UNIDAD 6

GENÉRICOS Y COLECCIONES

1. INTRODUCCIÓN.
2. GENÉRICOS.
   2.1. Puntos Importantes Sobre Genéricos.
   2.2. Limitar Tipo, Operador Diamante y Wilcard.
3. ITERADORES Y COMPARADORES.
   3.1. Iterar: Interfaces Enumeration e Iterator y For-Each.
   3.2. Comparar: equals(), hashCode(), Comparable<T> y Comparator
4. USAR COLECCIONES PRECONSTRUIDAS.
   4.1. Interface List<T>: Clases ArrayList, LinkedList, Vector...
   4.2. Interface Set<T>: Clases TreeSet, HashSet y EnumSet.
   4.3. Colas con Prioridad.
   4.4. Mapas.
   4.5. Clases legacy: Dictionary, Stack, Vector.
5. CREA TUS PROPIAS COLECCIONES.
   5.1. TDA's.
   5.2. Listas.
   5.3. Árboles: Binarios, de Búsqueda, Balanceados, ...
   5.4. Tablas Hash.
   5.5. Implementar Soporte de Bucle for-each.
6. EJERCICIOS.

BIBLIOGRAFÍA:
- Java, Cómo Programar (10ª Ed.) Pearson. Paul y Harvey Deitel (2016).
- Estructuras de Datos y Algoritmos en Java. Robert Lafore (Sams 1998)
- Algorithms (4 Ed). Robert Sedgewick and Kevin Wayne Princeton University (Addison-Wesley 2011).
- Master in Delphi (Data Structures and Algorithms), 2006.
- Estructuras de datos y algoritmos. Aho, Hopcroft, Ullman (Addison-Wesley)
- https://www.geeksforgeeks.org/java/

6.1. INTRODUCCIÓN.

Las estructuras de datos son colecciones de elementos (datos relacionados). Los arrays, por ejemplo, son estructuras de datos que consisten en elementos de datos relacionados y del mismo tipo y mantienen la misma longitud una vez creados.

Aunque se utilizan con frecuencia, tienen capacidades limitadas. Por ejemplo, si indicas el tamaño de un array y necesitas modificar ese tamaño en tiempo de ejecución, debes crear uno nuevo. Una de las estructuras de datos prefabricadas de Java (colecciones de la API de Java) es **ArrayList**. Los objetos **ArrayList** son similares a los arrays, sólo que proporcionan cambio de tamaño dinámico cuando es necesario.

Igual que resolver un problema estadístico con muchos datos (hacerle la media a los datos, buscar máximos, mínimos, etc.) es complicado sin arrays, muchos problemas se vuelven complicados si no tienes la estructura de datos que te ayude a resolverlo. **Usar la colección de datos adecuada, simplifica la solución de muchos problemas**.

En esta unidad veremos algunas estructuras de datos (colecciones) usuales y como utilizarlas, como usar las que tiene Java preconstruidas y como Java nos da soporte para que podamos crear las nuestras (comentamos algunas que nos pueden ser útiles, aunque no con la profundidad que este tema se merece. :-( ).

6.2 GENÉRICOS.

Cuando creas una estructura de datos (colección) debes indicar el tipo de datos que tienen los elementos que va a almacenar la estructura. Si por ejemplo haces una lista, tendrás que programar una lista para

datos int, otra para datos float, otra para datos double, otra para cada determinado tipo de objeto que tu programa quiera manejar (imágenes, facturas, ...).

Sin embargo, el código de la estructura es prácticamente el mismo en todos los casos. Este inconveniente podría solucionarse si pudieses **parametrizar el código**, **hacer una lista genérica**, es decir, cuando vayas a usar el código de la estructura, poder decirle en ese momento el tipo de dato que vas a guardar en ella. De esta forma, solo tendrías que programar **un solo código que se puede utilizar con cualquier tipo de dato**.

Las primeras versiones de Java no tenían esta característica, así que para conseguir código genérico, las clases se definían para contener tipos **Object** que permite almacenar cualquier cosa. Por ejemplo la clase **ArrayList** originalmente no estaba parametrizada, y cuando recuperabas un elemento de la colección, tenías un **Object** que luego tenías que convertir al tipo real del elemento con un cast, ejemplo:

```
String item = (String)lista.get(i);
```

Otros lenguajes como **Smalltalk** y **C++**, no comprueban el tipo de dato en tiempo de compilación, lo hacen en tiempo de ejecución. En Smalltalk, tenías errores en tiempo de ejecución en vez de en tiempo de compilación cuando intentabas añadir un elemento de un tipo distinto al de la estructura. Y en Java tienes un error en tiempo de ejecución si intentas convertir a una clase incompatible un elemento (excepción **ClassCastException**).

Java 5.0 introdujo los **tipos parametrizados**, que hacen posible crear colecciones genéricas que **retrasan la comprobación de tipos hasta que se ejecuta el código**. Así que puedes crear un **ArrayList<String>**,

y el compilador solo permite añadir elementos **String** al array. Y cuando recuperes un elemento, no es necesario realizar un cast de tipos, porque solo puedes obtener un **String**. Java usa las clases parametrizadas como lo hace C++. Crear clases parametrizadas es opcional, puedes usarlas normalmente (y estarán usando Object) o parametrizadas.

Java lo que hace es eliminar la información del tipo en tiempo de ejecución, y vigilar lo que se añade en tiempo de compilación. Esto significa que **el operador instanceOf no funciona correctamente** porque hace la comprobación en tiempo de ejecución. Ejemplos:

```
if(lista instanceOf ArrayList<String>){ // Falla
new ArrayList<String>[N];               // Falla
new ArrayList[N];                       // OK, ArrayList existe
```

Los tipos parametrizados **permiten crear clases, interfaces y métodos en los que el tipo de datos sobre los que operan se especifica como parámetro**. Una clase, interfaz o método que funciona con un tipo de parámetro se denomina **genérico**, como una clase genérica o método genérico o interface genérica.

La ventaja del código genérico es que trabaja automáticamente con el tipo de dato indicado en su parámetro de tipo. Muchos algoritmos son lógicamente los mismos, independientemente del tipo de datos a los que se apliquen. Por ejemplo, un **Quicksort** (algoritmo de ordenación) es el mismo si está ordenando elementos de tipo Integer, String, Object, o Hilos. Con los genéricos, puedes definir un algoritmo una vez, independientemente de cualquier tipo específico de datos, y luego aplicar ese algoritmo a una amplia variedad de tipos de datos sin ningún esfuerzo adicional.

Es importante entender que Java siempre ha tenido la habilidad de

crear clases, interfaces y métodos generalizados operando a través de referencias del tipo **Object**. Debido a que **Object** es la superclase de todas las demás clases, una referencia de **Object** puede referirse a cualquier tipo de objeto. Así, en el código pre-genérico, las clases, interfaces y métodos generalizados utilizaban referencias a objetos para operar en varios tipos de datos.

El problema era que no podían hacerlo con la seguridad del tipo porque se necesitaban conversiones para convertir explícitamente de **Object** al tipo real de datos sobre los que se operaba. Por lo tanto, era posible crear accidentalmente bugs de tipo. Los genéricos aportan la seguridad que faltaba porque hacen que estas conversiones sean automáticas e implícitas. En resumen, **los genéricos amplían la capacidad de reutilizar el código y permiten hacerlo de forma segura y confiable**.

EJEMPLO 1: Un ejemplo simple de uso de genéricos. Creamos una clase Genérica:

<T> es el parámetro de tipo

```
class Gen<T>{   // T es el parámetro de tipo genérico
   T obj;       // Declara un objeto de tipo T

   public Gen(T o){ obj = o; }  // o es ref. a objeto de tipo T

   public T getObj(){ return obj; } // getter que devuelve obj T

   public void mostrarTipo(){  // Muestra el tipo de T
      System.out.println("El tipo de T es: " +
                         obj.getClass().getName() );
   }
}

// Demostración de uso de la clase genérica anterior
public class Genericos {
   public static void main(String[] args) {
      Gen<Integer> iOb; //Crea una referencia Gen para Integers
      iOb = new Gen<Integer>(28);
      iOb.mostrarTipo();
```

```
        int v = iOb.getOb();  // No necesitas convertir
        System.out.println("Valor: " + v);
        System.out.println();
        Gen<String> strOb = new Gen<String>("Prueba de genéricos");
        strOb.mostrarTipo();
        String str = strOb.getOb();
        System.out.println("Valor: " + str);
    }
}
```

Antes de continuar, es necesario recalcar que el compilador de Java no crea realmente versiones diferentes de Gen, o de cualquier otra clase genérica. Aunque es útil pensar en estos términos, no es lo que realmente sucede. Realmente, el compilador elimina toda la información de tipo genérico, sustituyendo las conversiones necesarias, para que el código se comporte como si se hubiera creado una versión específica de Gen. Por lo tanto, en realidad solo existe una versión de Gen en tu programa. El proceso de eliminación de información de tipo genérico se denomina borrado (**erasure**) que se explica más adelante.

Observa que cuando se llama al constructor Gen, se especifica el argumento de tipo (Integer, String, etc.). Si no es así, se producirá un error de tiempo de compilación. Por ejemplo, la siguiente asignación provocará un error en tiempo de compilación:

```
iOb = new Gen<Double>(28.0); // Error!
```

Porque iOb es del tipo `Gen<Integer>`, no se puede usar para referirse a un objeto `Gen<Double>`. Este tipo de control es uno de los principales beneficios de los genéricos porque añade seguridad del tipo. La declaración

```
iOb = new Gen<Integer>(28);
```

hace uso de autoboxing para encapsular el valor 28, que es un **int**, en un **Integer**. Esto funciona porque `Gen<Integer>` crea un constructor que

toma un argumento `Integer`. Como se espera un `Integer`, Java colocará automáticamente 28 dentro de uno. Por supuesto, la tarea también podría haber sido escrita explícitamente, así:

```
iOb = new Gen<Integer>( Integer.valueOf(28) );
```

Pero no hay necesidad ni ningún beneficio al usar esta última versión. Cuando programes tu código genérico puedes usar cualquier letra para indicar el parámetro de tipo, pero se suelen usar estas:

E — Elemento (usado en el Framework de Java Collections)
K — clave
N — Número
T — Tipo
V — Valor
S,U,V etc. — segundo, tercer, cuarto tipo...

6.2.1. PUNTOS IMPORTANTES SOBRE GENÉRICOS.

Los genéricos funcionan solo con tipos referencia (objetos).

Al declarar una instancia de un tipo genérico, el argumento de tipo pasado al parámetro de tipo debe ser un tipo referencia. No se puede utilizar un tipo primitivo, como **int** o **char**. Por ejemplo, con Gen, es posible pasar cualquier tipo de clase como T, pero no se puede pasar un tipo primitivo a T. Por lo tanto, la siguiente declaración es ilegal:

```
Gen<int> intOb = new Gen<int>(28); // Error, no puedes usar int
```

Por supuesto, no poder especificar un tipo primitivo en la declaración no es una restricción seria porque Java puede usar los envoltorios de tipo para encapsular un tipo primitivo. Además, el mecanismo de autoboxing y auto-unboxing hace que el uso del wrapper de tipos sea transparente al programador. Es decir en el código puedes usar:

```
// En declaraciones (amarillo) Integer, para usarlo
// tanto int como Integer
Gen<Integer> intOb = new Gen<Integer>(28);
```

Los tipos genéricos difieren según sus argumentos de tipo

Una referencia de una versión específica de un tipo genérico no es compatible con otra versión del mismo tipo genérico. Por ejemplo, asumiendo que estamos en el programa del ejemplo 1 anterior, la siguiente línea de código es un error y no se compilará:

```
iOb = strOb; // Error!
```

Aunque tanto iOb como strOb son de tipo Gen<T>, son referencias a diferentes tipos porque sus argumentos de tipo son diferentes. Esto añade seguridad de tipo y evita posibles errores.

No puedes crear instancias de tipos genéricos

Si lo haces obtienes un error de compilación, por ejemplo:

```
public static <E> void amplia(List<E> lista) {
    E elemento = new E();  // error de compilación
    lista.add(elemento);
}
```

Puedes hacerlo usando reflection como alternativa:

```
public static <E> void amplia(List<E> lista, Class<E> clase)
throws Exception {
    E elemento = clase.newInstance();   // OK
    lista.add(elemento);
}
```

Y cuando llamas al método amplia:

```
List<String> ls = new ArrayList<>();
amplia(ls, String.class);
```

No puedes instanciar arrays con elementos genéricos

Si defines una clase genérica con una variable de instancia que sea un array fallará si instancias el array. Ejemplo:

```
public class C1<T> {
    private T[] a;
    public C1(int capacidad) {
        a = new T[capacidad];  // error
    }
}
```

Podemos hacerlo usando `java.lang.reflect.Array` de la siguiente manera:

```
public class C1<T> {
    private T[] a;
    public C1(Class<T> clase, int capacidad) {
        a = (T[])Array.newInstance(clase, capacidad); // Ok
    }
}
```

Si lo hacemos sin el constructor:

```
public class C1<T> {
    private T[] a = (T[])Array.newInstance(T.class, 10); // Ok
}
```

EJERCICIO 1: Define un método estático y genérico llamado intercambia(array, i, j) que intercambie los elementos i y j del array.

No puedes declarar variables static con tipos genéricos

La variable global estática es accedible por todos los objetos (no estáticos) de la clase. Por ejemplo:

```
public class DispositivoMovil<T> {
    private static T os; // error!!
    // ...
}
```

De permitirse, el siguiente código sería confuso:

```
DispositivoMovil<Smartphone> sp = new DispositivoMovil<>();
DispositivoMovil<Pager> p = new DispositivoMovil<>();
DispositivoMovil<Tablet> t = new DispositivoMovil<>();
```

Como la variable os es accesible desde todos los objetos, ¿De qué tipo es os? ¿Smartphone, Pager o Tablet? ¿Los 3 a la vez?, ¿ninguno?

No puedes usar Casting o instanceof con genéricos

Como se aplica erasure (pérdida de información de tipo), no puedes cambiar el tipo o averiguar el tipo porque no se conoce el tipo. Ejemplo:

```
public static <E> void rtti(List<E> lista) {
    if(lista instanceof ArrayList<Integer>){  // error compilación
       ArrayList<Integer> li = (ArrayList<Integer>)lista;
        // ...
    }
}
```

La alternativa que tienes es usar un wildcar que te asegure que la lista es un ArrayList de tipo ilimitado por la base (lo veremos):

```
public static void rtti(List<?> lista) {
    if(lista instanceof ArrayList<?>){   // OK
        // ...
    }
}
```

Y con el casting, a no ser que hayas parametrizado con wildcars ilimitados:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>)li; // error en t-compilación
```

Sin embargo, en algunos casos el compilador sabe que el parámetro de tipo siempre es válido y permite el cast. Ejemplo:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

No puedes crear arrays de tipos parametrizados

El array necesita conocer el tipo de elementos que almacena. Pero el erasure hace que se pierda. Por ejemplo:

```
List<Integer>[] al = new List<Integer>[2]; // error
```

No puedes crear, atrapar o lanzar tipos genéricos

Una clase genérica no puede extender la clase `Throwable` ni directa ni indirectamente.

```
// Indirectamente
class MathException<T> extends Exception{ /*...*/ } //error

// Directamente
class ColaException<T> extends Throwable{ /*...*/ } // error
```

Un método no puede haacer un catch en una instancia genérica:

```
public static <T extends Exception, J> void ejecuta(List<J>
trabajos) {
    try {
        for(J trabajo : trabajos)
            // ...
    } catch(T e) {  /*...*/ }  // error
}
```

Sin embargo, si puedes usar un tipo genérico dentro de un **throws**:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {/*...*/ }  // OK
}
```

No puedes sobreescribir métodos con parámetros formales

Si al sobreescribir usas un genérico como parámetro y pierdes el tipo, quizás queden con la misma firma y aparece una ambiguedad que no se permite:

```
public class Ejemplo {
    public void print(Set<String> sSet) { }
```

```
    public void print(Set<Integer> iSet) { } // error
}
```

UNA CLASE GENÉRICA CON DOS PARÁMETROS DE TIPO

Puedes declarar más de un parámetro de tipo en un genérico. Para especificar dos o más parámetros de tipo, simplemente escribes una lista separada por comas. Por ejemplo, la siguiente clase DosGen es una variación de la clase Gen que tiene dos parámetros de tipo:

```
// Una clase DosGen simple con dos parámetros de tipo: T y V
class DosGen<T, V> {
   T ob1; // Declara un objeto de tipo T
   V ob2; // Declara un objeto de tipo V

   DosGen(T o1, V o2){
      ob1 = o1;
      ob2 = o2;
   }

   T getOb1(){ return ob1; }
   V getOb2(){ return ob2; }

   void mostrarTipo(){
      System.out.println( "El tipo de T es: " +
                          ob1.getClass().getName() );
      System.out.println( "El tipo de V es: " +
                          ob2.getClass().getName());
   }
}

// Demostración de clase DosGen
public class Genericos2 {
   public static void main(String[] args) {
      DosGen<Integer,String> dosGen =
               new DosGen<Integer, String>(28,"Genericos");
      dosGen.mostrarTipo();
      int v = dosGen.getOb1();
      System.out.println("Valor: " + v);
      String str = dosGen.getOb2();
      System.out.println("Valor: "+str);
   }
```

```
}
```

SINTAXIS PARA CREAR UNA CLASE GENÉRICA

```
class NombreClase<lista_tipos> { /*...*/ }
```

Sintaxis para declarar una referencia a una clase genérica y crear una instancia genérica:

```
NombreClase<lista_tipos> referencia =
        new NombreClase<lista_tipos>(lista_parámetros);
```

SINTAXIS PARA CREAR UNA INTERFACE GENÉRICA

```
interface NombreInterface<lista_tipos> { /*...*/ }
```

SINTAXIS PARA CREAR UN MÉTODO GENÉRICO

```
class C<T> {
    // Método de instancia con parámetro genérico
    [modificadores] tipo m1(T t, ...){ tipo t; /*...*/ return t; }
    // Método de instancia que devuelve un tipo genérico
    [modificadores] T m2(tipo p, ...){  T t; /*...*/ return t; }
    // método de instancia con parámetro y valor genérico
    [modificadores] T m3(T t, ...){  T t1;/*...*/ return t1; }
}
```

SINTAXIS PARA CREAR UN MÉTODO ESTÁTICO GENÉRICO

Cuando un método se hace genérico, le ocurre lo mismo que a las clases, pero en el marco del método. En el caso de métodos estáticos, al igual que le ocurre a las variables, no deberían poder ser genéricos. Pero podemos hacerlos genéricos con un truco, ocultando el tipo genérico, tanto si el genérico aparece en los parámetros formales o en el tipo devuelto, se acompaña static con <T>: **static<T>**

```
class C<T> {
   // Método estático con parámetro genérico
   [modificadores] static<T> tipo m(T t){tipo t;/*...*/return t; }
```

```
        // Método estático que devuelve un tipo genérico
        [modificadores] static<T> T m2(){T t; /*...*/ return t; }
        // método estático con parámetro y valor genérico
        [modificadores] static<T> T m3(T t){ T t1;/*...*/ return t1; }
    }
```

Decir también que una clase no genérica puede tener un método genérico.

6.2.2. LIMITAR TIPO, OPERADOR DIAMANTE Y WILCARD

En los ejemplos anteriores, los parámetros de tipo pueden cambiarse por cualquier tipo. Esto es lo ideal para muchos propósitos. Pero a veces es útil limitar los tipos que se pueden usar con uno o varios de los parámetros de tipo.

Por ejemplo, imagina que necesitas crear una clase genérica que almacene un valor numérico y que sea capaz de realizar diversas funciones matemáticas, como calcular el recíproco u obtener la parte fraccionaria, o usar una raíz cuadrada. Además, quieres utilizar la clase para calcular estas cantidades para cualquier tipo de número, incluidos integers, floats y doubles. Por lo tanto, quieres especificar el tipo números genéricamente, utilizando un parámetro de tipo. Para crear una clase así, puedes intentar algo como esto:

```
// OperaMate intenta (sin éxito) crear una clase genérica
class OperaMate<T>{
   T num;

   public OperaMate(T n){ num = n; }
   double reciproco(){ return 1 / num.doubleValue(); } //Error!

   double parteDecimal(){
       return num.doubleValue() - num.intValue(); //Error!
   }
   //...
}
```

Desafortunadamente, OperaMate no se compila porque ambos métodos generan errores en tiempo de compilación. En el caso del método `reciproco()`, utiliza `doubleValue()`, que es un método que tienen todas las clases numéricas. El problema es que el compilador no tiene manera de saber que tienes la intención de crear objetos OperaMate utilizando únicamente tipos numéricos. El mismo tipo de error ocurre dos veces en `parteDecimal()`, que necesita llamar tanto a `doubleValue()` como a `intValue()`.

Para resolver este problema, necesitas alguna manera de decirle al compilador que tenemos la intención de pasar solo tipos numéricos al parámetro T. Además, necesitas alguna forma de asegurarte de que esa intención se plasme en la realidad y solo se pasen tipos numéricos.

