UD10. Estructuras de datos II. Colecciones

Módulo: Programación

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Germán Gascón Grau
g.gascongrau@edu.gva.es

Contenidos

- Estructuras estáticas vs Estructuras dinámicas
- Interfaces
- Colecciones

Estructuras estáticas vs Estructuras dinámicas

- En prácticamente todos los lenguajes de programación existen estructuras para almacenar colecciones de datos.
- Podemos ver a las estructuras de datos como conjunto de datos identificados por un único nombre.
- Estructuras **estáticas**
  - Tamaño fijo, es decir, no puede cambiar y conocido en **tiempo de compilación**.
  - Ejemplo: Arrays
- Estructuras **dinámicas**
  - El número de elementos se decide y modifica en **tiempo de ejecución**.
  - El número de elementos es ilimitado.
  - Ejemplos: ArrayList, las listas enlazadas, los árboles, los grafos, etc.

Interfaces

- Las **interfaces** son un **mecanismo** que ofrecen algunos lenguajes de programación entre los que se encuentra Java, que permite indicar **qué se quiere** ofrecer pero **sin determinar cómo** se debe hacer.
- Este mecanismo será muy útil en las siguientes situaciones:
  - Sabemos qué queremos hacer pero todavía no sabemos como hacerlo.
  - Sabemos qué queremos hacer pero lo va a hacer otro programador.
  - Necesitamos tener varias implementaciones de una misma cosa, es decir, definir varios comportamientos en respuesta a una misma cosa.
- En Java una interfaz se crea mediante la palabra reservada `interface`
- La forma de indicar qué se quiere es mediante **métodos que carecen de implementación**. De esta forma se expone qué métodos debe tener la interfaz pero no se determina cómo deben actuar.
- Por ejemplo:

```
public interface DestroyListener {
    void onDestroy(Item i);
}
```

- De esta forma, todas aquellas clases que **implementen** la interfaz, **podrán actuar** cómo si fuesen objetos del tipo que define la interfaz.
- Por tanto, implementar una interfaz es cómo firmar un contrato, ya que la clase que implementa la interfaz se compromete a implementar los métodos de dicha interfaz.
- En este ejemplo, podría existir una clase `Inventory` que implementase la interfaz `DestroyListener` para recibir avisos cada vez que un `Item` sea destruido. De esta forma podría actualizar los slots del inventario.

