ANEXO I: Resumen de uso de colecciones

Este apartado muestra un resumen de uso de las colecciones.

Es importante destacar que para utilizar las colecciones con tipos primitivos debemos utilizar el tipo envoltorio asociado al tipo de datos (`Integer` para `int`, `Double` para `double`, etc.). Así, por ejemplo, una lista de enteros se declara con el tipio `LinkedList<Integer>`.

Listas: `LinkedList`

La documentación de la clase `java.util.LinkedList` está disponible en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html o línea en Eclipse.

```
// > Construcción de una lista vacía

LinkedList<String> lista = new LinkedList<String>();

// > Añadir elementos

// Añadir elementos por el final

lista.add("uno"); // Opción 1
lista.addLast("dos"); // Opción 2

// Añadir un elemento al principio

lista.addFirst("cero");

// > Consultar el tamaño

System.out.println("Tamaño: " + lista.size());
System.out.println("¿Está vacía? " + lista.isEmpty());

// > Consultar elementos

// el primer elemento

String primero = lista.get(0); // Opción 1
primero = lista.getFirst(); // Opción 2

System.out.println(primero);

// el último elemento

String ultimo = lista.get(lista.size() - 1); // Opción 1
ultimo = lista.getLast(); // Opción 2

System.out.println(ultimo);

// consultar si contiene un elemento

System.out.println("¿Contiene 'cuatro'? " + lista.contains("cuatro"));

// > Reemplazar un elemento

lista.set(1, "nuevo");
```

```java
// > Crear una copia

LinkedList<String> copia = new LinkedList<String>(lista);

// > Borrado

lista.removeFirst(); // el primero
lista.removeLast(); // el último

// retorna si ha podido eliminarlo

boolean resultado = lista.remove("tres");
System.out.println("¿Borrado? " + resultado);

// > Recorridos

// Recorrido for each

for (String elemento : copia) {
      System.out.println("Elemento: " + elemento);
}
```

Conjuntos: HashSet

La documentación online está disponible en
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html

```
// > Creación de un conjunto vacío

HashSet<String> conjunto = new HashSet<String>();

// > Añadir elementos

conjunto.add("uno");
conjunto.add("tres");
conjunto.add("dos");
conjunto.add("cinco");

// El método add retorna un booleano indicando si el elemento
// es aceptado en el conjunto

// Un conjunto EVITA REPETIDOS

boolean insertado = conjunto.add("dos");
System.out.println("¿Insertado? " + insertado);

// > Consultar el tamaño

System.out.println("Tamaño: " + conjunto.size());
System.out.println("¿Es vacío? " + conjunto.isEmpty());

// > Consultar si contiene un elemento

System.out.println("¿Contiene 'cuatro'? " + conjunto.contains("cuatro"));

// > Borrado

// El método remove intenta borrar el elemento

boolean borrado = conjunto.remove("uno");
System.out.println("¿Borrado? " + borrado);

System.out.println("¿Borrado? " + conjunto.remove("seis"));

// > Recorridos
// No podemos recorrer una colección por índice
// Debemos recorrer con "for each" (iterador)

for (String elemento : conjunto) {
      System.out.println(elemento);
}

// IMPORTANTE: observa que el orden de los elementos no corresponde con la
inserción

// > Copia

HashSet<String> copia = new HashSet<String>(conjunto);

System.out.println(copia);
```

Mapas: HashMap

Un mapa es una estructura de datos que asocia parejas de valores. En cada asociación uno de los valores actúa como clave, el cual permite recuperar el valor asociado o borrar la pareja.

La documentación online está disponible en
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html

```
// > Creación de un mapa vacío: asocia cadenas con enteros

HashMap<String, Integer> mapa = new HashMap<String, Integer>();

// > Inserción de entradas (clave, valor)

mapa.put("uno", 1);
mapa.put("dos", 2);
mapa.put("tres", 3);

// > Consultar el tamaño

System.out.println("Tamaño: " + mapa.size());
System.out.println("¿Es vacío? " + mapa.isEmpty());

// > Recuperación de entradas por clave

System.out.println(mapa.get("tres"));

// > Un mapa no acepta dos entradas con la misma clave
// La segunda inserción reemplaza a la primera

mapa.put("tres", 2);
System.out.println(mapa.get("tres"));

// > Recorrido de las entradas por clave

for (String clave : mapa.keySet()) {

      Integer valor = mapa.get(clave);
      System.out.printf("Clave: %s - Valor %d \n", clave, valor);
}


// > Consultar si existe una entrada con una clave

System.out.println("¿Clave 'uno'? " + mapa.containsKey("uno"));
System.out.println("¿Clave 'cuatro'? " + mapa.containsKey("cuatro"));


// > Obtener una copia de las claves del mapa

HashSet<String> copiaClaves = new HashSet<String>(mapa.keySet());

System.out.println(copiaClaves);
```

```java
// > Obtener una copia de los valores
LinkedList<Integer> copiaValores = new LinkedList<Integer>(mapa.values());
System.out.println(copiaValores);
// Observa que los valores pueden estar repetidos

// > Obtener una copia del mapa
HashMap<String, Integer> copia = new HashMap<String, Integer>(mapa);
System.out.println(copia);

// > Eliminar entradas por clave
mapa.remove("tres");
System.out.println(mapa);
```

Ordenar una lista

```
// Algunos tipos de datos implementan un orden natural: interfaz Comparable
// Por ejemplo, las cadenas

LinkedList<String> lista = new LinkedList<String>();
lista.addAll(Arrays.asList("hola", "mundo", "juan", "pedro"));

// El método static java.util.Collections.sort() ordena una lista
Collections.sort(lista);

// El orden aplicado es el alfabético de las cadenas
System.out.println(lista);
```

Implementación de un orden natural (Comparable)

A continuación se muestra una clase que representa un pedido y que implementa su orden natural. El orden implementado corresponde al orden alfabético de su propiedad *producto*.

Es habitual al implementar el orden natural de un tipo de datos hacer uso del orden de los tipos de datos de los que es cliente. En el ejemplo, se utiliza el orden natural de la clase `String`.

```
public class Pedido implements Comparable<Pedido> {

	private final String producto;
	private final int cantidad;

	public Pedido(String producto, int cantidad) {
		this.producto = producto;
		this.cantidad = cantidad;
	}

	public int getCantidad() {
		return cantidad;
	}

	public String getProducto() {
		return producto;
	}

	@Override
	public String toString() {
		return getClass().getName()
		  + " [producto=" + producto + ", cantidad=" + cantidad + "]";
	}

	@Override
	public int compareTo(Pedido arg0) {

		// Ordena los pedidos alfabéticamente por nombre de producto

		// Se apoya en el orden natural implementado en String

		return this.producto.compareTo(arg0.producto);
	}
}
```

El tipo de datos `Pedido` implementa su orden natural. Por tanto, una lista de pedidos puede ser ordenada con el método `sort`.

```
 // > Caso de estudio: pedidos

LinkedList<Pedido> pedidos = new LinkedList<>();
pedidos.add(new Pedido("teclado", 5));
pedidos.add(new Pedido("ratón", 3));
pedidos.add(new Pedido("monitor", 3));
pedidos.add(new Pedido("monitor", 2));

Collections.sort(pedidos);

System.out.println(pedidos);
```

Cuando queremos ordenar una lista cuyo tipo de datos no implementa el orden natural o bien queremos aplicar un orden distinto al natural, podemos programar una clase implemente la interfaz `java.util.Comparator` para proporcionar un orden.

En el siguiente ejemplo se implementa un comparador de pedidos. Ordena los pedidos utilizando dos criterios. En primer lugar, ordena alfabéticamente por nombre de producto. En caso de empate, lo resuelve por cantidad de menor a mayor.

```
public class ComparadorPedidos implements Comparator<Pedido> {

	@Override
	public int compare(Pedido arg0, Pedido arg1) {

		// Compara por orden alfabético del producto
		// En caso de empate, resuelve por cantidad (de menor a mayor)

		// Nos apoyamos en el orden natural de las cadenas (String)
		// y el de los números implementado en las clases envoltorio

		int criterio1 =
		  arg0.getProducto().compareTo(arg1.getProducto());
		int criterio2 =
		  ((Integer)arg0.getCantidad()).compareTo(arg1.getCantidad());

		if (criterio1 == 0) { // empate criterio 1
			return criterio2;
		}
		else return criterio1;
	}
}
```

El método `sort` ofrece una versión sobrecargada que permite establecer un orden de comparación (`Comparator`)

```
ComparadorPedidos comparador = new ComparadorPedidos();

Collections.sort(pedidos, comparador);

System.out.println(pedidos);
```