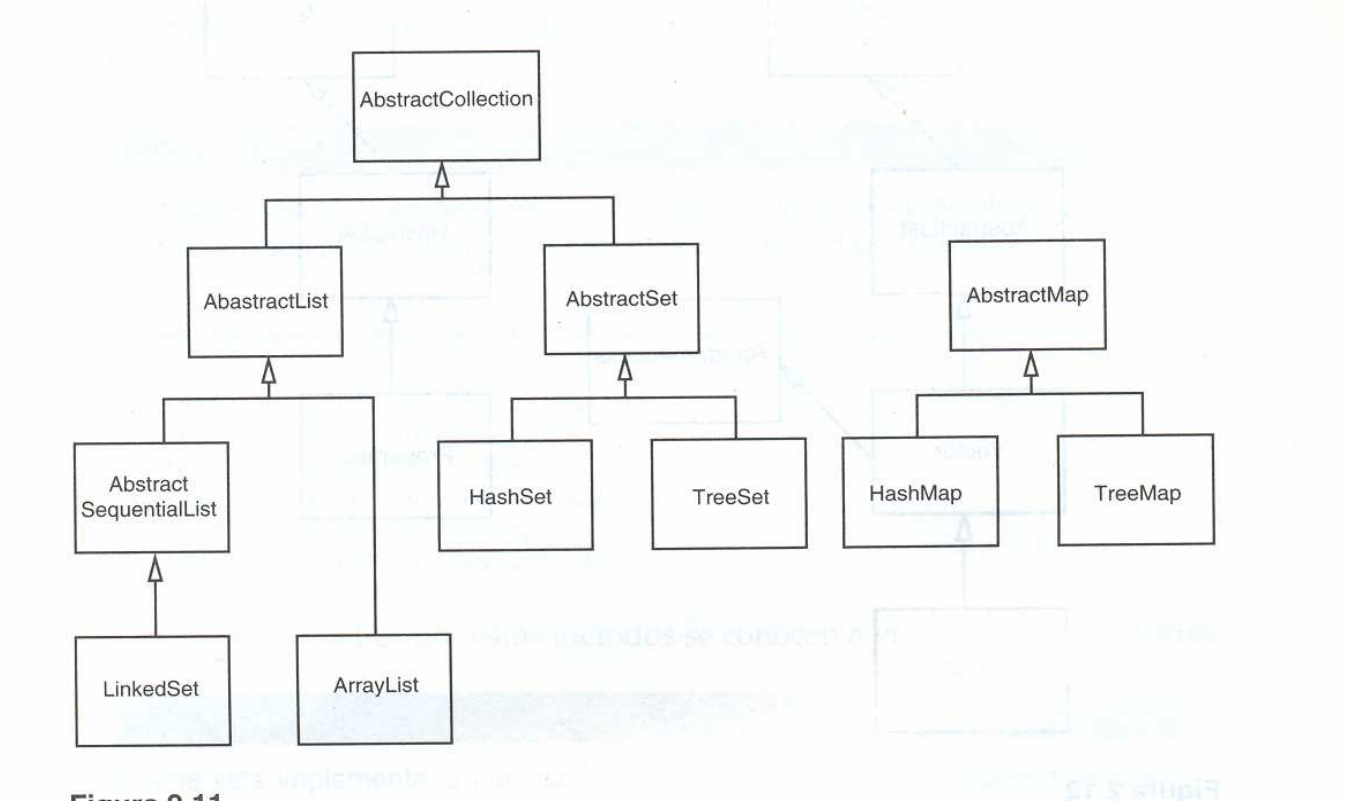
Indice Unidad I – Programación Orientada a Objetos avanzada

