Part 8

Hash Map

Learning Objectives

- You're familiar with the concept of a hash map and know how one works.
- You know how to use Java's hash map: you know how to create one, add information to it and retrieve information from it.
- You can describe situations where using a hash map could be useful.
- You know how to use a hash map as an instance variable.
- You know how to go through keys and values of a hash map using the for-each loop.

A HashMap is, in addition to ArrayList, one of the most widely used of Java's pre-built data structures. The hash map is used whenever data is stored as key-value pairs, where values can be added, retrieved, and deleted using keys.

In the example below, a HashMap object has been created to search for cities by their postal codes, after which four postal code-city pairs have been added to the HashMap object. At the end, the postal code "00710" is retrieved from the hash map. Both the postal code and the city are represented as strings.

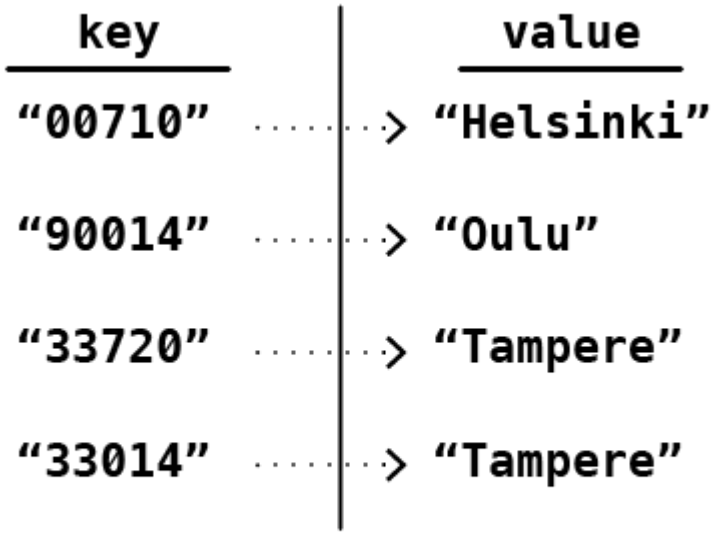
```
HashMap<String, String> postalCodes = new HashMap<>();
postalCodes.put("00710", "Helsinki");
postalCodes.put("90014", "Oulu");
postalCodes.put("33720", "Tampere");
postalCodes.put("33014", "Tampere");

System.out.println(postalCodes.get("00710"));
```

Sample output

```
Helsinki
```

The internal state of the hash map created above looks like this. Each key refers to some value.



If the hash map does not contained the key used for the search, its `get` method returns a `null` reference.

```
HashMap<String, String> numbers = new HashMap<>();
numbers.put("One", "Uno");
numbers.put("Two", "Dos");

String translation = numbers.get("One");
System.out.println(translation);

System.out.println(numbers.get("Two"));
System.out.println(numbers.get("Three"));
System.out.println(numbers.get("Uno"));
```

Sample output

```
Uno
Dos
null
null
```

Quiz:
Retrieve from a hash map

Points:
1/1

Which command retrieves the value 3 from the hash map?

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

conversionMap.put("3", 9);
conversionMap.put("6", 6);
conversionMap.put("9", 3);

