ArithProgression, los objetos usan los métodqs getFirstValueO y getNextValueO de ArithProgression. Este polimorfismo también es válido dentro de la versión heredada de printProgression(n) porque aquí las llamadas de los métodos GetFirstValueO y getNextValueO son implícitamente para el objeto "actual" (que en Java se llama this) y en este caso serán de la clase ArithProgression.

En la definición de la clase ArithProgression se han agregado dos métodos constructores, un método predeterminado (por omisión) que no tiene parámetros, y un método paramétrico, que toma un parámetro entero como incremento para la progresión. El constructor predeterminado en realidad llama al paramétrico, al usar la palabra clave **this** y pasar a 1 como el valor del parámetro de incremento. Estos dos constructores ilustran la sobrecarga del método, en la que un nombre de método puede tener varias versiones dentro de la misma clase porque en realidad un método se especifica por su nombre, la clase de objeto que llama y los tipos de argumentos que se le pasan a él: su firma. En este caso, la sobrecarga es para métodos constructores (un constructor predeterminado y uno paramétrico).

La llamada this(1) al constructor paramétrico como primera declaración del constructor predeterminado activa una excepción a la regla general de encadenamiento de constructor. Es decir, siempre que la primera declaración de un constructor C' llama a otro constructor C" de la misma clase usando la referencia **this**, el constructor de superclase no es llamado implícitamente para C'. Sin embargo, nótese que a la larga un constructor de superclase será llamado a lo largo de la cadena, sea en forma explícita o implícita. En particular, para la clase ArithProgression, el constructor predeterminado de la superclase (Progression) es llamado en forma implícita como primera declaración del constructor paramétrico de ArithProgression.

```
class ArithProgression extends Progression {
      protected long inc;     // incremento
      ArithProgression() {//  Constructor predeterminado, llama al parametrico
            this(1 );
      }
      ArithProgression(long increment) { //Constructor paramétrico (incremento)
            super();
            inc = increment;
      }
      protected long getNextValue(){      // metodo sobrescrito
            cur += inc; // Avanza la serie sumando el incremento al valor actual
            return cur; //    regresar siguiente valor de la serie

      //    Hereda getFirsValue();
      //    Hereda printProgression(..)
}
```

Una clase de serie geométrica

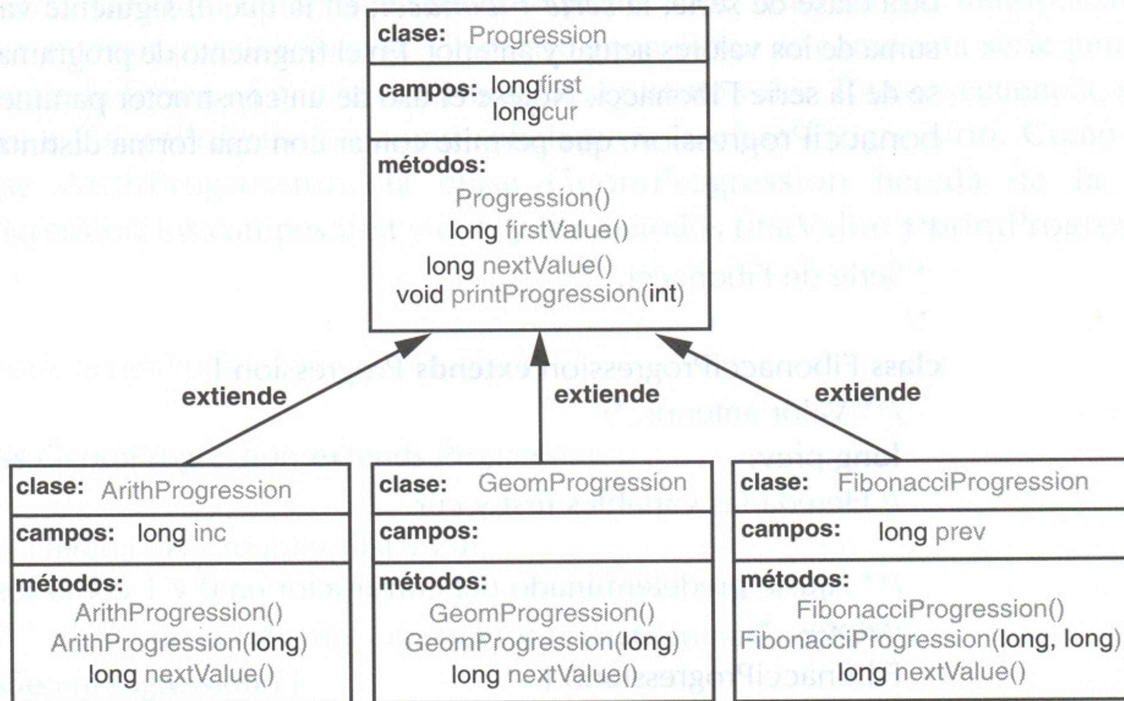
Se definirá la clase GeomProgression que se muestra a continuación, que implementa una serie geométrica, determinada multiplicando el valor anterior por una base b. Una serie geométrica es como una serie genérica, excepto la forma en que se determina el siguiente valor. En consecuencia, se declara a GeomProgression como subclase de la clase Progression. Como en la clase ArithProgression, la clase GeomProgression hereda de la clase Progression los campos first y cur, y los métodos GetFirstValue y printProgression.

```
class GeomProgression extends Progression { //  Hereda first y cur
      GeomProgression(){      // Constructor prdeterminado, llama al parametrico
                  this(2);
      }
      GeomProgression(long base){   // Constructor parametrico,
            super();
            first = base;     // Define el primer termino en funcion de base
            cur = first;      // y el corriente de igual forma
      }
      protected long getNextValue(){
            cur *= first;     // Avanza la serie
            return cur;       // Retorna el siguiente valor de la serie
      }
      //    Hereda getFirstValue()
      //    Hereda printProgression(int).
}
```

