Files, Static and Final

Course Evaluations in Miverva

File Input/Output

Reading a text file

You need the `FileReader` and `BufferedReader` classes

- The `FileReader` handles the opening and reading of text files, **character by character**
- The `BufferedReader` allows you to read the file **line by line** instead.

```
    // Open a file for reading
    FileReader fr = new FileReader("someFile.txt");

    // Create a BufferedReader so you can read it line by line
    BufferedReader br = new BufferedReader(fr);

    // Initialize a variable that will hold the latest line that you read. Initialize it to the fir
    String currentLine = br.readLine();

    // Read lines until there are no more line ( br.readLine()returns null in taht case)
    while ((currentLine != null){
        // print the line
        System.out.println(currentLine);

        // Read the next line
        currentLine = br.readLine();
    }

    // VERY IMPORTANT: this closes the file, so other programs can read or write to it
    br.close();
    fr.close();

```

This code will throw an exception if the file cannot be opened, or if it does not exist

Writing a text file

You need the `FileWriter` and `BufferedWriter` classes

- The `FileWriter` handles the opening and writing of text files, **character by character**
- The `BufferedWriter` allows you to write multiple characters at once.

```
    // Open a file for writing, character by character
    // This erases any previous file with the same name
    FileWriter fw  = new FileWriter("someOtherFile.txt");

    // Create a BufferedWriter, so you can write multiple characters at once
    BufferedWriter bw = new BufferedWriter(fw);

    bw.write("is this real life?");

    // VERY IMPORTANT: this closes the file, so other programs can read or write to it.
    // AND MORE IMPORTANTLY: it writes the file to your harddrive
    bw.close();
    fw.close();

```

This code will throw an exception if the file cannot be opened, or if it does not exist

Writing a text file

You need the `FileWriter` and `BufferedWriter` classes

- The `FileWriter` handles the opening and writing of text files, **character by character**
- The `BufferedWriter` allows you to write multiple characters at once.

```
    // Open a file for writing, character by character
    // If the file exists, the new text will be appended at the end
    boolean append = true;
    FileWriter fw  = new FileWriter("someOtherFile.txt", append);

    // Create a BufferedWriter, so you can write multiple characters at once
    BufferedWriter bw = new BufferedWriter(fw);

    bw.newLine();
    bw.write("is this just fantasy?");


    // VERY IMPORTANT: this closes the file, so other programs can read or write to it.
    // AND MORE IMPORTANTLY: it writes the file to your harddrive
    bw.close();
    fw.close();

```

This code will throw an exception if the file cannot be opened, or if it does not exist

The static and final keywords

Static

We use the `static` keyword when we want a variable or method to be accesible without creating an instance of a class

- A static variable will have **the same value** for all of the instances of a class

```
    public class Node{
        // non static variables are unique to each instance of this class
        private String data;
        private Node next;

        // static variables are the same for every instance of this class
        private static numberOfNodes = 0;

        public Node(String input_data){
            this.data = input_data;
            this.next = null;

            // we can access static variables inside non static methods
            numberOfnodes += 1;
        }
    }

```

- Only one copy of the static variable exists in the computer's memory.
- If you change the value of a static variable, this change will be seen in **all** instances of the class.

Static

We use the `static` keyword when we want a variable or method to be accesible without creating an instance of a class

- A static method **can be called without creating an instance of a class**

```java
public class Node{
    // non static variables are unique to each instance of this class
    public String data;
    public Node next;

    // static variables are the same for every instance of this class
    public static numberOfNodes = 0;

    public Node(String input_data){
        this.data = input_data;
        this.next = null;

        // we can access static variables inside non static methods
        numberOfnodes += 1;
    }

    // static methods of this class can be called as Node.static_method_name
    public static int NumberOfNodesSoFar(){
        return numberOfNodes;
    }
}

```

- Inside the class file where they are declared, you can call static methods directly by their name
- Outside that file, you should call them as `ClassName.staticMethodName(arguments)`
- Only one copy of the static method exists in the computer's memory.

Static

Some rules for using static methods and variables

- A non-static method can access: static methods and variables
- A non-static method can access: non-static methods and variables
- A static method can access: static methods and variables
- A static method **cannot** access: non-static methods and variables. It doesn't have access to the `this` reference.