if(/* code here */ == 3) {
    System.out.println("Correct!");
}
```

Select the correct answer

- conversionMap.get(3)
- ✓ conversionMap.get("9")
- conversionMap.get(9)
- conversionMap.get(6)
- conversionMap.get("3")
- conversionMap.get("6")

The answer is correct

Answered The number of tries is not limited

Using a hash map requires the `import java.util.HashMap;` statement at the beginning of the class.

Two type parameters are required when creating a hash map - the type of the key and the type of the value added. If the keys of the hash map are of type string, and the values of type integer, the hash map is created with the following statement `HashMap<String, Integer> hashmap = new HashMap<>();`

Adding to the hash map is done through the `put(*key*, *value*)` method that has two parameters, one for the key, the other for the value. Retrieving from a hash map happens with the help of the `get(*key*)` method that is passed the key as a parameter and returns a value.

Programming exercise:

Nicknames

Points 1/1

In the `main`-method create a new `HashMap<String,String>` object. Store the names and nicknames of the following people in this hashmap so, that the name is the key and the nickname is the value. Use only lower case letters.

- matthew's nickname is matt
- michael's nickname is mix
- arthur's nickname is artie

Then get Matthew's nickname from the hashmap, and print it.

There is no automated tests for this exercise. Just submit the exercise when you think it works as it should.

Exercise submission instructions

How to see the solution

Hash Map Keys Correspond to a Single Value at Most

The hash map has a maximum of one value per key. If a new key-value pair is added to the hash map, but the key has already been associated with some other value stored in the hash map, the old value will vanish from the hash map.

```
HashMap<String, String> numbers = new HashMap<>();
numbers.put("Uno", "One");
numbers.put("Dos", "Zwei");
numbers.put("Uno", "Ein");

String translation = numbers.get("Uno");
System.out.println(translation);

System.out.println(numbers.get("Dos"));
System.out.println(numbers.get("Tres"));
System.out.println(numbers.get("Uno"));
```

Sample output

```
Ein
Zwei
null
Ein
```

A Reference Type Variable as a Hash Map Value

Let's take a look at how a spreadsheet works using a library example. You can search for books by book title. If a book is found with the given search term, the library returns a reference to the book. Let's begin by creating an example class `Book` that has its name, content and the year of publication as instance variables.

```
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;
    }
}
```

Let's create a hash map that uses the book's name as a key, i.e., a String-type object, and the book we've just created as the value.

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

The hash map above uses a `String` object as a key. Let's expand the example so that two books are added to the directory, `"Sense and Sensibility"` and `"Pride and Prejudice"`.

```
Book senseAndSensibility = new Book("Sense and Sensibility", 1811, "...");
Book prideAndPrejudice = new Book("Pride and Prejudice", 1813, "....");

HashMap<String, Book> directory = new HashMap<>();
directory.put(senseAndSensibility.getName(), senseAndSensibility);
directory.put(prideAndPrejudice.getName(), prideAndPrejudice);
```

Books can be retrieved from the directory by book name. A search for `"Persuasion"` will not produce any results, in which case the hash map returns a `null`-reference. The book "Pride and Prejudice" is found, however.

```
Book book = directory.get("Persuasion");
System.out.println(book);
System.out.println();
book = directory.get("Pride and Prejudice");
System.out.println(book);
```

Sample output

```
null

Name: Pride and Prejudice (1813)
Content: ...
```

Quiz:

Object as a HashMap value

Points: 1/1

Which command can be used to retrieve the number 4 from the hash map?

```
public class Person {
    private String name;
    private int age;
```

```
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

-----------------

public static void main(String[] args) {
    HashMap<String, Person> personMap = new HashMap<>();

    Person casper = new Person("Casper", 55);
    Person mike = new Person("Mike", 4);
    Person matt = new Person("Matt", 12);

    personMap.put(casper.getNimi(), casper);
    personMap.put(mike.getNimi(), mike);
    personMap.put(matt.getNimi(), matt);

    if(/* code here */ == 4) {
        System.out.println("Correct!");
    }
}
```

Select the correct answer

personMap.get("Matt").getAge()

✕ personMap.get(mike).getAge()

personMap.get("Mike")

personMap.get(4)

personMap.get(matt).getAge()

personMap.get("Matt")

personMap.get("Mike").getAge()

The answer is not correct

Answered Tries remaining: 2

When Should Hash Maps Be Used?

The hash map is implemented internally in such a way that searching by a key is very fast. The hash map generates a "hash value" from the key, i.e. a piece of code, which is used to store the value of a specific location. When a key is used to retrieve information from a hash map, this particular code identifies the location where the value associated with the key is. In practice, it's not necessary to go through all the key-value pairs in the hash map when searching for a key; the set that's checked is significantly smaller. We'll be taking a deeper look into the implementation of a hash map in the Advanced Programming and Data Structures and Algorithms courses.

Consider the library example that was introduced above. The whole program could just as well have been implemented using a list. In that case, the books would be placed on the list instead of the directory, and the book search would happen by iterating over the list.

In the example below, the books have been stored in a list and searching for them is done by going through the list.