TIPOS LIMITADOS (bounded types)

Java proporciona tipos limitados (**bounded types**). Al especificar un parámetro de tipo, puedes crear un límite superior que declara la superclase de la cual se derivan todos los argumentos de tipo que son aceptables. Esto se logra mediante el uso de una cláusula **extends** al especificar el parámetro de tipo, como se muestra aquí:

<T extends superclase>

Esto indica que **T puede reemplazarse solo por superclase o subclases de la superclase**. Por lo tanto, la superclase define un límite superior inclusivo. La clase anterior se define con la siguiente modificación y ya funcionaría sin problemas:

```
class OperaMate<T extends Number>{ ... }
```

Compatibilidad de tipos

Los tipos limitados son especialmente útiles cuando necesitas asegurar

que un parámetro de tipo sea compatible con otro. Por ejemplo, usemos la siguiente clase llamada Pareja, que almacena dos objetos que deben ser compatibles entre sí:

```
class Pareja<T, V extends T>{ // V debe ser un@ T o subclase
   ...
```

EL OPERADOR DIAMANTE

El concepto de **Java Diamond Operator** aparece en la versión 7 de Java. Se le denomina así por la forma que tiene el operador "<>" y permite simplificar el manejo de genéricos. Muchas veces nos olvidamos de usarla ya que estamos acostumbrados a la sintaxis básica y es difícil cambiar hábitos. Vamos a ver un ejemplo sencillo de uso. Supongamos que tenemos el siguiente código en Java:

```
package od;

import java.util.ArrayList;
import java.util.List;

public class Principal {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<String>();
        lista.add("hola");
        lista.add("adios");
        lista.forEach( System.out::println );
    }
}
```

Todo funciona bien, sin embargo no es necesario especificar tantos tipos. Basta con definir el tipo de dato al declarar la referencia, no hace falta hacerlo de nuevo en el constructor.

```
package od;

import java.util.ArrayList;
import java.util.List;
```

```
public class Principal {
    public static void main(String[] args) {
        List<String> lista= new ArrayList<>();
        lista.add( "hola" );
        lista.add( "adios" );
        lista.forEach( System.out::println );
    }
}
```

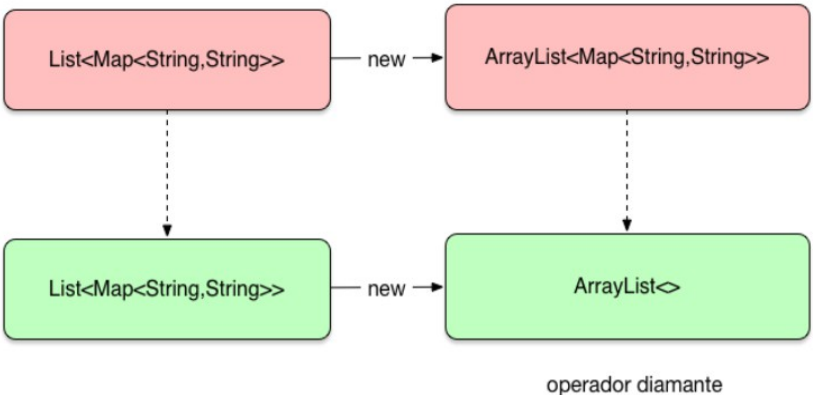
De esta forma se simplifica el manejo de genéricos. Si te parece que las ventajas de usarlo no son excesivas, vamos a ver otro ejemplo más complejo:

```
package od;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Principal2 {
   public static void main( String[] args ) {
       List< Map<String, String> > lista =
          new ArrayList<Map<String,String>>();
       Map<String,String> mapa = new HashMap<String,String>();
       mapa.put("clave1", "valor1");
       lista.add(mapa);
       lista.forEach(System.out::println);
    }
}
```

En este caso tenemos una lista donde cada elemento es un mapa y todo esta lleno de anotaciones genéricas. En este ejemplo el operador diamante nos puede ayudar bastante. El código:

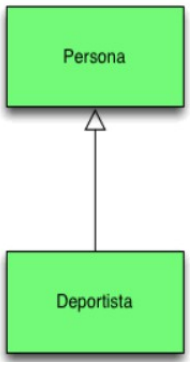
```
package od;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
public class Principal3 {
   public static void main(String[] args) {
      List<Map<String, String>> lista = new ArrayList<>();
      Map<String,String> mapa = new HashMap<>();
      mapa.put("clave1", "valor1");
      lista.add(mapa);
      lista.forEach( System.out::println );
    }
}
```



Si te acostumbras a usarlo harás más sencillo tu código genérico.

EL OPERADOR WILDCARD ILIMITADO



El operador wildcard se escribe con el carácter '?'. Vamos a suponer un ejemplo sencillo en el que tenemos dos clases. La clase Persona y la clase Deportista. El código de Persona:

```
package ow;

public class Persona {
    private String nombre;
    public String getNombre(){ return nombre; }
    public void setNombre(String n){ this.nombre= n; }
    public Persona(String nombre){ this.nombre= nombre; }
```

```
}
```

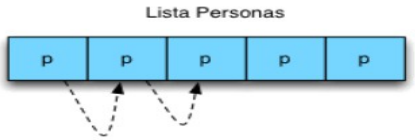
El código de Deportista:

```
public class Deportista extends Persona {
    private String deporte;
    public Deportista(String nombre, String deporte){
        super(nombre);
        this.deporte= deporte;
    }
    public String getDeporte(){ return deporte; }
    public void setDeporte(String d){ deporte = d; }
}
```

Podemos crearnos una lista de Personas que tenemos que recorrer e imprimir por pantalla. El programa Java sería así de sencillo:

```
package ow;
import java.util.ArrayList;
import java.util.List;

public class Principal {
    public static void main(String[] args) {
        List<Persona> lP = new ArrayList<>();
        lP.add( new Persona("Pepe") );
        lP.add( new Persona("Maria"));
        imprimir(lP);
    }

    public static void imprimir( List<Persona> lista ) {
        for(Persona p: lista) {
            System.out.println( p.getNombre() );
        }
    }
}
```

El programa simplemente construye una lista de Personas la rellena y luego la recorre.



Ahora bien, que pasaría si realizamos una modificación a la lista para que sea de Deportistas. El código quedaría de la siguiente forma:

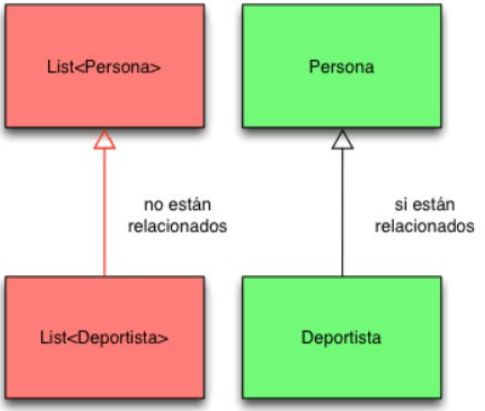
```
package ow;

import java.util.ArrayList;
import java.util.List;

public class Principal {
    public static void main(String[] args) {
        List<Deportista> lP= new ArrayList<>();
        lP.add( new Deportista("Pepe", "futbol") );
        lP.add( new Deportista("Maria", "tenis") );
        imprimir(lP);
    }

    public static void imprimir( List<Persona> lista ) {
        for( Persona p:lista ) {
            System.out.println( p.getNombre() );
        }
    }
}
```

Lamentablemente este código no compila y da el siguiente error:

```
The method imprimir(List<Persona>) in the type Principal is not
applicable for the arguments (List<Deportista>)
```

Parece extraño ya que estamos pasando una lista de Deportista y un Deportista también ES UNA Persona. Sin embargo tenemos que darnos cuenta de lo siguiente: una lista de Personas y una lista de Deportistas son dos tipos diferentes y no están relacionados. Aunque los objetos que ellos almacenen si lo estén.

Al tratarse de dos tipos de objetos que realmente no están relacionados por herencia no podemos apoyarnos en el polimorfismo para ejecutar el mismo código ya sea List<Persona> o List<Deportista> lo que llega de parámetro. Vamos a ver una solución usando wildcard.



El carácter Wildcard (?) nos sirve para decirle a Java que cuando usemos un tipo genérico se puede aplicar cualquier tipo al parámetro lista vamos a verlo:

```
public static void imprimir( List<?> lista ) {
    for(Persona p: lista) {
        System.out.println( p.getNombre() );
    }
}
```

De esta forma añadimos flexibilidad ya que podemos pasar cualquier tipo como parámetro genérico. Sin embargo tenemos todavia un problema al recorrer la lista de Personas estamos obligando a que la clase sea de tipo Persona cosa que el caracter Wildcard no obliga. Así pues seguiremos con problemas de compilación, para evitar esto podemos añadir al Wildcard una restricción de límite.

```
public static void imprimir( List<? extends Persona> lista){
    for(Persona p: lista) {
        System.out.println(p.getNombre());
    }
 }
```

Ahora sí podemos trabajar tranquilos y aprovechar el polimorfismo.

LIMITAR EL OPERADOR WILDCARD POR ABAJO

Además de poder limitar por arriba el tipo del parámetro genérico ilimitado, en el caso de usar el operador wildcard, también le podemos poner límite por debajo (como mínimo debe ser superclase de esta clase que indico) y usas la palabra **super**:

List<? super Deportista>

EL TIPO RAW

Cuando un genérico se declara sin indicar el tipo genérico, se usa por defecto Object y se le llama tipo Raw. Esto genera un warning. Siempre es mejor usar el tipo parámetro salvo que vayas a mezclar objetos de tipos de familas de clases diferentes. Por ejemplo:

```
public class Caja<T>{
    T dato;

    public static void main(String[] args){
        Caja c = new Caja();  // c es de tipo raw == Object
        c.dato = "Hola";
        c.dato = 38;
        c.dato = new Date();
    }
}
```

6.3. ITERADORES Y COMPARADORES.

6.3.1. ITERADORES.

Los iteradores se usan en Java para recorrer los elementos de las colecciones uno a uno. Java tiene 3 formas de iterar.

INTERFACE ENUMERATION<E>

Enumeration es una interfaz usada para obtener elementos de las

colecciones **Vector** y **Hashtable**. También se usan para indicar streams de entrada a un **SequenceInputStream**. Podemos crear un objeto **Enumeration** llamando al método **elements()** de una clase. Ejemplo:

```
// "v" es un objeto de la clase Vector y e es de tipo Enumeration
Enumeration e = v.elements();
```

La interfaz declara dos métodos a implementar:

```
// comprobar si la colección tiene más elementos
public boolean hasMoreElements();
// Obtener siguiente elemento, si no hay NoSuchElementException
public Object nextElement();
```

EJEMPLO 2: Programa de ejemplo del uso de Enumerator<E>:

```
import java.util.Enumeration;
import java.util.Vector;

public class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        for(int i = 0; i < 10; i++) v.addElement(i);
        System.out.println(v);
        // Al principio e apunta al primer elemento
        Enumeration e = v.elements();
        // Comprobar que hay otro elemento más
        while( e.hasMoreElements() ) {
           int i = (Integer)e.nextElement();
           System.out.print(i + " ");
        }
    }
}
```

Limitaciones de Enumeration:

- Solo se puede utilizar en las clases **Vector** y **Hashtable**.
- No puedes borrar elementos usando **Enumeration**.
- Solo puedes recorrer los elementos en una dirección.

INTERFACE ITERATOR<E>

Puede utilizarse con cualquier colección. Puedes eliminar elementos de la colección sin problemas. Se crean objetos **Iterator** llamando al método **iterator()** de la interfaz **Collection**.

```
// Si "c" es cualquier colección. itr es un iterator de "c"
Iterator itr = c.iterator();
```

Nota: ten en cuenta que es una interfaz, por tanto no puedes crear objetos suyos directamente (con el operador **new**) porque obtendrás un error:

```
Iterator it = new Iterator();  // Error, una interface!!
```

La interface Iterator define 3 métodos:

```
public boolean hasNext(); // true si hay más elementos
public Object next();     // siguiente / NoSuchElementException
public void remove();     // Borra el siguiente elemento
```

El método **remove()** puede lanzar dos excepciones:

- **UnsupportedOperationException**: si la operación no la soporta.
- **IllegalStateException**: si no se ha llamado al método **next()** o se ya se ha eliminado el elemento antes.

EJEMPLO 3: uso de un Iterator.

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test1 {
    public static void main(String[] args)  {
       ArrayList aL = new ArrayList();
       for(int i = 0; i < 10; i++) aL.add(i);
       System.out.println(aL);
       // Al principio itr no apunta al primer elemento
       // Hay que hacer un next() para apuntar al primero
       Iterator itr = aL.iterator();
       // comprobar si hay siguiente
       while( itr.hasNext() ) {
          int i = (Integer)itr.next();
```

```
            System.out.print(i + " ");
            if( i % 2 != 0 ) itr.remove(); // elimina impares
        }
        System.out.println("\n" + aL);
    }
}
```

Limitaciones de Iterator:

- Solo se recorren en una dirección.
- Cambiar y añadir elementos no está permitido.

INTERFACE LISTITERATOR<E>

Las listas son colecciones que pueden tener iteradores bidireccionales (desde el principio hasta el final y viceversa) mientras que en otras colecciones esto no tienen sentido. El iterador `ListIterator` permite recorrer los elementos de una lista en ambas direcciones. La interface List obliga a las listas a implementar el método `listIterator()` que crea un objeto de este tipo.

```
// Creación de un ListIterator en la lista lista
ListIterator li = lista.listIterator();
```

La interface `ListIterator` extiende a la interface `Iterator` y obliga a implementar estos 3 métodos además de otros:

```
// Dirección de principio a fin
public boolean hasNext();
public Object next();
public int nextIndex();

// Dirección de final al principio
public boolean hasPrevious();
public Object previous();
public int previousIndex();

// Otros
public void remove();
public void set(Object obj);
```

```
public void add(Object obj);
```

El método `set()` puede devolver excepciones (`UnsupportedOperationException`, `ClassCastException`, `IllegalArgumentException` y `IllegalStateException`) y el método `add()` también (`UnsupportedOperationException`, `ClassCastException` y `IllegalArgumentException`).

EJEMPLO 4: Uso de ListIterator.

```
import java.util.ArrayList;
import java.util.ListIterator;

public class TestListIterator {
    public static void main(String[] args) {
        ArrayList aL = new ArrayList();
        for(int i = 0; i < 10; i++) aL.add(i);
        System.out.println(aL);
        ListIterator li = aL.listIterator();
        while( li.hasNext() ) {
            int i = (Integer)li.next();
            System.out.print(i + " ");
            if( i % 2 == 0 ) {
                i++;
                li.set(i);
                li.add(i);
            }
        }
        System.out.println();
        System.out.println(aL);
    }
}
```

Limitaciones de ListIterator:

- Solamente se aplica a clases que implementen List.

SPLITERATOR<T>

JDK 8 añadió este nuevo tipo de iterator definido por la interface Spliterator. Es parecido a los iterator anteriores pero ofrece más

funcionalidad. Por ejemplo soporta recorrer porciones de la colección de manera paralela, pero además ofrece un efoque de trabajo en streaming donde combina las operaciones `hasNext()` y `next()` en un solo método. Tiene estos métodos:

- `int characteristics()`: devuelve un código con las características del iterator.
- `long estimateSize()`: estima el número de elementos que quedan por recorrer y si no puede estimarlo devuelve `Long.MAX_VALUE`.
- `default void forEachRemaining( Consumer<? Super T> accion)`: aplica una acción a cada elemento que le queda por recorrer.
- `default Comparator<? Super T>getComparator()`: devuelve el comparador usado para invocar al iterator o null si no se usa ninguno o `IllegalStateException` si la secuencia no está ordenada.
- `default long getExactSizeKnown()`: si tiene un tamño definido, indica la cantidad de elementos que le quedan por recorrer y -1 en otro caso.
- `Default boolean hasCharacteristics(int valor)`: devuelve true si cuenta con la características pasadas en el entero.
- `boolean tryAdvance( Consumer<? Super T> accion)`: ejecuta la acción sobre el siguiente elemento y devuelve true tras aplicarla, o false si no hay siguiente elemento.
- `SplitIterator<T> trySplit()`: si es posible, devuelve una referencia a una nueva instancia para un trozo de la colección y en otro caso devuelve `null`.

Usarlo en iteraciones básicas es bastante sencillo: llamas a `tryAdvance()` hasta que obtienes `false`. Si aplicas la misma acción a todos los elementos la alternativa es `forEachRemaining()`. Consumer es una interface funcional genérica de `java.util.function` (que especifica el método abstractos accept( )

EJEMPLO 5: Uso de Splitterator.

```
import java.util.ArrayList;
import java.util.Spliterator;

public class TestSplitIterator {
    public static void main(String[] args){
        ArrayList<Double> ald = new ArrayList<>();
        for(int i = 0; i < 1_000; i++)
            ald.add(Math.random() * 1_000);
        Spliterator<Double> sd = ald.spliterator();
        while(sd.tryAdvance( n -> System.out.println(n) ) );
        // Crear una lista con los cuadrados
        sd = ald.spliterator();
        ArrayList<Double> alc = new ArrayList<>();
        while(sd.tryAdvance( n -> alc.add( n * n) ) );
        // Mostrarla con forEachRemaining()
        sd = alc.spliterator();
        sd.forEachRemaining( (d) -> System.out.println(d) );
    }
}
```

BUCLE FOR EXTENDIDO (FOR-EACH)

El bucle de tipo **for-each** es otra forma de recorrer elementos a partir de Java5.

- Se usa la palabra clave **for**.
- En vez de declarar e inicializar una variable contador, usas una variable del mismo tipo que los elementos de la colección, seguida por dos puntos y el nombre de la referencia al objeto con el que se trabaja en cada iteración del bucle.
- En el cuerpo del bucle se usa el elemento que va cambiando en cada iteración, ya no tienes un índice (si la colección es un array), tienes un elemento.

Sintaxis:

for (**tipo** variable : colección) { sentencias que usan variable; }

EJEMPLO 6: Uso de for extendido.

```
class ForEach  {
    public static void main(String[] arg) {
        int[] marcas = { 125, 132, 95, 116, 110 };
        int peor = maximo(marcas);
        System.out.println("La peor marca " + peor);
    }

    public static int maximo( int[] numeros ) {
        int max = Integer.MIN_VALUE;
        for(int n : numeros ) {
            if (n > max) {
                max = n;
            }
       }
       return max;
   }
}
```

Limitaciones del bucle for-each:

- No sirve para modificar los elementos de la colección:

```
for( int n : marcas ) {
    n = n * 2;   // solo cambias la variable, no el elemento
}
```

- No puedes averiguar el índice:

```
for(int n : numeros) {
    if (n == buscado) {
        return ???;   // no sabes la posición del índice
    }
}
```

6.3.2 IGUALDAD Y COMPARADORES.

IGUALDAD DE OBJETOS

Ya sabemos que utilizar el operador == para comparar objetos no da el resultado esperado (o1 == o2 comprueba cuando son el mismo objeto, comparten la misma memoria física) y normalmente nosotros necesitaremos saber si contienen la misma información aunque esté almacenada en distintos lugares. Por ejemplo, para nosotros dos variables que referencien fechas (objetos `Date`), son iguales cuando describen el mismo momento de tiempo, mientras que para Java solo son iguales con == si referencian al mismo objeto.

En la clase `Object` se define el método `equals(Object)` donde se implementa el código que hace nuestra versión de la igualdad entre los objetos obj1 y obj2. Así que deberíamos sobreescribir este método en nuestras clases si pensamos almacenar sus objetos en colecciones.

EJEMPLO 7: Definir el método para poder saber si dos cartas de una baraja son iguales.

```
public class Carta {
   private int palo;   // espadas, copas, oros, bastos
   private int valor;  // Nº de 1 a 11

   @Override
   public boolean equals(Object obj) {
      try {
         Carta otra = (Carta)obj; // cast de tipos
         if (palo == otra.palo && valor == otra.valor)
            return true;
         else
            return false;
      }
      catch (Exception e) {
          // NullPointerException si obj es null
          // ClassCastException si tipo de obj no es Carta
          return false;
      }
   }
   // resto de la clase...
}
```

Sin `equals()` en la clase T, los métodos **`contains()`** y **`remove()`** de `Collection<T>` no funcionan bien.

Según la especificación del método equals definido en la clase *`Object`*, debe tener las siguientes propiedades:

- **Reflexiva:** `x.equals(x)` es true.
- **Simétrica**: `x.equals(y)` es true si `y.equals(x)` lo es.
- **Transitiva**: si `x.equals(y)` es true y `y.equals(z)` es true, entonces `x.equals(z)` debe ser true.
- **Consistente**: varias llamadas a `x.equals(y)` siempre son true o false si ni x ni y cambian.
- **`x.equals(null)`** siempre es `false`.

EL MÉTODO HASHCODE()

Este método complementa a `equals()` y sirve para comparar objetos **de una forma más rápida en estructuras Hash** ya que únicamente nos devuelve un número entero. **Cuando Java compara 2 objetos en estructuras de tipo hash (`HashMap`, `HashSet`, etc.) primero llama al método `hashCode()` y luego a `equals()`**. Si los métodos `hashCode()` de cada objeto devuelven diferente hash, no sigue comparando y considera a los objetos distintos. En el caso en que ambos objetos compartan el mismo hashcode, **Java llama a `equals()` y revisa en detalle si se cumple la igualdad. De esta forma las búsquedas se aceleran en estructuras hash.**

Por lo tanto si queremos sobreescribir correctamente los métodos equals() y hashcode() **debemos asegurar que cuando dos objetos sean iguales devuelvan el mismo hashcode**. Aunque **dos objetos diferentes pueden tener el mismo hashcode, eso no constradice lo anterior.**

El problema de un programador es como implementar el `hashCode()` de su clase, ya que el framework de colecciones lo usa para diseñar **las estructuras internas de sus tipos abstractos de datos**. En el siguiente ejemplo se muestra la clase `Persona` con el método `equals()` sobreescrito **para que dos objetos sean iguales si su nombre y apellidos coinciden**. El método es sencillo de entender.

```
package comp;

public class Persona {
    private String nombre;
    private String apellidos;

    public String getNombre() { return nombre; }
    public void setNombre(String nombre){ this.nombre = nombre; }
    public String getApellidos() { return apellidos; }
    public void setApellidos(String apes){ this.apellidos= apes; }

    @Override
    public boolean equals(Object obj) {
        if( this == obj) return true;
        if( obj == null) return false;
        if( getClass() != obj.getClass() ) return false;
        Persona otra = (Persona) obj;
        if(apellidos == null) {
            if(otra.apellidos != null) return false;
        }
        else if( !apellidos.equals(otra.apellidos)) return false;
        if(nombre == null) {
            if( otra.nombre != null) return false;
        }
        else if(!nombre.equals(otra.nombre)) return false;
        return true;
    }
} // fin clase Persona
```

El método anterior ha sido generado con las utilidades de Eclipse. De igual forma se puede generar el hashCode:

```
public int hashCode() {
    final int primo = 31;
```

```
        int resultado = 1;
        resultado = primo * resultado +
                    ((apellidos == null) ? 0 : apellidos.hashCode());
        resultado = primo * resultado +
                    ((nombre == null) ? 0 : nombre.hashCode());
        return resultado;
    }
```

Este método ya no es tan sencillo de entender. La lógica que hay detrás de este código es la de hacer grupos de objetos: por tanto los hashCodes deben tener una cierta variabilidad. Es decir, si todos los objetos generan el mismo `hashcode`, no ayuda a repartirlos en difernetes grupos, si todos generan el mismo habrá que utilizar siempre el método `equals()` a la hora de compararlos porque la mayoría estarán en el mismo grupo.

Por otro lado, si todos los objetos generan hashCodes diferentes, será imposible agruparlos **de una forma óptima** porque habrá casi tantos grupos como objetos y tampoco aumentarán el rendimiento.

De ahí que use este misterioso algoritmo. La implementación del **método hashCode** se debe realizar según los siguientes pasos:

- Almacenar un valor no 0 en una variable int, por ejemplo 1.
- Por cada campo f (variable de instancia) usado en el método `equals()` de la clase, debes obtener un hash code (int):
    - Si el campo f es un boolean: `(f ? 1 : 0)`
    - Si f es byte, char, short o int: `(int)f`
    - Si f es long: `(int) (f ^ (f >>> 32))`
    - Si f es float: `Float.floatToIntBits(f)`
    - Si f es double: `Double.doubleToLongBits(f)` y calcular el hash del long.
    - Si f es una referencia a un objeto y el método equals de esta clase compara recursivamente invocando el método

equals del campo, invocar su método `hashCode()`. Si el valor de campo es nulo, 0.

- Si f es un array, se aplican estas reglas a cada elemento. Si cada elemento del array es significativo se puede usar `Arrays.hashCode()`.
- Combinar los hash obtenidos como: `resultado = 31 * resultado + c`.

Existe **otra opción sencilla para generar la implementación** que es utilizar el API de Java que dispone en el paquete de utilidades métodos para generar hashCodes.

