Part 8

Similarity of objects

Learning Objectives

- You'll revise object equivalence comparison using the equals method.
- You'll know what the hashCode method does.
- You'll know how approximate equalities of objects can be compared.
- You'll know how to use the programming environment's ready-made tools to create equals and hashCode methods.

Let's revise the `equals` method used to compare objects, and become familiar with the `hashCode` method used in making approximate comparisons.

Method to Test For Equality - "equals"

The equals method checks by default whether the object given as a parameter has the same reference as the object it is being compared to. In other words, the default behaviour checks whether the two objects are the same. If the reference is the same, the method returns `true`, and `false` otherwise.

This can be illustrated with the following example. The `Book` class does not have its own implementation of the `equals` method, so it falls back on the default implementation provided by Java.

```
Book bookObject = new Book("Book object", 2000, "...");
Book anotherBookObject = bookObject;

if (bookObject.equals(anotherBookObject)) {
    System.out.println("The books are the same");
} else {
    System.out.println("The books aren't the same");
}

// we now create an object with the same content that's nonetheless its own
anotherBookObject = new Book("Book object", 2000, "...");
```

```
if (bookObject.equals(anotherBookObject)) {
    System.out.println("The books are the same");
} else {
    System.out.println("The books aren't the same");
```

Sample output

```
The books are the same
The books aren't the same
```

The internal structure of the book objects (i.e., the values of their instance variables) in the previous example is the same, but only the first comparison prints "`The books are the same`". This is because the references are the same in the first case, i.e., the object is compared to itself. The second comparison is about two different entities, even though the variables have the same values.

For strings, `equals` works as expected in that it declares two strings *identical in content* to be 'equal' even if they are two separate objects. The String class has replaced the default `equals` with its own implementation.

If we want to compare our own classes using the `equals` method, then it must be defined inside the class. The method created accepts an `Object`-type reference as a parameter, which can be any object. The comparison first looks at the references. This is followed by checking the parameter object's type with the `instanceof` operation - if the object type does not match the type of our class, the object cannot be the same. We then create a version of the object that is of the same type as our class, after which the object variables are compared against each other.

```
public boolean equals(Object comparedObject) {
    // if the variables are located in the same place, they're the same
    if (this == comparedObject) {
        return true;
    }

   // if comparedObject is not of type Book, the objects aren't the same
   if (!(comparedObject instanceof Book)) {
        return false;
    }
```

```
    // let's convert the object to a Book-olioksi
    Book comparedBook = (Book) comparedObject;

    // if the instance variables of the objects are the same, so are the object
    if (this.name.equals(comparedBook.name) &&
        this.published == comparedBook.published &&
        this.content.equals(comparedBook.content)) {
        return true;
    }

    // otherwise, the objects aren't the same
    return false;
```

The `Book` class in its entirety.

```
public class Book {
    private String name;
    private String content;
    private int published;

    public Book(String name, int published, String content) {
        this.name = name;
        this.published = published;
        this.content = content;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPublished() {
        return this.published;
    }

    public void setPublished(int published) {
        this.published = published;
    }

    public String getContent() {
        return this.content;
    }
```

```
    public void setContent(String content) {
        this.content = content;
    }

    public String toString() {
        return "Name: " + this.name + " (" + this.published +   ")\n"
            + "Content: " + this.content;
    }

    @Override
    public boolean equals(Object comparedObject) {
        // if the variables are located in the same place, they're the same
        if (this == comparedObject) {
            return true;
        }

        // if comparedObject is not of type Book, the objects aren't the same
        if (!(comparedObject instanceof Book)) {
            return false;
        }

        // let's convert the object to a Book-object
        Book comparedBook = (Book) comparedObject;

        // if the instance variables of the objects are the same, so are the ob
        if (this.name.equals(comparedBook.name) &&
            this.published == comparedBook.published &&
            this.content.equals(comparedBook.content)) {
            return true;
        }

        // otherwise, the objects aren't the same
        return false;
    }
```