Clase de la serie de Fibonacci

Como un ejemplo más se definirá una clase FibonacciProgression que representa otra clase de serie, la *serie Fibonacci,* en la que el siguiente valor se define como la suma de los valores actual y anterior. Nótese el uso de un constructor parametrizado en la clase FibonacciProgression, que permite contar con una forma distinta de iniciar la serie.

```
class FibonacciProgression extends Progression{
      long prev;  // Valor anterior
      // Hereda las variables first y cur
      FibonacciProgression(){ //  Constructor predeterminado
            this(0, 1);       // llama al parametrico
      }
      FibonacciProgression(long value1, long value2){//    Constructor
paramétrico
            super();
            first = value1;               //    Definimos primer valor
            prev = value2 - value1; // valor ficticio que antecede al primero
      }
      protected long getNextValue() {     //    Avanza la serie
            long temp = prev;
            prev = cur;
            cur += temp;                        //    sumando el valor anterior
al valor actual
            return cur;                         //    retorna el valor actual
      }
      // Hereda getFirstValue()
      // Hereda printProgression(int).
}
```



Para completar el ejemplo, se definirá una clase Tester, que ejecuta una prueba sencilla de cada una de las tres clases. En esta clase, la variable prog es polimórfica durante la ejecución del método main, porque se refiere a objetos de la clase ArithProgression, GeomProgression y FibonacciProgression en turno.

El ejemplo presentado en esta sección es pequeño, pero proporcjona una ilustración sencilla de la herencia en Java. Sin embargo, la clase Progression, sus subclases y el programa probador tienen varias limitaciones que no son aparentes de inmediato. Uno de los problemas es que las series geométrica y de Fibonacci crecen con rapidez, y no está previsto el manejo del desbordamiento inevitable de los enteros largos que se manejan. Por ejemplo, como 3 **40 > 2**63, *(** significa potenciación, lease 3 elevado al exponente 40)*una serie geométrica con la base *b* = 3 tendrá desbordamiento de entero largo después de 40 iteraciones. En forma parecida, el 94avo término de la serie de Fibonacci es mayor que 2**63, por lo que esta serie desbordará después de 94 iteraciones. Otro problema es que podrían no permitirse valores iniciales arbitrarios para una serie de Fibonacci. Por ejemplo, ¿se permite una serie Fibonacci que inicie con 0 y -1? Para manejar errores de entrada o condiciones de error que se presenten durante la ejecución de un programa de Java se requiere tener algún mecanismo que lo haga. Este tema se llama Tratamiento de Excepciones y lo vimos en AED.

```
/** Programa de prueba para las clases de series */
import ArithProgression;
import GeomProgression;
import FibonacciProgression;
class Tester{
      public static void main(String[] args){ Progression prog;
            System.out.println("\n\nSerie aritmética con incremento
predeterminado: ");
            prog = new ArithProgression();
            prog.printProgression(10);
            System.out.println("\n\nSerie aritmética con incremento 5: ");
```

```
            prog = new ArithProgression(5);
            prog.printProgression(10);
            System.out.println("\n\nSerie geométrica con la base predeterminada: 
");
            prog = new GeomProgression();
            prog.printProgression(10);
            System.out.println("\n\nSerie geométrica con base 3: \n");
            prog = new GeomProgression(3);
            prog.printProgression(10);
            System.out.println("\n\nSerie de Fibonacci con valores iniciales 
predeterminados: ");
            prog = new FibonacciProgression();
            prog.printProgression(10);
            System.out.println("\n\nSerie Fibonacci con valores iniciales 4 y 6: 
");
            prog = new FibonacciProgression(4,6);
            prog.printProgression(10);
      }
}
```

```
Serie aritmética con incremento predeterminado:
0 1 2 3 4 5 6 7 8 9

Serie aritmética con incremento 5:
0 5 10 15 20 25 30 35 40 45

Serie geométrica con la base predeterminada:
2 4 8 16 32 64 128 256 512 1024

Serie geométrica con base 3:
3 9 27 81 243 729 2187 6561 19683 59049

Serie de Fibonacci con valores iniciales
predeterminados:
0 1 1 2 3 5 8 13 21 34
```