```
import java.util.Objects;

@Override
public int hashCode() {
    // Observa que como en equals() se usa nombre y apellidos
    // Son los que usamos en hash
    return Objects.hash(nombre, apellidos);
}
```

De esta forma nos olvidamos de las complicaciones de este método. En cualquier caso, si por cualquier motivo quieres encargarte tu mismo de la implementación ten en cuenta que: **Si dos objetos son iguales, deben tener el mismo `hashCode()` (van al mismo grupo) y debería poder calcularse de forma más rápida que saber si son iguales.**

COMPARAR OBJETOS CON COMPARABLE<T>

Si necesitas establecer un orden en los objetos de una clase (saber si uno es mayor, menor o igual que otro), puedes hacer que la clase implemente la interface Comprable<T> donde T es la clase de los Objectos. Un orden es necesario si necesitas ordenar, buscar o recorrer los objetos en una colección, para hacer esas tareas necesitas indicar cuando se considera que un objeto es mayor que otro, menor o

igual.

La interface `java.lang.Comparable<T>` define un único método:

```
public int compareTo( T obj );
```

El valor devuelto por una llamada al método `obj1.compareTo(obj2)` debe ser:

- **negativo** cuando obj1 sea menor que obj2 (obj1 < obj2)
- 0 si son iguales (obj1.equals(obj2))
- **positivo** si obj1 es mayor que obj2 (obj1 > obj2).

EJEMPLO 8: Crear una clase que implemente la interface Comparable.

```
public class NombreComp implements Comparable<NombreComp> {
   private String nombre, ape1, ape2;

   public NombreComp(String n, String a1, String a2) {
      if (n == null || a1 == null || a2 == null)
         throw new IllegalArgumentException("Datos forzosos");
      nombre = n;
      ape1 = a1;
      ape2 = a2
   }
   public boolean equals(Object obj) {
      try {
         NombreComp nc = (NombreComp)obj;
         return nombre.equals(nc.nombre)  &&
                ape1.equeals(nc.ape1) &&
                ape2.equals(nc.ape2);
      }
      catch (Exception e) {
         return false;
      }
   }

   public int compareTo( NombreComp nc ) {
      String temp = nc.ape1 + nc.ape2 + nc.nombre;
      String este = ape1 + ape2 + nombre;
      return este.compareTo(temp);
   }
```

```
}
```

COMPARAR OBJETOS CON COMPARATOR<T>

En vez de que cada clase tenga ya en su interior un orden prefijado al implementar `Comparable<T>`, también puedes crear un objeto independiente que lo haga, en vez de que la clase tenga su método o al margen de que lo tenga. Es decir, una clase puede tener una manera estándar de comparar sus objetos, pero si en determinado momento te interesa compararlos de otra manera, en vez de cambiar la clase, usas un objeto `Comparator<T>`.

El objeto que se use para aplicar un orden a los objetos T (quizás distinto al que usan por defecto) debe implementar la interface `Comparator<T>`, donde T es el tipo de objetos a comparar. La clase comparadora debe implementar el método:

```
public int compare( T obj1, T obj2 );
```

Lo que devuelve es un valor negativo, cero o positivo, con la misma lógica que se usa en `compareTo()`. Los comparadores se usan mucho para no tener que implementar la interface `Comparable<T>` y para definir varias formas de comparar objetos, lo que permite ordenarlos en las colecciones por criterios distintos.

EJEMPLO 9: uso de Comparator.

```
package e1;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class JugadorFutbol implements Comparable<JugadorFutbol> {
    private String nombre;
    private int goles;
    private double ficha;
```

```
    public JugadorFutbol(String n, int g, double f) {
        nombre = n;
        golesMarcados = g;
        ficha = f;
    }

    public String getNombre() {  return nombre; }
    public int getGoles() {  return goles; }
    public double getFicha() {  return ficha; }

    // Implementar Comparable<T>
    public int compareTo(JugadorFutbol f) {
        if(golesMarcados > f.golesMarcados) return 1;
        if(golesMarcados < f.golesMarcados) return -1;
        return 0;
    }

    public String toString() {
        return String.format( "%s %d goles y %.2f€",
                              nombre, goles, ficha);
    }
}

public class Ej {
   public static void main(String[] args) {
      JugadorFutbol mes = new JugadorFutbol("Mesi", 20, 1000);
      JugadorFutbol cr7 = new JugadorFutbol("Cristiano", 17, 800);
      // Podrían estar todos los jugadores del mundo...
      ArrayList<JugadorFutbol> lj = new ArrayList<>();
      lj.add(mes);
      lj.add(cr7);
      // Ordenadar usando comparable (por goles)
      Collections.sort(lj); // Usando Comparable
      System.out.println( "Jugadores ordenados por goles" );
      for(JugadorFutbol f: lj) System.out.println( f );
      // Ordenadar y mostrar usando ficha
      System.out.println( "Jugadores ordenados por ficha" );
      lj.sort( // Usando Comparator<T>
         new Comparator<JugadorFutbol>() {
           public int compare(JugadorFutbol j1, JugadorFutbol j2){
             if(j1.getFicha() > j2.getFicha() ) return 1;
             if(j1.getFicha() < j2.getFicha() ) return -1;
```

```
                return 0;
            }
        }
    );
    for(JugadorFutbol f: lj) System.out.println( f );
  }
}
```

6.4. COLECCIONES PRECONSTRUIDAS EN JAVA.

EL FRAMEWORK DE COLECCIONES DE JAVA

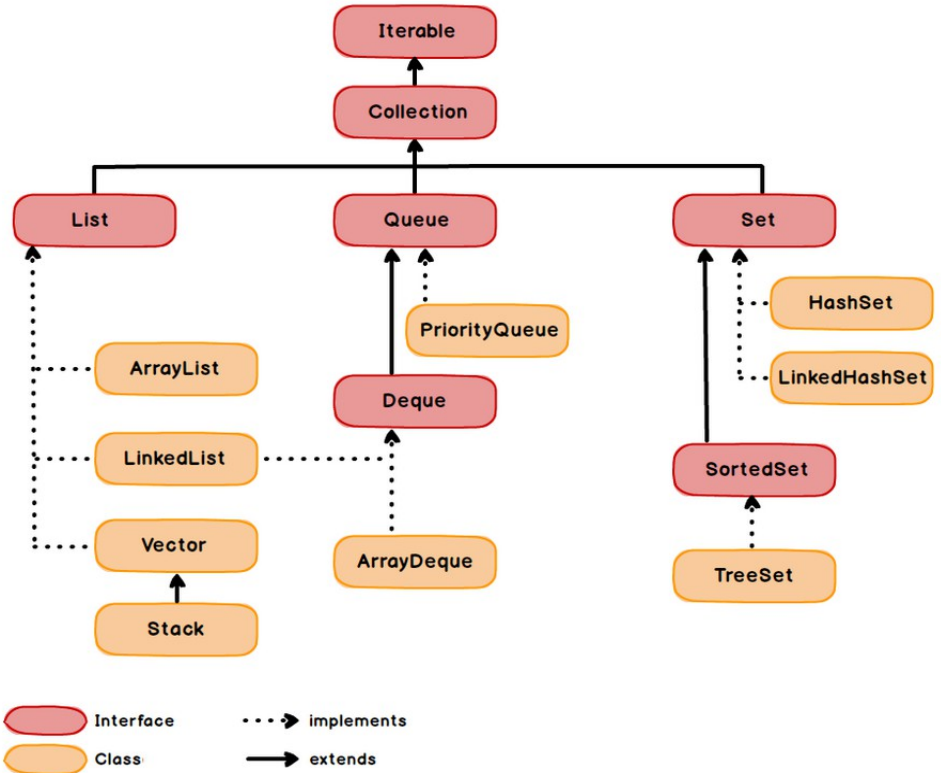
Las interfaces y clases preconstruidas de Java y parametrizadas que sirven como estructuras de datos preconstruidas se denominan **JFC**. Se dividen en dos grupos: colecciones y mapas. Las colecciones son almacenes de datos organizados y los mapas asocian objetos entre si de la misma forma que un diccionario asocia palabras con sus definiciones (se les conoce también como arrays asociativos o diccionarios en otros lenguajes de programación). En Java, colecciones y mapas se representan con las interfaces parametrizadas `Collection<T>` y `Map<T,S>`.

Hay principalmente dos **tipos de colecciones: listas (lists)** y **conjuntos (sets)**.

- `List<T>`: una lista es una colección en la que sus elementos se organizan en una secuencia lineal, tienen un elemento primero, segundo, etc. y todos los elementos menos el último tienen un elemento siguiente.
- `Set<T>`: tienen la propiedad de que no admiten elementos repetidos (como los conjuntos matemáticos) y no tienen porqué seguir ningún orden.

Las interfaces parametrizadas `List<T>` y `Set<T>` las definen y son

subinterfaces de `Collection<T>`. En la figura se ve la estructura de Collection:



INTERFACE COLLECTION<T>

La interface **`Collection<T>`** define estos métodos y obliga a que un objeto c de una clase que la implemente los ofrezca:

- `c.size()` — un **int** que indica el número de elementos que tiene.
- `c.isEmpty()` — devuelve true si no tiene elementos.
- `c.clear()` — quita todos los elementos.

- `c.add(object)` — añade el objeto a la colección. Devuelve true si la operación tiene éxito (si intentas añadir un repetido a un conjunto, devuelve false porque no puede añadirlo).
- `c.contains(object)` — devuelve true si c contiene el objeto (deben poder compararse los elementos).
- `c.remove(object)` — elimina el objeto de c. Si tiene éxito devuelve true, si falla devuelve false (necesita comparar los elementos).
- `c.containsAll(c2)` — devuelve true si c contiene todos los elementos de la colección c2. Es como la operación c contiene a c2.
- `c.addAll(c2)` — añade todos los objetos de c2 a c. Es como la operación unión de c2 a c.
- `c.removeAll(c2)` — elimina de c todos los elementos de c2, es como restar a c, c2. Lo que se denomina diferencia.
- `c.retainAll(c2)` — elimina todos los elementos de c que no estén en c2, es como la intersección de c y c2.
- `c.toArray()` — devuelve un array de tipo Object[ ].
- `c.toArray(tipo)` — devuelve un array de tipo T[ ].

Según como se implementen las clases y el tipo de colección, aunque el resultado sea el mismo, estas operaciones pueden variar mucho en eficiencia (unas implementaciones harán algunas muy rápido y otras implementaciones necesitarán consumir muchos recursos (sentencias / tiempo o memoria) para realizar la misma operación, y sin embargo puede que sea más eficiente en otras.

ITERATOR Y BUCLES FOR-EACH

La interfaz `Collection<T>` define un método para crear un iterador que permita recorrer sus elementos independientemente del tipo de dato que contenga llamando al método `c.iterator()` que devuelve un objeto

de una clase que implemente la interfaz parametrizada `Iterator<T>`. Ya hemos comentado que esta interfaz define 3 métodos: `iter.next()`, `iter.hasNext()` y `iter.remove()`.

También puede utilizarse un bucle for-each para iterar en una colección `Collection<T>` o sus descendientes:

```
for( T x : coleccion ) {
    // procesar x
}
```

6.4.1 CLASES ARRAYLIST, LINKEDLIST Y VECTOR.

Todas estas clases implementan la interfaz `List<T>` y `Collection<T>`. Hay dos formas de crear estructuras o colecciones de tipo lista:

- **Una es usando arrays como base**: pero el código se encarga de hacerlos dinámicos para que puedan aumentar de tamaño si es necesario.
- **Otra es usando referencias**: en este caso cada nodo de la lista tendrá una referencia al siguiente nodo (listas enlazadas) y opcionalmente también al anterior nodo (listas doblemente enlazadas).

Las clases de tipo lista que implementan `List<T>` son: `java.util.ArrayList`, `java.util.LinkedList` y `java.util.Vector`.

Un objeto de tipo `ArrayList<T>` representa una secuencia ordenada de objetos de tipo T, almacenados en un array que puede crecer de tamaño si es necesario al añadir un nuevo elemento.

Un objeto de tipo `LinkedList<T>` representa lo mismo pero se implementa usando nodos enlazados con referencias.

¿Porqué varias implementaciones para la misma colección? El

funcionamiento a nivel lógico de todas es igual, pero según la implementación que elijas, unas operaciones tardan más que otras. Cuando necesites que uno de tus programas use listas, debes pensar qué operaciones son las que más va a hacer, o en cuales necesitas más rapidez y eso te ayuda a elegir la que debes usar. Por ejemplo, **LinkedList** es más eficiente cuando se añaden o eliminan elementos no solo por el final de la lista sino al principio o en medio de ella: en **ArrayList** orden Θ(n) (el esfuerzo que hay que hacer depende de n que representa cuantos elementos tiene la lista, a más elementos, más esfuerzo). Pero en **LinkedList** el orden de las inserciones y borrados es Θ(1), es decir, da igual lo grande que sea la lista, el esfuerzo de realizar la operación es constante, no depende de la cantidad de elementos que tenga ni de donde se produzca la inserción o el borrado.

Por otro lado, `ArrayList` es más eficiente si se necesita acceder a los elementos de forma aleatoria (no secuencial): en ArrayList Θ(1) y en LinkedList si queremos acceder al elemento de la posición k, el orden es Θ(k).

Las operaciones que define la interface `List<T>` son:

- `lista.get(i)` — devuelve elemento de la posición i. Si indicas un i mayor de `lista.size()-1` tienes `IndexOutOfBoundsException`.
- `lista.set(i, obj)` — almacena obj en la posición i, reemplazando el objeto anterior de esa posición.
- `lista.add(i, obj)` — inserta obj en la posición i, la lista incrementa su tamaño en 1 y los elementos anteriores se desplazan para hacer hueco al nuevo. El valor de i debe estar entre 0 y `lista.size()`.
- `lista.remove(i)` — elimina el objeto de la posición i. El hueco que deja se rellena desplazando el resto de elementos. El tamaño de

la lista decrece. El valor de i debe estar entre 0 y `list.size()-1`

- `lista.indexOf(obj)` — devuelve un int con la posición de obj en la lista, o -1 si no está. Si hay varios (repetido) da el primero.

CLASE ARRAYLIST<T>

Aunque la capacidad de un objeto ArrayList se incrementa automáticamente cuando almacenas un objeto, también puedes incrementar manualmente su capacidad llamando al método `ensureCapacity(int nuevaCapacidad)`. Esto puede ser interesante si sabes con anterioridad la cantidad aproximada de elementos que vas a añadir. Incrementas una vez y evitas que cada cierto número de inserciones se deba reajustar el tamaño (que es menos eficiente porque hay que realojar el array interno que almacena los objetos, copiarlos, etc.)

De la misma manera, puedes reducir la capacidad del ArrayList si sabes que no va a crecer más. Si quieres que tenga la capacidad de los elementos que almacena ahora mismo, puedes llamar a `trimToSize()`.

A veces te puede interesar obtener un array a partir del ArrayList. En ese caso puedes llamar a toArray(). En ese caso tienes dos versiones sobrecargadas:

- `Object[] toArray()`: devuelve un array de Object.
- `<T> T[] toArray(T array[])`: devuelve un array de elementos que tienen el mismo tipo que T. Esta será la versión que normalmente utilizarás.

EJEMPLO 9: Uso de la clase ArrayList<E>

```
import java.util.ArrayList;

public class TestArrayList {
    public static void main(String[] args){
        String[] p = "En un lugar de la macha de cuyo".split(" ");
```

```
            ArrayList<String> als = new ArrayList<>();
            System.out.println("Tamaño inicial: " + als.size() );
            for(String s : p)   // Añadimos elementos
                  als.add(s);
            System.out.println("Tamaño final: " + als.size() );
            System.out.println("Contenido: " + als );
            als.remove("lugar");
            System.out.println("Contenido tras eliminar \"lugar\" y "
                              + als.remove(3) + ": " + als);
    }
}
```

EJEMPLO 10: Convertir una Lista en un array.

```
import java.util.ArrayList;

public class ArrayList2Array {
    public static void main(String[] args){
        ArrayList<Integer> ali = new ArrayList<>();
        ali.add(1); ali.add(2); ali.add(3);
        System.out.println("Contenido de la lista: " + ali);
        Integer[] ai1 = new Integer[ali.size()];
        ai1 = ali.toArray(ai1);
        int suma = 0;
        for(int i : ai1) suma += i;
        System.out.println("La suma es " + suma);
    }
}
```

EJERCICIO 2: Haz un método llamado `indiceDe()` que tenga de parámetros: un ArrayList y un posible elemento suyo y devuelva la posición que ocupa en la lista, o -1 si no se encuentra.

b) Luego mira a ver si la colección tiene ya uno parecido.

CLASE LINKEDLIST<T>

La clase `LinkedList<T>` añade algunos métodos adicionales:

- `getFirst()` — el primer elemento de la lista. Si está vacía se lanza `NoSuchElementException` (igual en los siguientes 3).
- `getLast()` — devuelve el último elemento de la lista.

- `removeFirst()` — elimina el primero de la lista y lo devuelve. Hacen lo mismo las funciones `remove()` y `pop()`.
- `removeLast()` — elimina el último de la lista y lo devuelve.
- `addFirst(obj)` — añade obj al principio. Lo mismo que `push(obj)`.
- `addLast(obj)` — añade obj al final.

Nota: *si solo usas para eliminar/insertar elementos los métodos `pop()` y `push()`, estás usando la lista como si fuese una pila (lista LIFO) y si solo usas `add()` y `remove()` la usas como una cola (lista FIFO).*

EJEMPLO 11: Ejemplo de uso de la clase LinkedList<T>

```
import java.util.LinkedList;

public class TestLinkedList {
    public static void main(String[] args){
        LinkedList<String> lls = new LinkedList<>();
        lls.add("FITO");
        lls.add("Y");
        lls.add("LOS");
        lls.add("FITIPALDIS");
        lls.addLast("PINK");
        lls.add(1, "FLOYD");
        System.out.println("Contenido de la lista: " + lls);
        System.out.println("ELimino dos: ");
        lls.remove("Y LOS");
        lls.remove(2);
        System.out.println("Contenido de la lista: " + lls);
        System.out.println("Borro primero y último);
        lls.removeFirst();
        lls.removeLast();
        System.out.println("Contenido de la lista: " + lls);
        // Obtener un cambiar un valor
        String valor = lls.get(2);
        lls.set(2, valor + " cambiado");
        System.out.println("Contenido de la lista: " + lls);
    }
}
```

Además tiene el método `iterator()`, pero para listas hay un iterador especial `ListIterator<T>`, que extiende a `Iterator<T>` y que permite además de `hasNext()`, `next()` y `remove()` los métodos `hasPrevious()`, `previous()`, `add(obj)` y `set(obj)`.

EJEMPLO 12: Queremos una lista ordenada siempre de forma creciente. Al añadir un nuevo elemento, usaremos un `ListIterator` para encontrar la posición que debe ocupar el objeto en la lista y mantener así el orden sin necesidad de ordenar de manera explícita.

```
import java.util.LinkedList;
import java.util.ListIterator;

public class DemoListIterator {
    public static void main(String[] args){
        LinkedList<String> lls = new LinkedList<>();
        ListIterator<String> li = null;
        String[] as = "SOLO SE QUE NO SE NADA".split(" ");
        for(int i = 0; i < as.length; i++) {
            // Añadir las cadenas ordenadas
            li = lls.listIterator();
            while( li.hasNext() ) {
                String item = li.next();
                if( as[i].compareTo(item) <= 0) {
                    li.previous();
                    break;
                }
            }
            li.add(as[i]);
        }
        System.out.println("Contenido de la lista: " + lls);
    }
}
```

ORDENAR LOS ELEMENTOS

Ordenar los elementos de una lista es una operación bastante común, pero la interface no la declara. Sin embargo hay métodos estáticos para hacerlo en la clase `java.util.Collections`.

Por ejemplo, si lista es de tipo `List<T>`, la llamada al método `Collections.sort(lista);` ordena la lista en orden ascendente. Los elementos de tipo T deben implementar la interfaz `Comparable<T>`. Funciona para Strings y wrappers de los tipos primitivos y para tus propios objetos si implementas la interface `Comparable<T>`.

El método `Collections.sort()` está sobrecargado con otro método al que le pasas un objeto `Comparator<T>` como segundo parámetro: `Collections.sort(lista, Comparator);`

> **Nota**: *el algoritmo usado es "merge sort" cuyo peor caso y caso medio tienen orden Θ(n*log(n)) para una lista de tamaño n (un poco más lento en promedio que QuickSort, pero un poco más rápido en el peor caso).*

Otros métodos estáticos son `Collections.shuffle(lista)` para reordenar los elementos de forma aleatoria y `Collections.reverse(lista)` para invertir el orden de los elementos.

CLASE VECTOR

Cuando creamos un Vector con el conetructor `Vector(int capacidad, int incremento)` podemos especificar su dimensión inicial y cuanto crecerá si la rebasamos. Por ejemplo la siguiente línea crea un vector con una capacidad inicial de almacenar 20 elementos. Si rebasamos dicha capacidad y guardamos 21 elementos, tu tamaño crece a 25 (se incrementa en 5 cada vez que aumenta).

```
Vector v = new Vector(20, 5);
```

A un segundo constructor solamente se le pasa la capacidad inicial. Si se rebasa, el tamaño del vector se duplica. Debes tener cuidado con este constructor, ya que si se pretende guardar un número grande de

elementos es mejor especificar el incremento si no quieres desperdiciar memoria el ordenador.

```
Vector v = new Vector(20);
```

El constructor por defecto define una capacidad inicial de 10 que se duplica si se rebasa.

