8. Colecciones e iteradores

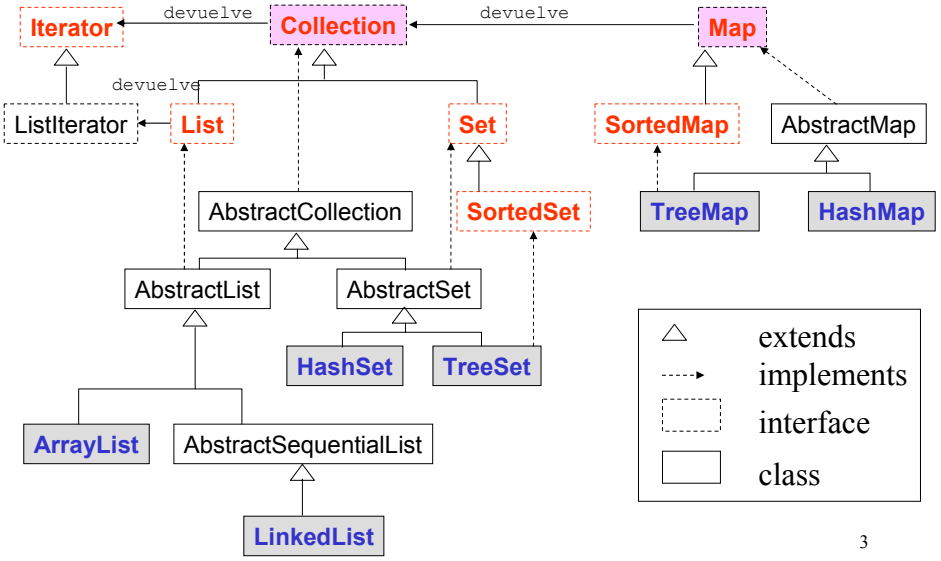
- interfaz `Collection`
  - clases `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`
- interfaz `Map`
  - clases `TreeMap`, `HashMap`
- Iteratores: interfaz `Iterator`

Colecciones en Java

- Permite *almacenar* y organizar *objetos* de manera útil para un acceso eficiente.
- Se encuentran en el paquete **`java.util`**
- Núcleo de abstracciones de colecciones de utilidad (interfaces) e implementaciones ampliamente útiles.
- Las interfaces proporcionan métodos para todas las operaciones comunes y las implementaciones concretas especifican la decisión de las operaciones no permitidas.

  (`java.lang.UnsupportedOperationException`)
- Sobre los elementos se puede iterar (**`Iterator`**)

Jerarquía de colecciones



Interfaz Collection (1/2)

- **Collection**
  - int **size**()
  - boolean **empty**()
  - boolean **contains**(Object elem)
  - Iterator **iterator**()
  - Object[] **toArray**(), Object[] **toArray**(Object dest[])
  - boolean **add**(Object elem),
  - boolean **remove**(Object elem)
  - void **clear**()

- **List** – Una colección cuyos elementos permanecen en un orden particular a menos que se modifique la lista (no significa lista enlazada aunque es una posible implementación).
  - void **add**(int index, Object element)
  - Object **remove**(int index)
  - Object **get**(int index)
  - Object **set**(int index, Object element)
  - int **indexOf**(Object o)
  - int **lastIndexOf**(Object o)
  - List **subList**(int min, int max)

Interfaz Collection (2/2)

-Set – Una colección (conjunto) donde no puede haber elementos repetidos, y cuyos elementos no se almacenan necesariamente siguiendo un orden particular.

- Mismos métodos que `Collection` con otro contrato.

-SortedSet – Conjunto con elementos ordenados.

```
-Object first()
-Object last()
-SortedSet subSet(Object fromElement, Object toElement)
-SortedSet headSet(Object toElement)
-SortedSet tailSet(Object fromElement)
```


Interfaz Map

- **Map**
  - Un objeto que asocia claves con valores.
  - No puede tener claves duplicadas.
  - `Object put(Object key, Object value);`
    `Object remove(Object key); Object get(Object key);`
  - **containsKey, containsValue, isEmpty, size**
  - Proporciona tres vistas de colección: colección de claves (**keySet**), colección de valores (**values**), colección de asociaciones clave-valor (**entrySet**).
- **SortedMap:** Un mapa cuyas claves están ordenadas.
  - `Object firstKey(), Object lastKey(), SortedMap subMap(Object minKey, Object maxKey), SortedMap headMap(Object maxKey), SortedMap tailMap(Object minKey)`

Iteración

- `Collection >> Iterator iterator();`

```
interface Iterator{
   boolean hasNext();
   /* Devuelve true si la iteración tiene mas elementos */
   Object next();
   /* Devuelve el siguiente elemento de la iteración
   Lanza excepción NoSuchElementException */
   void remove();
   /* Elimina el último elemento devuelto por la iteración
   Está capacitado para decir que no lo implementa
   UnsupportedOperationException */
}
```

- La interfaz **`ListIterator`** extiende a `Iterator` y maneja un objeto `List` ordenado. Permite iterar hacia delante y hacia atrás.