```
ArrayList<Book> books = new ArrayList<>();
Book senseAndSensibility = new Book("Sense and Sensibility", 1811, "...");
Book prideAndPrejudice = new Book("Pride and Prejudice", 1813, "....");
books.add(senseAndSensibility);
books.add(prideAndPrejudice);

// searching for a book named Sense and Sensibility
```

```
Book match = null;
for (Book book: books) {
    if (book.getName().equals("Sense and Sensibility") {
        match = book;
        break;
    }
}

System.out.println(match);
System.out.println();

// searching for a book named Persuasion
match = null;
for (Book book: books) {
    if (book.getName().equals("Persuasion")) {
        match = book;
        break;
    }
}

System.out.println(match);
```

Sample output

```
Name: Sense and Sensibility (1811)
Content: ...

null
```

For the program above, you could create a separate class method `get` that is provided a list and the name of the book to be fetched as parameters. The method returns a book found by the given name if one exists. Otherwise, the method returns a `null` reference.

```
public static Book get(ArrayList<Book> books, String name) {

    for (Book book: books) {
        if (book.getName().equals(name)) {
            return book;
        }
    }

    return null;
}
```

Now the program is a bit more clear.

```
ArrayList<Book> books = new ArrayList<>();
Book senseAndSensibility = new Book("Sense and Sensibility", 1811, "...");
Book prideAndPrejudice = new Book("Pride and Prejudice", 1813, "....");
books.add(senseAndSensibility);
books.add(prideAndPrejudice);

System.out.println(get(books, "Sense and Sensibility"));

System.out.println();

System.out.println(get(books, "Persuasion"));
```

Sample output

```
Name: Sense and Sensibility (1811)
Content: ...

null
```

The program would now work in the same way as the program implemented with the hash map, right?

Functionally, yes. Let's, however, consider the performance of the program. Java's `System.nanoTime()` method returns the time of the computer in nanoseconds. We'll add functionality to the program considered above to calculate how long it took to retrieve the books.

```
ArrayList<Book> books = new ArrayList<>();

// adding ten million books to the list

long start = System.nanoTime();
System.out.println(get(books, "Sense and Sensibility"));

System.out.println();

System.out.println(get(books, "Persuasion"));
long end = System.nanoTime();
double durationInMilliseconds = 1.0 * (end - start) / 1000000;
```

```
System.out.println("The book search took " + durationInMilliseconds + " millise
```

Sample output

```
Name: Sense and Sensibility (1811)
Content: ...

null
The book search took  881.3447 milliseconds.
```

With ten million books, it takes almost a second to find two books. Of course, the way in which the list is ordered has an effect. If the book being searched was first on the list, the program would be faster. On the other hand, if the book were not on the list, the program would have to go through all of the books in the list before determining that such book does not exist.

Let's consider the same program using a hash map.

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

// adding ten million books to the list

long start = System.nanoTime();
System.out.println(directory.get("Sense and Sensibility"));

System.out.println();

System.out.println(directory.get("Persuasion"));
long end = System.nanoTime();
double durationInMilliseconds = 1.0 * (end - start) / 1000000;

System.out.println("The book search took " + durationInMilliseconds + " millise
```

Sample output

```
Name: Sense and Sensibility (1811)
Content: ...
```

```
null
The book search took 0.411458 milliseconds.
```

It took about 0.4 milliseconds to search for two books out of ten million books with the hash map. The difference in performance in our example is over a thousandfold.