```
Vector v = new Vector();
```

Añadir elementos al vector

Hay dos formas de añadir elementos a un vector. Añadir a continuación del último elemento con `addElement()` o insertarlo en una determinada posición mediante `insertElementAt(e, pos)`. El segundo parámetro indica el lugar que ocupará el nuevo objeto en el vector. Si tratamos de insertar un elemento en una posición que no existe obtenemos una excepción del tipo `ArrayIndexOutOfBounds`. Por ejemplo, si tratamos de insertar un elemento en la posición 9 cuando el vector solamente tiene cinco elementos. Para insertar el string "tres" en la tercera posición del vector v, escribimos

```
v.insertElementAt("tres", 2);
```

Para saber cuantos elementos guarda un vector se llama a `size()`. Para saber la capacidad actual de un vector se llama a `capacity()`.

Podemos eliminar todos los elementos de un vector llamando a `removeAllElements()` o bien eliminar un elemento concreto con `removeElement(obj)` o usando su índice con `removeElementAt(2)`.

Acceso a los elementos de un vector

Es tan sencillo como acceder a los elementos de un array. Pasamos un índice a `elementAt(indice)`. Por ejemplo, `v.elementAt(4)` sería equivalente a a[4] si a fuese un array y v un vector. Para acceder a todos lo

elementos del vector, escribimos un código semejante al empleado para acceder a todos los elementos de un array.

```
for(int i=0; i < v.size(); i++){
    System.out.print( v.elementAt(i) + "\t");
}
```

Existe otra alternativa, que es la de usar las funciones de la interface `Enumeration` que son `hasMoreElements()`; y `Object nextElement()`.

EJEMPLO 13: Crea una clase que implemente la interface Enumerarion. La función miembro elements() de la clase Vector devuelve un objeto de la clase VectorEnumerator que implementa el interface Enumeration y tiene que definir las dos funciones `hasMoreElements()` y `nextElement()`.

```
import java.util.Vector;
import java.util.Enumeration;

public class DemoListIterator {
    public static void main(String[] args){
        Vector<String> vs = new Vector<>(5, 5);
        String[] as = "SOLO SE QUE NO SE NADA".split(" ");
        for(int i = 0; i < as.length; i++)
            vs.addElement(as[i]);
        Enumeration<String> e = vs.elements();
        while( e.hasMoreElements() ){
            String elemento = (String)e.nextElement();
            if(elemento.equals("NO") ){
                System.out.println("Encontrado NO");
                break;
            }
        }
    }
}
```

Para buscar objetos en un vector se puede usar una `Enumeration` y hecer una comparación elemento por elemento mediante *`equals`*`()`, tal como aparece en el ejemplo 13. Podemos usar alternativamente `contains()` si

no necesitamos el índice.

```
if(v.contains("tres")){
    System.out.println("Encontrado tres");
}
```

VECTOR<E> CONTRA ARRAYLIST<E>

`ArrayList` y `Vector` implementan la interface `List<T>` y ambos usan arrays dinámicos para hacerlo. Sus diferencias más importantes:

- **Sincronización**: `Vector` es sincronizada, lo que significa que solamente un thread (un hilo de ejecución) puede acceder al mismo tiempo a la estructura, mientras que un `ArrayList` no está sincronizado. Así que en un entorno concurrente (varios hilos ejecutándose a la vez) es más inseguro usar `ArrayList`.
- **Rendimiento**: `ArrayList` es más rápido al no estar sincronizado.
- **Cambio de tamaño**: `ArrayList` y `Vector` pueden aumentar y disminuir su tamaño de forma dinámica. `ArrayList` incrementa su tamaño un 50% y un vector un 100% (dobla su tamaño).
- **Recorrer elementos**: `Vector` puede usar `Enumeration` e `Iterator` pero `ArrayList` solo `Iterator`.

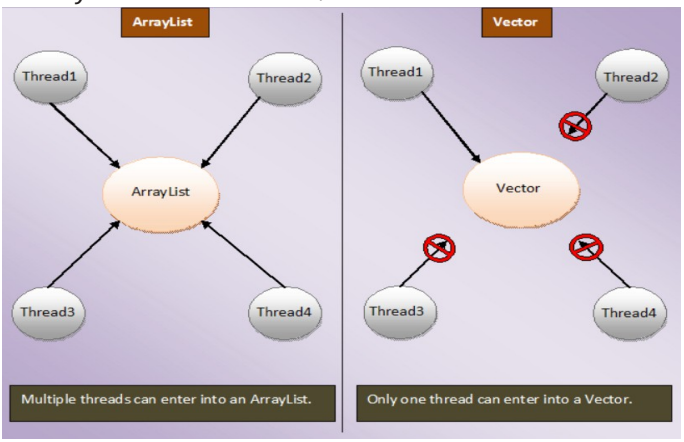


Figura 3: acceso seguro en programas multihilo.

6.4.2. INTERFACE SET<T>: CLASES HASHSET Y TREESET

Un conjunto (set) es una colección de datos (objetos) donde un mismo objeto no puede estar repetido. Implementan todos los métodos de la interfaz `Collection<T>`. Java predefine varias clases que implementan la interface `Set<T>` como: `java.util.TreeSet` y `java.util.HashSet`.

TREESET<T>

Tiene la propiedad de almacenar los elementos ordenados de forma ascendente (un `Iterator` o bucle `for-each` recorre sus elementos en orden) y por tanto los objetos que almacena deben implementar la interface `Comparable<T>` y por tanto definir `obj1.compareTo(obj2)` o altenativamente aportar un objeto `Comparator<T>` como parámetro en el constructor del `TreeSet` (definir el método `compare()` de este objeto `Comparator`). Un `TreeSet` no usa el método `equals()` para comparar sus objetos, sino el método `compareTo()` (o `compare()`).

Se implementan en una estructura similar a un árbol binario de búsqueda al que se añade balanceo (así todas las operaciones de inserción, borrado y búsqueda son eficientes, con un orden en el peor caso de Θ(log(n)), donde n es el número de elementos en el conjunto).

EJEMPLO 14: lee palabras de algún origen (como un fichero de texto o un array) y muestra un listado de las palabras que aparecen ordenadas alfabéticamente. Usamos un TreeSet para ordenar:

```
import java.util.TreeSet;

public class TestTreeSet {
    public static void main(String[] args){
        String[] origen = "SOPA PAELLA SOPA TORTILLA".split(" ");
        TreeSet<String> palabras = new TreeSet<String>();
        int i = 0; // Mientras haya palabras en origen añadirlas
        while( i < origen.length )
```

```
                palabras.add( origen[i++] );
            for(String p : palabras )
                System.out.println(p); // ordenadas y sin repetidos
        }
    }
```

EJEMPLO 15: Usar un TreeSet para ordenar y quitar repetidos de otra colección llamada c (un array por ejemplo):

```
import java.util.TreeSet;
import java.util.Arrays;

public class TestTreeSet {
    public static void main(String[] args){
       int[] origen = new int[20];
       for(int i = 0; i < origen.length; i++)
           origen[i] = (int)(Math.random() * 10);
       System.out.println("Original:" + Arrays.toString(origen) );
       // Instancias TreeSet y añades elementos
       TreeSet<Integer> tsi = new TreeSet<>();
       for(int i = 0; i < origen.length; i++)
           tsi.add(origen[i]);
       // Si origen fuese Integer[] podrías ahorrar el bucle
       //    TreeSet<Integer> tsi = new treeSet<>( origen );
       // Usas toArray() y no hace falta instanciar el array
       Object[] r = tsi.toArray();
       System.out.println("Resultado: " + Arrays.toString(r) );
       // Si usas toArray(T[]) debes instanciar el array
       Integer[] r1 = new Integer[tsi.size()];
       r1 = tsi.toArray(r1);
       System.out.println("Resultado: " + Arrays.toString(r1) );
    }
}
```

Si la colección de datos c tiene objetos, podrías expresarlo de forma más compacta:

```
ArrayList<Integer> ls = new ArrayList<>( new TreeSet<Integer>(c) );
```

HASHSET<T>

Utiliza una tabla hash para guardar sus elementos. Las operaciones de

búsqueda, insertar y eliminar elementos son muy eficientes, pero **no es capaz de aplicar ningún orden** y por tanto no obliga a sus elementos a implementar la interfaz **Comparable**, sin embargo necesita definir como calcular un código hash de los elementos: hashCode() .

Tiene 3 constructores:

- `HashSet<E>()`: una capacidad inicial de 16 elementos y un factor de carga de 0.75. El factor de carga indica que cuando tenga ese porcentaje ocupación (elementos/capacidad * 100) debería crecer para seguir siendo óptimo.
- `HashSet<E>(Collection <? extend E> c)`: crea la instancia con tantos elementos como tenga la colección c.
- `HashSet<E>(int capacidad)`: instancia uno vacío con la capacidad indicada y un factor de carga de 0.75.
- `HashSet<E>(int capacidad, float factor)`: instancia uno vacío con la capacidad y el factor de carga indicados.

Usa el método `E.equals()` para averiguar cuando dos objetos son iguales, aunque primero utiliza `E.hashCode()` para averiguar el grupo donde puede estar. Un `Iterator` puede visitar todos sus elementos, pero sin ningún orden.

> **Nota**: Te recuerdo que las operaciones de conjuntos *matemáticos existen en los conjuntos de Java, pero con otro nombre:*
>
> - *A.add(x) añadir elemento x al conjunto A.*
> - *A.remove(x) sacar el elemento x de A.*
> - *A.contains(x) comprueba si x pertenece (es miembro de) A.*
> - *A.addAll(B) -> A UNION B.*
> - *A.retainAll(B) -> A INTERSECCION B.*
> - *A.removeAll(B) -> A – B.*

EJEMPLO 16: Uso de un HashSet<E>

```
import java.util.HashSet;

public class TestHashSet {
   public static void main(String[] args){
     String[] griego = "alfa beta delta epsilon gamma".split(" ");
      System.out.println("Original:" + Arrays.toString(griego) );
      HashSet<String> hss = new HashSet<>();
      for(String g : griego) hss.add(g);
      System.out.println("Resultado: " + hss );
      System.out.println("Contienes lambda: " +
                         hss.contains("lambda") );
    }
}
```

LINKEDHASHSET<E>

LinkedHashSet es como un HashSet pero que mantiene una lista con los elementos en el orden en que fueron introducidos, esto permite recorrerlos en el mismo orden en que se insertaron cuando usas un Iterator

EJERCICIO 3: Repite el ejemplo 16 pero usa un LinkedHashSet en vez de un HashSet.

ENUMSET

Imagina que E es un tipo enumerado, como es una clase, es posible crear objetos de tipo `TreeSet<E>` y `HashSet<E>`. Sin embargo, como son muy simples y tienen pocos valores diferentes, consumen más recursos las propias estructuras que los datos que contienen. Por ese motivo, para estos casos, Java ofrece la clase `java.util.EnumSet` para dar mayor eficiencia.

Los sets de tipos enumerados se crean con métodos estáticos de la clase `EnumSet`. Por ejemplo si e1, e2 y e3 son valores del tipo enumerado

E, entonces el método `EnumSet.of( e1, e2, e3 )` crea y devuelve un conjunto de tipo `EnumSet<E>` que contiene los elementos e1, e2 y e3. Puedes usar la función `EnumSet.of()` con cualquier número de parámetros para crearlo, eso sí, todos deben ser el mismo tipo enumerado y no admite null.

La función `EnumSet.range(e1,e2)` que crea el EnumSet conteniendo todos los valores entre e1 y e2 incluidos ambos. También tienes la función `EnumSet.allOf(E.class)` que contiene todos los valores y `EnumSet.noneOf(E.class)` que está vacío.

EJEMPLO 17: tenemos un programa que maneja eventos que pueden ocurrir de forma repetitiva en ciertos días de la semana. Cada evento puede tener un conjunto de días en los que ocurre.

```
enum Dia { LUN, MAR, MIE, JUE, VIE, SAB, DOM }
EnumSet<Dia> ocurre = EnumSet.of( Dia.LUN, Dia.MIE );
```

6.4.3. COLAS CON PRIORIDAD: PriorityQueue.

Una cola con prioridad es un **ADT (Abstract Type Data)** o **TDA** en castellano (**Tipo Abstracto de Datos**) que representa una colección donde cada elemento tiene una prioridad que permite comparar unos elementos con otros. Las operaciones que tiene son `add()` para añadir un elemento, `remove()` que elimina y devuelve el elemento con la mínima prioridad.

Una forma de implementarla es usar una lista enlazada que almacene los elementos ordenados por la prioridad en orden creciente. La operación `remove()` elimina y devuelve el primer elemento de la lista y es rápido. Pero la operación `add()` obliga a dejar al elemento en la posición que le corresponde y eso supone un esfuerzo de media $\Theta(n)$, donde n es el número de elementos de la lista. Una mejor implementación consigue

que ambas operaciones se realicen en Θ(log(n)), usando una estructura llamada "heap".

La clase **`PriorityQueue<T>`** implementa una cola con prioridad de elementos de tipo T. Implementa la interfaz `Collection<T>` y `Queue<T>` extiende a **`AbstractQueue`** tiene las operaciones **`add(obj)`** y **`remove()`**.
Tiene 7 constructores. Puede crecer de manera dinámica. Los constructores:

- `PriorityQueue()`: capacidad de 11.
- `PriorityQueue(int capacidad)`: capacidad la indicada.
- `PriorityQueue(Comparator<? super E> comp) PriorityQueue(int capacidad, Comparator<? super E> comp)`: capacidad la indicada y se indica el comparador para conocer la prioridad de los elementos.
- `PriorityQueue(Collection<? extends E> c)`: capacidad la cantidad de elementos que tenga la colección (argumento c) y us ael criterio que esté usando c (o un comparator o los objetos son comparables).
- `PriorityQueue(PriorityQueue<? extends E> c)`: capacidad de la cola con prioridad que se le pasa y su mismo criterio de prioridad. Es un constructor de copia.
- `PriorityQueue(SortedSet<? extends E> c)`: la capacidad del conjunto c y su criterio de ordenación es que usa para calcular la prioridad.

Si los elementos implementan la interfaz **`Comparable`**, pueden compararse con el método **`compareTo()`**. Alternativamente puedes aportar un objeto **`Comparator`** como parámetro del constructor y en ese caso se utiliza el método **`compare()`**. Clases como String, Integer y Date implementan `Comparable`.

Si no se le indica el comparador, usa el comparador por defecto de los objetos que almacena. La cola se ordena en orden ascendente, por tanto, en la cabeza de la cola (por donde salen los elementos) estará el elemento con valor más bajo de prioridad (el menor).

Aunque según el comparador que aportes, puedes cambiar este comportamiento. Por ejemplo si la prioridad se basa en una marca de tiempo, puedes priorizar la cola para que el más antiguo esté el primero cambiando el comparador.

Si llamas a `comparator()` obtienes el comparador que usa, aunque si utilizas el orden de los propia clase E...

Las colas con prioridad **pueden utilizarse para ordenar otras colecciones (como un TreeSet) aunque no quita repetidos**. También pueden utilizarse **para organizar elementos dando prioridad para que salgan de la cola los que menos prioridad tengan**.

EJEMPLO 18: Un programa que devuelva el trabajo que hay que realizar en cierto momento, dando prioridad a los que menos tiempo necesiten:

```
import  java.util.PriorityQueue;

public class TestColaConPrioridad {
   public static void main(String[] args){
        PriorityQueue<Trabajo> cpt = new PriorityQueue<>();
        Trabajo t1 = new Trabajo("cambio aceite", 30);
        Trabajo t2 = new Trabajo("arraglar pinchazo", 15);
        cpt.add( t1 );
        cpt.add( t2 );  // quedará el primero de la cola
        Trabajo actual = cpt.remove();
        System.out.println("Trabajo actual: " + actual );
}

class Trabajo implements Comparable<Trabajo> {
```

```
    private String nombre;
    private int t;  // tiempo

    public String toString() {
        return "{ trabajo:" + nombre + ", tiempo: " + t + "}";
    }

    public int compareTo(Trabajo otro) { return  t - otro.t; }
}
```

Nota: *Aunque iteres por la cola usando un iterador, el orden de la iteración está indefinido.*

Algunas operaciones:

- `add(E e)` Inserta el elemento e.
- `clear()`borra todos los elementos.
- `comparator()` devuelve el comparador usado o null si la prioridad se calcula con el de E que debe ser Comparable.
- `contains(Object o)` devuelve true si contiene al objeto o.
- **`forEach(Consumer<? super E> action)`** realiza la acción para cada elemento del `Iterable` hasta que se procesan o lanza excepción.
- `Iterator()` devuelve un iterator de los elementos de la cola.
- `offer(E e)` inserta el elemento en la cola.
- `remove(Object o)` elimina un elemento y devuelve true si existe.
- **`removeAll(Collection<?> c)`** elimina los elementos que también estén en c.
- **`removeIf`**`(Predicate<? super E> filter)` elimina todos los elementos que cumplen el predicado indicado.
- **`retainAll`**`(Collection<?> c)` se queda con los elementos que están en la colección c.
- `spliterator()` Crea un `Spliterator` sobre los elementos.
- `toArray()` devuelve Object[] con los elementos.
- `toArray(T[] a)` devuelve T[] de elementos de la cola.

Métodos declarados en `java.util.AbstractQueue`

- `addAll(Collection<? extends E> c)` Añade los elementos de c.
- `element()` Recupera pero no borra la cabeza de la cola.
- `remove()` Recupera y elimina la cabeza de la cola.

Métodos declarados en `java.util.AbstractCollection`

- `containsAll(Collection<?> c)` true si c está contenido o igual.
- `isEmpty()` devuelve true si no tiene elementos.
- `toString()` sin comentarios.

Métodos Heredados de **`java.util.Queue`**

- `peek()` recupera pero no elimina la cabeza de la cola o devuelve null si está vacía.
- `poll()` recupera y elimina el elemento de la cabeza o devuelve null.

CLASE ArrayDeque<E>

Es una cola que implementa la interface **`Deque`**, basada en arrays dinámicos sin restricciones de capacidad que puede utilizarse como una cola (FIFO) o como una pila (LIFO). Además ofrece la posibilidad de insertar y borrar elementos tanto por la cabeza como por la cola.

No es segura en programación concurrente, así que habría que sincronizar su uso en aplicaciones multihilo. Es más eficiente que **`Stack`** cuando se usa como una pila y más eficiente que **`LinkedList`** cuando se usa como una cola.

Métodos:

- `add(e)` añade elemento por la parte trasera de la cola. Si su capacidad está limitada y no tiene espacio devuelve `IllegalStateException` y en otro caso devuelve true.

- `addFirst(e)` añade elemento por la cabeza de la cola y se comporta como `add()`.
- `addLast(e)` como add().
- `contains(e)` true si contiene el objeto e.
- `descendingIterator()` devuelve un iterator en orden del último (cola) al primer elemento (cabeza).
- `element()` devuelve sin eliminar el elemento de la cabeza de la cola.
- `getFirst()` devuelve y no elimina el primer elemento de la deque.
- `getLast()` devuelve y no elimina el última elemento de la dequeu.
- `iterator()` devuelve un iterator para la deque desde el primer elemento (la cabeza) hasta el último (la cola).
- `offer(e)` añade un elemento por la parte trasera de la cola y no lanza excepciones como `add()` sino que devuelve `false`.
- `offerFirst(e)` como el anterior pero añade por la cabeza.
- `offerLast(e)` como `offer()`.
- `peek()` recupera un elemento por la cabeza y no lo borra. Si no hay elementos devuelve null.
- `peekFirst()` como `peek()`.
- `peekLast()` recupera sin borrar el elemento de la parte trasera de la cola o null si está vacía.
- `poll()` recupera y borra la cabeza o devuelve null si está vacía.
- `pollFirst()` como `pull()`.
- `pollLast()` como pull() pero para el elemento de la parte trasera.
- `pop()` elimina y devuelve un elemento de la cabeza.
- `push(e)` añade un elemento por la cabeza.
- `removeFirst()` elimina el elemento de la cabeza.
- `removeLast()` elimina elemento de la parte de atrás.
- `size()` calcula y devuelve la cantidad de elementos en la deque.

EJEMPLO 19: Declarar e instancia una Dequeue y añadir y borrar elementos.