Ejemplo de uso de Iteradores

- Cálculo del gasto total de un departamento

```
public double gastoDpto(){
   double gasto=0;
   Iterator it=plantilla.iterator();
   while (it.hasNext()){
       gasto+=((Empleado)it.next()).getSueldo();
   }
   return gasto;
}
```

Siendo plantilla una colección que implemente la interfaz `Collection`

Implementaciones de `Collection`

-**`LinkedList`** – Una implementación de una lista doblemente enlazada. La **modificación** es **poco costosa** para cualquier tamaño, pero el **acceso aleatorio** es **lento**. Útil para implementar colas y pilas.

```
-getFirst, getLast, removeFirst, removeLast, addFirst, addLast
```

-**`ArrayList`** – Una lista implementada utilizando un array de dimensión modificable. Es **costoso añadir o borrar** un elemento cerca del principio de la lista si ésta es grande, pero es relativamente poco costoso de crear y **rápido** para **acceso aleatorio**.

-**`HashSet`** – Un `Set` implementado mediante una tabla *hash*. Es una buena implementación de propósito general por lo que la búsqueda, la adición y eliminación son **insensibles al tamaño** de los contenidos.

-**`TreeSet`** – Un `SortedSet` implementado utilizando un árbol binario equilibrado. Es **más lento para buscar o modificar que un `HashSet`**, pero mantiene los elementos ordenados. Asume que los elementos son *comparables* si no se le ha pasado un *comparator* en el constructor.

-Todas son **`Cloneable`** y **`Serializable`**

Convenciones sobre excepciones

- **`UnsupportedOperationException`**
  - Métodos opcionales en la implementación de una interfaz
- **`ClassCastException`**
  - El tipo del elemento que se desea insertar no es del tipo apropiado
- **`IllegalArgumentException`**
  - El valor del elemento no es apropiado para la colección
- **`NoSuchElementException`**
  - La colección de la que se quiere devolver un elemento está vacía
- **`NullPointerException`**
  - Se pasa como argumento una referencia con valor `null` cuando la colección no admite este valor.

Declaración de colecciones

Clase concreta

```
import java.util.*;

public class ColeccionSimple {
    public static void main( String args[] ) {
        List c = new ArrayList();
        for( int i=0; i < 10; i++ )
            c.add(new Integer(i));

        Iterator it = c.iterator();
        while( it.hasNext() )
            System.out.println(it.next());
    }
}
```

interfaz

Implementaciones de Map

- **HashMap**
  - Una implementación de `Map` con una *tabla hash*.
  - El método `hashCode` de cada clave se utiliza para seleccionar un lugar en la tabla
  - Una colección de utilidad muy general con tiempos relativamente cortos de búsqueda e inserción.

- **TreeMap**
  – Una implementación de `SortedMap` utilizando un árbol binario equilibrado que mantiene sus elementos ordenados por clave.
  – Útil para conjuntos de datos ordenados que requieren una búsqueda por clave moderadamente rápida.
  -Asume que los elementos son *comparables* si no se le ha pasado un *comparator* en el constructor.

Ejemplo 1/2

- Generar números al azar (`Math.random`) y contar cuantas veces sale cada uno.
- `HashMap` = Colección de pares (clave-valor)
  - Clave = número aleatorio generado
  - Valor = contador que acumula las veces que ha aparecido

```
class Contador {
     private int i;
     public Contador(){ i=1;}
     public void incrementar(){++i;}
     public String toString() {
        return Integer.toString(i);
     }
}
```


Ejemplo 2/2

```
class Estadistico {
    public static void main( String args[] ) {
        HashMap tabla = new HashMap();

        for(int i=0; i < 10000; i++) {
            // Generar un número entre 0 y 20
            Integer num = new Integer((int)(Math.random()*20));

            if(tabla.containsKey(num))
                 //Incrementamos el contador asociado al número
                 ((Contador)tabla.get(num)).incrementar();
            else
                //Añadimos nuevo par: numero-contador
                 tabla.put(num, new Contador());
            }
        System.out.println(tabla);
    }
}
```

Las utilidades de `Collections`

- `public static Object` **`min`**`(Collection col)`
- `public static Object` **`max`**`(Collection col)`
- `public static Object` **`min`**`(Collection col, Comparator comp)`
- `public static Object` **`max`**`(Collection col, Comparator comp)`
- `public static void` **`reverse`**`(List lista)`
- `public static void` **`copy`**`(List dst, List fnt)`
- `public static void` **`sort`**`(List lista)`
- `public static void` **`sort`**`(List lista, Comparator comp)`
- `public static int` **`binarySearch`**`(List lista, Object clave)`
- `public static int` **`binarySearch`**`(List lista, Object clave, Comparator comp)`

Conclusiones

- Si un método tiene que devolver (pasar como parámetro) una colección de objetos, el tipo será `Iterator` o cualquiera de las interfaces de colección.
- El tipo de la declaración de los atributos y variables locales será cualquiera de las interfaces de colección.
  - `List lista = new ArrayList();`
  - Excepción: `LinkedList` si la utilizamos como pila o cola.
- Utilizar SIEMPRE `Iterator` para el recorrido de cualquier colección.