Static + Final

The `final` keyword is used when we want the value of a `static` variable to be constant

```
    // The values of the following variables will remain unchanged over the execution of the program
    static final double PI = 3.141592653589793;

    static final double PLANCK_CONSTANT = 6.62606896e-34;

```

This is usually useful when you want to precompute certain numbers and keep them constants during the execution of your program.

```
    // The values of the following variables will remain unchanged over the execution of the program
    static final double PI = precomputePi();

    public static double precomputePi(){
        // execute some algorithm for computing pi
    }

```

How a Hashtable works

Hashtable

Think of a Hashtable as a dictionary

Key (String)	Value (String)
“***brillig***”	“the time when you begin broiling things for dinner”
“***slithy***”	“blend of slimy and lithe”
“***gyre***”	“to whirl”
“***gimble***”	“to make a face“
“***burble***“	“to make vocalized bubbles with the mouth”
“***chortle***”	“to chuckle and snort at the same time”
“***mimsy***”	“flimsy and miserable”

We use a Hastable to associate a key to a value

Hashtable

Keys, as well as values, can be of any type

Key (String)	Value (Long)
"Yogi"	869 567 3212
"Asako"	365 092 3744
"Marcel"	438 756 3321
"Ante"	438 887 1414
"Sylvie-Anne"	514 096 4638
"Luc"	604 394 6870
"Sofia"	206 384 8697

Hashtable

A hash table is a generalization of the simple fixed-size array

- In a fixed size array, you access elements by their **index**, which is a number between 0 and the size of the array minus 1

```
    // Declaring a string array
    String[] someArray = { "514 398 2186", "438 887 1414",
                           "609 234 7564", "555 567 9876" };

    // Accesing the value at index 2
    System.out.println( someArray[2] );

```

  This is called **direct addressing**, since we can access an element of an array with **just one operation**

- A hash table uses a similar idea, we want to access a value **directly** using its key

```
    // This is not valid java code, just an example

    // We would like to do something like:

    // Accesing the value at index "John"
    String phoneNumber  = someHashTable[ "John" ];

    // Accesing the value at index "Anita"
    System.out.println("Anita's phone number is "+someHashTable[ "Anita"] );

```

Hashtable

To get direct access, a hash table uses a fixed-size array to store data.

We would associate each key to a position in the array, and store the data in the corresponding slot in the array.

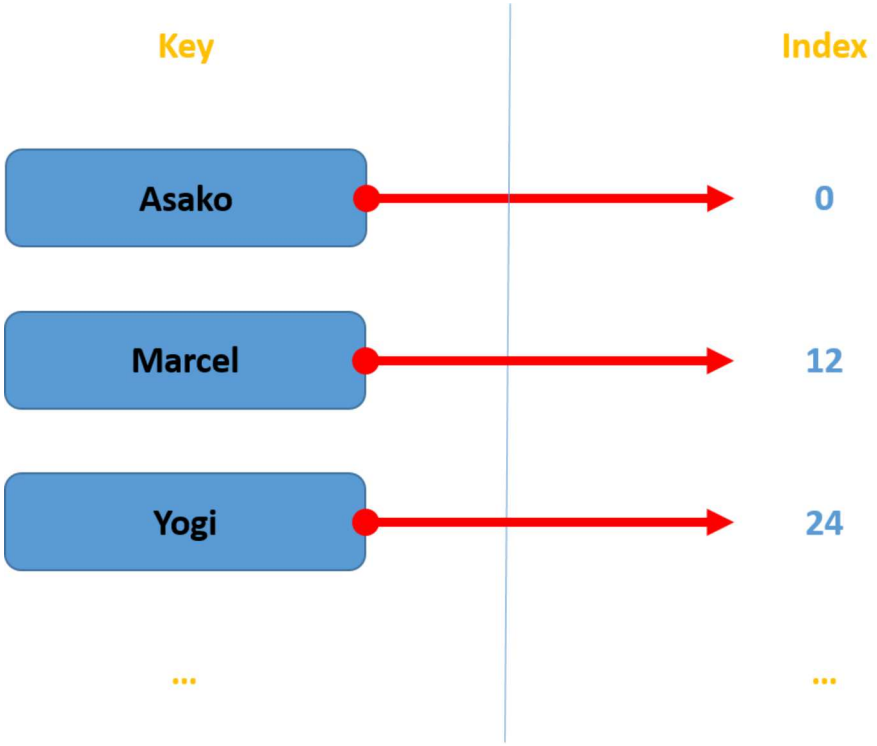
We could try and have one slot in the array for every possible key; i.e. map each possible name to a number in the array

This is not feasible if the number of keys is very big (How many possible names are there?)

