# 575 Software Engineering : Exercise 1 : Quotes - Work in pairs

In this tutorial you will write your first few Java programs, explore the use of some classes of the Java Standard Library, explore the difference between static and non-static elements and overriding.

*Look at pages 4 onwards of this spec for details of how to get the skeleton code, test it, and submit.*

## Hello World in Java

Following a long standing tradition in programming, write, compile and run a HelloWorld program as your first Java program. The program should simply print out "Hello, World!" to the standard output.

No need to submit this program.

## Famous and not so famous quotes

You are provided with a file Quotes.txt which contains some famous quotes (collected by Gabriel Robins, http://www.cs.virginia.edu/~robins). We do not vouch for their accuracy nor necessarily agree with their content! The file is organised as follows: Each line contains information about one quote: first the quote itself and then, after a semicolon, data about whom the quote is attributed to, and possibly some additional contextual data. An example of one line in the Quotes.txt file is:

*Assassins!;Arturo Toscanini (1867-1957) to his orchestra*

### `QuotesPrinter` Program

1) Write a program that can parse the content of the file and print out the content in a different format. More precisely, for each line of the file, replace the semicolon separating quote from additional data with a new line. For example, the line showed above should be printed as:

   *Assassins!*
   *Arturo Toscanini (1867-1957) to his orchestra*

   **What to do:**
   - Look at the file `QuotesPrinter.java` which contains a main method, with the following signature: `public static void main(String[] args) throws Exception`
   - Declare a constant that contains the path and filename that stores the quotes: `static final String quotesFile = … ;`
   - The standard Java library provides many different ways of reading a file. In this exercise you should use `BufferedReader` object. Note that to create one of these objects you will need a `FileReader` object. Refer to the Java Library Documentation online for more information on these classes.
   - For reformatting the quotes, methods provided by the standard class `String` should suffice. Refer to the Java Library Documentation online to find appropriate methods.

2) Once you have run the `QuotesPrinter` program and seen it work successfully, refactor the program to facilitate writing automated tests. Change your code to make the method `reformat(BufferedReader input) throws IOException` encapsulate the string manipulation. In other words, move code that manipulates the `BufferedReader` to the `reformat` method. The behaviour of the program should remain unchanged.

3) Test your program. First try running the unit test `reformatsSingleLine` from `QuotesPrinterTest.java` from within IntelliJ (click on the green "play" icon in the margin). Does it pass? If not, fix your implementation to make the test pass. **Do not change the test!** Are there any more edge cases that you think we should test? What could q quote have (or not have) that could break your code. Add these tests to `QuotesPrinterTest.java` following the same test pattern.

### `PickYourQuote` Program

Write a new program that works similarly to the previous one but that changes its behaviour according to the parameters with which the program is called. In this case, the program is to be called with two parameters, a filename from which to extract quotes and a quote number to be printed. For instance:

java *PickYourQuote Quotes.txt 4*

will print out the 4th quote from the Quotes.txt file.

First create a new class for your program called `PickYourQuote`, with a `main` method. Recall that the `main` must be declared in particular way for it to be executable from the command line. See the `QuotesPrinter`.

Also, within your `PickYourQuote` class, write a method `public static void reformat(int i, BufferedReader in) throws IOException` that reformats and prints line `i` from buffered reader `in`.

The main method of the class should extract appropriate parameters from the command line (i.e., args[0] and args[1]) and call method `reformat`. Note that you will need to parse the second parameter to convert it to an `int`. Use the static method `parseInt` from the `Integer` class.

Test your class with the test in `PickYourQuoteTest.java`, **do not change the test**. Write any additional tests you consider relevant.

### `PickManyQuotes` Program

Look at the file `PickManyQuotes.java` which contains a main method, with the following signature: `public static void main(String[] args) throws Exception.` The file also contains, contains within a comment, stubs for two methods: `public static List<Quote> loadQuotes(BufferedReader in) throws IOException`, and `public static void reformat(List<Quote> quotes, int[] choices).`

The aim is to complete the main method to support printing several (but not all) quotes from the file based on the user's choice. The method should first have the program load the quotes into a list. To make the program more object-oriented, we will define a type `Quote` for the list elements.

Create a `Quote` class that encapsulates the attributes of a quote, how to parse a quote line, and also how it is printed. `Quote` should have two `String` attributes, one for the quote itself and the other for the additional contextual daeta. The class should have one constructor that takes one `String` and parses out the quotes and contextual data. The class should have to getter methods `getQuote()` and `getContext()` that return the content of its attributes. The class should override the `toString()` method to return strings like the following (note the single quotation marks and the word "by").