Introducción. La estructura de las colecciones . . 2
Interfaces de colecciones . . . 3
Interfaces de colección e implementaciones separadas . 3
Colecciones y las interfaces de iteración en la librería Java 4
Colecciones concretas 7
Listas enlazadas (LinkedList). . . 7
public class LinkedListTest . . 11
public class SortedList extends LinkedList . 12
Array de listas (ArrayList). . . 15
Acceso a los elementos de un ArrayList. . 16
Inserción y eliminación de elementos intermedios 17
Una implementación de ArrItems extendiendo ArrayList 20
Conjuntos de hash 21
Usando class HashSet . . . 22
Arboles 24
Árboles rojo-negro . . . 24
Comparación de objetos . . 25
Class TreeSet 27
Arbol TreeSet ordenado según interfaz Comparable . 27
Arbol TreeSet ordenado según objeto Comparator 28
Conjuntos de bits . . . 30
public class CribaEras extends BitSet . . 30

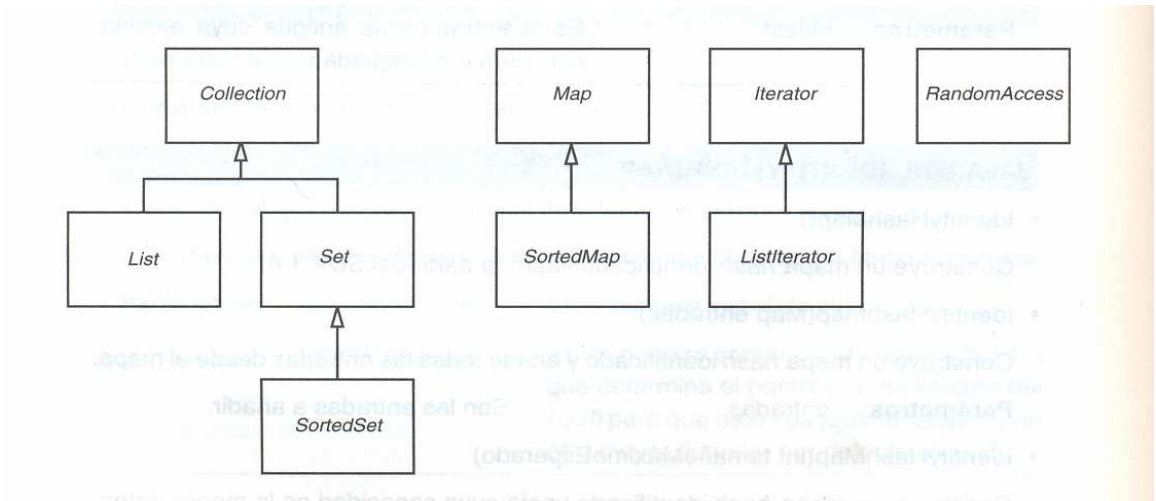
La estructura (framework) de las colecciones

- **Conjunto de clases para lograr funcionalidades avanzadas.**
- **Contienen superclases con mecanismos, políticas y funcionalidades**
- **El usuario compone subclases para heredar las funcionalidades**
- **Por ejemplo, Swing es una estructura para interfaces de usuario.**



clases abstractas	concretas	contenedor "heredadas"
AbstractCollection	LinkedList	Vector
AbstractList	ArrayList	Stack
AbstractSequentialList	HashSet	Hashtable
AbstractSet	TreeSet	Properties
AbstractMap	HashMap	
	TreeMap	

Estas clases implementan la estructura de interfaces:



Interfases de colección con implementaciones separadas

Java separa las interfaces de sus implementaciones en clases.
Sea una interfase de comportamiento muy conocido: la cola.