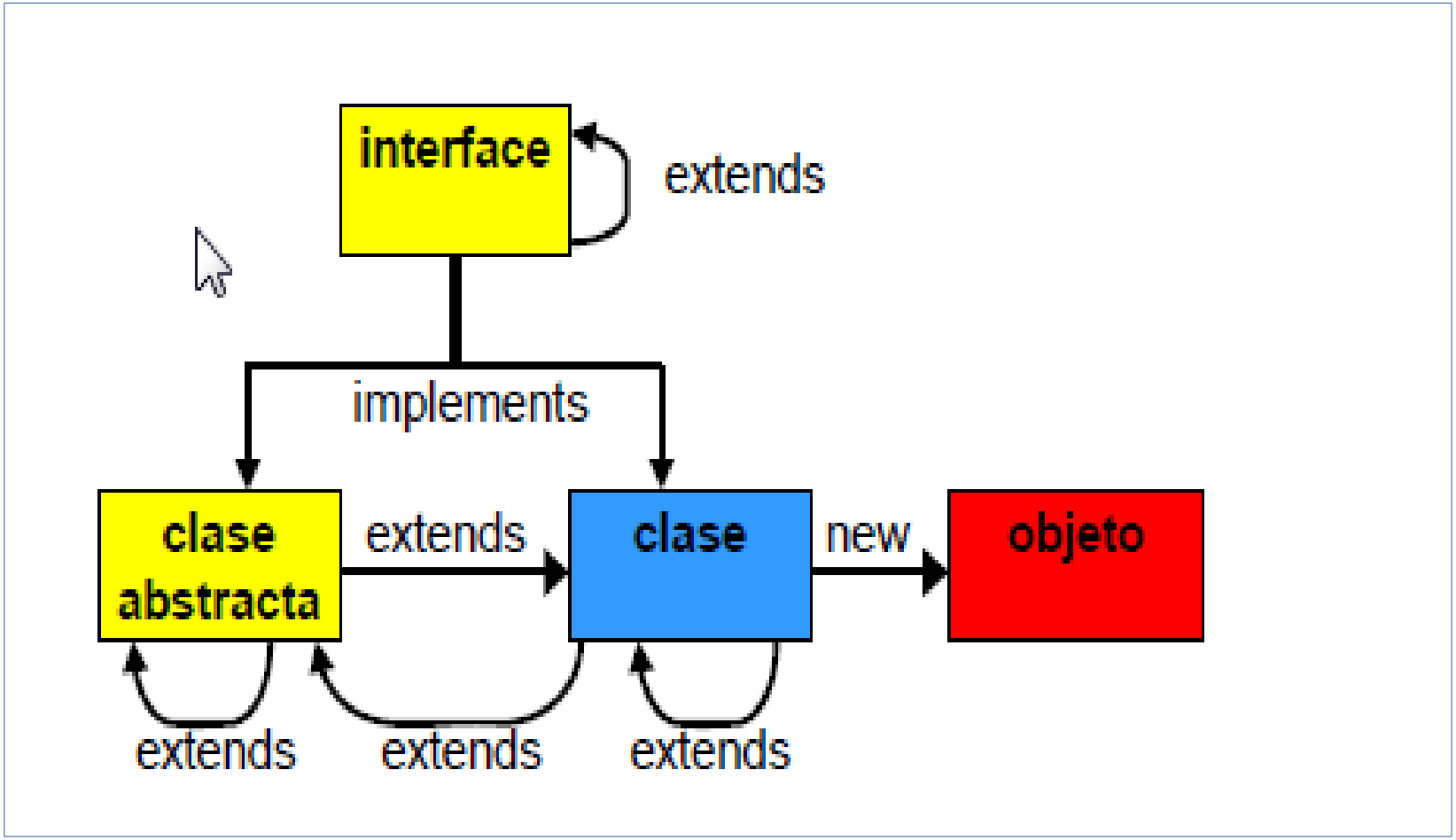
Interfaces

- Todos los métodos de una interfaz son públicos por defecto, por tanto, no es necesario poner la palabra reservada `público` delante.
- No tienen constructores ya que no son clases.
- Pueden incluir "**atributos**" que en realidad serán constantes, ya que aunque no lo indiquemos, son `public static final`.
- Las interfaces:
  - Se pueden extender con nuevas constantes y/o métodos
  - Se pueden implementar totalmente → clases.
  - Se pueden implementar parcialmente → clases abstractas.

Interfaces

- Una clase implementa una interfaz cuando proporciona código concreto para los métodos definidos en la interfaz.
  - De una misma interfaz pueden derivarse varias implementaciones.
  - Una misma clase puede implementar varias interfaces.
  - Si una clase no implementa todos los métodos definidos en una interfaces sino solo una parte, el resultado es una clase abstracta (implementación parcial).

