Programación

Estructuras de Almacenamiento
Colecciones de datos

Esta presentación ha sido elaborada a partir del material del módulo de Programación del IES El Majuelo (Gines)

ÍNDICE

1. Uso de clases y métodos genéricos

- El uso de genéricos es muy importante en las colecciones.
- Las colecciones admiten cualquier tipo de datos.
- Con los genéricos, indicamos el tipo de datos que se puede guardar en una colección.
- Evita tener que realizar conversiones de tipos.
- Si en un punto del programa intentamos insertar un elemento que no corresponde al tipo de dato indicado, dará un error de compilación.

1. Uso de clases y métodos genéricos

- Clase **sin** métodos generícos:

```
public static class util {
    public static int compararTam(Object[] a,Object[] b) {
        return a.length-b.length;
    }
}
```

Clase **con** métodos generícos:

```
public static class UtilGenericos {
    public static <T> int compararTam(T[] a, T[] b) {
        return a.length-b.length;
    }
}
```

1. Uso de clases y métodos genéricos

- Invocación al método NO genérico:

```
Integer []a={0,1,2,3,4};
Integer []b={0,1,2,3,4,5};
util.compararTam((Object[])a, (Object[])b);
```

Invocación al método genérico:

```
Integer []a={0,1,2,3,4};
Integer []b={0,1,2,3,4,5};
util.<Integer>compararTam(a, b);
```

1. Uso de clases y métodos genéricos

Permitiría añadir cualquier tipo de datos a la colección.

Problema:

A la hora de visualizar los datos hay que hacer un casting (obligatorio) y no siempre sería posible (datos de diferente tipo).

Los parámetros de tipo de las clases genéricas solo pueden ser clases. No pueden ser tipos de datos primitivos como int, double, char, etc. En su lugar, debemos usar las clases wrappers de cada tipo primitivo (Integer, Double, Character, …).

1. Uso de clases y métodos genéricos

Ejemplos de Código

2. Tipos de colecciones

Tipos de variables que se usan en un programa:

- Simples (dato único).
- Estructuradas (más de un dato del mismo tipo).
  - Internas (memoria RAM).
    - Estáticas: Tamaño fijo e invariable.
    - Dinámicas. Tamaño variable.
  - Externas (ficheros).

2. Tipos de colecciones
2.1. Colecciones de datos dinámicas

- Sus datos siempre son objetos.
- Los objetos reciben el nombre de nodos.
  - La clase a la que pertenece el objeto tiene un dato (campo de enlace) cuyo tipo es el propio nombre de la clase donde está; dicho campo se usa para poder enlazar dicho nodo con otro de la propia colección.
- Los nodos no se sitúan de forma continua en memoria.
- Tipos:
  - Lineales: pilas, colas y listas.
  - No lineales: Árboles y grafos.

3. Jerarquía de colecciones

- En la librería de Java existen una serie de clases e interfaces que permiten manipular este tipo de estructuras.

  https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html

- Dichas clases e interfaces se pueden agrupar en tres conjuntos:
  - Interfaz **`Collection`**.
  - Interfaz **`Map`**.
  - Interfaz **`Iterator`**.

  https://www.adictosaltrabajo.com/2015/09/25/introduccion-a-colecciones-en-java/

3. Jerarquía de colecciones
3.1. Interfaz *Collection*

- En una **`Collection`** se pueden recorrer todos los elementos que lo forman y se puede saber cuántos nodos tiene.
- A este nivel no hay un orden establecido.
- Las clases que heredan dicha interfaz impondrán más restricciones. Dichas clases aportarán más funcionalidades y tendrán, como información muy importante, el orden en el que están colocados los nodos.
- Hay dos interfaces que implementan la interfaz **`Collection`**: **`List`** y **`Set`**.

3. Jerarquía de colecciones
3.2. Interfaz *List*

- Una lista es una colección donde los nodos que lo forman están dispuestos en un cierto orden.
- La diferencia entre **`List`** y **`Collection`** es que la primera guarda información de cómo están colocados los nodos; esto es algo que no hace la interfaz **`Collection`**.
  - Por esta razón, las listas permiten poder acceder a un determinado nodo por su posición.
- La interfaz **`List`** es implementada por dos clases:
  - **`ArrayList`**.
  - **`LinkedList`**.