> *`Assassins!' by Arturo Toscanini (1867-1957) to his orchestra*

Uncomment the two stubs by removing the lines starting with `/**` and `**/`. The code between these lines was commented as it would not compile until your wrote your `Quote` class.

Complete the method `public static List<Quote> loadQuotes(BufferedReader in) throws IOException`, that loads all quotes from the stream `in` to a `List` of `quotes.`

Complete the method in class `PickManyQuotes` with signature `public static void reformat(List<Quote> quotes, int[] choices)` that for every `int i` in `choices` prints the result of `toString` of the corresponding `Quote` object in the list `quotes`.

Complete the method of `PickManyQuotes` to load quotes into a list of `Quote`, organise choices from the command line into an array of `int` and call `reformat`. A call such as:

> java *PickManyQuotes Quotes.txt 4 5 8*

should print out quotes 4, 5, and 8 from the Quotes.txt file.

Test your class with the test in `PickManyQuotesTest.java`, **do not change the test**. Write any additional tests you consider relevant.

## Questions

Consider the following questions. For each one, we strongly suggest first thinking what the answer might be, and then modifying your code to see if you were right or wrong.

### `QuotesPrinter` Program

- The attribute `quotesFile` is declared as `static`. If the static keyword is removed, what happens? Would the code compile? If not, why? If so, what is the behaviour of the new program?
- Did you remember to `close` the buffered reader? Why do you think it is better to do so? Why do you think that in this particular program it makes no practical difference?

### `PickManyQuotes` Program

- If we remove the `toString` method from class `Quote`, what would happen? Would the code compile? If not, why? If so, what would get printed?
- In the `loadQuotes` method, a variable of type `List<Quote>` is defined but is made to store the reference to an object of type `ArrayList<Quote>`. Could we not have created an object of type `List<Quote>`? If not, why? If so, why does the rest of the program not require changes.
- In the `loadQuotes` method, instead of creating an `ArrayList<Quote>` we could have created an object of type `LinkedList<Quote>`. This would require also including the appropriate `import java.util.LinkedList` statement at the top of the file. Would additional changes be required? Why?

### Getting The Skeleton Code

Get the outline from GitLab:

```
git clone https://gitlab.doc.ic.ac.uk/lab1920_spring/575_tut1_login.git
```

You should work in a pair, but only one person needs to clone the repository. The best way to work is two people sitting at one computer. Work together on the code and make commits back to this repository. All the commits will be in the name of whoever is logged in, but that is ok. A good convention is to add both your names (or initials) to each commit comment.

### Project Structure

When you clone from GitLab, you should find a structure similar to this one.

```
├── Quotes.txt
├── src
│   ├── PickManyQuotes.java
│   └── QuotesPrinter.java
└── test
    ├── QuotesPrinterTest.java
    └── ...
```

There are two source folders, **src** and **test**. Inside each of these folders you already have some classes which we are aiming to develop more fully. Under test, you should find a skeleton **QuotesPrinterTest.java**, where you should add more tests.

You will add more source files under both src and test during the exercise.

### Importing the code into an IDE…

For **IntelliJ IDEA**: you can *Import Project*, browse to cloned directory, *Create project from existing sources,* then keep clicking *Next* until finished. If no JDK is set, choose one >= JDK 8.

## Submission

When you have finished, make sure you have pushed all your changes to GitLab (source code only, not .class files generated by the compiler). You can then use LabTS (https://teaching.doc.ic.ac.uk/labts) to test the version of your work that is on GitLab. Then you need to submit a revision id to CATE so that we know which version of your code you consider to be your final submission.

Only one person from your pair needs to submit on LabTS - the other member of the pair should declare they are part of the group on CATE.

## Autotest with LabTS

The LabTS build has a number of stages and checks a number of qualities of your code:

**Compilation:** compiles your code - does it compile correctly?

**Test Results:** runs all of your test, plus our additional tests - do they pass?

**Style:** we run Checkstyle to check a number of things about the style of your code, following the **Google Java Style** standard conventions for naming, layout etc.

## Deadline

You are strongly encouraged to complete the exercise during the lab session on Thursday morning. However it is possible to submit up until **9am (in the morning!) Monday 21st January** (think of this as Sunday evening).

## Assessment

The first 50% of the marks are for the automated tests and checks:

- Code compiles
- Tests pass
- Style check passes

If you pass these automated checks then you are eligible for the other 50% of the marks, based on:

- Code meets requirements