Furthermore, we might not even need all of the possible positions in the array to store data

Hashing

Instead, we use a fixed-size array with a small size. For example, we could use just 26 slots for all possible names in a phone book.



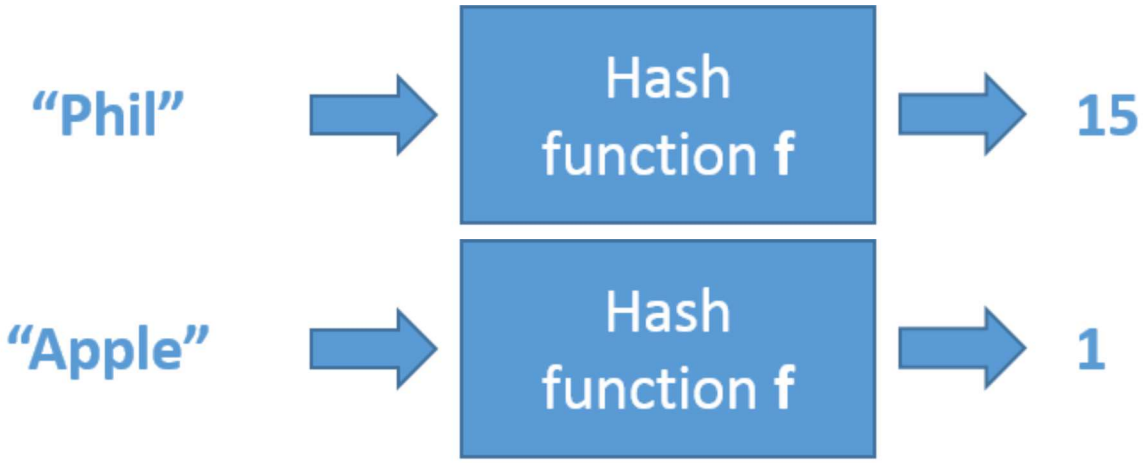
Array

Index	Content
0	365 092 3744
...	
12	438 756 3321
...	
24	869 567 3212
25	--

We need to compute an index for every key. This is called hashing .

Hashing function

A hashing function maps input data of arbitrary size to output data of fixed size; i.e. The set of all possible person names to 26 slots



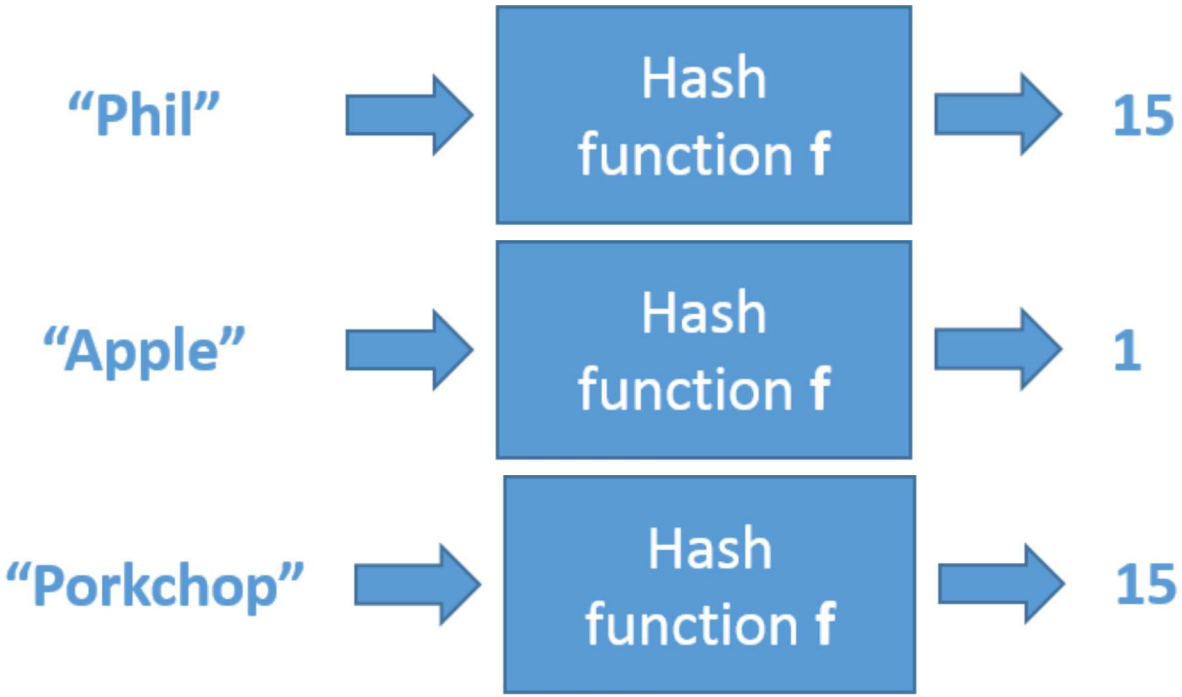
For a key `k`, we store its value in the slot given by the hashing function `f(k)`

