UD2

LENGUAJE DE PROGRAMACIÓN JAVA

MP_0485
Programación

4.3 Conjuntos

Introducción

Los arrays nos ofrecen una interesante forma de estructurar datos en la memoria, pero tienen el problema de que es necesario saber previamente la longitud o número de elementos que tendrá cada array. Para resolver este problema podemos utilizar otro recurso que nos ofrece Java; las clases que representan colecciones y las clases que representan mapas

Colecciones y mapas sirven para almacenar en la memoria un grupo o conjunto de elementos, al igual que los arrays, pero tienen la gran ventaja de crecer dinámicamente según las necesidades del programa en cada momento.

Java cuenta con muchísimas clases que sirven para representar colecciones de datos y mapas, clases que forman parte de una compleja jerarquía. En la siguiente imagen te mostramos la jerarquía que corresponde a las clases con las que vamos a trabajar.

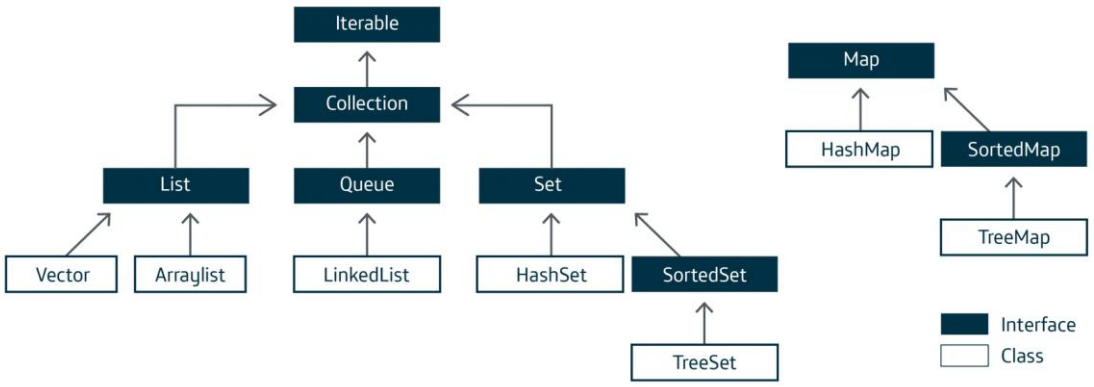

Figura 1: Estructura jerárquica de las interfaces Coleccion y Map.

En la imagen hemos simplificado el árbol jerárquico de las colecciones y mapas, eliminando algunas ramas y dejando lo más relevante. Lo importante es que comprendas que cuando utilizas una clase, esta no está aislada, sino que forma parte de una estructura jerárquica más compleja.

Ten en cuenta que en la imagen los rectángulos con fondo negro son interfaces y los rectángulos con fondo blanco son clases. Así puedes fácilmente deducir que la clase Vector implementa la interfaz List, e indirectamente, también las interfaces Collection e Iterable. También puedes comprobar que la clase LinkedList implementa las interfaces List y Queue, y también, de manera indirecta, las interfaces Collection e Iterable.

Interfaz set

Todas las clases que derivan directa o indirectamente de la interfaz Set pertenecen a un tipo especial de colecciones llamadas conjuntos. Estas colecciones están basadas en la teoría de conjuntos de matemáticas. La característica más importante de este tipo de colecciones es que no admiten duplicados.

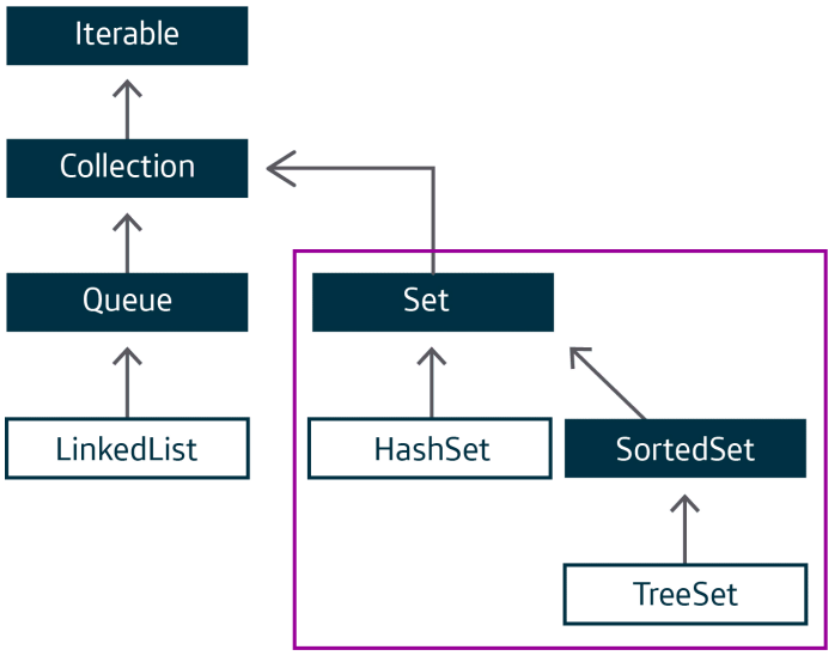

Figura 2: Estructura jerárquica de las interfaces Coleccion y Map.