El lenguaje Java no soporta la estructura cola.
Debemos implementarlas con otros recursos, (Ej. LinkedList)

El comportamiento de una interfaz cola(queue) especifica:

- **que los elementos se añaden por la parte final de la misma**
- **se eliminan por la cabecera**
- **permite determinar cuántos elementos existen dentro de ella.**
- **Apropiada para una colección de objetos que cumplan el requisito "primero en entrar, primero en salir" ("first in, first out")**

Si hubiera, podría parecer:

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

La interfaz no indica nada acerca de la implementación.
Clásicamente existen dos implementaciones:

- **Usando un "array circular"**

```
class ColaEnArrayCircular implements Queue{
…   …
}
```

- **Utilizando una lista de nodos vinculados**(Vide EAD).

```
class ColaConNodosVinculados implements Queue{
…   …
}
```

Al usarla, después de instanciar el objeto, no hay diferencias.

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

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

Como elegimos la implementación?

ColaEnArrayCircular:

- **Muy eficaz.**
- **Es una colección con una capacidad finita.**
- **Entonces método add puede fallar:**
  public void add(Object obj) throws CollectionFullException
  y problema: no se puede añadir clausula throws al sobrescribir add

ColaConNodosVinculados

- **No tan eficaz.**
- **Su capacidad no tiene límites explícitos.**

Colecciones y las interfaces de iteración en la librería Java

La interfaz fundamental es Collection.
Dos métodos fundamentales:

boolean add(Object obj)
- **Añade un objeto a la colección**
- **devuelve true si dicho objeto cambia la colección**
- **false en caso contrario..**

Iterator iterator()
- **El método iterator() de Collection devuelve un objeto que implementa otra interfaz: Iterator.**

La interfaz Iterator permite recorrer los elementos de una colección.
Su comportamiento consta de los siguientes métodos:

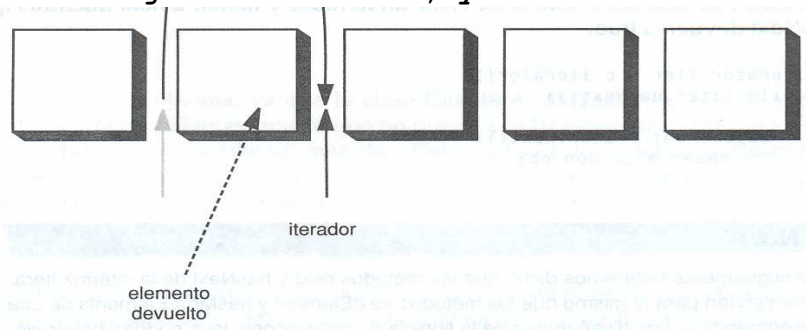
- **Object next()**
- **boolean hasNext()**
- **void remove()**

Usando next en ciclo puedo recorrer elementos de una colección uno a uno.
Problema: al llegar a su final, next lanza una NoSuchElementException.
Solución: invocar el método hasNext() antes de llamar a next().
Si quiere procesar todos los elementos de una colección debe
- **crear un iterador,**
- **llamar a next() mientras hasNext() devuelva true.**

```
Iterator iter = c.iterator(); //    creo iterador
while (iter.hasNext()){       //    mientras exista próximo itero
        Object obj = iter.next();
        // hacer algo con obj
}
```

remove() elimina el elemento devuelto por la última llamada a next.

La iteración recorre elementos. Cuando llame a next(), el iterador salta sobre el siguiente elemento, y devuelve una referencia al anterior.



Para borrar el objeto sobre el cual está posicionado, primero deberá avanzar al siguiente. Por ejemplo, para eliminar el primer elemento:

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

next() y remove() están vinculados.
Es ilegal invocar remove() sin previa a next().
Si lo hacemos, IllegalStateException.

para borrar dos elementos adyacentes:

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

Las interfaces Collection e Iterador son genéricas.
Es posible escribir métodos que operen sobre cualquier colección.
La colección es informada como parámetro del método.

```
public String myString(Collection e){
      String aux =”[“;
      Iterator iter = c.iterator();
      while (iter.hasNext()
            aux+=iter.next() + “, “;
      aux+="]";
      return aux;
}
```

Copiando todos los elementos de una colección a otra:

```
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;
}
```