3. Jerarquía de colecciones
3.2.1 Clase *ArrayList*

- Un **`ArrayList`**, al contrario de lo que sucedía con los `arrays` comunes (tablas), no tiene un tamaño fijo, es decir, puede variar la cantidad de nodos que lo forman según las necesidades que vaya habiendo a lo largo de la ejecución del programa.
- El acceso a los elementos de un **`ArrayList`** es rápido ya que, indicando su posición, accedemos directamente a ese elemento.
- El principal inconveniente de este tipo de colecciones es borrar un elemento, ya que requiere tiempo debido a la forma que tiene de eliminar un objeto:
  - Desplaza todos los elementos que van después del nodo que queremos borrar, un lugar hacia la izquierda, con el fin de eliminar el hueco que deja dicho nodo.
  - Lo mismo sucede si queremos insertar un elemento en un sitio determinado.

3. Jerarquía de colecciones
3.2.2. Clase *LinkedList*

- Tiene forma de lista doblemente enlazada.
- Cada nodo tiene dos campos de enlace: uno apunta al nodo que va por detrás de dicho nodo y el otro apunta al nodo que va delante del mismo.
- La principal ventaja de este tipo de listas es que permite borrar e insertar nodos de una manera fácil, ya que lo único que hace es cambiar las referencias de los nodos que van delante y detrás del que deseamos borrar o del que queramos insertar.

3. Jerarquía de colecciones
3.5. Interfaz *Set*

- Se usa cuando queremos que en la colección no haya objetos repetidos para unos determinados valores del objeto.
- Es implementada por clases como:
  - **`HashSet`**:
    - Es la clase más utilizada para implementar una colección sin duplicados, pero no se garantiza un orden.
    - Lo que hace esta clase es usar una tabla (tabla **`hash`**) con el fin de guardar los diferentes códigos de identificación que tiene la colección. Cada posición de la tabla apunta a los diferentes nodos que tienen el mismo código de identificación.
  - **`TreeSet`**:
    - Implementa la interfaz **`SortedSet`** (que a su vez implementa la interfaz **`Set`**), consiguiendo con ello que los elementos que la forman estén ordenados formando la estructura de un árbol binario.

3. Jerarquía de colecciones
3.6. Interfaz *Map*

- Los mapas guardan parejas de objetos, asociando un objeto por su clave con otro objeto por un valor concreto.
- Las claves no pueden estar repetidas y solo se pueden asociar con un valor.
- Dos clases que implementan este interfaz:
  - **`HashMap`**.
    - Hace uso de tablas hash con la misma lógica que la clase **`HashSet`**.
  - **`TreeMap`**.
    - Hace que los elementos salgan ordenados por la clave.

3. Jerarquía de colecciones
3.7. Interfaz *Iterator*

- Se utiliza para acceder y recorrer los elementos de una colección.
- La interfaz **`Collection`** tiene un método **`iterator()`** que devuelve un objeto de tipo **`Iterator`**. Mediante dicho objeto, usando sus métodos podemos recorrer la colección.
- La interfaz **`ListIterator`** implementa la interfaz **`Iterator`**, dando mayores posibilidades.
  - Se usa para recorrer listas (**`ArrayList`** o **`LinkedList`**).
  - No se puede usar para recorrer otro tipo de colecciones.

4. Operaciones con colecciones
4.1 Interfaz *Collection*

- Características de este tipo de interfaz:
  - Sólo se pueden recorrer los datos que lo forman en el mismo orden en el que fueron insertados. Hay que pasar por todos ellos para llegar a un dato concreto.
  - Se puede eliminar cualquier elemento de la colección.
  - Los elementos nuevos se van colocando al final de la colección.
- Los objetos que se guardan en una colección no se identifican por los valores que contienen sino por un identificador que se le asigna a cada objeto; dicho identificador es único para cada objeto.

4. Operaciones con colecciones
4.1 Interfaz *Collection*

Métodos más importantes:

- **`boolean add(Object objReci);`** Añade el objeto que recibe la colección. Devuelve *true* si la operación se ha podido realizar.
- **`boolean remove(Object objReci);`** Elimina un objeto que haya en la colección, cuyo identificador coincida con el del objeto que recibe.
- **`boolean contains(Object objReci);`** Comprueba si en la colección hay un objeto con el mismo identificador que el del objeto que recibe.

