

Índice

- Interfaz vs. Implementación
- Estructuras de datos
- Elementos de Java Collections Frameworks (JCF)
- Interfaces Collection e Iterator
- Interfaces Set, List y ListIterator
- Interfaz Map
- Interfaces Comparable y Comparator: clases con orden
- Clases Collections y Arrays
- Tipos genéricos

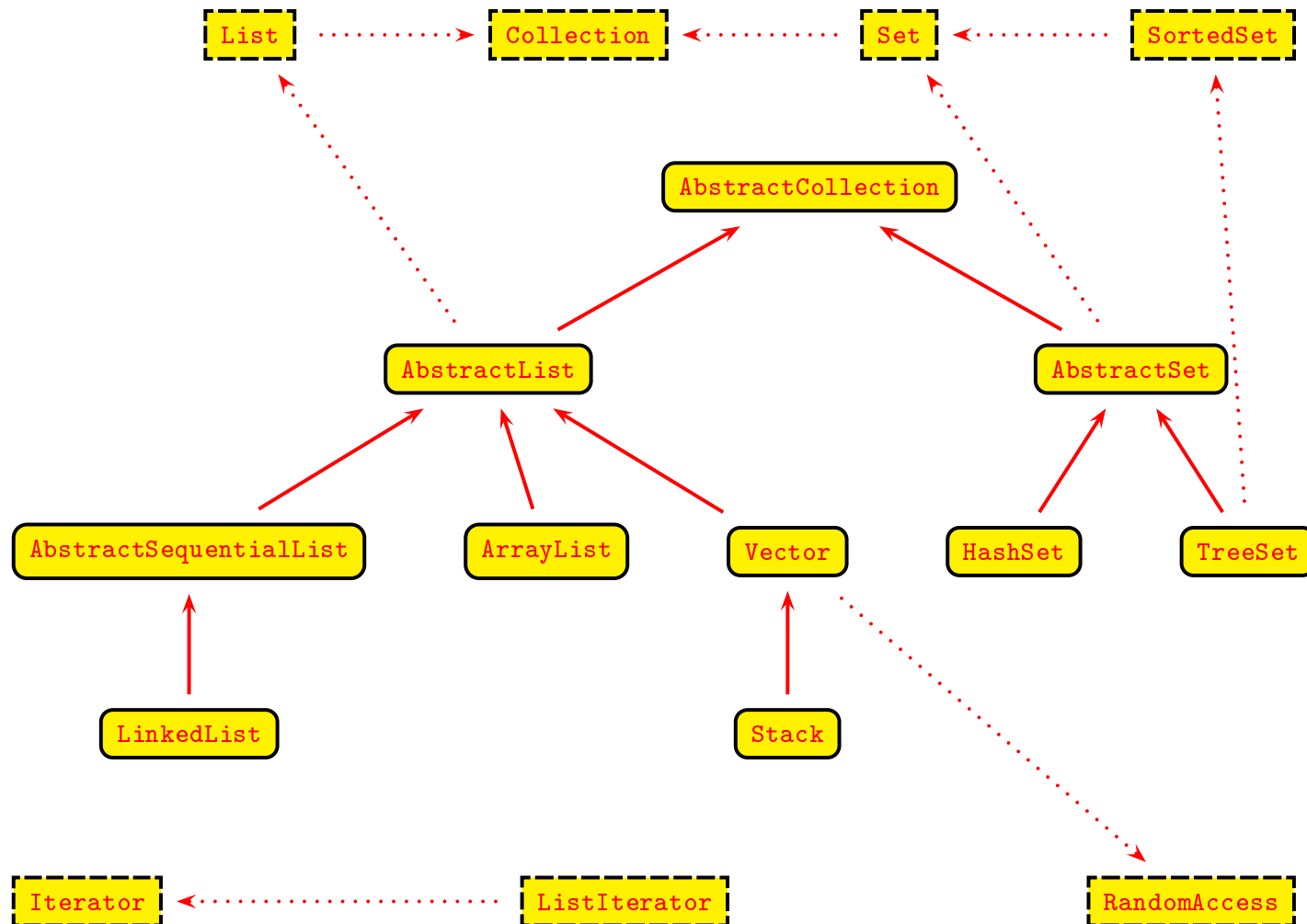
Estructuras de Datos

- **Lineales:** pilas, colas, listas, secuencias
- **Arborescentes:** árboles generales, árboles binarios, árboles binarios de búsqueda, árboles equilibrados (AVL, 2-3, 2-3-4, rojinegros, B), montículos
- **Funcionales:** tablas (dispersas y ordenadas)
- **Relacionales:** estructuras de partición, grafos