La interfaz Collection declara métodos (abstractos).

```
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
```

Para no tener que redefinir tantos métodos: class AbstractCollection:

- **Abstractos los métodos fundamentales, *add(), iterator()***
- **Implementa los demás métodos en función de los fundamentales:**

```
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;
      }
}
```

Colecciones concretas

Listas enlazadas

- **guarda cada objeto en un nodo separado.**
- **Cada nodo tiene una referencia al siguiente.**
- **Esto lo vimos en la unidad III – AED**
- **Allí hacíamos todo el trabajo (mucho).**
- **La colección LinkedList implementa una lista doblemente enlazada.**
- **LinkedList implementa la interfaz Collection.**
- **Utiliza los métodos tradicionales para recorrer la estructura.**

```
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
}
```

- **add() de LinkedList añade el objeto al final de la lista.**
- **Para insertar ordenado, usar un iterador.**
- **Usar iteradores en colecciones que deben mantener un orden natural.**
- **Ejemplos:**
  - **conjunto (set) no impone ningún orden a sus elementos.**
  - **(no es necesario el método add de la interfaz Iterator).**
  - **Si la lista requiere un orden, disponemos de subinterfaz: ListIterator que contiene un método add:**

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

- **Listlterator puede recorrer las listas hacia atrás.**
  - **Object previous()**
  - **boolean hasPrevious()**
  - **previous() devuelve el objeto que ha sido dejado atrás.**

- **Para implementar ListIterator en la clase LinkedList:**

```
ListIterator iter = equipo.ListIterator();
iter. next () ;
iter.add("Julieta");
```

Queda Angelina, Julieta, Pablito, Carlitos.

- **sucesivas llamadas a add(), los elementos son insertados en la posición anterior a la que marca en ese momento el iterador.**

- **Si el iterador apunta al comienzo de la lista enlazada, el nuevo elemento añadido pasa a ser la cabecera de la lista.**

- **Si el iterador alcanza último elemento (cuando hasNext() devuelve false), el elemento añadido pasa a ser la cola de la lista.**

- lista con n elementos, n+1 lugares posibles para añadir
  - 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:

- next(), nodo a izquierda.
- previous(), nodo a derecha.

- existe un método set()

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

- en programación concurrente, problemas:
  - si un iterador recorre y otro modifica.
  - Usando add() de la clase y add() del iterador (Compilación). (ConcurrentModificationException)

```
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
                     (No existe próximo de un nodo removido)
```

- Existen otros muchos métodos interesantes p/listas.
- Mayormente en la superclase AbstractCollection
- LinkedList extiende AbstractColIection.
- contains("Harry") devuelve true si la lista contiene "Harry".
- (Pero no posiciona iterador.)
- No soportan acceso directo.
- Si 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 con 5 demos:

- generando listas,
- invirtiéndolas,
- verificando existencia de elementos,
- generando una lista ordenada.

```
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);
    }
```

```
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]
```

```
    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);
    }
```

```
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]
```

```
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");
}
```

```
Demo03 - Verifica si todas de list01 existen en list02
list02  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");
}
```

```
Demo04 - remuevo ultimo de list02, luego idem demo03
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();
         …
  }
}
```

```
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 (ArrayList)

- encapsula un array Object[] reubicable **dinámicamente**.
- Idem Vector, pero con métodos no sincronizados.
- Normalmente lenguajes fijan tamaño de los arrays en compilación.
- Algunos permiten fijar el tamaño en tiempo de ejecución. (Pero una vez fijado, inamovible)
- ArrayList: expansión y reducción en tiempo de ejecución. (automatico)

Diferencias entre un array T[] y un ArrayList

- **T[]**: característica del lenguaje lava; Uno para cada tipo.
- **ArrayList** es una clase; "en el tamaño encaja todo" (Object)

Elementos se añaden con add()

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

ArrayList

- gestiona un array interno de referencias Object.
- no tiene límites.
- Si llama a add() y el array interno está lleno,
- crea otro array más grande y
- copia automáticamente los elementos.

Se puede asegurar/ajustar su capacidad:

- equipo.ensureCapacity(100);
- new ArrayList(100);
- equipo.size() devuelve el **número real** de objetos.
- Equipo.trimToSize() Ajusta la capacidad a size()

Acceso a los elementos de un ArrayList.

- No mas [], ahora get() y set()
- equipo.**set** (i, "Antonio");
- Empleado e = (Employee)equipo.**get**(i);
- Contabilizan sus elementos a partir de cero.
- puede contener objetos de **diversas clases** (heterogéneo, inseguro)
- (es posible añadir por accidente un elemento de un tipo erróneo)
- pero podemos chequear que estamos accediendo:

```
Date birthday = . . .;
equipo.set(i, birthday);
ArrayList list;
list.add (new Empleado(datos));
list.add(new Date(birthday));

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

- **A favor**: elementos de estructura jerárquica, OK: **polimorfismo**.
- **En contra**: no estructura jerárquica, mucho uso **instanceof**.

Inserción y eliminación de elementos intermedios en un ArrList

- elementos se pueden insertar en cualquier punto
  equipo.add(n,e);// n y siguientes ➔
  (Realocación dinámica de capacidad, si necesario)

  Empleado e = (Empleado)equipo.remove(n);
  ➔ n+1 elementos
- No son operaciones eficientes. Criterio:
  - mayoría accesos directos. Usar ArrayList.
  - mayoría inserción/eliminación. Usar LinkedList.

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.

Un ejemplo

La clase Item (Vista en AED)

```
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 (Vista en AED)

```
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){// según 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

Búsqueda por clave

LinkedList, método contains(). Búsqueda secuencial

ArrayList, método contains(). Idem

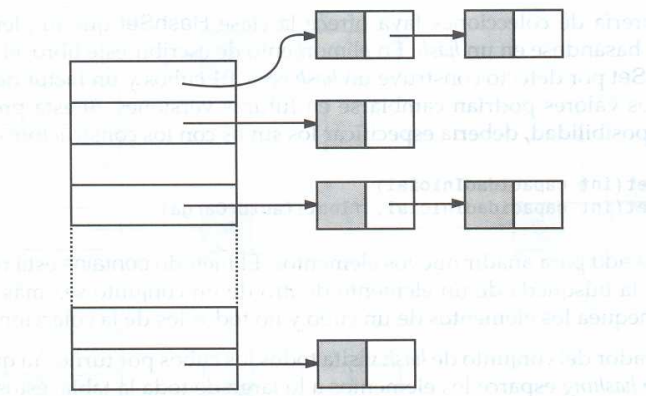
HashSet, HashMap, contains(), containsKey(). Acceso directo

Tabla hash: excelente para localizar objetos de una forma rápida.
(También arrays asociativos)

- Un hash procesa un entero, llamado código hash, para cada objeto.
- Un hash es un array de listas enlazadas.
- Cada lista recibe el nombre de cubo, o bucket.
- Para localizar el cubo se procesa su código hash.

Ejemplo de un algoritmo de localización:

- Código hash: 345
- Cant cubos: 101
- Indice cubo: 345 % 101 = 42



Mas sobre tablas Hash

- puede suministrar el número inicial de cubos.
- Si el algoritmo es adecuado, carga cubos pareja.
- Es usual sobredimensionar la capacidad inicial del hash.
- si necesita almacenar 100 entradas, prevea capacidad inicial 150.
- La tabla hash puede llenarse.
- ocurre cuando carga **factor de carga**. (Por defecto 0.75)
- Java **redimensiona automáticamente** duplicando cantidad de cubos.
- pueden usarse para varias estructuras de datos importantes.
- Ejemplo: conjunto o set.
- colección de elementos sin duplicaciones..