```
import java.util.Dequeue;

public class TestDeque {
    public static void main(String[] args) {
        Deque<String> dq = new LinkedList<String>();
        dq.add("E1(Tail)");  // Añadimos por el final
        dq.addFirst("E2(Head)"); // al principio
        dq.addLast("3(Tail)");// Add at the last
        dq.push("E4(Head)");
        dq.offer("E5(Tail)");
        dq.offerFirst("E6(Head)");
        System.out.println(dq + "\n");
        dq.removeFirst();
        dq.removeLast();
        System.out.println("Deque: " + dq);
        dq = new ArrayDeque<String>();
        dq.add("En");
        dq.addFirst("un");
        dq.addLast("lugar");
        dq.addLast("de");
        dq.addLast("la");
        dq.addLast("mancha");
        System.out.println(dq);
        System.out.println( dq.pop() );
        System.out.println( dq.poll());
        System.out.println( dq.pollFirst() );
        System.out.println( dq.pollLast() );
        for(Iterator itr = dq.iterator(); itr.hasNext() ) {
            System.out.print(itr.next() + " ");
        }
        System.out.println();
        for(Iterator it = dq.descendingIterator(); it.hasNext()){
            System.out.print(itr.next() + " ");
        }
    }
}
```

6.4.4. INTERFACE MAPS: CLASES HashMap y TreeMap...

Un array de N elementos puede describirse como una asociación de objetos con cada una de las posiciones enteras 0, 1, . . . , N-1. Con uno de esos enteros (sirve de índice o dirección del elemento) puedes usar las operaciones obtener el elemento (get) y asignar el elemento en una posición (put).

Un mapa es una generalización del array, en la que tienes las operaciones "get()" y "put()" pero los índices pueden ser objetos arbitrarios, no tienen porqué ser números enteros. En otros lenguajes de programación los llaman **arrays asociativos**. Puedes considerarlo como **una colección de parejas (clave,valor)** donde la clave puede ser de cualquier tipo (no como un array donde la clave siempre es int).

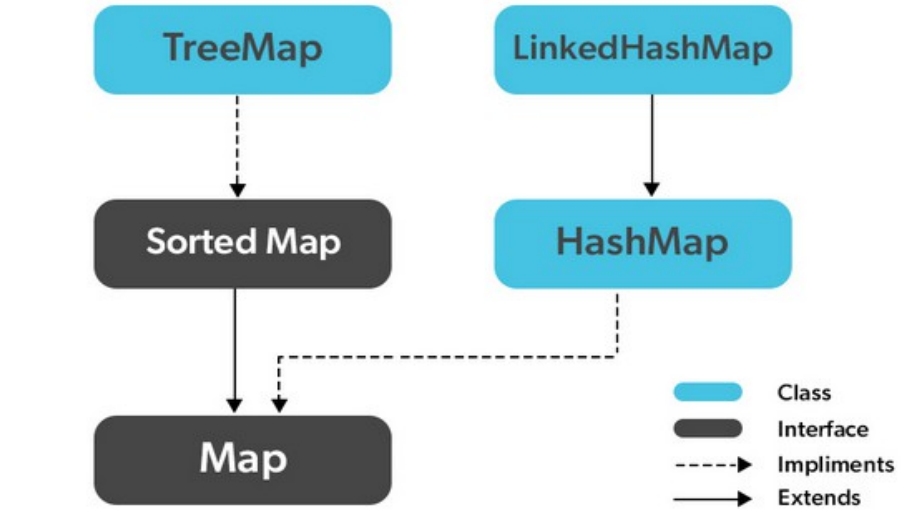


Figura 4: Clases e interfaces de tipo Mapa.

La interface `java.util.Map<K,V>` los define. K es el tipo de las claves y V el tipo de los valores. Las operaciones:

- `get(k)` — devuelve el objeto de tipo V asociado a la clave k o null si no está la clave.
- `getOrDefault(K,V)` — si k existe devuelve su valor, sino devuelve el valor por defecto v.
- `m.put(k,v)` — Asocia el valor v con la clave k y lo añade al mapa. Si ya hay otro valor asociado, reemplaza al valor anterior.
- `putIfAbsent(K,V)` — si no está k, añade (k, v).
- `putAll(m2)` — si m2 es otro mapa, añade todos sus elementos.
- `remove(k)` — elimina la asociación con la clave k si está.
- `containsKey(k)` — devuelve true si está la clave k.
- `containsValue(v)` — devuelve true si el valor v está asociado.
- `size()` — devuelve el número de parejas clave/valor.
- `isEmpty()` — devuelve true si no tiene parejas.
- `clear()` — vacía toda la colección.
- `entrySet()` — crea un Set externo de los valores del mapa.
- `Set()`— crea un Set externo con las claves.

Java incluye las clases `TreeMap<K,V>` y `HashMap<K,V>` que implementan la interface. Aunque en realidad hay más (`AbstractMap`, `EnumMap`, `WeakHashMap`, `LinkedHashMap`, `IdentityHashMap`), estas dos son las más importantes.

Las operación getOrDefault() es muy útil porque te permite procesar los objetos de manera cómoda. Por ejemplo, si en un mapa tienes asociada la cantidad de dinero que debe pagar un cliente, si el cliente no está, la suma debe comenzar por 0 y si está debe comenzar por lo que ya debe pagar:

```
Map<String, Double> m = new HashMap<>();
m.put("Santi", 10);
System.out.print("Cliente: ");
String cliente = sc.nextLine();
```

```
System.out.print("Importe: ");
double importe = sc.nextDouble();
if( m.get(cliente) == null )
    m.put(cliente, importe);
else
    m.put(cliente, m.get(cliente) + importe  );
```

Al ejecutar este código, si tecleamos un nombre distinto de "Santi", dejará a importe la deuda del cliente (no tenía un importe anterior), sin embargo si tecleamos "Santi" dejará en la clave la suma del importe anterior y el importe leído. Pero puede simplificarse (cambiamos lo verde por):

```
m.put(cliente, m.getOrDefault(cliente, 0) + importe);
```

CLASE HASHMAP<K,V>

Es similar a `HashTable` pero sin sincronizar (inserugos en multi-thread). Permiten guardar claves null aunque solo puede estar asociada con un objeto. No mantiene ningún orden en los elementos.

Su estructura es como una lista enlazada de nodos con este formato:

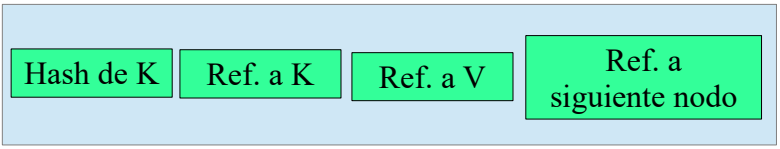


Figura 5: Estructura inicial de HashMap.

El rendimiento de un HashMap depende de 2 parámetros: la capacidad inicial y el factor de carga.

- **Capacidad Inicial**: el número de nodos que tiene para almacenar parejas (K, V). Por defecto 2^4= 16.
- **Factor de carga**: porcentaje de ocupación tras realizar un cambio de tamaño). Por defecto 0.75f. Por tanto cuando se alcanza el 75% de ocupación se realizará un cambio de tamaño y

y se volverán a recalcular todos los hashes de las parejas para reubicarlas.

- **Límite**: el producto del factor de carga y la capacidad. Por defecto es 12 (16 * 0.75 = 12). Esto indica que tras insertar 12 parejas se produce el proceso de Rehashing (doblar la capacidad y recalcular hashes).

Si la capcidad inicial es alta, quizás nunca sea necesario el rehashing, pero de producirse en mapas de gran tamaño, puede penalizar la eficiencia de la aplicación si ocurre a menudo.

> **Nota:** *Desde Java 8 se utilizan árboles binarios auto balanceados BST en lugar de listas enlazadas porque el peor caso de una búsqueda es de $O(Log_2\ n)$.*

Cuando necesites un HashMap para aplicaciones concurrentes puedes envolver al HashMap con el método `Collections.synchronizedMap()` en vez de implementar la sincronización de manera manual. Ejemplo:

```
Map m = Collections.synchronizedMap( new HashMap(...) );
```

La eficiencia de sus operaciones: **get()** y **put()** son constantes si la función hash dispersa de manera efectiva las claves de las parejas entre los nodos disponibles y no se producen muchas colisiones (dos claves intentan ocupar el mismo nodo). De ahí el uso del factor de carga.

EJEMPLO 20: Declarar e instancia un HashMap y añadir y borrar elementos.

```
import java.util.HashMap;

public class TestHashMap {
    public static void main(String args[]) {
        HashMap<Integer, String> hm1 = new HashMap<>();
```

```
            HashMap<String, Integer> hm2 = new HashMap<>();
            // Añadir códigos postales (casi como arrays)
            hm1.put(46, "Valencia");
            hm1.put(18, "Granada");
            hm1.put(28, "Madrid");
            // Ciudades y sus miles de habitantes habitantes
            hm2.put("Valencia", 800);
            hm2.put("Granada", 300);
            hm2.put("Madris", 3000);
            System.out.println("Mapa hm1: " + hm1);
            System.out.println("Mapa hm2: " + hm2);
            // Borrar un elemento
            hm2.remove("Madrid");
            System.out.println("Mapa hm2: " + hm2);
            // Recorrer los elementos
            for(Map.Entry<String, Integer> e : map.entrySet() )
                System.out.printf("(K: %s, V: %s) ",
                                  e.getKey(), e.getValue() );
        }
    }
```

Un `HashMap` no necesita un orden, así que los objetos usados como clave no tienen porqué ser comparables aunque la clase de las claves debería tener una definición correcta de los métodos `equals()` y `hashCode()`.

Un Map no admite claves duplicadas. Si haces un segundo `put()` con la misma clave, "machacas" el valor anterior. Pero puedes tener varias veces el mismo objeto si lo añades con diferentes claves. De todas formas, si necesitas tener repetidos en una misma clave, puedes usar una lista como elemento del mapa (o del conjunto) y meter en ella todos los objetos de una misma clave (como hacer "grupos" de elementos):

```
HashMap<Integer, List<String> > hm = new HashMap<>();
if(!hm.contains(clave)) {
    hm.put(clave, new LinkedList());
}
hm.get(clave).add("elemento1");
hm.get(clave).add("elemento2");
```

CLASE TREEMAP<K,V>

En un `TreeMap`, las parejas (K, V) se guardan en un árbol de búsqueda, ordenadas por la clave en orden ascendente. Por eso las claves deben poder compararse implementando **`Comparable<K>`**, o crearse el map con un objeto **`Comparator<K>`** como parámetro del constructor.

La implementación no está sincronizada (no aconsejable en aplicaciones multithread). Extiende **`AbstractMap`** e implementa `NavigableMap`. No permite claves null. Si necesitas usarlo en entornos multithread puedes sibronizarlo manualmente o bien:

```
TreeMap m = Collections.synchronizedSortedMap( new TreeMap(...) );
```

La implementación de Java se basa en un árbol rojo-negro donde cada nodo tiene

- 3 Variables con datos: (K, V, color)
- 3 Referencias a otros nodos: (left, right, padre)

Los constructores disponibles son: `TreeMap()`, `TreeMap(Comparator comp)`, `TreeMap(Map)`, `TreeMap(SortedMap)`.

EJEMPLO 21: Declarar y usar un TreeMap.

```
public class TestTreeMap {
    public static void main(String args[]){
        // Inicializar con tipo raw
        TreeMap tm1 = new TreeMap();
        // Insertar elementos
        tm1.put("C++", 3);
        tm1.put("Java", 1);
        tm1.put("Python", 2);
        // Usando genéricos
        TreeMap<Integer, String> tm2 = new TreeMap<>();
        tm2.put(new Integer(3), "C++");
        tm2.put(new Integer(2), "Python");
        tm2.put(new Integer(1), "Java");
```

```
            // Imprimir elementos
            System.out.println(tm1);
            System.out.println(tm2);
            // Modificar elementos
            tm2.put(2, "C#");
            System.out.println(tm);
        }
    }
```

EJERCICIO 4: Define un listín telefónico con parejas (String, Integer) para contener nombres de personas y números de teléfonos.

VISTAS DE UN MAP, LIST ó SET

Como un mapa no es exactamente una colección, no implementa iteradores. Para acceder a todos los elementos, el método `keySet()` devuelve un conjunto de todas las claves y usando las claves, accedes a los valores.

EJEMPLO 22: Acceder a todos los valores de un Hashmap:

```
Set<String> claves = directorio.keySet();
Iterator<String> iter = claves.iterator();
System.out.println("El mapa contiene:");
while(iter.hasNext()) {
   String k = iter.next();
   Double v = map.get(k);
   System.out.println( " (" + k + "," + v + ")" );
}
```

EJEMPLO 23: Acceder a todos los valores de un `Hashmap<String, Double)` con for-each:

```
...
for(String k : mapa.keySet() ) {
   Double v = mapa.get(k);
   System.out.println( " (" + k + "," + v + ")" );
}
```

También hay otras dos vistas en los mapas llamadas:

- `values()` devuelve una colección de valores (no un conjunto,

porque dos claves podrían tener el mismo valor) con los valores del mapa.

- `entrySet()` devuelve un conjunto con las asociaciones (parejas) del mapa.

La interface `List<T>` puede definir una sublista con el método `lista.subList(desde, hasta)` que incluye en la sublista (una nueva lista) los elementos que ocupan las posiciones desde, desde+1, … hasta-1. Si cambias los elementos en la sublista, también cambias la lista a partir de la que se forma.

De igual modo, puedes definir subconjuntos de conjuntos `TreeSet` con `subSet(desde,hasta)`. El método `headSet(e)` tiene todos los elementos menores que e y `tailSet(e)` los mayores o iguales que e.

Y la clase `TreeMap<K,V>` define 3 subvistas llamadas:

- `subMap(desdeClave, hastaClave)`,
- `headMap(hastaClave)` parejas con k menor de hastaClave
- `tailMap(desdeClave)` parejas con k mayor o igual que hastaClave

6.4.5. CLASES E INTERFACES HEREDADAS (LEGACY).

En las primeras versiones de Java el FrameWork de colecciones no estaba creado aunque se usaban algunas interfaces y clases de java.util. Cuando aparece, algunas funcionalidades se duplican al mantenerse las primeras y las del FrameWork.

Cuando te plantees si usar una heredada de las primeras versiones de Java o una del FrameWork, normalmente es preferible escoger la más moderna, la del framework Collections. Incluso sabiendo que ninguna está sincronizada mientras que las heredadas si lo están. Las clases

heredadas definidas por java.util son: **Hashtable** es una clase similar a HashMap que implementa las interfaces Serializable, Cloneable, Map<K,V> y extiende Dictionary<K,V>. La sublase inmediata es Properties, La clase Vector ya la hemos comentado antes.

CLASE STACK<E>

Hereda de **Vector** y es preferible usar **ArrayDeque** si no usas concurrencia. Tienes los métodos:

- empty() devuelve true si no tiene elementos y false si los tiene.
- peek() devuelve sin borrarlo, el tope de la pila.
- pop() devuelve y borra el elemento del tope de la pila. Si no hay elementos lanza EmptyStackException.
- push(e) guarda un elemento en el tope de la pila.
- search(e) busca e en la pila y devuelve su posición desde el tope ó -1 si no está.

DICTIONARY

Es una clase abstracta y representa parejas (K,V) almacenadas en en una lista. Funciona como un Map y no admite claves duplicadas.

EJEMPLO 24: Ejemplo de uso de un diccionario.

```
public class TestClasesHeredadas {
    public static void main(String[] args) {
        Dictionary d = new Hashtable();
        d.put("123", "CP");
        d.put("456", "Programa");
        for(Enumeration i = d.elements(); i.hasMoreElements() ){
            System.out.println("Valor: " + i.nextElement() );
        }
        System.out.println("\nValor clave 6: " + d.get("6") );
        System.out.println("Valor clave 456: " + d.get("456"));
        System.out.println("\nVacío: " + d.empty() + "\n");
        for(Enumeration k = d.keys(); k.hasMoreElements();)
            System.out.println("Claves: " + k.nextElement() );
        }
```

```
            System.out.println("\nBorra" + d.remove("123"));
            System.out.println("Comprueba borrado: " + d.get("123"));
            System.out.println("\nNúmero elementos: " + d.size());
    }
}
```

6.5. CREAR NUESTRAS COLECCIONES.

¿Porqué es interesante que seamos capaces de fabricar nuestras propias colecciones?

EJEMPLO 25: En muchas ocasiones es suficiente con utilizar el framework de Colecciones (ArrayList, HashSet, etc.) para solucionar nuestros problemas. Sin embargo, en algunos casos, ninguna de las colecciones de fábrica encajan con lo que estamos buscando. En estos casos debemos construir una **Java Custom Generics** que se encargue de solventar el problema al que nos enfrentamos.

Un ejemplo de esto es el concepto de Nodo. Este concepto es conocido por todos ya que HTML lo usa en su famoso DOM (Document Object Model). Recordemos que un Nodo es un elemento que esta relacionado con otro elemento de su mismo tipo de forma jerárquica.

Vamos a ver como construimos una clase genérica que gestione el concepto de Nodo.

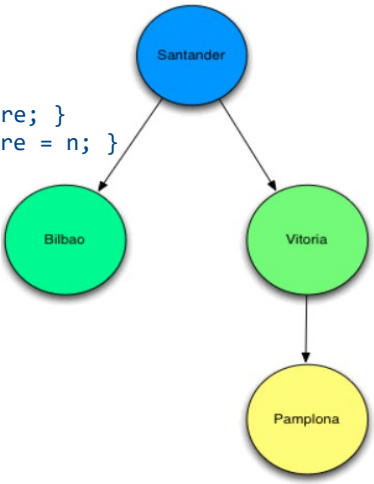
```
package nodos;

public class Nodo<T> {
 private T dato;
 private Nodo<T> siguiente;
 private Nodo<T> anterior;

 public Nodo(T dato) {
   this.dato = dato;
 }
```

```
    public void setDato(T dato) { this.dato = dato; }
    public T getDato() { return this.dato; }
    public void setSiguiente(Nodo<T> s) { this.siguiente = s; }
    public Nodo<T> getSiguiente() { return siguiente;  }
    public Nodo<T> getAnterior() { return anterior; }
    public void setAnterior(Nodo<T> a) { anterior = a; }
   }
```

Como podemos ver, la clase Nodo dispone de un enlace con el Nodo siguiente y con el Nodo anterior. Se trata además de una clase genérica así pues podemos asignar cualquier tipo de objeto. Vamos a usar el concepto de Ciudad para construir una estructura de Nodos.

```
package nodos;

public class Ciudad {
    private String nombre;
    public Ciudad(String n){ nombre= n; }
    public String getNombre(){ return nombre; }
    public void setNombre(String n) { nombre = n; }
}
```

Una vez tenemos el concepto de Ciudad vamos a crearnos la siguiente estructura de nodos. Vamos a verlo en código:




```
public static void main(String[] args){
    Ciudad c1= new Ciudad("Santander");
    Ciudad c2= new Ciudad("Bilbao");
    Ciudad c3= new Ciudad("Vitoria");
    Ciudad c4= new Ciudad("Pamplona");
    Nodo<Ciudad> n1= new Nodo<Ciudad>(c1);
    Nodo<Ciudad> n2= new Nodo<Ciudad>(c2);
    Nodo<Ciudad> n3= new Nodo<Ciudad>(c3);
    Nodo<Ciudad> n4= new Nodo<Ciudad>(c4);
    n1.setSiguiente(n2);
    n2.setAnterior(n1);
    n2.setSiguiente(n4);
    n3.setAnterior(n1);
    n4.setAnterior(n3);
```

```
    viaje(n4);
  }
```

Acabamos de crear una estructura de Nodos y nos queda ver el código de la función viaje() que de forma recursiva nos mostrará como los nodos están relacionados para hacer un viaje entre Santander y Pamplona (si te das cuenta, forman una lista).

```
private static void viaje( Nodo<Ciudad> nodo ) {
     System.out.println( nodo.getDato().getNombre() );
     if( nodo.getAnterior() != null ) viaje(nodo.getAnterior());
 }
```

El resultado es el siguiente :

```
Pamplona
Vitoria
Santander
```

Los Java Custom Generics pueden sernos muy útiles en aquellos casos donde las estructuras de fábrica no son capaces de aportarnos una buena solución.

6.5.1 TDA's (TIPOS DE DATOS ABSTRACTOS).

Un tipo de datos es una colección de valores y un conjunto de operaciones que se pueden realizar a esos valores. Por ejemplo, el tipo **boolean** son los valores **true** y **false**. Y tiene las operaciones **and**, **not**, **xor**, **or**...

Los tipos de datos simples, pueden unirse en registros para formar datos compuestos que tienen mayor complejidad, por ejemplo un número de cuenta bancaria tiene 4 piezas de información (entidad, sucursal, dígito de control y número de cuenta), cada una puede ser de tipo **int**, otro tipo de dato.

Hay que distinguir entre el **concepto lógico del tipo de dato y su implementación** en un programa de ordenador. **Un mismo concepto**

lógico puede construirse de varias formas.

Un **Tipo de Dato Abstracto** (**TDA**, ADT en inglés) es la realización de un tipo de dato como un componente de software. Su interfaz está definida en términos de un tipo de dato y un conjunto de operaciones. **No indica nada de como se implementa, únicamente describe las operaciones que debe tener (su comportamiento)**. Es equivalente al concepto de interface en Java.

Una estructura de datos se utiliza para ayudar a implementar un TDA en un lenguaje de programación. Si el lenguaje es orientado a objetos como Java, un TDA es una interface y su implementación una clase.

En este apartado aprenderemos como podemos implementar nuestras propias estructuras de datos cuando Java (u otro lenguaje que usemos) no nos ofrezca la que necesitamos. Recuerda también que **disponer de la colección adecuada puede resolver un problema con mucho menos esfuerzo que si no cuentas con ella**.

6.5.2 LISTAS.

Todos sabemos intuitivamente lo que es una lista. El concepto importante es el de posición de cada elemento: primer elemento, segundo elemento, ... último elemento. Es decir, **los elementos forman una secuencia**.

Una lista la podemos definir como una secuencia finita y ordenada de elementos. Ordenada en el sentido de que cada elemento ocupa una posición dentro de la lista, no en el sentido de que esa posición ocupada por el elemento sea forzosamente la que debe tener porque responde a un criterio por el que se ordena al elemento. Por ahora, solo vamos a pensar en listas que no ordenen los elementos por un criterio. Todos los elementos de la lista que vamos a implementar van a ser del mismo

tipo (si no, ya sabes que puedes usar Object como elemento).

Las operaciones del TDA Lista se definen en esta interfaz:

```
/** TDA Lista<E> */
public interface Lista<E> {
   /** Vaciar la lista de elementos */
   public void vaciar();
   /** Inserta elemento en posición actual.
       @param e Elemento a insertar. */
   public void inserta(E e);
   /** Añade elemento al final de la lista.
       @param e El alemento a añadir */
   public void pega(E e);
   /** Elimina de la lista y devuelve el elemento actual.
       @return El elemento que se elimina. */
   public E borra();
   /** Cambia la posición actual al principio de la lista */
   public void inicio();
   /** Cambia la posición actual al final de la lista */
   public void fin();
   /** Mueve la posición actual hacia la izquierda, posición
       anterior o previa. No cambia si ya está al principio */
   public void ant();
   /** Mueve a la derecha la posición actual (siguiente)
       no cambia si ya está al final */
   public void sig();
   /** @return El número de elementos en la lista. */
   public int longitud();
   /** @return Poosición del elemento actual. */
   public int posicion();
   /** Cambiar la posición actual.
       @param pos La nueva posición. De 0 a lista.length()-1*/
   public void mueveA(int pos);
   /** @return El elemento actual. */
   public E elemento();
}
```

EJEMPLO 26: Hacer un trozo de código de cómo se puede acceder a todos los elementos de la lista L desde el primero hasta el último:

```
for(L.inicio(); L.posicion() < L.longitud(); L.sig() ) {
```

```
        E e = L.elemento();
        HacerAlgo(e);
    }
```

EJEMPLO 27: Buscar la posición de un elemento e si está en la lista:

```
    /** @return int la posición del elemento en la lista o -1 */
    public static int busca(Lista<Integer> L, int e) {
       for(L.inicio(); L.posicion() < L.longitud(); L.sig() )
          if( e == L.elemento() ) return L.posicion();  // encontrado
       return -1;
    }
```

No necesitamos conocer la implementación para usarlos, solo el TDA / interface que define su comportamiento.

IMPLEMENTACIÓN BASADA EN ARRAYS

Vamos a usar arrays para implementar la interfaz **Lista<E>** y a la clase la llamamos **ListaA<E>**. Los miembros privados de la lista contienen todo lo necesario para poder ofrecer el funcionamiento que se espera indicado en la interface:

```
    class ListaA<E> implements Lista<E> {
       private static final int tama = 10;    // tamaño por defecto
       private int maxTama;  // Limite de tamaño
       private int ne;       // Nº de elementos
       private int actual;   // Posición actual
       private E[] a;        // Array que almacena elementos

       /** Constructores */
       /** Crea una lista con la capacidad por defecto */
       ListaA() { this(tama); }
       /** Crea un nuevo objeto lista.
           @param tama Max # de elementos */
       @SuppressWarnings("unchecked") // Almacenamiento Generico array
       ListaA(int tama) {  // solución cutre para no usar reflection
          maxTama = tama;
          ne = actual = 0;
          a = (E[]) new Object[tama];   // Crea el array
       }
```