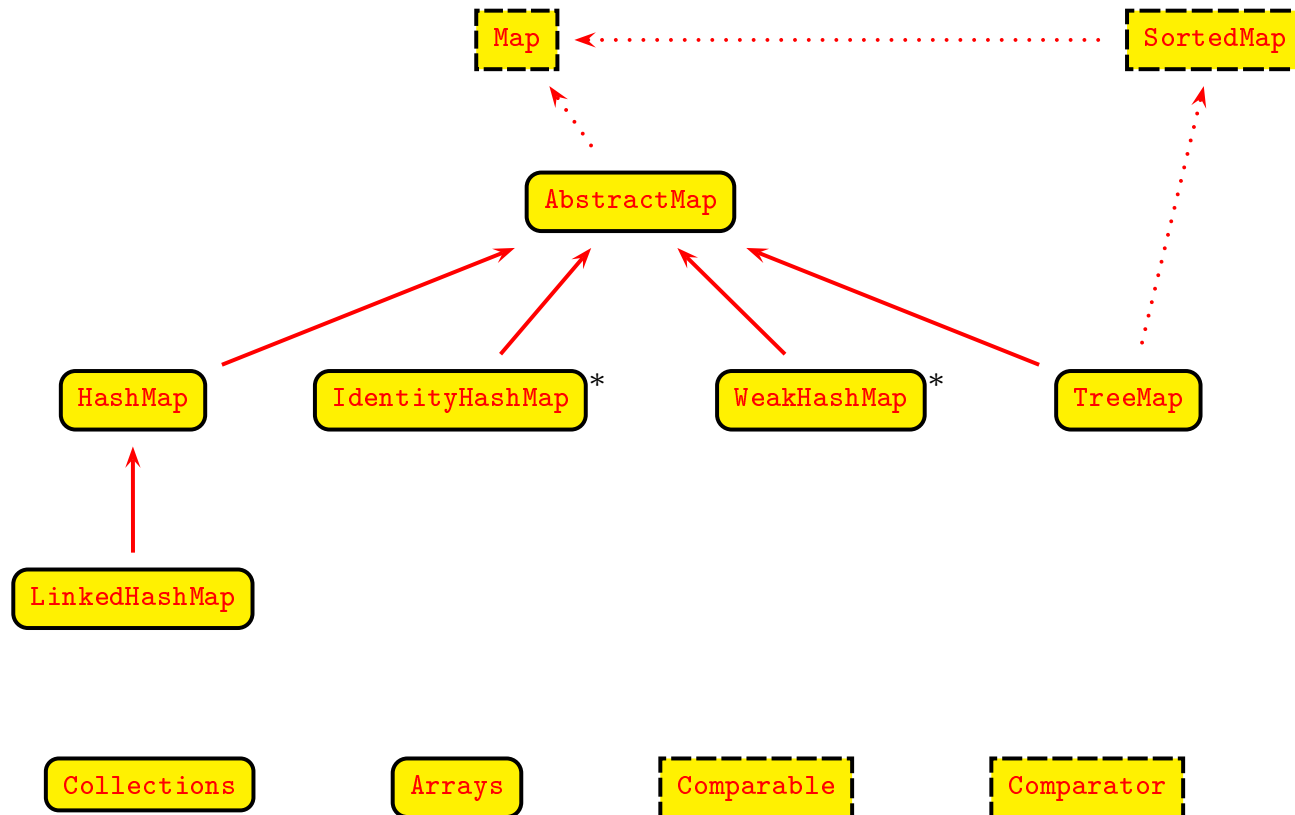
Java Collections Framework

- **Interfaces:** manipulación independiente de la implementación
- **Clases abstractas:** implementan parcialmente una interfaz. Sirven de base para crear nuestras propias clases.
- **Implementaciones** concretas de las interfaces. Dos clases que implementan la misma interfaz se usan del mismo modo, solo se diferencian en su implementación. Ej.: `ArrayList` y `LinkedList`.
- **Algoritmos:** realizan de manera eficiente tareas habituales como búsquedas, ordenación etc.

Jerarquía Collection



Jerarquía Map



Interfaz Collection

- Se puede ver como un **multiconjunto**.
- `boolean add(Object obj)`: devuelve true si ha modificado la colección, es opcional (ojo: `UnsupportedOperationException`)
- `boolean contains(Object obj)`: devuelve true si contiene el elemento
- `void clear()` vacía la colección (opcional)
- `boolean isEmpty()`: comprueba si tiene elementos
- `boolean remove(Object obj)`: elimina una aparición del elemento (opcional)
- `boolean retainAll(Collection c)`: conserva los elementos de la colección argumento
- `Object[] toArray`: devuelve un vector con los objetos
- Versiones con colecciones: `containsAll`, `addAll`, `removeAll`

Interfaz Iterator

- Permite recorrer los elementos de una colección y eliminarlos durante el recorrido
- `boolean hasNext()`: devuelve true si queda algún elemento por recorrer
- `Object next()`: avanza y devuelve el elemento que se ha saltado. Lanza `NoSuchElementException` si no hay más
- `void remove()`: elimina el último elemento devuelto por el iterador. Lanza `IllegalStateException` si no se ha hecho antes un `next()`
- Sucesivas llamadas a `next()` permiten **recorrer todos los elementos**

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

Interfaz List

■ Acceso posicional:

- `Object get(int i)`: obtiene el elemento de la posición `i`