Java ofrece la clase **HashSet** para implementar un conjunto

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

- contains() sólo chequea los elementos de su cubo.
- El iterador del conjunto hash visita todos los cubos.

Ejemplo:
El programa:

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

```
import java.util.*;
import java.io.*;
public class SetTest{
   public static void main(String[] args){
      Set palabras = new HashSet();
      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
```

Funciones hash

- Podemos insertar cadenas en tabla hash
- String dispone de hashCode, genera código hash de la cadena

Árboles (Colección TreeSet)

- es similar al hash, con ventajas: **colección ordenada**.
  - búsqueda por clave con mínima cantidad de comparaciones.
  - Recorriendo el árbol, elementos en secuencia correcta.
- **TreeSet** usa un árbol **rojo-negro** (red-black tree).
- la ordenación es ejecutada por el propio árbol:
  - cada elemento agregado ➔ posición correcta, entonces
  - el recorrido del iterador ➔ secuencia correcta

Árboles, Introducción **muy muy** suscinta.

- Capacidad de representar relaciones 1:n
- Es un grafo con restricciones
  - No puede tener ciclos
  - No puede tener elementos disyuntos
  - Cumpliendo restricciones anteriores ➔ árbol libre
  - Si definimos nodo raíz, ➔ árbol dirigido
  - Si agregamos orden, ➔ árbol ordenado
- Dependiendo de la cantidad de descendientes:
  - Mas de dos, n-ario o multicamino
    (Ej, BTree, estructura de índice de archivos)
  - Hasta dos, binarios
    - Estructuras de búsqueda en memoria dinámica
      (Conocidos: (2-4), AVL, rojo-negro)

Árboles rojo-negro

- árboles AVL y (2,4), buenos, pero:
  - no se adaptan bien gestionando claves duplicadas
    (Ejemplo: aplicaciones de diccionario)
  - AVL requieran muchas operaciones de reestructuración (rotaciones) después de la remoción de un elemento,
  - (2,4) pueden requerir muchas operaciones de fusión o de partición después de cada inserción o remoción.
- árbol **rojo-negro**, no tiene estos inconvenientes
  - Soporta bien claves duplicadas
  - sólo requiere hacer O(1) cambios estructurales después de una actualización para permanecer balanceado.

Idea fuerza: *TreeSet provee eficiente árbol de búsqueda binario balanceado en altura)*

Definición de árbol rojo-negro

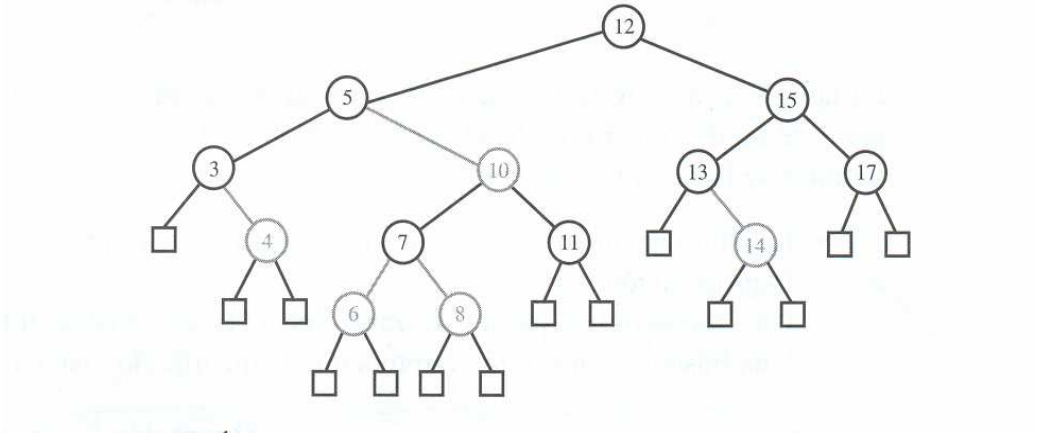
- es un árbol de búsqueda binaria
  - con los nodos "iluminados" de rojo y de negro
  - en una forma que satisface los siguientes

requisitos:

- **Propiedad de la raíz**: La raíz es "negra"
- **Propiedad externa**: Todo nodo externo es "negro". (Cuadrados)
- **Propiedad interna**: Los hijos de un nodo "rojo" son "negros".
- **Propiedad de profundidad**: Todos los **nodos externos** tienen **la misma profundidad negra: cantidad de antepasados negros.**

Presentamos un **árbol rojo-negro**

- Claves mayores se insertan a derecha
- Claves menores a izquierda
- (Claves iguales, investigar)
- Imagine "**negros**" los nodos 12, 5, 15, 3, 13, 17, 7 y 11
- Imagine "**rojos**" los nodos 10, 4, 14, 6 y 8



- Cada nodo externo tiene 3 antepasados negros,
- en consecuencia, su profundidad negra es 3.
- los datos están guardados en los nodos internos
- El algoritmo puede ver nodos externos del árbol:
  - vacíos
  - No existir
  - Ser un único objeto NODO_NULO

- Añadir un elemento a un árbol es
  - más lento que hacerlo a un hash,
  - mucho más rápido que hacerlo en una lista enlazada.

  (ejemplo, en un con 1.000 elementos, 10 comparaciones)

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 conteniendo colección.

Comparación de objetos

¿Cómo sabe un TreeSet la forma de ordenar los elementos?

- Por defecto, asume objetos que **implementan la interfaz Comparable.** Esta interfaz define un **solo** método:

  int compareTo(Object otro)

  La llamada a.compareTo(b) devuelve:
  - 0 si a y b son iguales,
  - un valor negativo si a < b
  - un valor positivo si a > b

String **implementan la interfaz Comparable**
Su compareTo ➔ orden alfabético (o lexicográfico).

Si usa objetos propios, debe redefinir el compareTo.
Si usamos Ítems, substituimos esMenor() y esMayor().

```
class Item implements Comparable{// claves asociadas a un valor
      protected int codigo;
      protected float valor;
      …
      public int compareTo(Object other){
            Item otro = (Item)other;
            return codigo - otro.codigo;
      }
}
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)
```

- La interfaz Comparable para definir orden tiene limitaciones.
- La clase sólo puede implementar la interfaz una vez y entonces
- los objetos vienen con su ordenamiento pre-definido

- qué podemos hacer si?
  - necesitamos distintos ordenamientos
  - los objetos no implementan la interfaz Comparable?

- **Solución flexible**: que el árbol compare según los objetos Comparator que le 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)
La llamada compare(a, b) devuelve:
- 0 si a y b son iguales,
- un valor negativo si a < b
- un valor positivo si a > b

- Para ordenar los elementos por un atributo determinado definimos
- una clase que implementa la interfaz Comparator para cada atributo.

```
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;
    }
}

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);
    }
}
```

```
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 **compare()**, podrá ordenar los elementos como desee.

Conjuntos de bits

- **BitSet** almacena secuencias de bits en un vector de bits.
- recomendable para eficientes secuencias de indicadores (mucho mejor que un ArrayList de objetos Bolean)
- **BitSet** le proporciona una interfaz adecuada para
  - la lectura,
  - el almacenamiento y
  - la reinicialización de bits individuales.

- Por ejemplo, sea el objeto bucketOfBits, la llamada:
  bucketOfBits.get(i) // true si bit(i) activo
  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.

Demo: Algoritmo "criba de Eratóstenes"

- el primer método desarrollado para enumerar primos.
- se ha popularizado para comparar performance de compiladores
- **Este programa**
  - Cuenta los números primos contenidos entre 2 y 1.000.000
  - muestra los 10 primeros.

```
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();
        for (count = 0,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,
```