```
public void vaciar() {    // borra elementos
   ne = actual = 0;       // Simplemente cambia valores
}

/** Inserta "e" en la posición actual */
public void inserta(E e) {
   assert ne < maxTama: "Capacidad excedida";
   for(int i = ne; i > actual; i--)  // Desplazar anteriores
      a[i] = a[i-1];                 // Hacer hueco al nuevo
      a[actual] = e;
   ne++;   // Incrementa el tamaño
}

//** Agrega "e" a la lista */
public void pega(E e) {
   assert ne < maxTama: "Capacidad excedida";
   a[ne++] = e;
}

/** Elimina y devuelve el elemento actual */
public E borra() {
   if( (actual < 0) || (actual >= ne) ) // No hay elemento
      return null;
   E e = a[actual];    // Copia una referencia al elemento
   for(int i = actual; i < ne-1; i++) // Tapar el hueco
      a[i] = a[i+1];
   ne--;  // Decrementa el tamaño
   return e;
}

public void inicio() { actual = 0; }
public void fin() { actual = ne; }
public void ant() { if(actual != 0) actual--; }
public void sig() { if(actual < ne) actual++; }

/** @return tamaño de lista */
public int longitud() { return ne; }

/** @return posición actual */
public int posicion() { return actual; }

/** Cambiar la posición actual a "p" */
public void mueveA(int p) {
```

```
            assert (p >= 0) && (p <= ne) : "Posición fuera de rango";
            actual = p;
        }

        /** @return elemento de posición actual */
        public E elemento() {
            assert (actual >= 0) && (actual < ne) : "No hay elemento";
            return a[actual];
        }
    }
```

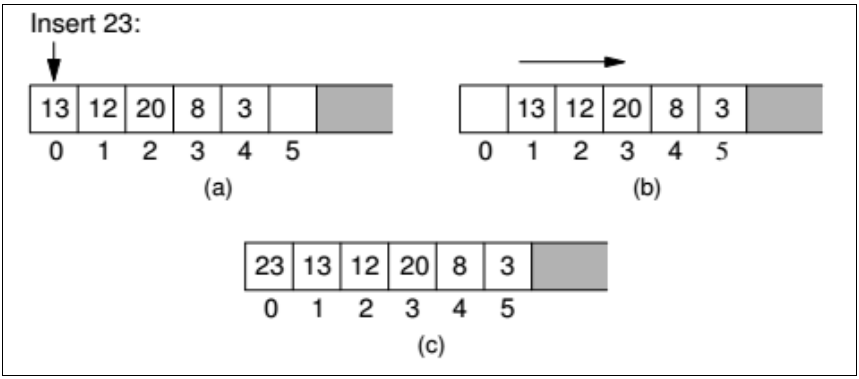


Figura 4: Insertar un elemento obliga a mover una posición al resto. Igual ocurre en el borrado de un elemento. Si la lista es grande, es ineficiente. Peor cuanto más grande sea la lista.

El inconveniente de esta implementación es que las operaciones inserta() y borra() deben mover elementos para hacer hueco y rellenar el hueco respectivamente.

Otro inconveniente es la limitación de tamaño. Si queremos evitarla podemos usar un array dinámico (como ArrayList), aunque el crecimiento y el decrecimiento también consume recursos.

IMPLEMENTACIÓN CON ENLACES

Si usamos enlaces (referencias), los elementos que almacenamos

estarán dentro de nodos. Uno nodo será un objeto de una clase que contiene un hueco para el elemento (dato) y uno o varios enlaces (referencias) al siguiente nodo de la lista, al nodo que contiene el elemento que le sigue en la lista. Otra posibilidad sería que cada nodo tenga un enlace al siguiente y otro al anterior (doblemente enlazada). Cuando el nodo sea el primero, su enlace anterior estará a null, y cuando sea el último de la lista lo que tendrá a null es el enlace al siguiente.

Para tener operaciones rápidas, la lista podría recordar el nodo donde se inicia la lista (cabeza), el nodo que la acaba (cola) y el nodo actual. Vamos a implementar una lista con un solo puntero o enlace al nodo siguiente (esto hará que ahorremos un enlace en cada nodo, pero no podamos tener operación anterior ant() que sea eficiente.

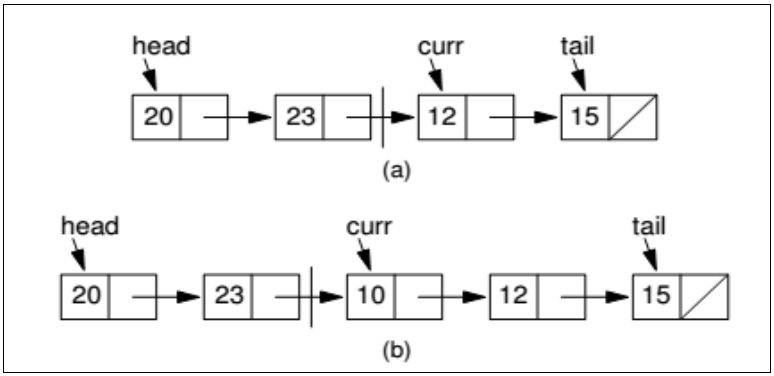


Figura 5 Una lista enlazada de elementos Integer. a) En cierto momento. b) Tras insertar el nodo con el elemento 10 en la posición actual.

Esta implementación permite mejorar la eficiencia de las operaciones inserta(), borra() y no tiene límite de tamaño (más que el de la memoria que tenga el ordenador, claro). A pesar de no ser necesario,

mantendremos los mismos constructores por compatibilidad, aunque internamente el funcionamiento será diferente.

Un asunto clave para la eficiencia es como respresentar la posición actual. Aunque lo lógico parace apuntar al elemento actual, hay una gran ventaja si apuntamos al elemento anterior al actual.

La figura 5 usa un puntero al elemento actual (curr). La línea vertical indica la posición lógica del elemento actual (entre los nodos 23 y 12). Piensa lo que ocurre si insertamos un nuevo nodo con el elemento 10 en la lista. El resultado debe ser lo que aparece en la figura 5b), sin embargo hay un problema: para enganchar el nuevo nodo al anterior, tenemos que cambiar el puntero del anterior. Pero si no lo tenemos, tendremos que recorrer toda la lista desde el principio hasta encontrarlo. Para solucionarlo, lo más sencillo es que la variable actual apunte al elemento anterior al actual.

Como muestra la figura 6, actual apunta al nodo que da acceso al actual, y así permite insertar de forma eficiente. Cuando la lista está vacía, o la posición actual esté al principio y al final, el código debe contemplar varios casos especiales que son posibles causas de fallo.

Para eliminar esos casos especiales (y simplificar el código) usamos un nodo especial llamado cabeza que será el primer nodo de la lista, no contiene elemento (dato), solo nos sirve para simplificar el código, el precio que pagamos es consumir un poco de memoria, que compensamos con el ahorro del código que necesitamos para los casos especiales. La figura 7 muestra el estado de una lista vacía nada más crearla.

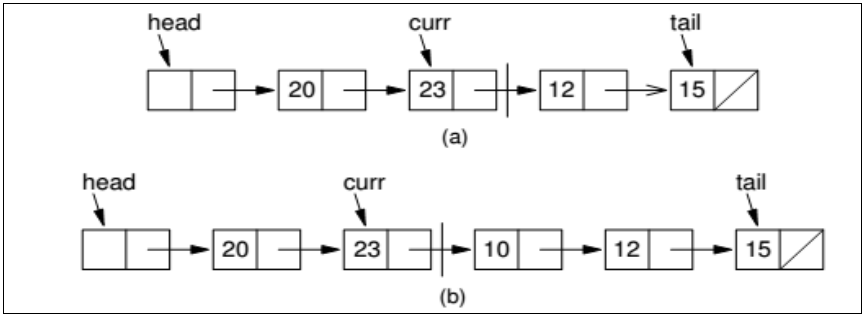


Figura 6 Inserción usando nodo cabeza y actual apuntando al anterior.

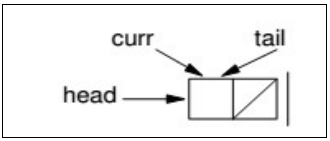


Figura 7 Aspecto inicial de la lista enlazada vacía, con su nodo cabeza.

Un posible listado de código para esta implementación:

```
/** ListaE Lista enlaza que implementa Lista<E> */
class ListaE<E> implements Lista<E> {
   private static class Nodo<E> {
       E dato;
       Nodo<E> sig;
       public Nodo() {}
       public Nodo(E dato){ this.dato = dato; }
       public Nodo(E dato, Nodo<E> s){ this.dato = dato; sig = s;}
   }

   private Nodo<E> cabeza;     // Puntero a la cabeza
   private Nodo<E> cola;       // Puntero al ultimo
   protected Nodo<E> actual;   // Acceso al actual
   private int ne;             // Nº de elementos

   /** Constructores */
   ListaE(int tama) { this(); }  // Constructor, se ignora tama
   ListaE() {
      actual = cola = cabeza = new Nodo<E>(null); // Crea cabeza
      ne = 0;
   }
```

```
/** Borra todos los elementos */
public void vaciar() {
   cabeza.setSig(null);   // Borra acceso a los nodos
   actual = cola = cabeza = new Nodo<E>(null); // Crea cabeza
   ne = 0;
}

/** Inserta "e" en la posición actual */
public void inserta(E e) {
   actual.setNext(new Nodo<E>(e, actual.sig()));
   if(cola == actual) cola = actual.sig(); // Nueva cola
   ne++;
}

/** Añade "e" al final de la lista */
public void pega(E e) {
   cola = cola.setSig( new Nodo<E>(e, null) );
   ne++;
}

/** Borra el elemento actual */
public E borra() {
   if(actual.sig() == null) return null; // No hay nodo
   E e = actual.sig().elemento();        // Recordar valor
   if(cola == actual.sig()) cola = actual; // borra último
   actual.setSig( actual.sigt().sig() ); // desenlazar
   ne--;   // Un elemento menos
   return e;
}

/** Al principio de la lista */
public void inicio() { actual = cabeza; }

/** Al final de la lista */
public void fin() { actual = cola; }

/** Mover a una posición anterior */
public void ant() {
   if(actual == cabeza) return; // No hay anterior
   Nodo<E> temp = cabeza;
   // Recorrer hasta encontrar el previo
   while( temp.sig() != actual) temp = temp.sig();
```

```
            actual = temp;
        }

        /** Mover al siguiente */
        public void sig(){
            if(actual != cola) actual = actual.sig();
        }

    /** @return Número de elementos de la lista */
        public int longitud() { return ne; }
        /** @return Posición actual */
        public int posicion() {
            Nodo<E> temp = cabeza;
            int i;
            for(i=0; actual != temp; i++)
                temp = temp.sig();
            return i;
        }

        /** Mover a una posición arbitraria "p" */
        public void mueveA(int p) {
            assert (p >= 0) && (p <= ne) : "Posición fuera de rango";
            actual = cabeza;
            for(int i = 0; i < p; i++) actual = actual.sig();
        }

        /** @return Elemento de la posición actual */
        public E elemento() {
            if( actual.sig() == null) return null;
            return actual.sig().elemento();
        }
    }
```

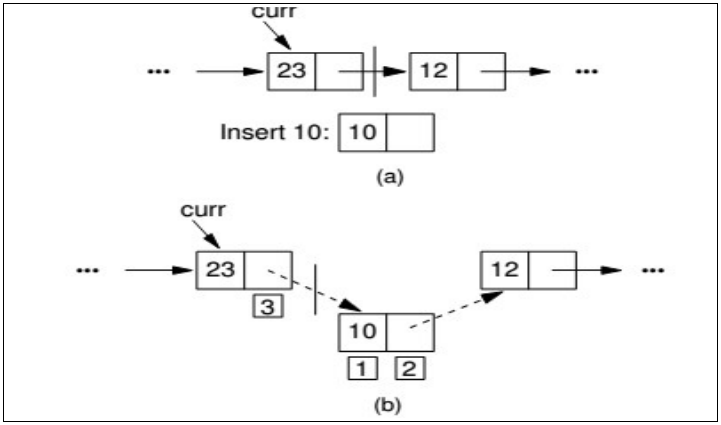


Figura 8: Proceso de inserción.

El operador **new** crea un nuevo nodo llamando al constructor **Nodo<E>** que acepta un valor para el elemento y un enlace para el campo sig. Para borrar el nodo basta con desenlazar el nodo de la lista.

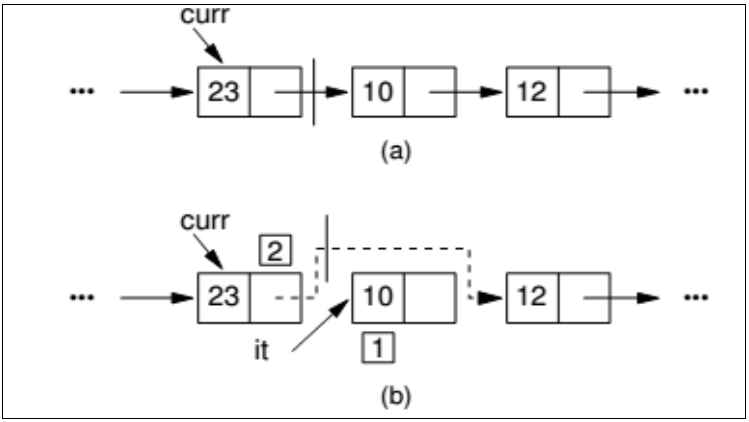


Figura 9. Proceso de borrado de un nodo.

MEJORAR EL RENDIMIENTO CON Freelists

El operador **new** es la operación más costosa que utilizan las estructuras enlazadas, debido a que la máquina virtual debe localizar

memoria cuando se le pide y estar pendiente de liberarla cuando no es necesaria.

Una posible mejora de eficiencia consiste en vez de estar constantemente haciendo llamadas a **new**, que la clase **Nodo** pueda manejar su propia lista de nodos reutilizables, eso es a lo que se llama una **freelist**. Contiene los nodos que actualmente están sin usar. Cuando un nodo se borre de la lista, se coloca en la primera posición de la freelist. Cuando un nodo se añade o inserte en la lista, en vez de crearlo, se comprueba si la freelist tiene nodos, y si es así, en vez de llamar a new, se reutiliza. El listado de la clase **Nodo<E>** con la freelist implementada sería este:

```
/** Nodo de ListaE */
class Nodo<E> {
   private E e; // Valor a almacenar
   private Nodo<E> sig; // Puntero al siguiente nodo

   /** Constructores */
   Nodo(E item, Nodo<E> s) { e = item; sig = s; }
   Nodo(Nodo<E> s) { sig = s; }

   /** Geeetr y setters */
   Nodo<E> sig() { return sig; }
   Nodo<E> setSig(Nodo<E> s) { return sig = s; }
   E elemento() { return e; }
   E setElemento(E item) { return e = item; }
   /** Mejora de eficiencia con freelist */
   static Nodo<E> freelist = null;

   /** @return Un nuevo nodo */
   static Nodo<E> get(E e, Nodo<E> s) {
      if(freelist == null)
         return new Nodo<E>(e, s);  // Lo obtiene de "new"
      Nodo<E> temp = freelist;      // Lo obtiene de freelist
      freelist = freelist.sig();
      temp.setElemento(e);
      temp.setSig(s);
      return temp;
```

```
    }

    /** Devolver un nodo a freelist */
    void libera() {
       elemento = null; // Borra la referencia al elemento
       sig = freelist;
       freelist = this;
    }
}
```

Y las operaciones modificadas con esta mejora aparecen a continuación:

```
/** Inserta */
public void inserta(E e) {
   actual.setSig( Nodo.get(e, actual.sig()) );
   if(cola == actual) cola = actual.sig();
   ne++;
}

/** Añade "e" a la lista */
public void pega(E e) {
  cola = cola.setSig( Nodo.get(e, null) );
  ne++;
}

/** Borrar un elemento y devolverlo */
public E borra() {
   if (actual.sig() == null) return null; // No hay
   E e = actual.sig().elemento();
   if(cola == actual.sig()) cola = actual;
   Nodo<E> temp = actual.sig();
   actual.setSig(actual.sig().sig());
   temp.libera(); // Se devuelve a la freelist
   ne--;
   return e;
}
```

Listas Doblemente enlazadas

Son listas implementadas con enlaces pero cada nodo tiene dos enlaces, uno al siguiente y otro al anterior. Esto supone consumir más memoria y

un poco de tiempo para mantener actualizados los enlaces al anterior cuando añades o eliminas elementos, a cambio de mejorar muchísimo las operaciones **nodo.ant()**.

6.5.3. ÁRBOLES BINARIOS.

Las estructuras secuenciales tienen inconvenientes sobre todo si se necesita tener elementos ordenados. Un árbol es una estructura no secuencial, sino jerárquica. Se pueden implementar con arrays o con punteros. La implementación de punteros en esta colección es más ventajosa.

Dentro de los árboles, los binarios son muy usados para implementar heaps, sets, árboles binarios de búsqueda, analizadores de expresiones, etc.

Un árbol binario A, es un conjunto de nodos que tiene estas propiedades:

- A es vacío, o
- A tiene un nodo raíz, r y 2 subárboles binarios hijos (AI, árbol izquierdo) y AD (árbol derecho), de forma que A = { r, AI, AD }

La siguiente clase **ArbolB<E>** implementa un árbol binario genérico.

```
package arboles;

public class ArbolB<E> {
   protected ArbolB<E> padre;   // Nodo padre
   protected int ne;            // Cantidad de nodos
   protected E e;               // Elemento que almacena
   protected ArbolB<E> izq;     // Puntero al arbol Hijo izquierdo
   protected ArbolB<E> der;     // Puntero al arbol Hijo derecho

   // Constructores
   public ArbolB(E k, ArbolB<E> i, ArbolB<E> d) {
      this.e = k;
      this.padre = null; // Es la raíz del árbol
      this.izq = i;
```

```
        this.der = d;
        this.ne = 1;
    }

    public ArbolB() { this(null, null, null); }

    public ArbolB(E k) { this(k, null, null); }

    public E getE() { return e; }

    public void setE(E k){ e = k; }

    public ArbolB<E> getIzq()  { return izq; }

    public ArbolB<E> getDer()  { return der; }

    // Conectar un árbol como hijo izquierdo
    public void setIzq(ArbolB<E> i) {
        ArbolB<E> nodo;
        int sePierden = (izq == null) ? 0 : izq.ne;  // Nodos se van
        izq = i; // Se conecta el árbol a la nueva rama
        if (i != null) {
            sePierden -= i.ne;  // Si vienen otros, pierde menos...
            i.padre = this;
        }
        // Actualizar la cantidad de nodos subiendo hasta la raíz
        ne -= sePierden;
        nodo = this.padre;
        while(nodo != null) {
            ne -= sePierden;
             nodo = nodo.padre;
        }
    }

    // conectar un arbol como hijo derecho
    public void setDer(ArbolB<E> d) {
        ArbolB<E> nodo;
        int sePierden = (der == null) ? 0 : der.ne; //Nodos q pierde
        der = d; // Se conecta el árbol a la nueva rama
        if (d != null) {
            sePierden -= d.ne;  // Si el nuevo tiene nodos..
            d.padre = this;
        }
        // Actualizar la cantidad de nodos subiendo hasta la raíz
        ne -= sePierden;
        nodo = this.padre;
        while(nodo != null) {
            ne -= sePierden;
             nodo = nodo.padre;
```

```
        }
    }

    public int getSize() { return ne; }

    // Vacía el nodo y devuelve el nº de elementos vaciados
    public void vaciar() {
        if(padre != null) padre.ne -= ne;
        ne = 0;
        e = null;
        izq = null;
        der = null;
    }

    // Desconecta el árbol de su padre, podría llamarse podar
    public void desconectar() { if(padre != null) padre = null; }
}
```

En la figura aparecen un árbol binario.

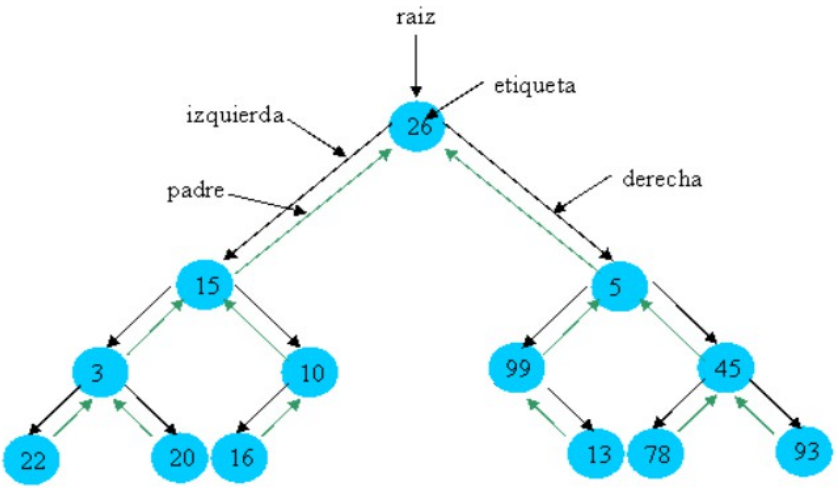


Figura 11: representación de dos árboles binarios.

RECORRIDOS DE UN ÁRBOL

Hay varias formas de recorrer todos los elementos de un árbol. En una lista tenías dos: del principio al final o del final al principio. El árbol T

(raíz en negro, árbol izquierdo en azul y árbol derecho en verde) de la figura 12 nos va a servir de ejemplo para mostrar varias formas de acceder a todos sus nodos.

T = { A, {B, {C}}, {D, {E, {F], {G}}, {H, {I}}} }

RECORRIDO EN PREORDEN DE UN ÁRBOL BINARIO T

Se define de forma recursiva:

1. Visitar el nodo raíz y luego:
2. Hacer el preorden del subárbol izquierdo y luego el preorden del subárbol derecho.

Para el árbol de la figura 12, los nodos se visitan en este orden:

A, B, C, D, E, F, G, H, I

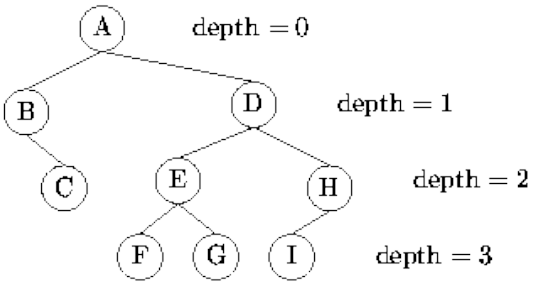


Figura 12: Árbol binario T.

RECORRIDO EN POSTORDEN

Primero se visita a los hijos, y luego al nodo raíz:

1. postorden del hijo izquierdo
2. postorden del hijo derecho
3. visitar el nodo raíz

En el ejemplo: C, B, F, G, E, I, H, D, A

RECORRIDO EN INORDEN

Primero el hijo izquierdo, el nodo raíz se queda en el centro, y el hijo derecho a continuación:

1. inorden del hijo izquierdo
2. visitar el nodo raíz
3. inorden del hijo derecho.

En el ejemplo: B, C, A, F, E, G, D, I, H

RECORRIDO EN ANCHURA

Primero visita los nodos de un nivel de profundidad 0, luego los del nivel de produndidad 1, etc. En cada nivel de profundidad los nodos se visitan de izquierda a derecha.

En el ejemplo: A, B, D, C, E, H, F, G, I

Vamos a centrarnos en el preorden. Realizarlo de forma recursiva es muy sencillo. Por ejemplo si quieres imprimir en texto el árbol al estilo {raíz, {preorden(izquierda)}, {preorden(derecha)} } o quieres fabricar una lista de elementos o nodos en preorden, etc. Es una solución sencilla de implementar.