Interfaces

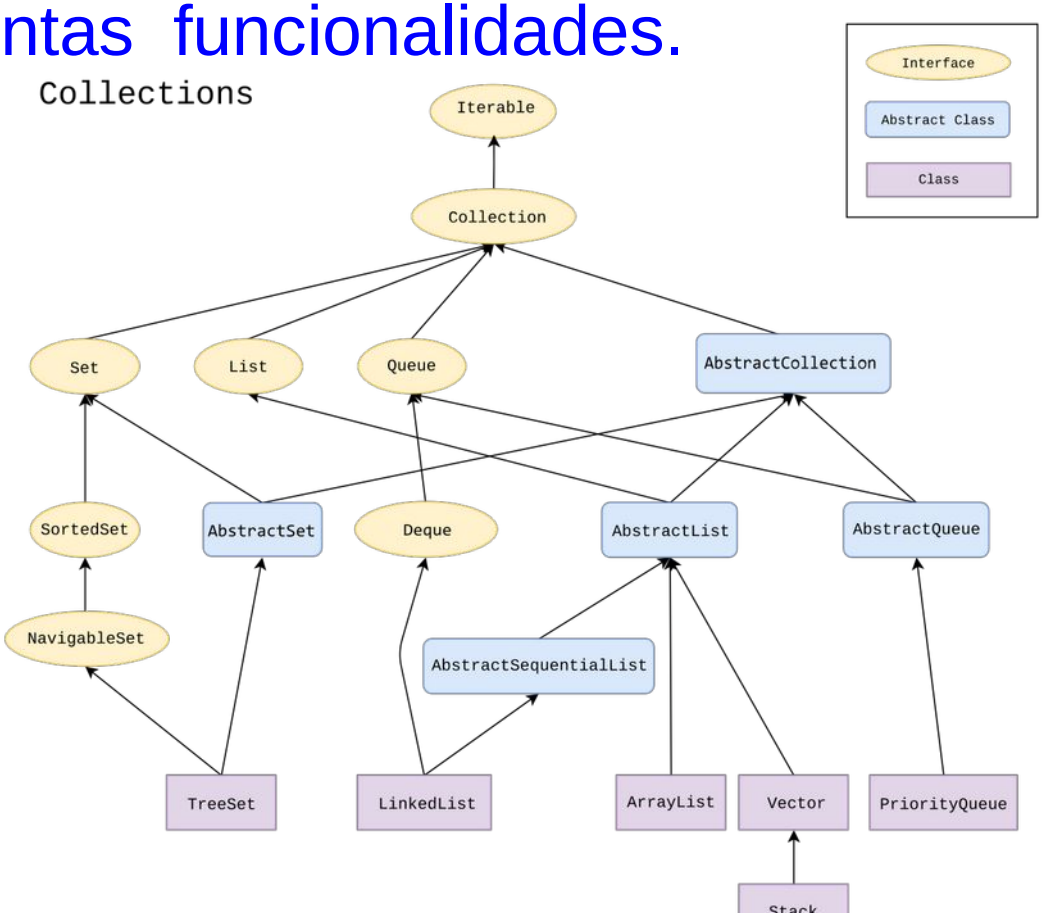

interface
extends
implements
clase abstracta
extends
clase
new
objeto
extends
extends
extends

Colecciones

- Una colección representa un grupo de objetos.
- Cada uno de los objetos de la colección son conocidos como elementos.
- Para trabajar con un conjunto de elementos, necesitamos un almacén donde guardarlos.
- Java ofrece un mecanismo general para poder trabajar con colecciones de objetos mediante la interfaz genérica `Collection`.
- Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección…

Colecciones

- Partiendo de la interfaz genérica `Collection` extienden otra serie de interfaces genéricas que cuyo objetivo es aportar distintas funcionalidades.



Colecciones

- La jerarquía de las Colecciones es extensa, pero nos centraremos en sólo en algunas implementaciones de las siguientes interfaces.

Colección	Descripción
List	Colección de objetos con una secuencia determinada
Set	Colección de objetos que no se pueden repetir
Map	Colección de objetos almacenados en parejas (clave-valor)

Algunas implementaciones

		Implementaciones				
		Tablas Hash	**Arrays redimensionables**	**Árbol balanceado**	**Listas enlazadas**	**Tablas Hash + Listas enlazadas**
Collection	**Set**	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

List

- La interfaz `List` (`java.util.List`) es una colección para representar secuencias de elementos.
- En la librería estándar de Java existen diversas clases que implementan esta interfaz.
- En nuestro caso vamos a centrarnos en la clase **`ArrayList`**, pero también tenemos las siguientes clases que implementan la interfaz `List`.
  - `LinkedList`
  - `Vector`
  - `Stack`
  - `PriorityQueue`

ArrayList

- La clase `ArrayList` permite el almacenamiento de datos en memoria de forma parecida a los arrays convencionales pero con una gran ventaja: la cantidad de elementos que puede guardar es dinámica.
- La cantidad de elementos de un array convencional está limitado por el número indicado en el momento de su creación o inicialización.
- Los `ArrayList` pueden guardar un número variable de elementos sin estar limitado por un número prefijado.
- Forma parte del paquete `java.util.ArrayList`

