



Lenguaje Java Avanzado

Sesión 2: Colecciones de datos



Índice

- Introducción
- Colecciones
- Enumeraciones e iteradores
- Polimorfismo e interfaces
- Wrappers de tipos básicos



Introducción

- Java proporciona un amplio conjunto de clases útiles para desarrollar aplicaciones
- En esta sesión veremos algunos grupos de ellas:
 - Clases útiles para crear y gestionar diferentes tipos de colecciones
 - Clases para recorrer, ordenar y manipular las colecciones
 - Clases para encapsular nuestros tipos de datos



Colecciones

- En el paquete `java.util`
- Representan grupos de objetos, llamados elementos
- Podemos encontrar de distintos tipos, según si sus elementos están ordenados, si permiten repetir elementos, etc
- La interfaz `Collection` define el esqueleto que deben tener todos los tipos de colecciones
- Por tanto, todos tendrán métodos generales como:
 - `boolean add(Object o)`
 - `boolean remove(Object o)`
 - `boolean contains(Object o)`
 - `void clear()`
 - `boolean isEmpty()`
 - `Iterator iterator()`
 - `int size()`
 - `Object[] toArray()`



Listas de elementos

- La interfaz `List` hereda de `Collection`
 - Operaciones propias de una colección tipo *lista*
 - Los elementos tienen un orden (posición en la lista)
- Así, tendremos otros nuevos métodos, además de los de `Collection`:
 - `void add(int posicion, Object o)`
 - `Object get(int indice)`
 - `int indexOf(Object o)`
 - `Object remove(int indice)`
 - `Object set(int indice, Object o)`



Tipos de listas

- ArrayList: implementa una lista de elementos mediante un *array* de tamaño variable
 - *NO sincronizado*
- Vector: existe desde las primeras versiones de Java, después se acomodó al marco de colecciones implementando la interfaz List.
 - *Similar a ArrayList, pero SINCRONIZADO. Tiene métodos anteriores a la interfaz List:*
 - `void addElement(Object o) / boolean removeElement(Object o)`
 - `void insertElementAt(Object o, int posicion)`
 - `void removeElementAt(Object o, int posicion)`
 - `Object elementAt(int posicion)`
 - `void setElementAt(Object o, int posicion)`
 - `int size()`
- LinkedList: lista doblemente enlazada. Útil para simular pilas o colas
 - `void addFirst(Object o) / void addLast(Object o)`
 - `Object getFirst() / Object getLast()`
 - `Object removeFirst() / Object removeLast()`



Conjuntos

- Grupos de elementos donde no hay repetidos
- Consideramos dos objetos de una clase iguales si su método `equals` los da como iguales
 - `o1.equals(o2)` es `true`
- Los conjuntos se definen en la interfaz `Set`, que, como `List`, también hereda de `Collection`
- El método `add` definido en `Collection` devolvía un `booleano`, que en este caso permitirá saber si se insertó el elemento en el conjunto, o no (porque ya existía)



Tipos de conjuntos

- HashSet: los objetos del conjunto se almacenan en una tabla *hash*.
 - El coste de inserción, borrado y modificación suele ser constante
 - La iteración es más costosa, y el orden puede diferir del orden de inserción
- LinkedHashSet: como la anterior, pero la tabla *hash* tiene los elementos enlazados, lo que facilita la iteración
- TreeSet: guarda los elementos en un árbol
 - El coste de las operaciones es logarítmico



Mapas

- No forman parte del marco de colecciones
- Se definen en la interfaz Map, y sirven para relacionar un conjunto de claves (*keys*) con sus respectivos valores
- Tanto la clave como el valor pueden ser cualquier objeto
 - `Object get(Object clave)`
 - `Object put(Object clave, Object valor)`
 - `Object remove(Object clave)`
 - `Set keySet()`
 - `int size()`