- `Object set(int i, Object obj)`: sustituye el elemento de la posición `i` por `obj` (opcional)
- `Object remove(int i)`: elimina el objeto en la posición `i` y devuelve el objeto eliminado (opcional)
- `void add(int i, Object obj)`: añade `obj` en la posición `i` desplazando el resto de los elementos (opcional)

■ Búsquedas:

- `int indexOf(Object obj), int lastIndexOf(Object obj)`: devuelve la primera/última posición donde se encuentra el objeto, -1 si no aparece

- **Recorridos:**

- `ListIterator listIterator()`: devuelve un iterador al principio de la lista
- `ListIterator listIterator(int i)`: devuelve un iterador situado en la posición indicada

- **Generación de vistas:**

- `List subList(int i, int j)`: vista desde i incluido hasta j excluido. Una modificación en la vista se refleja en la original.

Interfaz ListIterator

- Permite recorrer una lista **en los dos sentidos**.
- `boolean hasPrevious()`: devuelve true si hay un elemento delante del iterador
- `Object previous()`: retrocede y devuelve el elemento que se ha saltado
- `int nextIndex(), int previousIndex()`
- `void set(Object obj)`: sobrescribe el último elemento devuelto por `next()` o `previous`. Lanza una `IllegalStateException` si la estructura de la lista se ha modificado desde la última llamada a `next()` o `previous`.
- `void add(Object o)`: añade un elemento delante del iterador

Implementaciones de List

- ArrayList: con un **vector dinámico de objetos**. Acceso posicional eficiente, pero las inserciones y eliminaciones ineficientes.
- LinkedList: con una **lista doblemente enlazada**. Acceso posicional ineficiente, flexibilidad de crecimiento, inserción y eliminación eficientes. Tiene métodos adicionales de acceso a los extremos:
 - `void addFirst(Object obj), void addLast(Object obj)`
 - `Object getFirst(), Object getLast()`
 - `Object removeFirst(), Object removeLast()`