ArrayList. Declaración de una variable

- De forma genérica: `ArrayList nombre;`
  - Si no se especifica el tipo de datos se asume que el tipo es `Object`. Esto permite listas heterogéneas, pero tiene el inconveniente de tener que estar haciendo castings en la recuperación de los elementos.
  - Es recomendable especificar el tipo de datos que contendrá la lista. Así se utilizarán las operaciones y métodos adecuados para el tipo de datos concreto.
- Para especificar el tipo de datos: `ArrayList<Clase> nombre;`
  - En caso de guardar datos de un tipo básico de Java (char, int, double, etc...), se tiene que especificar el nombre de la clase asociada (Wrapper class): Character, Integer, Double, etc.
  - Ejemplos:
    - `ArrayList<String> paises;`
    - `ArrayList<Integer> edades;`

ArrayList. Creación de un objeto

```
ArrayList<Float> temperaturas;
temperaturas = new ArrayList<>();
```

- Se puede declarar y crear al mismo tiempo:

```
ArrayList<Float> temperaturas = new ArrayList<>();
```

- Otros ejemplos:

```
ArrayList<Grupo> grupos = new ArrayList<>();
ArrayList<Profesor> profesores = new ArrayList<>();
ArrayList<Alumno> alumnos = new ArrayList<>();
ArrayList<Asignatura> asignaturas = new ArrayList<>();
ArrayList<Aula> aulas = new ArrayList<>();
ArrayList<Matricula> matriculas = new ArrayList<>();
```

ArrayList. Creación de un objeto

- Tres constructores:
  - `ArrayList()`

    Constructor por defecto. Crea un ArrayList vacío.
  - `ArrayList(int capacidadInicial)`

    Crea un ArrayList vacío con una capacidad inicial igual a la indicada como parámetro.
  - `ArrayList(Collection c)`

    Crea un ArrayList a partir de los elementos de la colección indicada como parámetro.

ArrayList. Añadir elementos al final

- `boolean add(Object elemento)`
  - Los elementos que se van añadiendo se colocan después del último elemento.
  - El primer elemento se colocará en la posición 0.
- Ejemplo:

```
ArrayList<String> paises = new ArrayList<>();
paises.add("España");    // Ocupa la posición 0
paises.add("Francia");   // Ocupa la posición 1
paises.add("Portugal"); // Ocupa la posición 2

// Se pueden crear ArrayList para guardar datos numéricos
ArrayList<Integer> edades = new ArrayList<>();
edades.add(22);
edades.add(31);
edades.add(18);
```

ArrayList. Añadir elementos en una posición

- `void add(int posicion, Object elemento)`
  - Inserta el elemento en la posición indicada y desplaza todos los elementos uno hacia la derecha.
  - Si se intenta insertar en una posición que no existe, se producirá una excepción del tipo `IndexOutOfBoundsException`.
- Ejemplo:

```
ArrayList<String> paises = new ArrayList<>();
paises.add("España");    // Ocupa la posición 0
paises.add("Francia");   // Ocupa la posición 1
paises.add("Portugal"); // Ocupa la posición 2

// El orden hasta ahora es: España, Francia, Portugal
paises.add(1, "Italia");
// El orden ahora es: España, Italia, Francia, Portugal
```

ArrayList. Consultar un elemento

- `Object get(int indice)`
  - Obtiene el elemento que ocupa el índice indicado
- Ejemplo:

```
System.out.println(paises.get(3));
// Siguiendo con el ejemplo anterior,
mostraría: Portugal
```

ArrayList. Modificar un elemento

- `Object set(int indice, Object elemento)`
  - Permite modificar un elemento previamente guardado en la lista.
    - El primer parámetro indica el índice que ocupa el elemento a modificar.
    - El segundo parámetro indica el nuevo elemento que sustituirá al anterior.
- Ejemplo:

```
paises.set(1, "Alemania");
// Los países serían ahora: España,
Alemania, Francia, Portugal
```

ArrayList. Buscar un elemento

- `int indexOf(Object elemento)`
  - Devuelve la posición del elemento buscado.
    - Si el elemento se encuentra más de una vez, indicará la posición de la primera aparición.
    - El método `lastIndexOf` obtiene la posición del último elemento encontrado.
    - Si el elemento no se encuentra, devolverá -1.
- Ejemplo:

```
String paisBuscado = "Francia";
int pos = paises.indexOf(paisBuscado);
if (pos != -1)
    System.out.println(paisBuscado + " en la posición: " +
pos);
else
    System.out.println(paisBuscado + " no se ha encontrado");
```