Tipos de mapas

- **HashMap:** Utiliza una tabla *hash* para almacenar los pares *clave=valor*.
 - Las operaciones básicas (get y put) se harán en tiempo constante si la dispersión es adecuada
 - La iteración es más costosa, y el orden puede diferir del orden de inserción
- **Hashtable:** como la anterior, pero **SINCRONIZADA**. Como **Vector**, está desde las primeras versiones de Java
 - `Enumeration keys()`
- **TreeMap:** utiliza un árbol para implementar el mapa
 - El coste de las operaciones es logarítmico
 - Los elementos están ordenados ascendentemente por clave



Genéricos

- Colecciones de tipos concretos de datos
 - A partir de JDK 1.5
 - Aseguran que se utiliza el tipo de datos correcto

```
ArrayList<String> a = new ArrayList<String>();  
a.add("Hola");  
String s = a.get(0);
```

- Podemos utilizar genéricos en nuestras propias clases



Enumeraciones e iteradores

- Las enumeraciones y los iteradores no son tipos de datos en sí, sino objetos útiles a la hora de recorrer diferentes tipos de colecciones
- Con las *enumeraciones* podremos recorrer secuencialmente los elementos de una colección, para sacar sus valores, modificarlos, etc
- Con los *iteradores* podremos, además de lo anterior, eliminar elementos de una colección, con los métodos que proporciona para ello.



Enumeraciones

- La interfaz `Enumeration` permite consultar secuencialmente los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método `nextElement`:

```
Object item = enum.nextElement();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método `hasMoreElements`:

```
if (enum.hasMoreElements()) ...
```



Enumeraciones

- Con lo anterior, un bucle completo típico para recorrer una colección utilizando su enumeración de elementos sería:

```
// Obtener la enumeracion
Enumeration enum = coleccion.elements();
while (enum.hasMoreElements())
{
    Object item = enum.nextElement();
    ...// Convertir item al objeto adecuado y
        // hacer con el lo que convenga
}
```



Iteradores

- La interfaz `Iterator` permite iterar secuencialmente sobre los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método `next`:

```
Object item = iter.next();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método `hasNext`:

```
if (iter.hasNext()) ...
```

- Para eliminar el elemento de la posición actual del iterador, utilizamos su método `remove`:

```
iter.remove();
```



Iteradores

- Con lo anterior, un bucle completo típico para recorrer una colección utilizando su iterador sería:

```
// Obtener el iterador
Iterator iter = coleccion.iterator();
while (iter.hasNext())
{
    Object item = iter.next();
    ...// Convertir item al objeto adecuado y
    // hacer con el lo que convenga, por ejemplo
    iter.remove();
}
```



Bucles sin iteradores

- Nueva versión del for en JDK 1.5
- Permite recorrer tanto *arrays* como colecciones
- Previene salirse del rango de forma segura

```
List<String> lista = obtenerLista();  
for(String cadena: lista)  
    System.out.println (cadena);
```



Polimorfismo e interfaces

- Hacer referencia siempre mediante la interfaz
 - Permite cambiar la implementación sin afectar al resto del programa

```
public class Cliente {  
    List<Cuenta> cuentas;  
    public Cliente() {  
        this.cuentas = new ArrayList<Cuenta>();  
    }  
    public List<Cuenta> getCuentas() {  
        return cuentas;  
    }  
}
```



Ejemplo: Algoritmos

- La clase `Collections` dispone de una serie de métodos útiles para operaciones tediosas, como ordenar una colección, hacer una búsqueda binaria, sacar su valor máximo, etc
 - `static void sort(List lista)`
 - `static int binarySearch(List lista, Object objeto)`
 - `static Object max(Collection col)`
 - ...
- Sirven para cualquier implementación de `List`



Comparación de objetos

- Los objetos deben ser correctamente comparables para ser compatibles con las estructuras de datos y algoritmos.
- Comparación de igualdad: equals()
- Comparación de mayor o menor: clase Comparator o interfaz Comparable

Sobrecarga de Equals

- Object.equals(Object o)

```
public class MiClase {  
    ...  
    @Override  
    public boolean equals(Object o) {  
        // return true o false, según un criterio  
    }  
}
```



Evitar la sobrecarga de Equals si:

- Cada instancia es intrínsecamente única. Por ejemplo, instancias de hilos, que representan entidades activas, y no tan sólo un conjunto de valores.
- Cuando no es necesaria una comparación lógica. Por ejemplo, dos números aleatorios, donde la igualdad puede ocurrir pero su comprobación no es necesaria.
- Una superclase ya sobrecarga equals, y el comportamiento de éste es apropiado para la clase actual.