Interfaz Set

- Colección **sin elementos repetidos**.
- No tiene métodos adicionales a los de `Collection`, pero al intentar añadir un elemento que ya está en la colección se rechaza la petición.
- Por ello es importante saber cuando dos elementos son iguales: hace falta definir adecuadamente `equals` (y `hashCode` de forma coherente).
- Usando los métodos de `Collection` se pueden llevar a cabo las operaciones habituales de los conjuntos:
 - `c1.containsAll(c2)` corresponde a $s2 \subseteq s1$
 - `c1.addAll(c2)` corresponde a $s1 = s2 \cup s1$
 - `c1.retainAll(c2)` corresponde a $s1 = s2 \cap s1$
 - `c1.removeAll(c2)` corresponde a $s1 = s1 \setminus s2$

Clase HashSet

- Implementación de conjuntos mediante **tablas dispersas abiertas** (hash).
- El constructor por defecto construye una tabla de 101 posiciones y factor de carga límite 0,75 para redimensionar, pero se puede modificar:
`HashSet(int cap, float factor)`
- El iterador visita los elementos de forma aparentemente aleatoria, ya que no hay control del orden en que se encuentran los elementos.
- La clase `LinkedHashSet` mantiene el orden en que se añaden los elementos al conjunto.
- La función de dispersión (hash) viene dada por el método `int hashCode()` definido en la clase `Object`. Mejor redefinirlo para nuestras clases.
- Debe ser coherente con `equals`: si `x.equals(y)` entonces `x.hashCode() == y.hashCode()`.

Interfaz Map

- Estructura de datos agrupados en parejas clave/valor. La clave es única y se usa para acceder al valor.
- Operaciones básicas:
 - `Object put(Object o1, Object o2)`: añade una pareja clave/valor. Si la clave ya está se sustituye el valor. Devuelve el valor anterior.
 - `Object get(Object o)`: devuelve el valor asociado a una clave, `null` si no está.
 - `Object remove(Object o)`: elimina la pareja con la clave dada y devuelve el valor asociado.
 - `boolean containsKey(Object o)`: devuelve `true` si la clave está.
 - `boolean containsValue(Object o)`: devuelve `true` si hay alguna clave con ese valor asociado.

- **Vistas de un Map:** con las vistas se pueden eliminar elementos en el Map, pero no añadirlos
 - `Set keySet()`: devuelve el conjunto de claves
 - `Collection values()`: colección de valores guardados
 - `Set entrySet()`: conjunto de pares clave/valor. Sus elementos son objetos de la interfaz `Map.Entry` que dispone de los métodos:
 - `Object getKey(), Object getValue()`: obtiene la clave/el valor
 - `Object setValue(Object o)`: modifica el valor asociado.
- La clase `HashMap` implementa `Map` utilizando tablas dispersas.
- La clase `WeakHashMap` coopera con el recolector de basura para eliminar aquellas parejas que solamente están referenciadas desde la tabla.

Interfaces Comparable y Comparator

- Comparable declara el método `int compareTo(Object o)`: devuelve un entero negativo, 0 o positivo según `this` sea menor, igual o mayor que `o`
- Las clases `String`, `Character`, `File`, `Integer` ... la implementan. Permite definir un orden natural.
- El método `compareTo` debe ser coherente con `equals` y cumplir la propiedad transitiva.
- `Comparator` se utiliza cuando se quieren definir otros órdenes distintos del natural, definiendo el método `int compare(Object o1, Object o2)`.
- El programador debe proporcionar siempre una implementación de esta interfaz.
- Los objetos de las clases que la implementan se pueden usar como argumentos en los métodos de ordenación y búsqueda, o en los constructores de las clases `TreeSet` y `TreeMap`.

Interfaz SortedSet

- Extiende a Set y define los métodos:
 - `Comparator comparator()`: devuelve el objeto que establece el orden.
 - `Object last()`, `Object first()`: el primero y el último en el orden indicado
 - `SortedSet headSet(Object o)`, `SortedSet tailSet(Object o)`: para obtener subconjuntos al principio y al final.
 - `SortedSet subSet(Object o1, Object o2)`: para obtener un subconjunto en el medio. No se incluye el límite superior.
- La clase TreeSet implementa esta interfaz. Puesto que tanto HashSet como TreeSet disponen de constructores con argumento de tipo Collection permite la conversión entre las dos implementaciones.
- TreeSet está implementada internamente como un árbol rojinegro.