Trabajaremos con las clases HashSet y TreeSet. Observa que derivan de Collection e Iterable, por lo que puedes deducir que puedes iterar con una estructura for ... each, usar el método iterator() y otros métodos como add(), remove(), clear(), etc. Sin embargo, no podrás utilizar el método get(), propio de la interfaz List.

Diferencias entre HashSet y TreeSet

Ambas clases **no admiten elementos duplicados**, ya que son conjuntos. Para evitar los duplicados, por cada nuevo elemento que se va a almacenar se comprueba que la función equals() sobre el nuevo elemento comparado con todos los anteriores devuelva siempre el valor false.

Recuerda que **dos objetos se consideran iguales** cuando tienen el **mismo hashCode**. Si crees que lo necesitas, repasa de nuevo el apartado sobre los métodos hashcode() y equals() del capítulo anterior.

En las colecciones de tipo **HashSet los elementos no se colocan en el orden en que han sido insertados**, no se puede garantizar en qué orden se recuperan los elementos.

En las colecciones de tipo **TreeSet los elementos se ordenan atendiendo al criterio que dicta el método compareTo()**.

Conjuntos sin orden: HashSet

Una colección de tipo HashSet representa un conjunto de elementos del mismo tipo, donde no pueden existir duplicados y los elementos no guardan el orden en que se introdujeron. Veamos un ejemplo:

```
import java.util.HashSet;
import java.util.Set;

public class Principal {
	public static void main(String[] args) {
		Set<String> nombres = new HashSet<String>();

		nombres.add("Rosa");
		nombres.add("Carlos");
		nombres.add("Miguel");
		nombres.add("Carlos"); // Este no se añadirá.
		nombres.add("Sole");
		nombres.add("Adrián");
		nombres.add("Angel");
		nombres.add("Amelia");
		nombres.add("Fernando");
		nombres.add("Sebas");
		nombres.add("Lucas");

		for (String n : nombres) {
			System.out.println(n);
		}
	}
}
```

Hemos añadido dos veces el nombre "Carlos", sin embargo, el segundo no se añadirá. Observa también que, a la hora de mostrar los elementos en pantalla con la estructura for, no se recuperan en el mismo orden en el que se insertaron.

Con las cadenas de texto está muy claro cuándo hay un duplicado, pero, ¿qué pasa con objetos Triangulo? ¿Cuándo se considera que dos objetos son iguales? La respuesta **está en los métodos hashCode() y equals().**

Recuerda que dos objetos **se consideran iguales si la expresión obj1.equals(obj2) es true**, y será true solo si tienen el mismo hashCode. Cada clase puede sobrescribir los métodos hashCode() y equals(). En conclusión, el programador que implementa una clase puede decidir qué criterio utiliza para decidir que dos objetos de dicha clase son iguales.

Vamos a modificar la clase Triangulo para considerar que dos triángulos son iguales si la suma de sus lados es igual.

```
public class Triangulo {
	private int lado1;
	private int lado2;
	private int lado3;

	public Triangulo(int lado1, int lado2, int lado3) {
		this.lado1 = lado1;
		this.lado2 = lado2;
		this.lado3 = lado3;
	}

	@Override
	public String toString() {
		return "Triangulo [" + lado1 + ", " + lado2 + ", " + lado3

	}

	public String verTipo() {
		if (this.lado1==this.lado2&&this.lado2==this.lado3) {
			return this.toString() + " Equilátero";
		}
		else if (this.lado1==this.lado2||this.lado2==this.lado3||th
			return this.toString() + " Isósceles";
		}
		else {
			return this.toString() + " Escaleno";
		}
	}

	@Override
	public int hashCode() {
		return this.lado1+this.lado2+this.lado3;
	}
	@Override
	public boolean equals(Object obj) {
		if (this.hashCode() == obj.hashCode())
			return true;
		else
			return false;
	}
}
```

Ahora podemos crear una colección de objetos Triangulo donde no puedan existir dos triángulos cuya suma de los lados sea igual.

```
import java.util.HashSet;

public class Principal {
        public static void main(String[] args) {
                HashSet<Triangulo> trians = new HashSet<Triangulo>();

                trians.add(new Triangulo(1, 1, 1));
                trians.add(new Triangulo(2, 2, 1));
                trians.add(new Triangulo(3, 1, 1)); // No se guarda, se con
                trians.add(new Triangulo(7, 1, 3));
                trians.add(new Triangulo(9, 8, 7));
                trians.add(new Triangulo(9, 8, 7));
                trians.add(new Triangulo(5, 5, 2));
                trians.add(new Triangulo(1, 1, 2));

                for (Triangulo t : trians) {
                        System.out.println(t.verTipo());
                }
        }
}
```

De nuevo observa en el resultado de la ejecución que los objetos Triangulo mostrados no guardan el mismo orden en que se han ido añadiendo y, además, el tercer triángulo con lados 3, 1, 1 no se ha añadido.

Conjuntos ordenados: TreeSet

La **clase TreeSet funciona igual que la clase HashSet**, pero en este caso **los elementos se ordenan según el criterio que dicte el método compareTo()**. Además, la clase TreeSet implementa la interfaz SortedSet.

Vamos a probar el mismo ejemplo de los nombres del apartado anterior, pero ahora con un objeto TreeSet.

```
import java.util.Set;
import java.util.TreeSet;


public class Principal {
	public static void main(String[] args) {
		Set<String> nombres = new TreeSet<String>();

		nombres.add("Rosa");
		nombres.add("Carlos");
		nombres.add("Miguel");
		nombres.add("Carlos"); // Este no se añadirá.
		nombres.add("Sole");
		nombres.add("Adrián");
		nombres.add("Angel");
		nombres.add("Amelia");
		nombres.add("Fernando");
		nombres.add("Sebas");
		nombres.add("Lucas");

		for (String n : nombres) {
			System.out.println(n);
		}
	}
}
```

Comprueba que, igual que antes, no admite duplicados, pero ahora además los elementos están ordenados alfabéticamente.

Está muy claro cómo deberían ordenarse un grupo de objetos String (alfabéticamente) o un grupo de objetos Integer (numéricamente de menor a mayor). Sin embargo, no resulta tan claro cómo deben ordenarse los objetos. En este caso, es el programador que desarrolla la clase quien debe decidir qué tipo de ordenación resulta más interesante, y para ello debe implementar el método compareTo() de la interfaz Comparable.

Para implementar el método compareTo() hay que respetar las siguientes reglas:

- Dada la expresión obj1.compareTo(obj2), devolverá un número positivo si obj1 es mayor que obj2.
- Dada la expresión obj1.compareTo(obj2), devolverá un número negativo si obj1 es menor que obj2.
- Dada la expresión obj1.compareTo(obj2), devolverá un cero si obj1 es igual que obj2.

Vamos a implementar **el método compareTo() en la clase Triangulo** para que los triángulos se ordenen según la suma de sus lados. Para lograrlo, la clase Triangulo deberá implementar la

interfaz Comparable, así que lo primero que haremos será modificar la cabecera de la clase de la siguiente manera:

```
public class Triangulo implements Comparable { }
```

Además, la interfaz Comparable es genérica, por eso incluimos la expresión. La nueva clase Triangulo quedará así:

```
public class Triangulo implements Comparable<Triangulo> {
	private int lado1;
	private int lado2;
	private int lado3;

	public Triangulo(int lado1, int lado2, int lado3) {

		this.lado1 = lado1;
		this.lado2 = lado2;
		this.lado3 = lado3;
	}

	@Override
	public String toString() {
		return "Triangulo [" + lado1 + ", " + lado2 + ", " + lado3
	}

	public String verTipo() {
		if (this.lado1==this.lado2&&this.lado2==this.lado3) {
			return this.toString() + " Equilátero";
		}
		else if (this.lado1==this.lado2||this.lado2==this.lado3||th
			return this.toString() + " Isósceles";
		}
		else {
			return this.toString() + " Escaleno";
		}
	}

	@Override
	public int hashCode() {
		return this.lado1+this.lado2+this.lado3;
	}
	@Override
	public boolean equals(Object obj) {
		if (this.hashCode() == obj.hashCode())
			return true;
		else
			return false;
	}

	@Override
	public int compareTo(Triangulo o) {
		if (this.hashCode()==o.hashCode())
			return 0;
		else if (this.hashCode()>o.hashCode())
			return 1;
		else
			return -1;
	}
}
```

Ahora puedes crear una colección de objetos Triangulo exactamente igual que lo hiciste con un HashSet, pero ahora con un objeto TreeSet.

```
import java.util.TreeSet;

public class Principal {
	public static void main(String[] args) {
		TreeSet<Triangulo> trians = new TreeSet<Triangulo>();

		trians.add(new Triangulo(1, 1, 1));
		trians.add(new Triangulo(2, 2, 1));
		trians.add(new Triangulo(3, 1, 1)); // No se guarda, se con
		trians.add(new Triangulo(7, 1, 3));
		trians.add(new Triangulo(9, 8, 7));
		trians.add(new Triangulo(9, 8, 7));
		trians.add(new Triangulo(5, 5, 2));
		trians.add(new Triangulo(1, 1, 2));

		for (Triangulo t : trians) {
			System.out.println(t.verTipo());
		}
	}
}
```

Si has ejecutado ya, habrás comprobado que sigue sin admitir duplicado. El triangulo de lados 3, 1, 1 no ha sido añadido, pero ahora, además, los triángulos se muestran ordenados ascendentemente según la suma de sus lados.

Referencias

Apuntes elaborados a partir de la siguiente documentación:

[1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.

[2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.

[3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.

[4] Apuntes de Programación Edix Digital Workers.