Sobre los métodos equals()y hashCode()
Ya están explicados, si dudas repasa *Sobrescritura* donde se explica estos métodos. Un mini resumen es que estos métodos, heredados de Object, se basan en el valor único de la referencia o posición en memoria del objeto, pero si queremos que en lugar de en la referencia se basen en su estado, es decir, en su contenido, debemos sobreescribirlos.

LA NECESIDAD DE SOBREESCRIBIR EQUALS AL TRABAJAR CON COLECCIONES
Para las colecciones es importante el método equals por ejemplo, el método contains de la interface List usa equals()

contains

`boolean contains(Object o)`

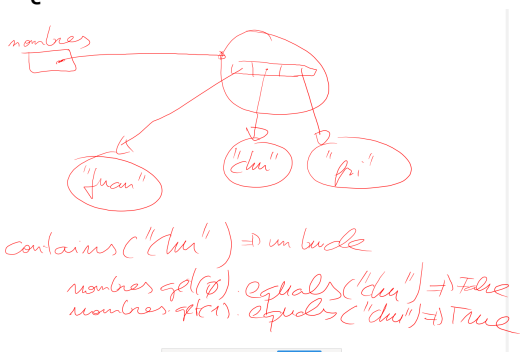
Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element e such that `(o==null ? e==null : o.equals(e))`.

Es decir, para saber si una lista contiene un objeto, recorre la lista y va comparando dicho objeto con cada elemento usando para comparar equals().

Debes de entender a la perfección la salida del siguiente programa:

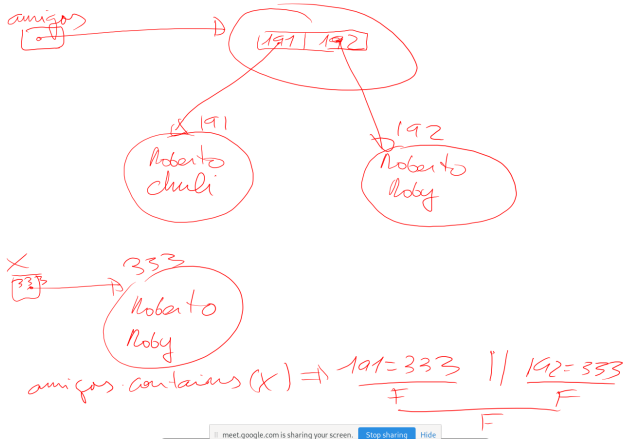
```
import java.util.ArrayList;
import java.util.List;


class App{
    public static void main(String[] args) {
      List<String> nombres= new ArrayList<>();
      nombres.add("Juan");nombres.add("Chu");nombres.add("Fri");
      System.out.println(nombres.contains("For"));
      System.out.println(nombres.contains("Chu"));
    }
}
```

El código anterior, funciona como esperamos PORQUE LA CLASE STRING TIENE EQUALS Y HASHCODE SOBRESCRITOS.



Supongamos ahora que queremos tener una lista de la clase Amigo. Un amigo tiene un nombre y un mote. Para este ejemplo considero que la combinación nombre/mote es única, de forma que para identificar a un amigo no basta sólo con el nombre o con el mote, necesito por tanto los dos valores. Si no se sobrescribe equals() el contains no funciona bien

```
import java.util.ArrayList;
import java.util.List;

//suponemos que un la combinación de nombre y mote es única
class Amigo{
    String nombre;
    String mote;
    Amigo(String nombre, String mote){
        this.nombre=nombre;
        this.mote=mote;
    }
}
class App{
    public static void main(String[] args) {
        List<Amigo> amigos= new ArrayList<>();
        amigos.add(new Amigo("Roberto","chuli"));
        amigos.add(new Amigo("Roberto","Roby"));
        Amigo x=new Amigo("Roberto","Roby");
        System.out.println(amigos.contains(x));
    }
}
```



En el código anterior, el *equals()* de la clase Amigo es el heredado por Object y no funciona por contenido. Por lo tanto contains() compara referencias como se indica en el dibujo de arriba.

En cambio, si se sobreescribe equals() en Amigo el funcionamiento de contains() es el esperado

```
import java.util.ArrayList;
import java.util.List;

//suponemos que un la combinación de nombre y mote es única
class Amigo{
    String nombre;
    String mote;
    Amigo(String nombre, String mote){
        this.nombre=nombre;
        this.mote=mote;
    }
    public boolean equals(Object o){
        if (o==null) return false;
        if(!(o instanceof Amigo)) return false;
        Amigo a = (Amigo) o; //si no hago cast no puedo acceder a nombre y mote
        return (this.nombre.equals(a.nombre)&&this.mote.equals(a.mote));
    }
}
class App{
    public static void main(String[] args) {
```

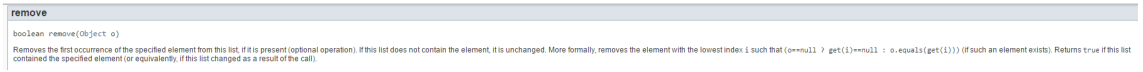
```
        List<Amigo> amigos= new ArrayList<>();
        amigos.add(new Amigo("Roberto","chuli"));
        amigos.add(new Amigo("Roberto","Roby"));
        Amigo x=new Amigo("Roberto","Roby");
        System.out.println(amigos.contains(x));
    }
}
```

Observa la forma típica de sobrescribir equals:

1. Primero comprueba si la referencia vale null o es de otro tipo en cuyo caso devuelve lógicamente false(por ej. Un Objeto Coche nunca será igual a un objeto Persona, ni será a igual a null)
2. Se hace un cast para manipular el contenido del Objeto de la forma que nos interese considerar que dos objetos son iguales. Frecuentemente el criterio será que todos los valores de sus atributos sean iguales, pero no necesariamente.

Como es bastante mecánico y predecible escribir esta sobreescritura normalmente nos quedamos con la versión que genera automáticamente el IDE.

Otro método de listas que usa equals() es remove(). Hay dos versiones, una borra por índice y otra por contenido. Esta segunda versión también necesita saber "que es igual".


remove

boolean remove(Object o)

Removes the first occurrence of the specified element from this list, if it is present (optional operation). If this list does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))) (if such an element exists). Returns true if this list contained the specified element (or equivalently, if this list changed as a result of the call).

Conclusión: sobrescribir equals() es casi obligatorio para aquellas clases que quiero integrar en colecciones. Además aunque en el ejemplo anterior no fue necesario, siempre que se sobreescribe equals() se debe sobreescribir hashCode() por coherencia. Imagina que cuelgas la clase Amigo en Internet y otros programadores quieren almacenar Amigos en un hashMap en lugar de en una lista, si no sobreescribimos hashCode(), habrá problemas como veremos a continuación

LA NECESIDAD DE SOBRESCRIBIR HASHCODE

Si se sobrescribe equals, es obligatorio "de facto" sobrescribir hashCode() ya que en el "contrato" especificado por Oracle de hashCode indica que si dos objetos se comparan con equals y dan true, deben de tener el mismo hashCode(). Si no hacemos esto los usuarios de nuestras clases se volverán locos al usarlas ya que cuentan con que "cumplimos el contrato" y al no hacerlo no les salen las cuentas en ciertas situaciones. Así, las colecciones java que manejan equals y hashCode funcionan de forma que cuentan con que los objetos que almacenan "cumplen el contrato". En el caso anterior sólo escribimos equals() y funcionaba correctamente el ejemplo, pero vamos a pasar a ver otra situación en la que si no sobrescribimos hashCode() el funcionamiento falla.

Ejemplo de HashMap con el que detectaremos la necesidad de sobreescribir equals y hashCode

Fíjate ahora en el siguiente código que usa el ejemplo

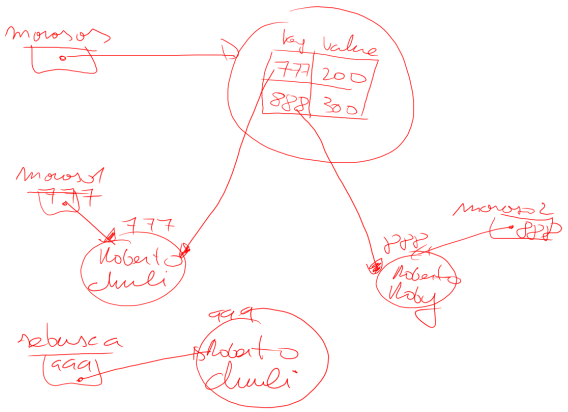
```
Amigo seBusca=new Amigo("Roberto","chuli");
System.out.println("sebusCa.hashCode(): "+ seBusca.hashCode());
```

```
seBusca) :"+ morosos.containsKey(seBusca));
 System.out.println("morosos.get(seBusca) :"+ morosos.get(seBusca));
```

El ejemplo:

```
import java.util.HashMap;
import java.util.Set;
class Amigo{
    String nombre;
    String mote;
    Amigo(String nombre, String mote){
        this.nombre=nombre;
        this.mote=mote;
    }

    @Override
    public String toString(){
        return nombre+" el "+mote+ " ";
    }
}

class App{
    public static void main(String[] args) {
        HashMap<Amigo,Integer> morosos= new HashMap<>();
        Amigo moroso1=new Amigo("Roberto","chuli");
        Amigo moroso2=new Amigo("Roberto","Roby");
        System.out.println("Codigo hash de moroso1 el chuli: " +moroso1.hashCode());
        System.out.println("Codigo hash de moroso2 Robi: " +moroso2.hashCode());
        morosos.put(moroso1,200);
        morosos.put(moroso2,300);
        Set<Amigo> claves=morosos.keySet();
        for(Amigo a:claves){
            System.out.println(a+ "  "+morosos.get(a)+"   Codigo hash de clave: "+a.hashCode());

        }
        Amigo seBusca=new Amigo("Roberto","chuli");


        System.out.println("morosos.get(seBusca) ¡ERROR! :"+ morosos.get(seBusca));
        System.out.println("morosos.get(moroso1) ¡OK! :"+ morosos.get(moroso1));


    }
}
```

Creamos un objeto "seBusca", ya que es un objeto diferente, aunque con el mismo contenido que otro existente en el mapa, voy a tener problemas con containsKey() y con get().
En el siguiente dibujo para simplificar los objetos value como son Integer ponemos directamente su valor en nuestra tabla imaginaria, realmente tendría que hacer flechas a objetos Integer
Y con este dibujo tienes que ser capaz de razonar el porqué de los dos últimos println()



¿Por qué pueden fallar métodos tipo containsKey(), get etc. al trabajar con mapas?
Al trabajar con mapas se usa:

- equals() para las comparaciones de forma similar a las listas.
- Pero además con mapas se usa hashCode() para conseguir acceso directo típico de esta estructura

Y en muchos casos sólo funciona como queremos si están sobreescritos ambos.

¿Por qué necesitamos sobreescribir hashCode() para trabajar con mapas?
Una explicación poco detallada, pero sencilla para salir del paso es la siguiente.
Un mapa es una estructura que permite el acceso directo, por lo tanto para localizar un elemento en el mapa no se hace un barrido secuencial del mapa. ¿Cómo se consigue el acceso directo?. De forma "parecida" al acceso directo de un array, haciendo un cálculo de posición de memoria. En un array el cálculo se hace a partir del valor de un índice i. En un mapa el cálculo se hace con el valor que devuelve hashCode() del objeto que hace de clave.

Por lo tanto, el ejemplo anterior lo solucionamos a continuación de forma que todos los objetos con el mismo contenido devuelve el mismo valor a través de hashCode().

El ejemplo completo con equals() y hashcode sobreescrito para que funcionen por contenido:

```
import java.util.HashMap;
import java.util.Set;

class Amigo{
    String nombre;
    String mote;
    Amigo(String nombre, String mote){
        this.nombre=nombre;
        this.mote=mote;
    }

    @Override
    public String toString(){
        return nombre+" el "+mote+ " ";
    }

    @Override
    public boolean equals(Object o){
      if (o==null) return false;
      if(!(o instanceof Amigo)) return false;
      Amigo a = (Amigo) o;
      return (this.nombre.equals(a.nombre)&&this.mote.equals(a.mote));
    }
    @Override
```

```
    public int hashCode(){//malucho inventado por mi,luego usaré el del IDE
        int sumaNombre=0;
        for(int i=0;i<nombre.length();i++){
            sumaNombre+=nombre.charAt(i);
        }
        int sumaMote=0;
        for(int i=0;i<mote.length();i++){
            sumaNombre+=mote.charAt(i);
        }
        return sumaNombre+sumaMote;
    }


}


class App{
    public static void main(String[] args) {
        HashMap<Amigo,Integer> morosos= new HashMap<>();
        Amigo moroso1=new Amigo("Roberto","chuli");
        Amigo moroso2=new Amigo("Roberto","Roby");
        System.out.println("Codigo hash de moroso1 el chuli: " +moroso1.hashCode());
        System.out.println("Codigo hash de moroso2 Robi: " +moroso2.hashCode());
        morosos.put(moroso1,200);
        morosos.put(moroso2,300);
        Set<Amigo> claves=morosos.keySet();
        for(Amigo a:claves){
            System.out.println(a+ "  "+morosos.get(a)+"   Codigo hash de clave: "+a.hashCode());

        }
        Amigo seBusca=new Amigo("Roberto","chuli");


        System.out.println("morosos.get(seBusca) ¡AHORA HACE LO QUE QUIERO! :"+ morosos.get(seBusca));
        System.out.println("morosos.get(moroso1) ¡OK! :"+ morosos.get(moroso1));

    }
}
```

Se puede escribir hashCode() de muchas maneras, pero ojo, la "calidad" de hashCode() puede influir en el rendimiento del HashMap. Para entenderlo bien debimos de haber hecho una "tabla hash casera", de todas formas nos podemos hacer una idea: si la función hashCode() es mala, provocará muchas colisiones, es decir, muchos objetos con el mismo hashCode y esto provocará internamente listas enlazadas muy largas con lo cual perdemos velocidad de acceso.

Generar hashcode() y equals() con IDE

Las técnicas para generar buenos métodos hashcode() y equals() son rutinarias y por tanto los IDEs pueden generar código para sobrescribir estos métodos.

El hashcode basado en contenido deja de ser único

Observa que al basarnos en contenido, el hashCode() deja de ser único y hay dos observaciones importantes:

1. Mismo contenido tiene que tener mismo hashCode(). Si un Objeto1 y otro Objeto2 son iguales según el método equals(), también tienen que tener el mismo código hash
2. Diferente contenido puede provocar mismo hashCode() y el funcionamiento es correcto(salvo empeoramiento de rendimiento).

CONCLUSIÓN

Entender el porqué es necesario sobrescribir hashcode y equals es más laborioso que su aplicación práctica que simplemente se reduce a:
Cuando consideremos que una clase puede formar parte de colecciones debe de tener sobrescrito el hashCode() y equals(). Utilizar la generación automática de código de los IDES para sobrescribir es una buena opción.

Ejercicio U7_B8A_E1: Hicimos el siguiente ejercicio en el boletín de mapas
Tenemos la siguiente tabla de latitudes/longitudes de ciudades nacionales e internacionales.

CIUDAD	LATITUD	LONGITUD
LUGO	43.01 N	7.33 O
BARCELONA	41.23 N	2.11 E
MADRID	40.24 N	3.41 O
LIMA	12.03 S	77.03 0

Las coordenadas son obligatoriamente objetos de la siguiente clase, a la que añadirás los métodos necesarios

```
class Coordenadas{
   private String latitud;
   private String longitud;
}
```

Los datos de la tabla anterior se almacenan obligatoriamente en un hashmap cuya clave es el nombre de la ciudad, y el valor almacenado será un objeto coordenada

```
Map<Coordenadas,String> ciudades= new HashMap<>();
```

Tu programa debe:

- Insertar los datos (para simplificar, directamente en el código, no por teclado).
- Imprimir el contenido del mapa.

AHORA SE PIDE: ahora la clave debe ser el objeto Coordenada (en lugar del nombre de la Ciudad)