Now the book comparison returns `true` if the instance variables of the books are the same.

```
Book bookObject = new Book("Book Object", 2000, "...");
Book anotherBookObject = new Book("Book Object", 2000, "...");

if (bookObject.equals(anotherBookObject)) {
    System.out.println("The books are the same");
} else {
```

```
    System.out.println("The books aren't the same");
}
```

Sample output

The books are the same

Quiz:
Equals

Points:
1/1

Below there is a Java class that represents a minimal notepad.

```
public class Notepad {
    private String name;
    private int year;

    public Notepad(String name, int year) {
        this.name = name;
        this.year = year;
    }

    public boolean equals(Object object) {
        if (object == null || this.getClass() != object.getClass()) {
            return false;
        }

        if (object == this) {
            return true;
        }

        Notepad compared = (Notepad) object;

        return this.nimi.equals(compared);
    }
}
```

What does the following program print?

```
Notepad basics = new Notepad("Equals basics", 2000);
Notepad advanced = new Notepad("Equals advanced", 2001);

System.out.println(basics.equals(basics));
System.out.println(basics.equals(advanced));
```

```
System.out.println(basics.equals(new Notepad("Equals basics", 2000)));
System.out.println(basics.equals(new Notepad("Equals basics", 2001)));
```

Select the correct answer

- ✓ true
false
false
false
- true
false
true
false
- true
false
true
true
- false
false
false
false

The answer is correct

Answered Tries remaining: 1

The ArrayList also uses the `equals` method in its internal implementation. If we don't define the `equals` method in our objects, the `contains` method of the ArrayList does not work properly. If you try out the code below with two Book classes, one with the `equals` method defined and another without it, you'll see the difference.

```
ArrayList<Book> books = new ArrayList<>();
Book bookObject = new Book("Book Object", 2000, "...");
books.add(bookObject);

if (books.contains(bookObject)) {
    System.out.println("Book Object was found.");
}

bookObject = new Book("Book Object", 2000, "...");

if (!books.contains(bookObject)) {
    System.out.println("Book Object was not found.");
}
```

This reliance on default methods such as `equals` is actually the reason why Java demands that variables added to ArrayList and HashMap are of reference type. Each reference type variable comes with default methods, such as equals, which means that you don't need to change the internal implementation of the ArrayList class when adding variables of different types. Primitive variables do not have such default methods.

Quiz:

Equals and ArrayList

Points:

1/1

Below is a Java class that represents a message.

```
public class Message {
    private String name;

    public Message(String name) {
        this.name = name;
    }
}
```

What does the following program print?

```
Message sms = new Message("SMS");
Message mms = new Message("MMS");

System.out.println(sms.equals(sms));
System.out.println(sms.equals(mms));
```

```
ArrayList<Message> messages = new ArrayList<>();
if (!messages.contains(sms)) {
    messages.add(sms);
}

if (!messages.contains(sms)) {
    messages.add(sms);
}

if (!messages.contains(new Message("SMS"))) {
    messages.add(sms);
}

System.out.println(messages.size());
```

Select the correct answer

false
false
3

false
false
2

false
false
0

true
false
3

✓ true
false
2

true

false

0

The answer is correct

Answered Tries remaining: 2

Approximate Comparison With HashMap

In addition to `equals`, the `hashCode` method can also be used for **approximate comparison of objects**. The method creates from the object a "hash code", i.e, a number, that tells a bit about the object's content. If two objects have the same hash value, they may be equal. On the other hand, if two objects have different hash values, they are certainly unequal.

Hash codes are used in HashMaps, for instance. HashMap's internal behavior is based on the fact that key-value pairs are stored in an array of lists based on the key's hash value. Each array index points to a list. The hash value identifies the array index, whereby the list located at the array index is traversed. The value associated with the key will be returned if, and only if, the exact same value is found in the list (equality comparison is done using the equals method). This way, the search only needs to consider a fraction of the keys stored in the hash map.