We say that `f(k)` is the hash value of the key `k`

A good hashing function would give *different* hash values to every key

Hashing function

But since we are limiting the size of the fixed-size array (the number of slots available) ...

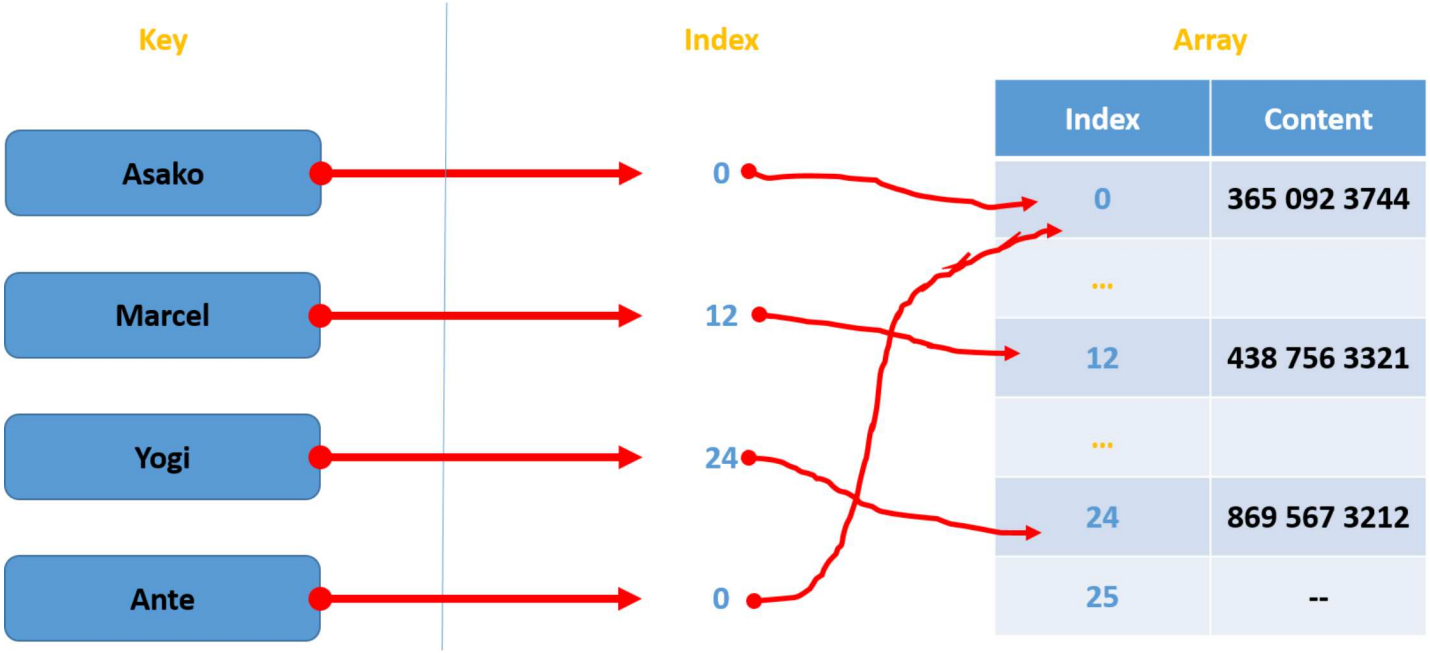


.. multiple keys will have the same hash value. This is called a collision

- In our example, the hash function maps the first letter of a name to a number from 0 to 25; representing its position in the alphabet
- Inevitably, as we add more elements to the hash table, multiple elements will have the same same hash value.

Collisions in hashing functions