4. Operaciones con colecciones
4.1 Interfaz *Collection*

Una vez añadidos los datos en una colección, para poder recorrerlos es absolutamente obligatorio usar un objeto de tipo **`Iterator`**. Este objeto sirve como cursor que se va moviendo por la colección. Dicho interfaz tiene los siguientes métodos:

- **`Object next();`** Devuelve el siguiente objeto que hay después del objeto que apunta el iterador y avanza dicho elemento. Si no hay ningún elemento más, lanzará el error **`NoSuchElementException`**.
- **`boolean hasNext();`** Devuelve true si hay algún elemento después del objeto donde apunta el iterador (éste no se mueve).
- **`void remove();`** Elimina el objeto donde apunta el iterador.

La interfaz **`Collection`** tiene métodos para borrar objetos de la colección siempre que la búsqueda del objeto a borrar se haga por el identificador. Si no es así, sino que queremos buscar objetos en función de los valores que tenga, hay que usar obligatoriamente un iterador.

4. Operaciones con colecciones
4.2 Interfaz *List*

Esta interfaz amplia la interfaz `Collection` añadiendo más posibilidades, como son:

- Mantiene el orden de los elementos que contiene, permitiendo con ello acceder a un determinado elemento con solo indicar su posición.
  - Las posiciones se empiezan a numerar desde el 0.
- Se puede insertar un elemento entre dos ya existentes.

4. Operaciones con colecciones
4.2 Interfaz *List*

Métodos más importantes:

- **`void add(int pos, Object objReci);`** Inserta en la lista el objeto que recibe en la posición indicada. Devuelve *true* si la operación se ha podido realizar.
- **`Object remove(int pos);`** Elimina el objeto que está en la posición que recibe. Devuelve el objeto borrado.
- **`Object set(int pos, Object objReci);`** Sustituye el objeto que está en la posición *`pos`* por el objeto que recibe. Devuelve el objeto original.
- **`Object get(int pos);`** Devuelve el objeto que está en la posición *`pos`*.

Si se intentara acceder a una posición que no existiera dentro de la colección, se lanzaría un error del **`tipo IndexOutOfBoundsExcepcion.`**

4. Operaciones con colecciones
4.2 Interfaz *List*

Para recorrer los elementos de una lista se puede usar el iterador **`Iterator`** o bien la interfaz **`ListIterator`**. Éstos últimos permiten mayores posibilidades ya que a través de ellos podemos añadir y modificar objetos de la lista, además de poder recorrer la lista en cualquier sentido. Métodos más importantes de **`ListIterator`**:

- **`void add(Object objReci);`** Inserta en la lista el objeto que recibe, después del objeto al que apunta el iterador.
- **`void set(Object obj);`** Reemplaza el objeto al que apunta el iterador por el objeto que recibe.
- **`int nextIndex();`** Devuelve la posición del objeto que está después del objeto al que apunta el iterador.
- **`int previousIndex();`** Devuelve la posición del objeto que está antes del que apunta el iterador.
- **`Object previous();`** Devuelve el objeto al que apunta el iterador.