ArrayList. Buscar un elemento

- `int size()`
  - Devuelve el número de elemento de la lista.
- Ejemplo:

```
for (int i=0; i < paises.size(); i++)
    System.out.println(paises.get(i));
```

ArrayList. Recorrido de una lista

- Todas las colecciones son iterables, esto es, implementan la interfaz Iterable, y por tanto pueden recorrerse mediante ForEach o con iteradores.
- Ejemplo:

```
for (String pais : paises)
    System.out.println(pais);
```

- Con iteradores:

```
Iterator<String> iter = paises.iterator();
while (iter.hasNext()) { // True si hay más elementos
    System.out.println(iter.next()); // devuelve el
elemento y apunta al siguiente
}
```

ArrayList. Otros métodos

`boolean remove(Object o)`

Elimina de la colección lo object indicado.

`void clear()`

Borra todo el contenido de la lista.

`Object clone()`

Devuelve una copia de la lista.

`boolean contains(Object o)`

Devuelve true si el elemento se encuentra en la lista y false en caso contrario.

`boolean isEmpty()`

Devuelve true si la lista está vacía.

`Object[] toArray()`

Convierte la lista en un array.

ArrayList vs Array

- **Array**
  - Tamaño fijo conocido en tiempo de compilación.
  - Todos los elementos son del mismo tipo.
  - Puede almacenar valores primitivos y referencias (Objetos).
  - Los elementos se almacenan en posiciones consecutivas de memoria.
- **ArrayList**
  - Tamaño modificable en tiempo de ejecución.
  - Todos los elementos son del mismo tipo.
  - Sólo puede almacenar referencias (Objetos)
  - Los elementos (referencias) se almacenan en posiciones consecutivas de memoria.

ArrayList vs Array

Arrays	List
`String[] paises;`	`List<String> paises;`
`paises = new String[1000];`	`paises = new ArrayList<>(1000);`
`paises[0] = "España";`	`paises.add("España");`
`String pais = paises[0];`	`String pais = paises.get(0);`
`paises[4] = "Alemania";`	`paises.set(4, "Alemania");` `// Fallará si no existe ya un país en el índice 4`

Map

- La interfaz `Map` (`java.util.Map`) se utiliza para representar colecciones de parejas de elementos en forma de **clave-valor**.
- Esta estructura de datos es conocida en otros lenguajes de programación como “Diccionarios”. Aunque en cada lenguaje la implementación puede variar, la idea final es la misma.
- Como todas las clases que implementan `Map` son colecciones, se pueden recorrer con iteradores.

Map

- Hay varias clases que implementan esta interaz. Las más empleadas son:
  - `HashMap`: los elementos no tienen un orden específico. Permite una clave con valor nulo y puede tener varios valores nulos.
  - `LinkedHashMap`: los elementos están ordenados según se han insertado. Permite una clave con valor nulo y puede tener varios valores nulos.
  - `TreeMap`: los elementos se ordenan por la clave de forma "natural". Por ejemplo, si la clave son valores enteros los ordena de menor a mayor. No permite claves con valor null pero puede tener varios valores nulos.
- En nuestro caso, vamos a centrarnos en `HashMap`.

HashMap. Declaración de una variable

- De forma general:

```
HashMap nombre;
```

  - Si no se especifica el tipo de datos, tanto la clave como el valor sería de tipo `Object`. Esto permite mapas heterogéneas, pero es necesario hacer castings en la recuperación de los elementos.
  - Es recomendable especificar el tipo de datos que contendrán las parejas clave-valor. Así se utilizarán las operaciones y métodos adecuados para el tipo de datos concreto.

- Para especificar el tipo de datos:

```
HashMap<Integer, String> nombre;
```

HashMap. Creación de un objeto

```
HashMap<Integer, String> nombre;
nombre = new HashMap<>();
```

- Se puede declarar y crear al mismo tiempo.

```
HashMap<Integer, String> nombre = new HashMap<>();
```

- Ejemplos:

```
HashMap<String, String> diccionario = new HashMap<>();
HashMap<Moneda, Double> cotizaciones = new HashMap<>();
HashMap<Jugador, Integer> puntuaciones = new HashMap<>();
HashMap<String, Integer> repeticiones = new HashMap<>();
HashMap<Entity, Sprite> sprites = new HashMap<>();
```

HashMap. Principales métodos

- `Object put(Object key, Object value)`

  Asocia el valor value con el elemento que tiene la clave key. Si ya existe un elemento con esa clave reemplaza su valor. Si el elemento con la clave key ya tenía un valor, devuelve el valor viejo, en caso contrario devuelve null.