```
// Devuelve un array a los elementos de un ArbolB en preorden
public E[] preorden(Arbol<E> a) {
   E[] lista = new E[a.getSize()];
   ayudantePreorden(a, lista, 0);
   return lista;
}

public ayudantePreorden(Arbol<E> a, E[] lista, int pos) {
   if(a == null) return;   // por precaución;
   lista[pos++] = a.getE();
   if(a.getIzq() != null)
      ayudantePreorden(a.getIzq(), lista, pos);
   if(a.getDer() != null)
      ayudantePreorden(a.getDer(), lista, pos);
```

```
    }
```

Sin embargo, si queremos crear un iterador, para poder ir recorriendo los elementos del **ArbolB<E>** de uno en uno (interfaz Iterator implementa los métodos **hasNext()** y **Next()**) sin preocuparnos en nuestro código de navegar por la estructura del árbol, debemos eliminar la recursividad, porque se va llamando a **Next()** de forma espaciada en el código y en el tiempo. Para eliminar la recursividad hay que utilizar una solución iterativa, para ello vamos a usar una pila que recuerde el estado en el que está el iterador:

1. Crear una pila vacía y meter al nodo raíz.
2. Cuando la pila no esté vacía hacer lo siguiente:
    - Pop un nodo de la pila y procesarlo.
    - Push hijo derecho del nodo popeado
    - Push hijo izquierdo del nodo popeado

```
// Iterador en preorden sin recursividad
// Aún queda poder utilizarlo de forma espaciada, no seguida
public void preorden(ArbolB<E> raiz) {
   if(raiz == null)
     return;
   Stack<ArbolB<E>> pila = new Stack<ArbolB<E>>();
   pila.push(raiz);
   while(!pila.empty()){
      ArbolB<E> next = pila.pop();
      System.out.printf("{ %d, ",next.e); // Procesarlo
      if(next.getDer() != null){
         pila.push( next.getDer() );
      }
      if(next.getIzq() != null){
         pila.push( next.getIzq() );
      }
   }
}
```

Ya no hay llamadas recursivas, pero se sigue utilizando de forma continua. Si queremos tener la funcionalidad de un Iterator, debemos

olvidarnos del bucle e ir haciendo las mismas operaciones del algoritmo pero no de forma continuada para que pueda llamarse a **next()**. Aquí tienes una posible solución, de un Iterator que devuelve nodos de tipo **ArbolB<E>**.

```
package arboles;

import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Stack;

public class ArbolBPreorden<E> implements Iterator<ArbolB<E>> {
    Stack<ArbolB<E>> pila;
    // Constructor
    public ArbolBPreorden(ArbolB<E> raiz) {
        if(raiz == null) return;
        pila = new Stack<ArbolB<E>>();
        pila.push(raiz);
    }

    public boolean hasNext(){ return !pila.empty(); }  // pila no vacía
    }

    public ArbolB<E> next() {
       if(!hasNext()) throw new NoSuchElementException();
       ArbolB<E> next = pila.pop();
        if(next.getDer() != null){
           pila.push( next.getDer() );
       }
        if(next.getIzq() != null) {
           pila.push( next.getIzq() );
       }
        return next;
    }
 }
```

ÁRBOLES BINARIOS DE BÚSQUEDA

Son árboles binarios donde la posición que ocupa un elemento **e** está restringida por el valor de alguna clave **K**: todos los elementos e en un surárbol izquierdo de un nodo son menores que su elemento e, y todos los valores e del subárbol de la derecha son mayores o iguales (si se usan para implementar conjuntos, y no se admiten repetidos, mayor

estricto).

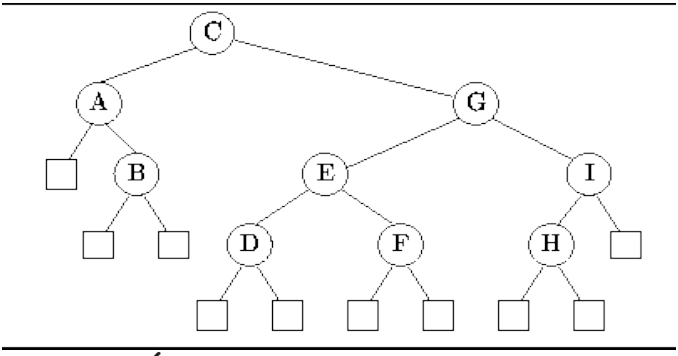


Figura 13: Árbol binario de búsqueda de caracteres.

Por tanto, los elementos E deberían poder ser comparables. También se suelen usar nodos que guardan dos valores, una pareja de clave-valor. En este caso las claves deben ser las comparables.

```
Public interface BBTree<E> extends BTree<E> {
   E minimo();
   E maximo();
}
```

La clase **ArbolBB<K extends Comparable<K>, E>** implementa un sencillo árbol de búsqueda binario que almacena parejas clave-valor :

```
package arboles;

import java.lang.RuntimeException;

public class ArbolBB<K extends Comparable<K>, E> {
   protected K k;                    // Clave del dato
   protected E e;                    // Valor del dato
   protected ArbolBB<K,E> padre, izq, der;  // Padre e hijos

   public ArbolBB(K clave, E valor) {
      if (clave == null)
         throw new IllegalArgumentException("ArbolBB,clave null");
      if (valor == null) return;
      k = clave;
```

```
            e = valor;
            izq = null;
            der = null;
            padre = null; // es raíz
        }

        // Leer valor buscado por clave b
        public E get(K b) {
            if( e == null ) return null;  // Está vacío
            int diff = b.compareTo( k );
            if(diff == 0) return e;
            else if(diff < 0 && izq != null) return izq.get(b);
            else if(der != null) return der.get(b);
            else return null;  // No lo ha encontrado
        }
```

Para añadir un elemento, usamos el método **put(clave,valor)** que debe hacer algo similar al método **get(clave)** navegando por el árbol desde la raíz hasta llegar a un nodo hoja. Una vez que encontremos ese nodo, lo transformamos en un nodo interno llamando a **conecta(nodo, clave, valor)** Después de realizar la conexión se llama al método **balancear()** que por ahora está vacío pero que comentaremos más adelante en una nueva clase **ArbolBBB** (árbol binario de búsqueda balanceado).

```
// Inserta el elemento (clave,valor)
public void put(K clave, E valor) {
  if(clave == null)
  throw new IllegalArgumentException("ArbolBB.put() clave null");
  int diff = clave.compareTo( k );
  // Si la clave es igual, cambiamos el valor
  if(diff == 0)
     e = valor; //throw new IllegalArgumentException "Duplicado";
  else if(diff < 0) {
     if(izq != null)
        izq.put(clave, valor);
     else {
        izq= new ArbolBB<K,E>(clave, valor);
        izq.padre = izq;
     }
  }
```

```
        else {
            if( der != null)
                der.put(clave,  valor);
            else {
                der = new ArbolBB<K,E>(clave, valor);
                der.padre = der;
            }
        }
        balancea();
    }

    public void balancea() {  return; }  // nada por ahora
```

BORRAR ELEMENTOS

Cuando borramos un elemento, es forzoso que el árbol permanezca ordenado.Si el elemento borrado es un nodo hoja, es muy sencillo eliminarlo, no puede causar problemas. En la figura 14 si borramos el nodo con el elemento 4, la operación no afecta al orden.

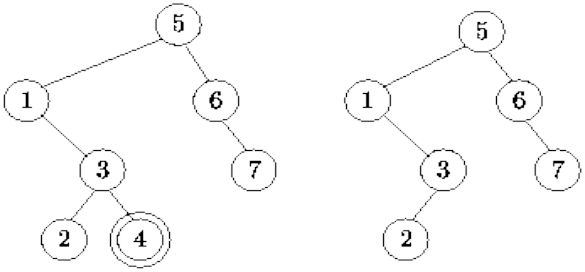


Figura 14: Borrar un nodo hoja.

Para borrar un nodo no hoja, lo movemos hacia abajo por el árbol hasta que se convierta en un nodo hoja. Para conseguirlo, iremos intercambiando son otros nodos que encontremos en el camino. Por ejemplo, en la figura 15, vamos a borrar el nodo con el valor 1. Al no ser un nodo hoja (aunque no tiene árbol izquierdo, lo tiene derecho) . Lo intercambiamos por la clave más pequeña que encontremos en el subárbol derecho (en este caso el nodo 2). Ahora el nodo 1 si es un

nodo hoja y puede borrarse sin problemas.

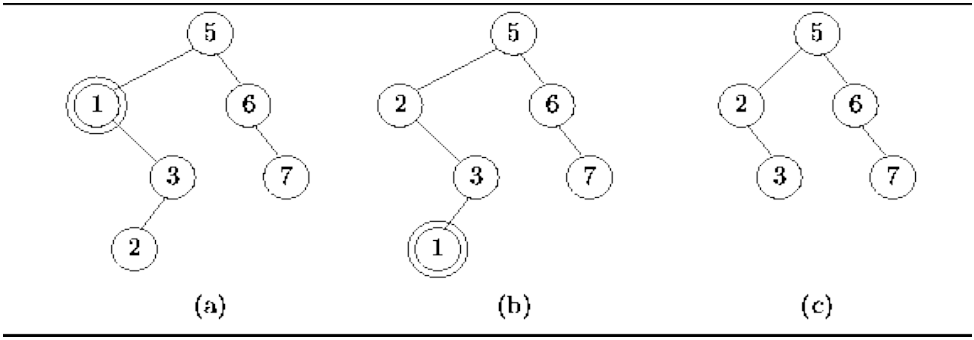


Figura 15: Borrar un nodo no hoja de un ArbolBB.

Siempre lo intercambiamos con el valor más pequeño de la derecha o con el valor más grande de la izquierda. Si no tiene ni uno ni otro, es porque es un nodo hoja. Si después de un intercambio, sigue sin ser un nodo hoja, volvemos a realizar otro intercambio. Al final el nodo debe alcanzar la parte baja del árbol y en ese momento se borra.

```
// Devuelve la clave más pequeña
public ArbolBB<K,E> min() {
   if(k == null) return null;
   if(izq != null) return izq.min();
   else return this;
}

// Devuelve la mayor clave
public ArbolBB<K,E> max() {
   if ( k == null ) return null;
   if(der != null) return der.max();
   else return this;
}

// Borra y devuelve el valor de la clave k o null si no existe
public void delete(K clave) {
       if(clave == null)  return;
       if( k == null )  // También podrías no hacer nada con un return
              throw new RuntimeException("ArbolBB con clave null");
      int diff = clave.compareTo( k );
      if ( diff == 0 ) {   // Bajar el nodo hasta hacerlo una hoja
```

```
                if ( izq != null ) {
                  ArbolBB<K,E> max = izq.max();
                  e = max.e;
                  k = max.k;
                  izq.delete(max.k);
                }
                else if( der != null ) {
                  ArbolBB<K,E> min = der.min();
                  e = min.e;
                  k = min.k;
                  der.delete(min.k);
                }
                else { // Desconectar el árbol del nodo
                  if(padre.izq == this) padre.izq = null;
                  else if(padre.der == this) padre.der = null;
                }
                balancea();
        }
        else if(diff < 0 & izq != null) izq.delete(clave);
        else if(diff >0 && der != null) der.delete(clave);
        else return; // Estoy en un nodo hoja y no se encuentra clave
    }
```

ÁRBOLES BINARIOS DE BÚSQUEDA BALANCEADOS.

El problema de los árboles binarios de búsqueda es que aunque las operaciones **get()**, **put()** y **delete()** en promedio tienen un orden de O(log(n)) en el peor caso lo tienen de O(n). El motivo es que cuando su altura crece, algunos tramos se parecen más a una lista que a un árbol.

La solución de este problema consiste en **balancear el árbol**: que cuando aumente su altura, se transforme y crezca en anchura, no en altura (si es posible). Un árbol balanceado tiene la misma altura (más menos 1) en su parte izquierda que en su parte derecha.

La clase **ArbolBBB<K,E>** podría extender la clase **ArbolBB<K,E>** para añadirle la propiedad de balanceo, pero para no mezclar cosas, voy a hacerla independiente. Podemos partir de una copia del código y añadir y modificar algunas cosas (por ejemplo no son punteros **ArbolBB<K,E>** sino **ArbolBBB<K,E>** tanto padre, izq, der y otras variables.

Cada nodo debe conocer la **altura del árbol** del que es raíz. Un **ArbolBBB** de un solo nodo tiene altura 0. Además cada nodo debe conocer su **factor de balanceo** **FB = altura(izq) – altura(der)**. Para que todo el árbol esté balanceado el FB de todos los nodos puede ser -1, 0 ó 1 (una diferencia de un nodo de altura como máximo tanto a la izquierda como a la derecha). La clase será una cosa similar a esta (solo te paso lo que sea significativamente diferente a lo anterior):

```
package arboles;

import java.lang.RuntimeException;

public class ArbolBBB<K extends Comparable<K>,E>{
   protected K k;  // clave
   protected E e;  // valor
   protected ArbolBBB<K,E> padre, izq, der; // padre e hijos
   protected int altura;   // Distancia a las hojas
   protected int fb;       // Factor de balanceo

   public ArbolBBB(K clave, E valor) {
      if (clave == null)
      throw new IllegalArgumentException("ArbolBBB() clave null");
      k = clave;
      e = valor;
      izq = null;
      der = null;
      padre = null; // es raíz
      altura = 0;
      fb = 0;
   }

   public E get(K b) {
      if( k == null ) return null;  // No se puede comparar
      int diff = b.compareTo( k );
      if(diff == 0) return e;
      else if(diff < 0 && izq != null) return izq.get(b);
      else if(der != null) return der.get(b);
      return null;  // No lo ha encontrado
   }

   // Lo que falta...
}
```

El método **setAltura()** debe llamarse cada vez que se modifica un

subárbol, para manetener actualizada la información de la altura de sus nodos. El método **fb()** calcula y devuelve la diferencia de alturas entre los árboles izquierdo y derecho de un nodo. Por definición, un nodo vacío está balanceado.

```
// Devuelve la clave más pequeña
public ArbolBBB<K,E> min() {
   if(k == null) return null;
   if(izq != null) return izq.min();
       else return this;
}

// Devuelve la mayor clave
public ArbolBBB<K,E> max() {
   if ( k == null ) return null;
   if(der != null) return der.max();
   else return this;
}

public int getAltura() { return altura; }

protected void setAltura() {
    if( k == null )
           altura = -1;
        else {
           int ai = 0, ad = 0; // Altura izquierda y derecha
           if( izq != null) ai = izq.getAltura();
           if( der != null) ad = der.getAltura();
           altura = 1 + Math.max(ai, ad);
        }
}

// Calcula el factor de balanceo de un nodo
protected int fb() {
        if( e == null ) return - 1;
        else {
          int ai= 0, ad = 0;
          if(izq != null) ai = izq.altura;
          if(der != null) ad = der.altura;
          fb = ai - ad;
          return fb;
        }
}
```

INSERTAR ELEMENTOS

Es un proceso de dos partes. Primero, se inserta el nodo en la posición que le corresponda según su valor. Después de insertarlo hay que comprobar que el árbol quede balanceado y balancearlo en caso de no estarlo.

Al insertar un nodo, se cambia la altura de la rama donde se añade (camino de la raíz hasta su posición), así que hay que calcular las alturas de los nodos y comprobar el balanceo. A veces incrementar la altura no incumple la condición de balanceo. Por ejemplo en un árbol T = {r, TI, TD}. Si ai y ad son las alturas de Ti y Td respectivamente, se cumple que |ai – ad| <= 1. Así que si por ejemplo ai = ad + 1 e insertamos en Td, no se incumple la condición.

```
// Inserta el elemento (clave,valor)
public void put(K clave, E valor) {
   if(clave == null)
  throw new IllegalArgumentException("ArbolBBB.put() con clave null");
   int diff = clave.compareTo( k );
   // Si la clave es igual, cambiamos el valor
   if(diff == 0)  // throw new IllegalArgumentException
       e = valor;  "Duplicado";
       else if(diff < 0) {
          if(izq != null)
              izq.put(clave, valor);
          else {
              izq= new ArbolBBB<K,E>(clave, valor);
              izq.padre = izq;
          }
       }
       else {
           if( der != null)
              der.put(clave,  valor);
           else {
              der = new ArbolBBB<K,E>(clave, valor);
              der.padre = der;
           }
       }
       balancea();
     }
```

BALANCEANDO LOS ÁRBOLES

Para volver a dejar un árbol balanceado, podemos aplicar unas operaciones llamadas **rotaciones**. Hay 4 casos posibles y cada uno tiene su propia rotación. La figura 16 tiene en a) el árbol A balanceado y en el B con un factor de +1 (tb está balanceado).

En el casso b) de la figura 16, insertamos un elemento en AL y eso hace que el árbol pierda el balanceo, porque queda con un factor de +2.

Para volver a dejarlo balanceado, en el caso c) reorganizamos los nodos A y B y los 3 subárboles llamados en la figura AL, AR y BR. Esta rotación se llama **rotación LL** porque los primeros dos nodos del camino de la inserción van a la izquierda. Hay 3 propiedades importantes de la rotación LL:

1. No destruye el orden de los datos. **AL** permanece a la izquierda del nodo **A**, **AR** se queda entre los nodos **A** y **B**, y el subárbol **BR** se queda a la derecha de **B**.
2. Tras la rotación, **A** y **B** están balanceados.
3. Tras la rotación, el árbol no ha cambiado de altura.

Se ha usado la **rotación LL** porque el factor de balanceo es positivo (había crecido la altura del subárbol izquierdo). La **rotación RR** se usa cuando el factor es negativo (el árbol derecho es demasiado alto). A estas dos rotaciones se las llama **rotaciones sencillas**.

Pero con estas dos rotaciones quedan por cubrir otros dos casos. En la figura 17, en el apartado b) el árbol queda sin balancear, aparece el nodo C con un factor de +2, pero el subárbol izquierdo tiene un factor negativo. Se soluciona haciendo una rotación RR en el nodo A y una rotación LL en el nodo C. **La combinación de dos rotaciones simples se llama rotación doble**, y en este caso se llama **rotación LR** porque

los primeros vértices en la ruta de la inserción son a la izquierda (L) y a la derecha (R).

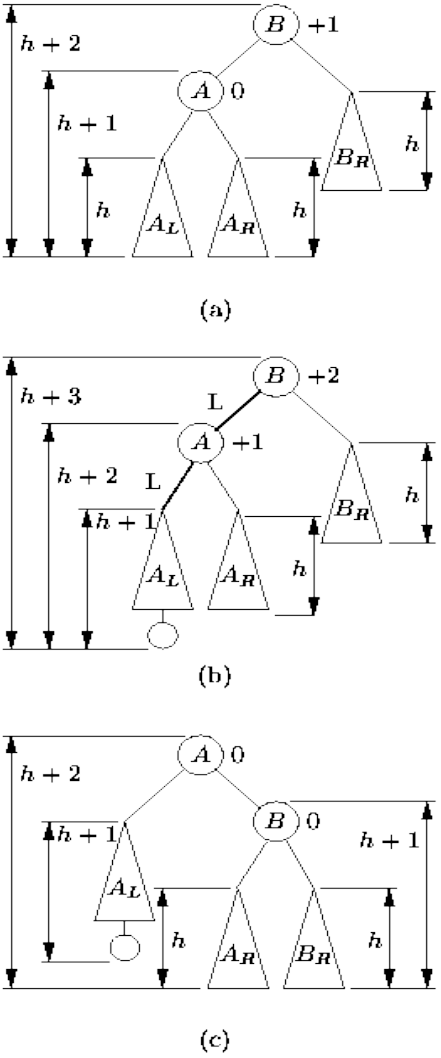


Figura 16: Balancear con una rotación LL.

La otra rotación doble es la **rotación RL**, que se usa cuando el árbol queda con factor negativo, pero su subárbol derecho lo tiene positivo y se ha realizado una inserción que ha seguido el camino RL.

Bueno, pues las cuatro rotaciones LL, RR, LR y RL cubren las posibilidades. ¿Cómo detecto cuando usar cada una? El siguiente teorema resuelve esa cuestión.

> **TEOREMA**: Cuando un árbol pierde el balanceo tras una inserción, una sola rotación simple o doble es suficiente para balancearlo.

Cuando se inserta el elemento x en T y acaba en un nodo hoja, los únicos nodos afectados son los del camino de la raíz a x. Pensemos en alguno de los nodos c de ese camino, su altura se incrementa en 1 o sigue igual. Si no cambia, no necesita rotación ni c ni ninguno de los que tenga por debajo.

Si la altura de c cambia, hay dos posibilidades: c sigue balanceado o pierde el balanceo. Si c permanece balanceado, c no necesita rotación. Aunque podría ser necesaria si alguien por encima de c la necesita y está en el camino de inserción de x.

Si c pierde el balanceo, hay que realizar una rotación en c. Tras hacerla, su altura queda igual que antes de la inserción. Así que ya no son necesarias las rotaciones en los nodos de ecima.

El teorema sugiere: comenzando en el nodo que acaba de insertarse, moviéndonos hacia atrás en el camino de inserción hasta la raíz. Para cada nodo, calcular la altura y comprobar el balanceo. Si la altura no se incrementa, no hace falta investigar más. Si queda sin balanceo, aplicar rotación. Tras la rotación, si recupea su altura no hay que considerar más nodos. En otro caso, investigamos el siguiente nodo del camino.

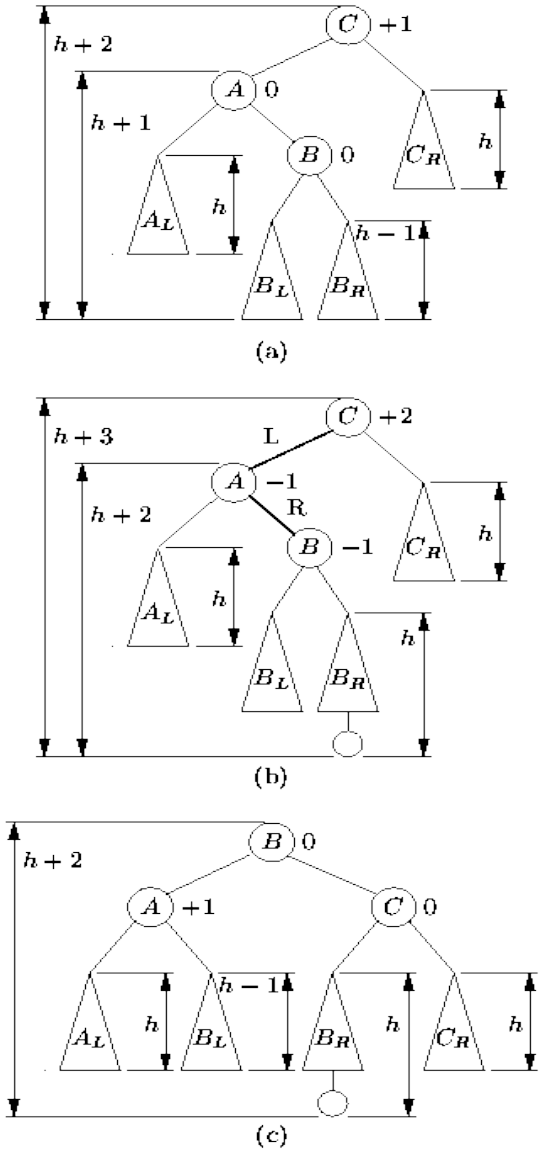


Figura 17: Balanceando con rotación LR.