Collisions happen when more than two keys have the same hash value

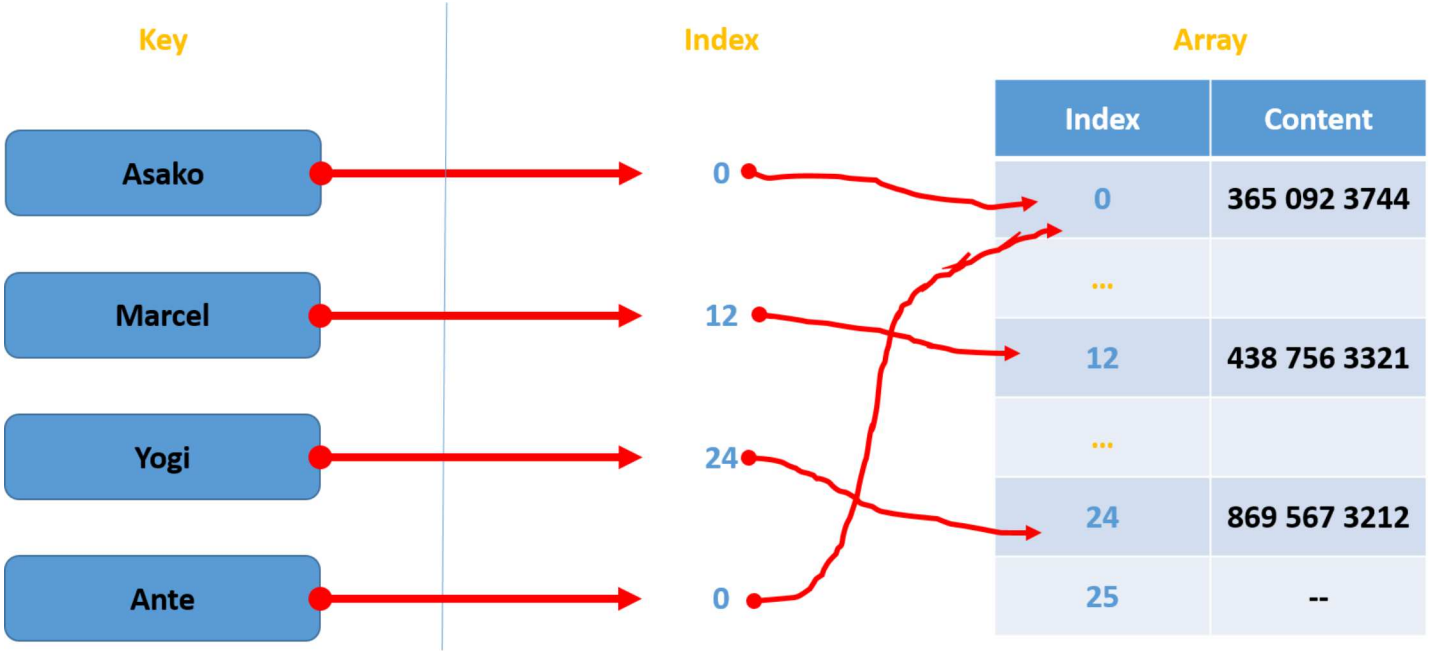


We can resolve collisions by

- Increasing the size of the array that holds the content, and finding a *better* hashing function
- Chaining multiple values in the same slot, using a `LinkedList`

Collisions in hashing functions

Collisions happen when more than two keys have the same hash value



We can resolve collisions by

- Increasing the size of the array that holds the content, and finding a *better* hashing function
- Chaining multiple values in the same slot; e.g. using a `LinkedList`