The difference in performance is due to the fact that when a book is searched for in a list, the worst-case scenario involves going through all the books in the list. In a hash map, it isn't necessary to check all of the books as the key determines the location of a given book in a hash map. The difference in performance depends on the number of books - for example, the performance differences are negligible for 10 books. However, for millions of books, the performance differences are clearly visible.

Does this mean that we'll only be using hash maps going forward? Of course not. Hash maps work well when we know exactly what we are looking for. If we wanted to identify books whose title contains a particular string, the hash map would be of little use.

The hash maps also have no internal order, and it is not possible to search the hash map based on the indexes. The items in a list are in the order they were added to the list.

Typically, hash maps and lists are used together. The hash map provides quick access to a specific key or keys, while the list is used, for instance, to maintain order.

Hash Map as an Instance Variable

The example considered above on storing books is problematic in that the book's spelling format must be remembered accurately. Someone may search for a book with a lowercase letter, while another may, for example, enter a space to begin typing a name. Let's take a look at a slightly more forgiving search by book title.

We make use of the tools provided by the String-class to handle strings. The `toLowerCase()` method creates a new string with all letters converted to lowercase. The `trim()` method, on the other hand, creates a

new string where empty characters such as spaces at the beginning and end have been removed.

```
String text = "Pride and Prejudice ";
text = text.toLowerCase(); // text currently "pride and prejudice "
text = text.trim(); // text now "pride and prejudice"
```

The conversion of the string described above will result in the book being found, even if the user happens to type the title of the book with lower-case letters.

Let's create a `Library` class that encapsulates a hash map containing books, and enables you to case-independent search for books. We'll add methods for adding, retrieving and deleting to the `Library` class. Each of these is based on a sanitized name - this involves converting the name to lowercase and removing extraneous spaces from the beginning and end.

Let's first outline the method for adding. The book is added to the hash map with the book name as the key and the book itself as the value. Since we want to allow for minor misspellings, such as capitalized or lower-cased strings, or ones with spaces at the beginning and/or end, the key - the title of the book - is converted to lowercase, and spaces at the beginning and end are removed.

```
public class Library {
    private HashMap<String, Book> directory;

    public Library() {
        this.directory = new HashMap<>();
    }

    public void addBook(Book book) {
        String name = book.getName()
        if (name == null) {
            name = "";
        }

        name = name.toLowerCase();
        name = name.trim();

        if (this.directory.containsKey(name)) {
            System.out.println("Book is already in the library!");
        } else {
```

```
            directory.put(name, book);
        }
    }
}
```

The `containsKey` method of the hash map is being used above to check for the existence of a key. The method returns `true` if any value has been added to the hash map with the given key. Otherwise, the method returns `false`.

We can already see that code dealing with string sanitization is needed in every method that handles a book, which makes it a good candidate for a separate helper method. The method is implemented as a class method since it doesn't handle object variables.

```
public static String sanitizedString(String string) {
    if (string == null) {
        return "";
    }

    string = string.toLowerCase();
    return string.trim();
}
```

The implementation is much neater when the helper method is used.

```
public class Library {
    private HashMap<String, Book> directory;

    public Library() {
        this.directory = new HashMap<>();
    }

    public void addBook(Book book) {
        String name = sanitizedString(book.getName());

        if (this.directory.containsKey(name)) {
            System.out.println("Book is already in the library!");
        } else {
            directory.put(name, book);
        }
    }

    public Book getBook(String bookTitle) {
```

```
        bookTitle = sanitizedString(bookTitle);
        return this.directory.get(bookTitle);
    }

    public void removeBook(String bookTitle) {
        bookTitle = sanitizedString(bookTitle);

        if (this.directory.containsKey(bookTitle)) {
            this.directory.remove(bookTitle);
        } else {
            System.out.println("Book was not found, cannot be removed!");
        }
    }

    public static String sanitizedString(String string) {
        if (string == null) {
            return "";
        }

        string = string.toLowerCase();
        return string.trim();
    }
}
```

Let's take a look at the class in use.

```
Book senseAndSensibility = new Book("Sense and Sensibility", 1811, "...");
Book prideAndPrejudice = new Book("Pride and Prejudice", 1813, "....");

Library library = new Library();
library.addBook(senseAndSensibility);
library.addBook(prideAndPrejudice);

System.out.println(library.getBook("pride and prejudice");
System.out.println();

System.out.println(library.getBook("PRIDE AND PREJUDICE");
System.out.println();

System.out.println(library.getBook("SENSE"));
```

Sample output

Name: Pride and Prejudice (1813)
Content: ...

Name: Pride and Prejudice (1813)
Content: ...

null

In the above example, we adhered to the DRY (Don't Repeat Yourself) principle according to which code duplication should be avoided. Sanitizing a string, i.e., changing it to lowercase, and *trimming*, i.e., removing empty characters from the beginning and end, would have been repeated many times in our library class without the `sanitizedString` method. Repetitive code is often not noticed until it has already been written, which means that it almost always makes its way into the code. There's nothing wrong with that - the important thing is that the code is cleaned up so that places that require tidying up are noticed.

Programming exercise:
Abbreviations

Points
1/1

Create a class `Abbreviations` for managing common abbreviations. The class must have a constructor, which does not take any parameters. The class must also provide the following methods:

- `public void addAbbreviation(String abbreviation, String explanation)` adds a new abbreviation and its explanation.
- `public boolean hasAbbreviation(String abbreviation)` checks if an abbreviation has already been added; returns `true` if it has and `false` if it has not.
- `public String findExplanationFor(String abbreviation)` finds the explanation for an abbreviation; returns `null` if the abbreviation has not been added yet.

Example:

```
Abbreviations abbreviations = new Abbreviations();
abbreviations.addAbbreviation("e.g.", "for example");
abbreviations.addAbbreviation("etc.", "and so on");
abbreviations.addAbbreviation("i.e.", "more precisely");

String text = "e.g. i.e. etc. lol";

for (String part: text.split(" ")) {
```

```
    if(abbreviations.hasAbbreviation(part)) {
        part = abbreviations.findExplanationFor(part);
    }

    System.out.print(part);
    System.out.print(" ");
}

System.out.println();
```

Sample output

```
for example more precisely and so on lol
```

Exercise submission instructions

How to see the solution

Going Through A Hash Map's Keys

We may sometimes want to search for a book by a part of its title. The `get` method in the hash map is not applicable in this case as it's used to search by a specific key. Searching by a part of a book title is not possible with it.

We can go through the values of a hash map by using a for-each loop on the set returned by the `keySet()` method of the hash map.

Below, a search is performed for all the books whose names contain the given string.

```
public ArrayList<Book> getBookByPart(String titlePart) {
    titlePart = sanitizedString(titlePart);

    ArrayList<Book> books = new ArrayList<>();

    for(String bookTitle : this.directory.keySet()) {
        if(!bookTitle.contains(titlePart)) {
            continue;
        }
```

```
        // if the key contains the given string
        // we retrieve the value related to it
        // and add it tot the set of books to be returned

        books.add(this.directory.get(bookTitle));
    }

    return books;
}
```

This way, however, we lose the speed advantage that comes with the hash map. The hash map is implemented in such a way that searching by a single key is extremely fast. The example above goes through all the book titles when looking for the existence of a single book using a particular key.

Programming exercise:

Print me my hash map

Points 1/1

Exercise template contains a class `Program`. Implement the following class methods in the class:

- `public static void printKeys(HashMap<String,String> hashmap)`, prints all the keys in the hashmap given as a parameter.
- `public static void printKeysWhere(HashMap<String,String> hashmap, String text)` prints the keys in the hashmap given as a parameter, which contain the string given as a parameter.
- `public static void printValuesOfKeysWhere(HashMap<String,String> hashmap, String text)`, prints the values in the given hashmap whichs keys contain the given string.

Example of using the class methods:

```
HashMap<String, String> hashmap = new HashMap<>();
hashmap.put("f.e", "for example");
hashmap.put("etc.", "and so on");
hashmap.put("i.e", "more precisely");

printKeys(hashmap);
System.out.println("---");
printKeysWhere(hashmap, "i");
```

```
System.out.println("---");
printValuesOfKeysWhere(hashmap, ".e");
```

Sample output

```
f.e
etc.
i.e
---
i.e
---
for example
more precisely
```

NB! The order of the output can vary, because the implementation of hashmaps does not guarantee the order of the objects in it.

Exercise submission instructions

How to see the solution

Going Through A Hash map's Values

The preceding functionality could also be implemented by going through the hash map's values. The set of values can be retrieved with the hash map's `values()` method. This set of values can also be iterated over with a for-each loop.

```
public ArrayList<Book> getBookByPart(String titlePart) {
    titlePart = sanitizedString(titlePart);

    ArrayList<Book> books = new ArrayList<>();

    for(Book book : this.directory.values()) {
        if(!book.getName().contains(titlePart)) {
            continue;
        }

        books.add(book);
```

```
    }

    return books;
}
```

As with the previous example, the speed advantage that comes with the hash map is lost.

Programming exercise:

Print me another hash map

Points 1/1

The exercise template contains the already familiar classes `Book` and `Program`. In the class `Program` implement the following class methods:

- `public static void printValues(HashMap<String,Book> hashmap)`, which prints all the values in the hashmap given as a parameter using

the toString method of the Book objects.

- `public static void printValueIfNameContains(HashMap<String,Book> hashmap, String text)`, which prints only the Books in the given hashmap which name contains the given string. You can find out the name of a Book with the method `getName`.

An example of using the class methods:

```
HashMap<String, Book> hashmap = new HashMap<>();
hashmap.put("sense", new Book("Sense and Sensibility", 1811, "..."));
hashmap.put("prejudice", new Book("Pride and prejudice", 1813, "...."));

printValues(hashmap);
System.out.println("---");
printValueIfNameContains(hashmap, "prejud");
```

Sample output

```
Name: Pride and prejudice (1813)
Contents: ...
Name: Sense and Sensibility (1811)
Contents: ...
```

```
---
Name: Pride and prejudice (1813)
Contents: ...
```

NB! The order of the output may vary. The implementation of a hashmap does not guarantee the order of the objects in it.

Exercise submission instructions

How to see the solution

Primitive Variables In Hash Maps

A hash map expects that only reference-variables are added to it (in the same way that `ArrayList` does). Java converts primitive variables to their corresponding reference-types when using any Java's built in data structures (such as ArrayList and HashMap). Although the value `1` can be represented as a value of the primitive `int` variable, its type should be defined as `Integer` when using ArrayLists and HashMaps.

```
HashMap<Integer, String> hashmap = new HashMap<>(); // works
hashmap.put(1, "Ole!");
HashMap<int, String> map2 = new HashMap<>(); // doesn't work
```

A hash map's key and the object to be stored are always reference-type variables. If you want to use a primitive variable as a key or value, there exists a reference-type version for each one. A few have been introduced below.

Primitive	Reference-type Equivalent
int	Integer
double	Double

char	Character

Java converts primitive variables to reference-types automatically as they are added to either a HashMap or an ArrayList. This automatic conversion to a reference-type variable is termed *auto-boxing* in Java, i.e. putting something in a box automatically. The automatic conversion is also possible in the other direction.

```
int key = 2;
HashMap<Integer, Integer> hashmap = new HashMap<>();
hashmap.put(key, 10);
int value = hashmap.get(key);
System.out.println(value);
```

Sample output

```
10
```

The following examples describes a class used for counting the number of vehicle number-plate sightings. Automatic type conversion takes place in the `addSighting` and `timesSighted` methods.

```
public class registerSightingCounter {
    private HashMap<String, Integer> allSightings;

    public registerSightingCounter() {
        this.allSightings = new HashMap<>();
    }

    public void addSighting(String sighted) {
        if (!this.allSightings.containsKey(sighted)) {
            this.allSightings.put(sighted, 0);
        }

        int timesSighted = this.allSightings.get(sighted);
        timesSighted++;
        this.allSightings.put(sighted, timesSighted);
    }

    public int timesSighted(String sighted) {
        return this.allSightings.get(sighted);
```

```
    }
}
```

There is, however, some risk in type conversions. If we attempt to convert a `null` reference - a sighting not in HashMap, for instance - to an integer, we witness a *java.lang.reflect.InvocationTargetException* error. Such an error may occur in the `timesSighted` method in the example above - if the `allSightings` hash map does not contain the value being searched, it returns a `null` reference and the conversion to an integer fails.

When performing automatic conversion, we should ensure that the value to be converted is not null. For example, the `timesSighted` method in the program program should be fixed in the following way. ->

```
public int timesSighted(String sighted) {
    return this.allSightings.getOrDefault(sighted, 0);
}
```

The `getOrDefault` method of the HashMap searches for the key passed to it as a parameter from the HashMap. If the key is not found, it returns the value of the second parameter passed to it. The one-liner shown above is equivalent in its function to the following.

```
public int timesSighted(String sighted) {
    if (this.allSightings.containsKey(sighted)) {
        return this.allSightings.get(sighted);
    }

    return 0;
}
```

Let's make the `addSighting` method a little bit neater. In the original version, 0 is set as the value of the sighting count in the hash map if the given key is not found. We then get retrieve the count of the sightings, increment it by one, and the previous value of the sightings is replaced with the new one by adding the incremented count back into the hash map. A part of this can also be replaced with the `getOrDefault` method.

```
public class registerSightingCounter {
    private HashMap<String, Integer> allSightings;

    public registerSightingCounter() {
        this.allSightings = new HashMap<>();
    }

    public void addSighting(String sighted) {
        int timesSighted = this.allSightings.getOrDefault(sighted, 0);
        timesSighted++;
        this.allSightings.put(sighted, timesSighted);
    }

    public int timesSighted(String sighted) {
        return this.allSightings.getOrDefault(sighted, 0);
    }
}
```

Programming exercise:

I owe you

Points 1/1

Create a class called `IOU` which has the following methods:

- constructor `public IOU()` creates a new IOU
- `public void setSum(String toWhom, double amount)` saves the amount owed and the person owed to to the IOU.
- `public double howMuchDoIOweTo(String toWhom)` returns the amount owed to the person whose name is given as a parameter. If the person

cannot be found, it returns 0.

The class can be used like this:

```
IOU mattsIOU = new IOU();
mattsIOU.setSum("Arthur", 51.5);
mattsIOU.setSum("Michael", 30);

System.out.println(mattsIOU.howMuchDoIOweTo("Arthur"));
System.out.println(mattsIOU.howMuchDoIOweTo("Michael"));
```

The code above prints:

Sample output

```
51.5
30.0
```

Be careful in situations, when a person does not owe anything to anyone.

NB! The IOU does not care about old debt. When you set a new sum owed to a person when there is some money already owed to the same person, the old debt is forgotten.

```
IOU mattsIOU = new IOU();
mattsIOU.setSum("Arthur", 51.5);
mattsIOU.setSum("Arthur", 10.5);

System.out.println(mattsIOU.howMuchDoIOweTo("Arthur"));
```

Sample output

```
10.5
```

Exercise submission instructions

How to see the solution

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

→ 3. Similarity of objects

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

In this part:

1. Short recap

2. Hash Map

3. Similarity of objects

4. Grouping data using hash maps

5. Fast data fetching and grouping information

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