```
    protected void rotarLL(){
           if( k == null )
               throw new RuntimeException("Nodo de ArbolBB con clave null");
           ArbolBBB<K,E> tmp = der;
           der = izq;
           izq = der.izq;
           der.izq = der.der;
           der.der = tmp;
           E et = e;
           e = der.e;
           der.e = et;
           der.setAltura();
           setAltura();
           fb();
         }

    //precond: el árbol necesita una rotacion RR
    protected void rotarRR(){
       if( k == null )
          throw new RuntimeException("Nodo de ArbolBB con clave null");
        ArbolBBB<K,E> tmp = izq;
        der = izq;
        izq = der.izq;
        der.izq = der.der;
        der.der = tmp;
        E et = e;
        e = der.e;
        der.e = et;
        der.setAltura();
        setAltura();
        fb();
}

//precond: el árbol necesita una rotacion LR
protected void rotarLR(){
   izq.rotarRR();
   rotarLL();
}

//precond: el árbol necesita una rotacion RL
protected void rotarRL(){
     der.rotarLL();
     rotarRR();
 }
```

El lugar donde se comprueba el tipo de rotaciones que hay que hacer (si es que hay que hacer alguna) es en el método **balancea()**.

```
public void balancea() {
    setAltura();
    fb = fb();
    if(fb > 1) {
           if( izq != null && izq.fb > 0 )
              rotarLL();
           else
              rotarLR();
    }
    else if ( fb < -1) {
           if( der != null && der.fb < 0)
              rotarRR();
           else
              rotarRL();
    }
}
```

6.5.4. TABLAS HASH.

Si el objetivo de una colección de objetos es acceder rápidamente a un elemento localizándolo a través de uno de sus datos (digamos su clave), si mantienes un orden en la colección por esa clave, la búsqueda binaria es la mejor opción. Pero si el orden te da igual, y lo único que necesitas es asociar la clave con el objeto para saber si una clave está o no y acceder rápidamente al objeto a modo de un índice en un array:

objeto1 = colección[clave1];

las tablas hash son la mejor opción. Son una colección que transforman cualquier clave (numérica, fecha, string, etc.) en un valor numérico utilizando una función hash llamada h: h(clave)= número. Ese número se utiliza de índice en un array donde se almacenan los objetos en posiciones que indican la función hash. Para no generar posiciones fuera del array, se usa el operador módulo posiciones del array.

> **Nota:** *la búsqueda binaria necesita ordenar las colecciones y recorrer la estructura a través de las posiciones en busca del elemento. Las tablas hash generan la posición que ocupa un*

elemento usando el propio elemento, no deben buscarlo, solo calcular donde está (si es que está).

Una tabla hash necesita dos algoritmos: El primero para calcular el hash y convertir la clave en una posición del array. En el mundo perfecto de las gominolas y los unicornios, dos claves distintas generan dos posiciones distintas, pero en nuestro maldito mundo real, a veces dos claves distintas generan la misma posición en el array, esto se conoce como **una colisión**. Así que es necesario un segundo algoritmo que resuelva el caso de las colisiones.

FUNCIÓN HASH

Es el primer algoritmo que necesita una tabla hash. Su trabajo es aceptar un valor que sirve de clave para un objeto y transformarlo en una posición o índice en una tabla donde almacenar ese objeto. Si la tabla tiene n huecos donde almacenar objetos, la posición que genera irá desde 0 hasta n-1 (si usa un array).

Como la clave puede ser de cualquier tipo, la función hash debe ser capaz de trabajar con enteros o con strings o con cualquier tipo de dato. Además, claves muy similares deberían generar posiciones diferentes, así que deben introducir algún tipo de proceso que no dependa de la clave pero que sea repetible (si a la media hora vuelves a pedir la posición, debe devolverte el mismo valor).

El caso más simple sería sacar el módulo a claves enteras. Si el array donde almacenar objetos tiene n huecos, y la clave k es entera positiva o negativa:

```
FUNCION hash( ENTERO k)
        DEFINE ENTERO LARGO h = k MODULO n;
        SI h < 0 ENTONCES DEVUELVE n – h; FINSI;
        DEVUELVE h;
```

```
FINFUNCION
```

Si los valores de las claves que vengan a nuestra colección estuviesen perfectamente distribuidos en todos los posibles valores, funcionaría bien, pero en el mundo real no ocurrirá, así que para optimizar la distribución de posiciones generadas, el tamaño del array (n) de la tabla debería ser un número primo (El motivo puedes encontrarlo en el libro "The Art of Computer Programming: Sorting and Searching").

Para claves String, la idea es convertir el string en un valor numérico y aplicarle una transformación similar. Por ejemplo combinar cada valor numérico de una letra con su posición.

```
FUNCION hash( STRING k)
    DEFINE ENTEROLARGO h= 0;
    PARA i DESDE 0 HASTA k.longitud - 1 HACER
        h= ( (h * 17) +  ASCII( k[i] )) MODULO n;
    FINPARA
    SI h < 0 ENTONCES DEVUELVE n - h; FINSI;
    DEVUELVE h;
FINFUNCION
```

Una función mejor que la anterior, fue descrita P.J. Weinberger y se la conoce como Executable and Linking Format (ELF) hash. Mejora a la anterior en que al utilizar desplazamientos de bits y operaciones lógicas, es más rápida. Además, dispersa mejor los valores generados reduciendo las colisiones y además no genera números negativos.

```
FUNCION hash_PJW(STRING k)
    DEFINE ENTEROLARGO g, h= 0;
    PARA i DESDE 0 HASTA k.longitud - 1 HACER
        h = ( (h << 4) +  ASCII( k[i] ));
        g = h AND 0xF0000000;
        SI g distinto 0 ENTONCES h= h XOR (g >> 24) xor g; FINSI
    FINPARA
    DEVUELVE h % n;
FINFUNCION
```

Si prefieres aplicar el mismo hash para cualquier tipo de dato, podrías mapearlos todos a string por ejemplo.

RESOLVER COLISIONES

Por muy buena que sea la función hash, tarde o temprano aparecerán colisiones salvo que la cantidad de huecos en relación con la cantidad de elementos que almacena (factor de ocupación) sea muy bajo. Si este factor sube demasiado, la tabla hash puede volverse muy ineficiente.

Hay varios tipos de algoritmos que se han pensado para resolver este asunto en las tablas hash. Unos se llaman abiertos, y se centran en aprovechar los huecos de la tabla que actualmente están vacías para dejar ahí un objeto que haya colisionado (deba ir a una posición que ya está ocupada). El más sencillo de este tipo se denomina de prueba lineal. Y es el que veremos ahora.

Básicamente consiste en ir recorriendo una tras otra las celdas contiguas a la que le correspondería (de forma circular, cuando llega a la última del array continuará buscando hueco por la primera). Pero claro, aquí hay que considerar si la tabla admite repeticiones de la clave o no. Por simplicidad, vamos a suponer que no, cada clave solo puede tener un valor asociado.

Veamos un ejemplo: Imagina que insertamos la clave “Santi” cuyo hash es 42. En la celda 42 almacenados los datos con clave "Santi". Ahora queremos insertar “Juan” y desafortunadamente obtenemos también 42. El código comprueba si la celda está vacía (a null) y ve que no, así que debe buscar otro hueco libre. Prueba 43, 44, …. Bueno pues va probando hasta encontrar uno libre o llegar de nuevo a 42 (estaría llena, factor de ocupación a 1) con las posiciones (42 + i) % n. De ahí el nombre del algoritmo, vas probando (viendo si está vacía) cada posición

consecutiva (lineal). Así que ocupará la posición 43.

Si al insertar resolvemos así el localizar un hueco, la operación de buscarlo debe seguir el mismo procedimiento. Imagina que buscamos en la tabla a "Miguel" que tiene un hash de 43. Vamos a la celda 43, y vemos que ahí no está Miguel, sino "Juan", así que seguimos buscando hasta que encontremos al que buscamos o el primer hueco. Si encontramos antes un hueco ¿podemos parar?

Veamos este sencillo caso. Una tabla con 5 huecos (slots) donde insertamos las asociaciones (clave, valor) siguientes:

("A", VA) hash("A")= 2
("B", VB) hash("B") = 2
("C", VC) hash("C")= 2

0	null
1	null
2	A
3	B
4	C

Si quitamos la clave asociación con clave "B" y dejamos un null:

```
Localizamos “B“ calculando el
hash (2) y buscando linealmente
si no está...se encuentra en la
posición 3, se deja un null y
se devuelve.
```

0	null
1	null
2	A
3	null
4	C

Ahora buscamos el dato con clave "C":

```
Localizamos “C“ calculando el
hash (2) y buscando linealmente
si no está hasta el primer
hueco sin usar...no se
encuentra y en la posición 3
hay null así que paramos!!
```

0	null
1	null
2	A
3	null
4	C

El problema es que cuando se insertó "C" no había huecos, pero luego se creó un hueco. ¿Cómo solucionarlo? Bueno, si se borra un dato, en

vez de dejar a null el slot, lo que dejamos a null es el dato que contiene, pero ponemos un chivato que nos indique que se ha borrado lo que hay (borrado=true). De forma que cuando localicemos una clave mediante prueba lineal, pararemos si encontramos el slot a null, pero no pararemos si encontramos un slot borrado. Así la prueba lineal funciona. Dejamos de buscar si nunca ha habido slot, o si hemos llegado a donde comenzamos a buscar.

Bien pasemos al código: Necesitaremos definir la interface pública de un TDA de tipo TablaHash con prueba lineal:

```
Interface TablaHashPL<K, V>
long hash(K k);
boolean inserta(K k, V v);// Inserta/cambia v asociado a clave k
V borra(K k);          // quita dato de clave k y lo devuelve/null
V busca(K k);          // dato v si tiene clave k/null en otro caso
void vaciar()          // Borra todos los elementos
boolean estaVacia();   // true si no tiene elementos
int cuantos();         // Nº de elementos almacenados
// Podríamos añadir iteradores, vistas, etc...
```

Además necesitamos definir una clase genérica que represente un slot de la tabla con sus datos, constructor, toString, getters y setters:

```
class Slot<K,V>
    K clave;
    V valor;
    boolean borrada; // Si true, tendremos q. Seguir localizando
```

La clase debe seguir la interfaz y a parte de un constructor debería tener algunos métodos internos (privados) que no ofrezca al exterior:

```
Class TablaHashPruebaLineal<K, V>
   long ne             // Nº de elementos
   long n;             // Tamaño de array de slots. Un nº Primo
   Slot[] t;           // Tabla de Slots de tamaño n.

   Constructor(int tama); // Crea array t con n >= tama y n primo
   // Métodos de uso interno (Privados)
```

```
    long siguientePrimo(long p);
    void extiende();   // Si 3 * ne > 2 * n aumenta tamaño de t
                       // al siguiente primo, recalcula hashes para
                       // reconstruir la nueva tabla hash
    long localiza(K k);       // devuelve posición donde debería
                              // estar k (puede ser slot vacío o
                              // borrado o donde está)
    // Métodos públicos: los de la interfaz y toString()
```

6.5.5 IMPLEMENTAR SOPORTE DE SENTENCIA FOR-EACH

El bucle for each en Java tiene esta sintaxis:

```
for( T variable : colección<T>) {
    // cuerpo del bucle
}
```

Que se transforma internamente en un bucle "normalito" (observa que tiene anulado el incremento, subrayado en amarillo):

```
for( Iterator<T> ite = new coleccion.iterator(); ite.hasNext();) {
    T variable = ite.next();
    // Cuerpo del bucle
}
```

Si quieres que tu colección permita recorrer sus elementos de tipo T con el bucle for-each, debe implementar la interface Iterable<T>

```
public class MiColeccion<T> implements Iterable<T> {
    // ... Otros elementos...
    public int size() {  /* nº de elementos */ }
    public T get(int i) {  /* el elemento i */ }
    public Iterator<T> iterator() { return new MiIterador(); }
    public class MiIterador implements Iterator<T> {
        private int actual = 0;
        public boolean hasNext() { return actual < size(); }
        public T next() {  return get(actual++); }
        public void remove(){
            throw new UnsupportedOperationException( ":-(" );
        }
    }
```

```
}
```

EJEMPLO 29: Para un String, permitir recorrer sus caracteres con for-each.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

class StringIterable implements Iterable<Character> {
  private String s;
  private int cont = 0;

  public StringIterable(String s) {  this.s = s;  }

    // Implementar Iterator.
    class StringIterator implements Iterator<Character> {
      public boolean hasNext() { return cont < s.length();  }
      public Character next() {
        if(cont == s.length()) throw new NoSuchElementException();
        cont++;
        return s.charAt(cont - 1);
      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
    }

    // Implementar Iterable
    public Iterator<Character> iterator() {
      return new StringIterator();
   }
}

public class MainClass {
  public static void main(String args[]) {
    StringIterable x = new StringIterable( "Es una prueba ;-)" );
    for(char ch : x){
      System.out.println(ch);
    }
  }
}
```

6.6. EJERCICIOS.

TEST

T1. ¿Cuál es el mayor beneficio que aportan los genéricos?

- Hacen el código más rápido.
- Hacen que el código sea más optimizado y entendible.
- Añaden estabilidad al código al detectar más errores en tiempo de compilación.
- Añaden estabilidad al código al detectar más errores en tiempo de ejecución.

T2. Cuales de estos tipos de parámetros se suele usar para que una clase genérica acepte cualquier tipo de objeto?

- V
- T
- G
- K

T3. ¿Cuales de los siguientes tipos de referencias no pueden ser genéricos?

- Clases inner anónimas
- Interfaces
- Clases inner.
- Todos los mencionados.

T4. ¿Qué son los métodos genéricos?

- Los que no tienen parámetros.
- Los definidos en una clase genérica.
- Los que exteienden métodos de una clase genérica.
- Los que introducen sus propios parámetros de tipo.

T5. ¿Cuál de estos tipos de datos no puede tener un tipo

parametrizado?

- Array
- Set
- List
- Map

T6. ¿Cuales de estas clases que implementan colecciones estándar usa arrays dinámicos?

- ArrayList
- AbstractSet
- AbstractList
- LinkedList

T7. ¿Map implementa la interface Collection?

- Falso
- Verdadero

T8. ¿Cuáles de estos métodos puede utilizarse para aumentar la capacidad de un objeto ArrayList de manera manual?

- ensureCapacity()
- increaseCapacity()
- increasecapacity()
- Capacity()

T9. ¿Cuales de estos métodos de la clase ArrayList se usa para obtener la cantidad de elementos actual de uno de esus objetos?

- capacity()
- length()
- size()
- index()

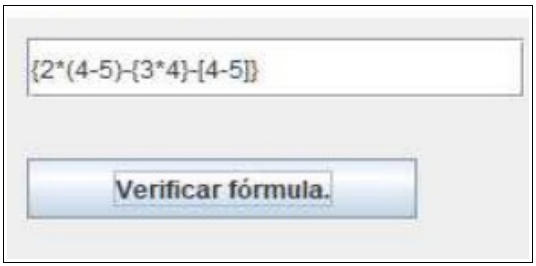
T10. ¿Cuales de estos métodos pueden utilizarse para obtener un array estático con los elementos de un ArrayList?

- Array()
- covertArray()
- toArray()
- covertoArray()

EA1. Define un objeto de la clase genérica java.util.Stack que almacene datos String y añades "Juan", "Ana" y "Luis". Luego muestra:

- Cantidad de elementos en la pila:
- Extrae un elemento.
- Cantidad de elementos en la pila.
- Consulta sin extraer el primer elemento.
- Cantidad de elementos en la pila.
- Extrae uno a uno cada elemento y lo imprimes mientras la pila no esté vacía.

EA2. Usa una pila para comprobar si una expresión que utiliza (), {} y [] para agrupar expresiones está bien o mal balanceada.



Ejemplo de fórmula: (2+[3-12]*{8/3})

Un algoritmo de validación consiste en ayudarnos de una pila e ir

examinando carácter a carácter la presencia de los paréntesis, corchetes y llaves.

- Si vienen símbolos de apertura, los almacenamos en una pila.
- Si vienen símbolos de cierre, extraemos de la pila y verificamos si está el símbolo equivalente de apertura: en caso negativo podemos inferir que la fórmula no está correctamente balanceada.
- Si al finalizar el análisis del último caracter de la fórmula la pila está vacía podemos concluir que está correctamente balanceada.

Ejemplos de fórmulas no balanceadas:

- }(2+[3-12]*{8/3})
- {[2+4}]

EA3. Una lista se comporta como una cola si las inserciones las hacemos al final y las extracciones las hacemos por el frente de la lista. También se las llama listas FIFO (First In First Out, primero en entrar primero en salir). Confeccionaremos un programa que permita utilizar la interfaz Queue y mediante la clase LinkedList administre la lista tipo cola.

EA4. Una cola con prioridad es una variante de una cola clásica. La implementa la clase PriorityQueue. Cuando se agregan elementos a la cola se organizan según su valor, por ejemplo, si es un número se ordenan de menor a mayor. Crea una cola con prioridad de enteros, añade los valores 70, 120 y 6 y luego muestra los valores que tiene mientras no esté vacía.

Vamos a usar una clase Persona para hacer colecciones de sus objetos:

```
public class Persona {
    private int dni;
    private String nombre;
    private int altura;

    public Persona(int dni, String nombre, int altura) {
        this.dni = dni;
        this.nombre = nombre;
        this.altura = altura;
    }

    public int getAltura(){ return altura; }
    // Resto omitido... pero para simplificar el ejercicio
    @Override
    public String toString() {
        return "Persona-> DNI: " + dni + " Nombre: " + nombre +
               " Altura: "+altura+"\n";
    }
}
```

EA5. Usar ArrayList. Haz una clase en Java que permita almacenar una lista de personas de longitud arbitraria a la que podremos dar un nombre y que nos sirva para almacenar una lista de pacientes de una consulta por ejemplo. Debe permitir conocer cuantos elementos hay, añadir un elemento por el final, obtener el elemento de una posición, borrar el elemento de una posición. Utiliza la clase ArrayList para almacenar las Personas.

EA6. La clase Vector. Aprende como usar la clase Vector y repite el ejercicio anterior con esta Colección. Puedes leer estas notas:

EA7. Interface Iterator. Imagina que queremos eliminar de la lista del ejercicio 1, las personas cuya estatura sea menor de un determinado valor. Hazlo definiendo una clase Iterator interna a la clase ListaPersonas.

EA8. Interface Comparable. Vamos a dar a nuestra clase Persona la posibilidad de comparar unas personas con otras, comprometiéndola a

implementar la interface apropiada y sobreescribiendo el método **public int compareTo(Persona p);** Si tuviéramos 2 Personas y quisiéramos compararlas u ordenarlas lo podríamos hacer por ejemplo por su altura, o por su edad, la pregunta a hacerse sería cuándo una persona es más grande que otra o cuándo es menor, o cuándo son iguales atendiendo a algún tipo de atributo. En nuestro ejemplo nos vamos a inventar esta ordenación y vamos a decir que una persona es más grande que otra si tiene mayor altura, será menor si tiene una altura menor y serán iguales si y solo si su altura es igual y el dni también es igual.

EA9. El uso de métodos de Java 9 Factory nos ayuda a trabajar de forma más cómoda con la inicialización de colecciones. En la mayoría de casos, si tenemos que trabajar con colecciones siempre hacemos operaciones como las siguientes a la hora de inicializarlas.

```
package p9;

import java.util.ArrayList;

public class E9A {

    public static void main(String[] args) {
        ArrayList<String> lst = new ArrayList<>();
        lst.add("hola");
        lst.add("que");
        lst.add("tal");
        lst.add("estas");
        for(String s : lst) { System.out.println(s); }
    }
}
```

Este tipo de operaciones son tediosas. Para mejorr la forma de inicializar se puede usar la clase Arrays y su método **asList()**.

a) Modifica el código anterior para inicicializar la lista de esta forma.

Sin embargo esto solo nos sirve para las listas y existen otros tipos de colecciones como Set y Map. A partir de Java 9 puedes usar los Factory Methods. Podemos hacer uso del método **of()** de la interface List y automáticamente podremos construir una lista. Del mismo modo podemos aplicar esta solución a los Sets y los Maps (pero una lista de claves y valores).

b) Crea la lista del ejemplo usando of.

c) Crea un Set con las palabras del ejemplo.

d) Crea un mapa de (enteros, String) con los elementos (1,"Java"), (2,"JavaScript") y (3,"Kotlin").

EA 10: Las hojas de cálculo tienen funciones para contar valores que hay en cierto grupo de celdas:

- contar(): cuenta la cantidad de celdas con valores.
- contarSI(condición): cuenta los valores que cumplen una condición.

Define la clase llamada Contador que implemente un método estático y genérico llamado `contarSI()` que acepta dos parámetros: el primero cualquier colección de objetos y el segundo una condición que deben cumplir los objetos dentro de la colección para ser contados. El método devuelve el número de objetos que cumplen la condición. Prueba el método en la propia clase con dos colecciones distintas y dos criterios.