4. Operaciones con colecciones
4.3 Interfaz *Set*

https://docs.oracle.com/javase/7/docs/api/java/util/Set.html

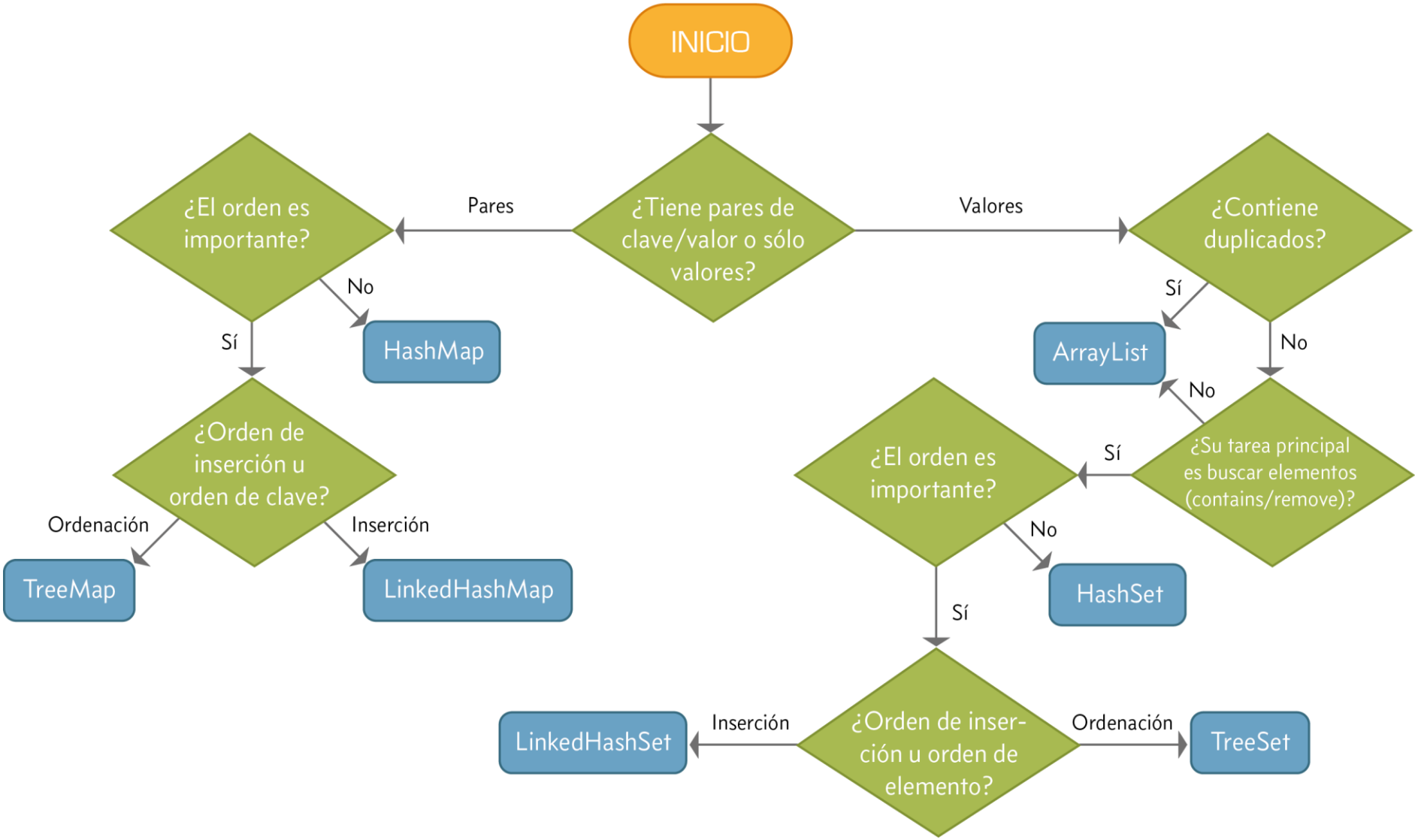
4. Operaciones con colecciones
4.3 Interfaz *Set*

Para recorrer los elementos de un conjunto se puede usar el iterador **`Iterator`**:

- **`boolean hasNext();`** `Devuelve true si existe un elemento más en el conjunto.`
- **`int next();`** Devuelve el objeto que está después del objeto al que apunta el iterador.
- **int hashCode();** Devuelve el valor del código hash del objeto.
- **void remove();** Elimina el objeto del conjunto.

Pero para añadir elementos al conjunto **`add(Object objeto)`** hay que recorrer de forma manual los elementos del conjunto, con un **`for`** por ejemplo.

Diagrama de decisión para uso de colecciones Java



5. Genéricos y colecciones

Gracias al uso de los genéricos, podemos recorrer una colección muy fácilmente usando este tipo de *for*:

```
for(nombreClase nomObjActual: nombreColeccion)
```

Para poder usar este tipo de recorrido es obligatorio que la colección sea genérica.

Este tipo de *for* sirve para recorrer la colección, pero en él no se podrá borrar ningún objeto.

También el uso de genéricos se puede hacer con el iterador:

```
Iterator <nombre_clase> obj = nombre_coleccion.iterator();
```

En este caso no hace falta que la colección sea genérica (y por tanto evitamos tener que hacer un casting).