- `Object get(Object key)`

  Obtiene el elemento del Map que tiene como clave key. Devuelve null si no encuentra el elemento.

- `Collection<Object> values()`

  Obtiene la lista de los valores que están al Map.

- `Set<Object> keySet()`

  Obtiene el conjunto de claves que están al Map.

- `boolean replace(Object key, Object value)`

  Cambia el valor del elemento con la clave key por newValue. Devuelve true si se ha podido cambiar el valor.

- `boolean replace(Object key, Object oldValue, Object newValue)`

  Cambia el valor el elemento con la clave key que tiene el valor oldValue por newValue. Devuelve true si se ha podido cambiar el valor.

HashMap. Principales métodos

- Los métodos de la interfaz `Map` son similares a los que hemos visto para las listas.
- `void clear()`

  Borra todo el contenido del Map.
- `Object clone()`

  Devuelve una copia del Map.
- `boolean containsKey(Object key)`

  Devuelve true si hay algún elemento que tenga como clave key.
- `boolean containsValue(Object value)`

  Devuelve true si hay algún elemento que tenga como valor value.
- `boolean isEmpty()`

  Devuelve true si lo Map está vacío.
- `int size()`

  Devuelve el número de elementos que tiene el Map.

Sobreescribir equals y hashCode

- La interfaz `Map` hace uso de los métodos **`equals`** y **`hashCode`** para determinar si una clave ya existe.
- Por ello, es MUY IMPORTANTE que SIEMPRE que utilicemos clases que implementan la interaz Map, tengamos sobreescritos y bien definidos los métodos:
  - `boolean equals(Object o)`
  - `int hashCode()`

Método equals

- Se utiliza para **determinar la igualdad de un Objeto con otro**. El método debe devolver true si los objetos se consideran “iguales” o false si se consideran distintos.
- Java no puede saber cuando nosotros consideramos que dos objetos son el mismo, por lo que por defecto equals considera que dos objetos son iguales si apuntan al mismo objeto, es decir, apuntan a la misma dirección de memoria y por tanto son el mismo objeto.
- Pero, ¿qué pasa si son distintas instancias pero representan el mismo objeto? Entonces hay que ayudar a Java sobreescribiendo el método equals de la clase que queramos comparar, indicando qué atributos debe utilizar y cómo debe compararlos para considerar que dos objetos son el mismo.
- **Siempre** que sobreescribamos el método `equals`, debemos sobreescribir también el método `hashCode`.

Características del método equals

- El método `equals` debe cumplir los siguientes requisitos para cualquier valor distinto de null:
  - **Reflexivo**: es decir x.equals(x) debe devolver true.
  - **Simétrico**: x.equals(y) == y.equals(x)
  - **Transitivo**: si x.equals(y) == true y y.equals(z) == true entonces x.equals(z) == true
  - **Consistente**: múltiples invocaciones de x.equals(y) deben devolver el mismo valor si x e y no han cambiado.
  - Si x.equals(y) == true entonces x.hashCode() == y.hashCode().
  - x.equals(null) debe devolver false.
- Los IDEs traen ayudas para generar correctamente el método `equals` y el método `hashCode`.

Método hashCode

- El objetivo del método **`hashCode`** es devolver un número entero que identifique al objeto.
- Por defecto, el método **`hashCode`** implementado en la clase `Object` devuelve una representación numérica de la dirección de memoria (interna a la JVM no la real) donde se encuentra el objeto.
- La interfaz `Map` utiliza el valor devuelto por `hashCode` para determinar el índice en el que se encuentra un objeto dada una clave determinada.

Características del método hashCode

- El método `hashCode` debe cumplir los siguientes requisitos:
  - **Distintas ejecuciones** de la aplicación deberían consistentemente devolver el **mismo** `hashCode` dado el mismo objeto. (Esto no lo cumple Java)
  - Si dos objetos se consideran **iguales**, deben devolver el **mismo** `hashCode`.