Interfaz SortedMap

- Extiende a Map y añade los siguientes métodos:
 - `Comparator comparator()`: devuelve el objeto que establece el orden.
 - `Object firstKey(), Object lastKey()`: devuelve la primera y última clave en el orden establecido
 - Vistas:

```
SortedMap headMap(Object o)
SortedMap tailMap(Object o)
SortedMap subMap(Object o1, Object o2)
```
- En este caso **el orden se establece en base a las claves**.
- La clase `TreeMap` implementa esta interfaz.

Clases Collections y Arrays

■ Algoritmos:

- `void sort(List l, Comparator c)`: ordenación por mezclas
- `void shuffle(List l)`: eliminación aleatoria del orden.
- `void reverse(List l)`: inversión del orden.
- `int binarySearch(List l, Object o)`: búsqueda binaria.
- `void copy(List l1, List l2)`: copia.
- `void fill(List l1, Object o)`: sustituye todo el contenido por o.
- `Object max/min(Collection c, Comparator c)`: máximo/mínimo de una colección.

■ Utilidades:

- `Set singleton(Object o)`: conjunto unitario.
- `List nCopies(int i, Object o)`: crear una lista con i copias de o.
- Constantes: `EMPTY_SET`, `EMPTY_LIST`, `EMPTY_MAP`.

- Conjuntos de métodos para convertir colecciones en **inmutables**:

`List unmodifiableList(List l)`

de forma que solamente se pueden leer, si no se lanza `UnsupportedOperationException`.

- o en colecciones **sincronizadas**, de forma que puedan acceder varias hebras:

`List synchronizedList(List l)`

- La clase `Arrays` dispone del método `List asList(Object[] a)` para generar una vista del vector como lista inmutable.

Tipos genéricos

- Enriquecen el sistema de tipos eliminando conversiones. Están parametrizados por variables de tipo

```
public interface List<E>
{ void add(E elemento);
  Iteraror<E> iterator();}
```

que se instancian según el contexto:

```
List<Integer> lent = new LinkedList<Integer>();
lent.add(new Integer(0));
```

- Todas las apariciones de las variables de tipo se sustituyen por los argumentos actuales.
- Los tipos genéricos se compilan una sola vez generándose un único fichero .class

Subtipado y Comodines

- En general si SubTipo es subtipo de Tipo entonces G<SubTipo> no es subclase de G<Tipo>

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = listaString;
```

produce un error de compilación

- La superclase de todas las colecciones es Collection<?>. No se pueden añadir objetos a ella excepto null porque se desconoce el tipo de los elementos que contiene.
- Podemos **comodines acotados** para indicar que deseamos que un método acepte colecciones de determinadas clases de elementos:

```
public void ejemplo(List<? extends/super Clase> l)
```

Es incorrecto añadir elementos a una colección cuyo tipo incluye comodines porque se desconoce el tipo que almacena, pero se pueden invocar los métodos de la cota superior de forma segura.

- Los **métodos genéricos** deben prefijar sus firmas con la declaración de las variables de tipo

```
public static <T> void copy(List<T> destino,  
                             List<? extends T> origen)
```

- Cuando un tipo genérico se emplea sin sus parámetros correspondientes, se dice que es un tipo “crudo” (raw) y la comprobación de tipos es menos estricta.
- **Limitaciones de los tipos genéricos:**
 1. Las variables de tipo no pueden sustituirse por tipos primitivos.
 2. Las variables de tipo no pueden emplearse en los miembros estáticos de la clase.
 3. Las variables de tipo no pueden emplearse con el operador instanceof y no deben usarse para hacer conversiones de tipo.
 4. Las variables de tipo no pueden emplearse en operaciones de creación de objetos ni en cláusulas extends o implements.