So far, we've only used String and Integer-type objects as HashMap keys, which have conveniently had ready-made `hashCode` methods implemented. Let's create an example where this is not the case: we'll continue with the books and keep track of the books that are on loan. We'll implement the book keeping with a HashMap. The key is the book and the value attached to the book is a string that tells the borrower's name:

```
HashMap<Book, String> borrowers = new HashMap<>();

Book bookObject = new Book("Book Object", 2000, "...");
borrowers.put(bookObject, "Pekka");
borrowers.put(new Book("Test Driven Development", 1999, "..."), "Arto");
```

```
System.out.println(borrowers.get(bookObject));
System.out.println(borrowers.get(new Book("Book Object", 2000, "...")));
```

Sample output

```
Pekka
null
null
```

We find the borrower when searching for the same object that was given as a key to the hash map's `put` method. However, when searching by the exact same book but with a different object, a borrower isn't found, and we get the *null* reference instead. The reason lies in the default implementation of the `hashCode` method in the `Object` class. The default implementation creates a `hashCode` value based on the object's reference, which means that books having the same content that are nonetheless different objects get different results from the hashCode method. As such, the object is not being searched for in the right place.

For the HashMap to work in the way we want it to, that is, to return the borrower when given an object with the correct *content* (not necessarily the same object as the original key), the class that the key belongs to must overwrite the `hashCode` method in addition to the `equals` method. The method must be overwritten so that it gives the same numerical result for all objects with the same content. Also, some objects with different contents may get the same result from the hashCode method. However, with the HashMap's performance in mind, it is essential that objects with different contents get the same hash value as rarely as possible.

We've previously used `String` objects as HashMap keys, so we can deduce that the `String` class has a well-functioning `hashCode` implementation of its own. We'll *delegate*, i.e., transfer the computational responsibility to the `String` object.

```
public int hashCode() {
    return this.name.hashCode();
}
```

The above solution is quite good. However, if `name` is *null*, we see a `NullPointerException` error. Let's fix this by defining a condition: if the value of the `name` variable is *null*, we'll return the year of publication as the hash value.

```
public int hashCode() {
    if (this.name == null) {
        return this.published;
    }

    return this.name.hashCode();
}
```

Now, all of the books that share a name are bundled into one group. Let's improve it further so that the year of publication is also taken into account in the hash value calculation that's based on the book title.

```
public int hashCode() {
    if (this.name == null) {
        return this.published;
    }

    return this.published + this.name.hashCode();
}
```

It's now possible to use the book as the hash map's key. This way the problem we faced earlier gets solved and the book borrowers are found:

```
HashMap<Book, String> borrowers = new HashMap<>();

Book bookObject = new Book("Book Object", 2000, "...");
borrowers.put(bookObject, "Pekka");
borrowers.put(new Book("Test Driven Development",1999, "..."), "Arto");

System.out.println(borrowers.get(bookObject));
System.out.println(borrowers.get(new Book("Book Object", 2000, "...")));
System.out.println(borrowers.get(new Book("Test Driven Development", 1999)));
```

Output:

Sample output

Pekka
Pekka
Arto

Let's review the ideas once more: for a class to be used as a HashMap's key, we need to define for it:

- the `equals` method, so that all equal or approximately equal objects cause the comparison to return true and all false for all the rest
- the `hashCode` method, so that as few objects as possible end up with the same hash value

Assisted creation of the equals method and hashCode

NetBeans provides support for the creation of both `equals` and `hashCode`. You can select Source -> Insert Code from the menu and then select *equals() and hashCode()* from the drop-down list. NetBeans then asks for the instance variables used in the methods. The methods developed by NetBeans are typically sufficient for our needs.

Use NetBeans's support in creating the equals and hashCode methods until you know that your methods are better than those created automatically by NetBeans.

Programming exercise:
Same date

Points
1/1

The exercise template contains a class `SimpleDate`, which defines a date object based on a given day, month, and year. In this exercise you will expand the SimpleDate class with an equals method, which can tell if the dates are exactly the same.

Create a method `public boolean equals(Object object)` for the `SimpleDate` class, which returns true if the date of the object passed

to the method as a parameter is the same as the date of the object used to call the method.