Propiedades que debe cumplir

- Reflexividad: `x.equals(x)` devuelve siempre verdadero.
- Simetría: para cualquier par de instancias no nulas, `x.equals(y)` devuelve verdadero si y sólo si `y.equals(x)` también devuelve verdadero.
- Transitividad: si `x.equals(y)==true` y `y.equals(z)==true`, entonces `x.equals(z)` también será verdadero, para cualesquiera instancias no nulas.
- Consistencia: múltiples llamadas al método con las mismas instancias devuelven el mismo resultado.
- Comparación con null falsa: `x.equals(null)` devuelve falso.



Sobrecargar hashCode()

- Cuando hashCode es invocado varias veces para el mismo objeto, debe devolver consistentemente el mismo entero, siempre que no se haya modificado ninguna información que afecte al resultado de equals. Esta consistencia debe mantenerse entre distintas ejecuciones de la misma aplicación.
- Si dos objetos son iguales según equals, entonces los métodos hashCode de ambos deben devolver el mismo entero.
- Si dos objetos no son iguales según equals, no se requiere que devuelvan hashCode diferentes. No obstante en la medida de lo posible deben ser distintos porque esto puede mejorar la eficiencia de las tablas hash.



Comparar implementando Comparable

- Permite establecer un orden entre objetos
- Se necesita decidir qué características del objeto establecen dicho orden

```
public class Persona implements Comparable<Persona> {  
    public int id;  
    public String apellido;  
    ...  
    @Override  
    public int compareTo(Persona p) {  
        return this.id - p.id;  
    }  
}
```



Comparador externo: Comparator

- Puede extenderse o bien desde una clase externa, o bien desde la propia clase cuyos objetos deben ser comparados

```
public class ComparaPersonaPorNombre implements Comparator<Persona>{  
    public int compare(Persona p1, Persona p2) {  
        return p1.apellido.compareToIgnoreCase(p2.apellido);  
    }  
}
```

...

```
List personas = new ArrayList<Persona>();  
personas.add(p1); personas.add(p2); personas.add(p3); //...
```

```
Collections.sort(personas); //Comparable.compareTo  
Collections.sort(personas, new ComparaPersonaPorNombre()); //Comparator.compare
```



Ejemplo: Wrappers de colecciones

- Objetos que envuelven la instancia de una colección existente
- Implementa la misma interfaz (p.ej `List`)
 - No conocemos la clase concreta del *wrapper*
- Cambia el comportamiento de algunos métodos
 - Sincronizar acceso a la colección

```
List Collections.synchronizedList(List l)
```

- Hacerla de sólo lectura

```
List Collections.unmodifiableList(List l)
```



Wrappers

- Los tipos simples (`int`, `char`, `float`, `double`, etc) no pueden incluirse directamente en colecciones, ya que éstas esperan subtipos de `Object` en sus métodos
- Para poderlos incluir, se tienen unas clases auxiliares, llamadas *wrappers*, para cada tipo básico, que lo convierten en objeto complejo
- Estas clases son, respectivamente, `Integer`, `Character`, `Float`, `Double`, etc.
- Encapsulan al tipo simple y ofrecen métodos útiles para poder trabajar con ellos



Wrappers

- Si quisiéramos incluir un entero en un `ArrayList`, lo podríamos hacer así:

```
int a;  
ArrayList al = new ArrayList();  
al.add(new Integer(a));
```

- Si quisiéramos recuperar un entero de un `ArrayList`, lo podríamos hacer así:

```
Integer entero = (Integer)(al.get(posicion));  
int a = entero.intValue();
```



Autoboxing

- Nueva característica de JDK 1.5
- Conversiones automáticas entre tipos básicos y sus *wrappers*

```
Integer n = 10;  
int num = n;  
  
List<Integer> lista= new ArrayList<Integer>();  
lista.add(10);  
int elem = lista.get(0);
```



¿Preguntas...?