Collisions in hashing functions

We can resolve collisions by

- Increasing the size of the array that holds the content, and finding a *better* hashing function

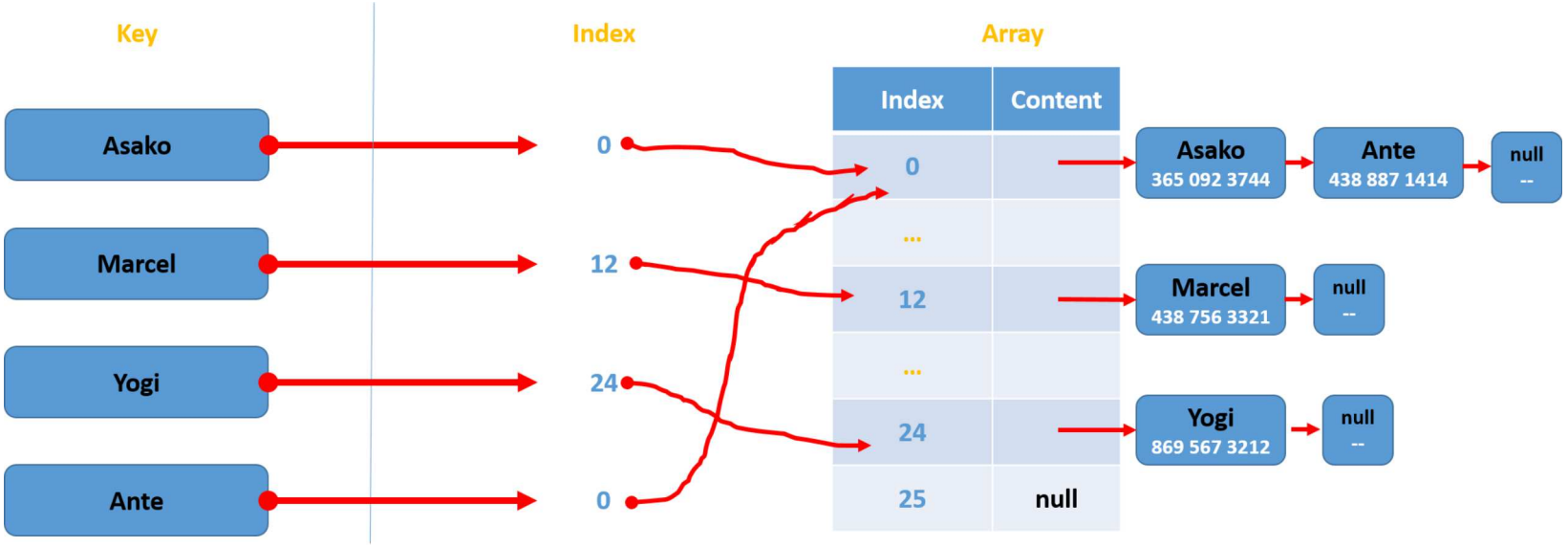
Picking a good hashing function is a good idea: In our example, there are a lot more people with a name starting with J, than there are people with a name starting with Q

But as long as there are more keys than available slots (hash values), there will be collisions.

Collisions in hashing functions

We can resolve collisions by

- Chaining multiple values in the same slot; e.g. using a `LinkedList`



We lose some of the benefit of direct addressing. But this is: 1) faster than a Linked list on average 2) More flexible than a fixed-size array

An implementation of a HashTable

See the provided code in the course website

An exercise for you:

Download the baby-names.csv file from the course website

- Using a Hashtable, count how many times **each name appears on the list** . See how popular your name is
- Compute the percentage of baby names with a given starting letter; e.g. how what is the percentage of names starting with a J
- Can you come up with a hashing function that maps baby names to 100 slots? Try loading the names in the baby-names.csv file to your Hashtable. Can you change your hashing function so that every slot has more or less the same number of elements?

Resources

- Exceptions: https://docs.oracle.com/javase/tutorial/essential/exceptions/
- Reading and writing files: http://www.homeandlearn.co.uk/java/read_a_textfile_in_java.html
- Static and Final: https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html