The method should work as follows:

```
SimpleDate d = new SimpleDate(1, 2, 2000);
System.out.println(d.equals("heh"));
System.out.println(d.equals(new SimpleDate(5, 2, 2012)));
System.out.println(d.equals(new SimpleDate(1, 2, 2000)));
```

Sample output

```
false
false
true
```

Exercise submission instructions

How to see the solution

Programming exercise:

Hash for date

Points 1/1

Let's expand the `SimpleDate` class from the previous exercise to also have its own `hashCode` method.

Create a method `public int hashCode()` for the `SimpleDate` class, which calculates a hash for the the SimpleDate object. Implement the calculation of the hash in way that there are as few similar hashes as possible between the years 1900 and 2100.

Exercise submission instructions

How to see the solution

Programming exercise:

Vehicle Registry (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Part 1: Equals and hashCode for the LicensePlate class

European license plates have two parts: a two letter country code and a nationally unique license number. The license number is made up of numbers and characters. License plates are represented by the following class:

```
public class LicensePlate {

    // these instance variables have been defined as final, meaning
    // that once set, their value can't be changed
    private final String liNumber;
    private final String country;

    public LicensePlate(String country, String liNumber) {
        this.liNumber = liNumber;
        this.country = country;
    }

    @Override
    public String toString() {
        return country + " " + liNumber;
    }
```

We want to be able to save the license plates in ArrayLists and to use them as keys in a HashMap. This, as explained above, means that the `equals` and `hashcode` methods need to be overwritten, or they won't

work as intended. Implement the methods `equals` and `hashCode` for the LicensePlate class.

Example program:

```
public static void main(String[] args) {
    // the following is the same sample program shown in ex 8.13 descriptio

        LicensePlate li1 = new LicensePlate("FI", "ABC-123");
        LicensePlate li2 = new LicensePlate("FI", "UXE-465");
        LicensePlate li3 = new LicensePlate("D", "B WQ-431");

        ArrayList<LicensePlate> finnishPlates = new ArrayList<>();
        finnishPlates.add(li1);
        finnishPlates.add(li2);

        LicensePlate newLi = new LicensePlate("FI", "ABC-123");
        if (!finnishPlates.contains(newLi)) {
            finnishPlates.add(newLi);
        }
        System.out.println("finnish: " + finnishPlates);
        // if the equals-method hasn't been overwritten, the same license n

        HashMap<LicensePlate, String> owners = new HashMap<>();
        owners.put(li1, "Arto");
        owners.put(li3, "Jürgen");

        System.out.println("omistajat:");
        System.out.println(owners.get(new LicensePlate("FI", "ABC-123")));
        System.out.println(owners.get(new LicensePlate("D", "B WQ-431")));
        // if the hasCode-method hasn't been overwritten, the owners won't
}
```

Implement the methods `equals` and `hashCode`. When they are implemented correctly the example program above will print:

Sample output

```
Finnish: [FI ABC-123, FI UXE-465]
owners:
Arto
Jürgen
```

Part 2: Pairing plates with owners

Implement the class `VehicleRegistry`, which has the following methods:

- `public boolean add(LicensePlate licensePlate, String owner)` assigns the owner it received as a parameter to a car that corresponds to the license plate received as a parameter. If the license plate doesn't have an owner, the method returns true. If the license already has an owner attached, the method returns false and does nothing.
- `public String get(LicensePlate licensePlate)` returns the owner of the car corresponding to the license plate received as a parameter. If the car isn't in the registry, the method returns null.
- `public boolean remove(LicensePlate licensePlate)` removes the license plate and attached data from the registry. The method returns true if removed successfully and false if the license plate wasn't in the registry.

Part 3: Expanded vehicle registry

Add the following methods to the `VehicleRegistry`:

- `public void printLicensePlates()` prints the license plates in the registry.
- `public void printOwners()` prints the owners of the cars in the registry. Each name should only be printed once, even if a particular person owns more than one car.

Useful tip! In the printOwners method, you can create a list used for remembering the owners that were already printed. If an owner is not on the list, their name is printed and they are added to the list; conversely, if an owner is on the list, their name isn't printed.

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 4. Grouping data using hash maps

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI