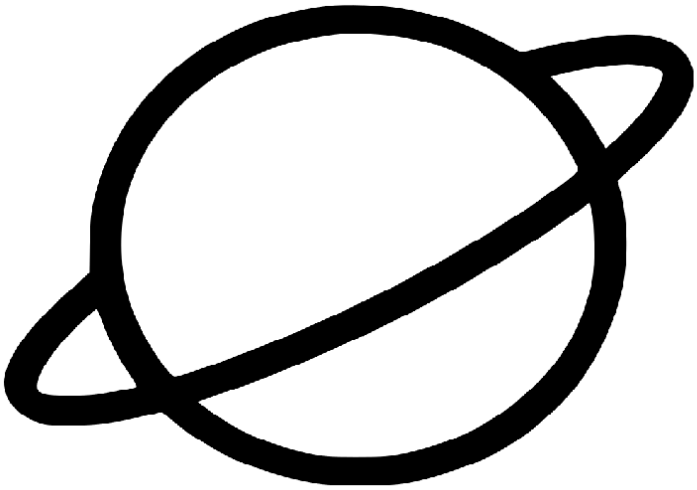
ASCII art

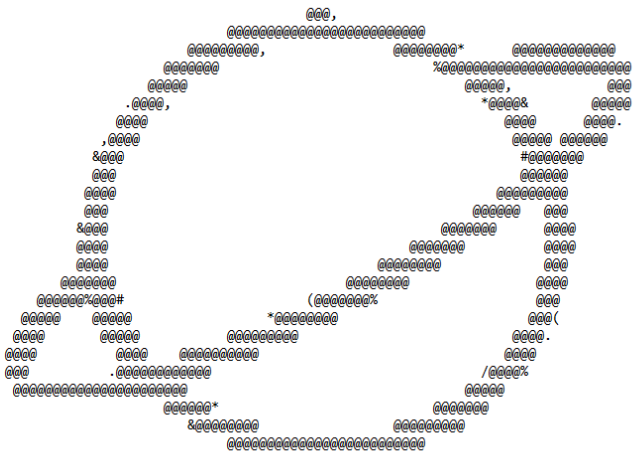
Did you spot any mistakes? Do you have any questions? Please contact Martin Drozdík (drozdma4) via MS Teams.

Fork the ASCIIArt repository (https://gitlab.fit.cvut.cz/BI-OOP/B231/asciiart) to start working on the project. **Do not forget to make your repo private!**

There might have been some corrections, clarifications or minor updates to the project. The last changes have been made 7. 10. 2023. Check out file history (https://gitlab.fit.cvut.cz/BI-OOP/bi-oop/commits/master/projects/ASCII-art.md) for more details. You will be notified of any notable changes.

The idea of this project is to load images, translate them into ASCII ART images, optionally apply filters, and save them. Something like this:





It is also possible to lower the font-size, increase density, and create more detailed art:

Application basics

The app will be a simple console-executable, that:

1. Loads an image
2. Translates the image into ASCII art
3. Applies filters if required – no filters by default
4. Outputs the image into an output (console, file, …)

These steps are not necessarily how your application should work internally. It is just a high-level description to explain the process. You may approach the problem in a way you seem fit (for example, delay the image loading, etc.). However, the input and output of the application must remain the same – load an image and export an ASCII version of it, possibly with applied filters.

When inside the sbt shell (after running the `sbt` command), the `run` command can look like this:

```
run --image "../images/test-image.jpg" --rotate +90 --scale 0.25 --invert --output-console

run --image "test-image.jpg" --output-file "../outputs/output.txt"

run --image "test-image.jpg" --rotate +90 --invert --output-file "../outputs/output.txt" --output-console --
```

Grading

The project is separated into sections. To receive full points from each section, the student must implement the application using objects and propper object modeling. Usage of polymorphism, immutable objects, and common design patterns is expected, but not necessarily required – otherwise, students must defend their design decisions.

The applications' design and code should be clean, clear, and as readable as possible. Complicated and hard-to-understand methods/designs are not appreciated. Simplicity is valued. Do not over-engineer, but keep the application expandable and scalable.

Bonus challenges will have minimal impact on the resulting number of points. The focus should be mainly on the core required functionalities.

Testing is a big part of development. The student is expected to automate tests that will check the functionality of all major classes. Tests must be just as clean and clear as the rest of the application. This ensures that the application is functional and testable. **A section that is not properly tested will receive up to a 50 % point penalty.**

The main focus of this semestral project is quality OOP design. User interface and algorithms do not need to be perfect.

Please try not to write a lot of extra complicated designs and classes. More code means more space for errors. If you want to be sure you get a good amount of points, focus on the core functionalities, and do not get carried away with crazy ideas.

Libraries and dependencies

It is not allowed to use external libraries, except for testing.

Note: It is recommended to use ImageIO to load images. Since ImageIO is from JDK, you can use it freely.

Testing – 50 %

Each feature/module of your application needs to be covered with unit tests. Tests are worth 50 % of your points. **Test each module individually.**

Many test cases for regular uses-cases and edge-cases are expected.

System tests that test your whole application are also welcomed; however, they are optional, unlike very much-needed and essential unit tests.

Do not underestimate the (points) value of testing.

Detailed description

UI – 8 points

The application provides a user interface via the console. Feel free to handle application arguments in a way you think is the best and makes the most sense. All UI solutions that make some sort of sense for your solution will be accepted.

Some of the recommended and possible ways to handle arguments are:

- Ignore the order of arguments and handle operations in a pre-defined order.
- Load all arguments one-by-one and process them all one-by-one in the inputted order.
- **The most recommended**: Read some arguments regardless of their order (load), but read and then execute all other arguments (filters, outputs) in their inputted order.

It is expected that the UI part of the program will be handled in an OOP manner and well thought out. Feel free to use any suitable pattern (MVC, MVP, MVVM) or make your own system.

Load an image – 10 points

The application must always process an image. **The minimum requirement for receiving full points from this section is the support of at least two different formats and a random image generator.**

If one of the arguments is `--image-random`, the application will generate a random image. The process of generating an image is up to you. Use trivial randomness of pixels, a noise function of whatever your heart desires. Dimensions of the random image should be also random, but with reasonable limits.

At least one of the two required image formats must be either:

- jpg
- png
- gif

When specifying a concrete image, the path can be absolute or relative. The specific image is passed as an argument `--image "path"`. If a user uses unsupported extensions, he must be notified.

Only one `--image*` argument can be specified. If there is not exactly one `--image*` argument, the user must be notified.

It is expected that even tho libraries such as ImageIO can load various types of images without any setup, the student will at least check the image type and use appropriate tools. **Loading must be extendable, meaning it should be possible to easily add more loading sources, such as exotic file formats, network data, random image generation methods, and other various data sources.**

ASCII conversion – 18 points

Any loaded image must be, at some point, translated into ASCII art. A simple and recommended method works as follows:

1. Load individual pixels in RGB (Red, Green, and Blue values in the 0-255 range).
2. Calculate each pixels greyscale value using the following formula: `greyscale value = ((0.3 * Red) + (0.59 * Green) + (0.11 * Blue))` (explained on Tutorials Point (https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm)).
3. Convert greyscale values into ASCII characters using a conversion table / algorithm. The result ASCII image must resemble its source image (at least slightly).

The minimum requirements for receiving full points from this section are:

- **The support of one linear transformation table**, for example, a Paul Bourkes' table (http://paulbourke.net/dataformats/asciiart/) . A linear transformation table is a table where the range of 255 greyscale values is equally divied between a set of ASCII characters (`255 / characterCount`).
- **The support for the user to manually set a linear table.**
- **The support of one non-linear transformation table**, where 255 greyscale values are divided un-equally. For example, one character is for values 0-200, and the rest is divided between 20 other characters. Feel free to be creative. Fry the image into oblivion if you need to.

More tables and ASCII algorithms with arbitrary logic should be easily added if needed.

The argument for setting a predefined table is `--table name` , where "name" is the name/identification of the predefined table. The argument for setting a user-defined table is `--custom-table characters` , where "characters" is a set of characters representing a linear scale (for example, `.:-=+*#%@`). If no conversion is explicitly defined, a default table is used – the default table is up to the student. Exactly one table should be used.

Note that **the pixel conversion is supposed to be one-to-one**, meaning one pixel translates into one character. You may notice that the resulting ASCII images have a different aspect ratio, but the currently used font probably causes that to display the text. If you want to fix this, make a filter that will correct the aspect ratio according to the most popular fonts and their letter aspect ratio. It is not mandatory tho.

Other more innovative conversion methods are welcomed as well.

The ASCII conversion process should be easily modifiable and extendable.

Image filters – 10 points

The minimum requirement for receiving full points from this section is the support of **at least three different filters**. **The program can run with multiple filters at once, applied in series** (for example rotate, scale and rotate again).

It should be possible to easily add more filters if needed.

Filters can be:

Rotate

The rotate filter rotates the ASCII image. Rotations dividable by 90 degrees are mandatory, rotations by any number of degrees are a bonus challenge.

The rotate argument is `--rotate degrees` – "degrees" parameter can be for example: `+90`, `90` (same as `+90`), `-180`, …

Scale

Scale filter scales the ASCII image. The filter should support scaling by `0.25`, `1`, and `4`, which looks as follows:

```
Original / Scale 1:
AB
CD

Scale 0.25 (take one of those 4 symbols or calculate average):
A

Scale 4:
AABB
AABB
CCDD
CCDD
```

Other scales are a bonus challenge. The scale arguments is `--scale value`.

Invert

Invert filter inverts the greyscale value of pixels. Inversion is done as follows:

`Inverted greyscale = white - greyscale`, where white is usually 255

The invert argument is `--invert`.

Flip

Flip filter flips the image on "y" or "x" axes.

```
Original:
AB
CD

Flip on x-axes:
CD
AB

Flip on y-axes:
BA
DC
```

The flip argument can be `--flip x` or `--flip y`.

Note: User can input both `--flip x` and `--flip y` at once.

Bonus challenge: Flip on diagonal axes.

Brightness

The brightness filter changes the greyscale value of pixels without losing any precision. Brightness is changed as follows:

The brightness argument is `--brightness value` – "value" parameter can be for example: `+1`, `1` (same as `+1`), `-5`, …

If a pixel brightness / greyscale value gets below 0, it should be considered as 0. If the brightness gets above 255, it should be considered as 255.

Font aspect ratio

The aspect ratio filter changes the aspect ratio of the output image according to a font's aspect ratio.

Fonts usually do not have an aspect ratio of 1:1. They are usually higher, which means the resulting image will probably be stretched out vertically. This filter should fix that.

Using parameter `--font-aspect-ratio x:y`, where `x:y` is the aspect ratio of a font character. We presume the usage of a monospaced font (all letters have the same width).

Note: This filter is definitely the hardest to make and is intended for people that want the ASCII conversion to look good.

Save an image – 6 points

The resulting image can be **saved to a file, printed in the console, or both**. Printing in the console is done using the `--output-console` argument and saving in a file is done using the `--output-file "path"` argument. The file path can be relative or absolute.

Whenever you print, you should print only the ASCII text. It is not necessary to convert the ASCII representation into image formats such as png, jpg, and so on. Simple *dump* into a txt file is good enough, even preferred! An important property of ASCII art is that it's just a text.

It should be possible to easily add more save targets, such as email, HTTP POST, and more.

Before-you-submit checklist (and tips)

Intro

First of all, make sure your approach is indeed object-oriented. What is one of the strong and easy approaches? – **Think in modules**. Separate your program into parts. Those parts should have only one responsibility. For example:

- Importers should import

- Exporters should export
- Filters should filter
- …

Support those services with more packages:

- Data models that carry *only* data (and maybe **offer** some data operations)
- User interface that uses services to communicate with users

Data models are essential because they connect your modules and abstractions. How else would you transfer data from importers to exporters if you want them to be truly universal and expandable? *Well, certainly not using a BufferedImage!*

Creating data models is very domain-dependent. Thinks about what you are creating. ASCII image works with images – how about some image model? What does an image consist of?

Designing modules is easy in theory but hard in practice. First of all, **design a unique high-abstract interface** that your modules will implement. Keep in mind, this interface must be extremely universal and **you should not think about concrete implementations** you currently need. Do not use some “random” value parameter that some implementations might or might not need. Keep it simple. Use domain data models to keep it expandable. The simpler the better – let the implementations worry about the details. For example, what does an export service do? Well, it exports (images). That sounds like one easy peasy method.

If you designed the interface, now comes the easy part. Just inherit from it and create concrete implementations. Do not forget to **design the inheritance tree well** so that no duplicate code is needed.

When using the modules or some modules require other modules, use as general interfaces as possible. Use the most abstract one if possible. This will ensure very good expandability and is very much maintenance-friendly.

What we do and don’t care about

We care about:

- Your **Object-Oriented design** and its primary principles, like
  - **High cohesion** - how you split the code
  - **Low coupling** - how the code/modules are dependent on each other
- **Abstraction** - how well you decouple interfaces from implementations and how well you use those interfaces
  - Do not forget that quality business models are crucial for good abstraction
- **Expandability** - how refactorings or additions to your code affect the rest of the codebase
  - Changes or additions should touch (almost) only the encapsulating module and not cause an avalanche of changes
  - **Do not confuse expandability with preemptively adding more features** *”just in case …”*. More features != expandable code. Less is more.
- **Code quality and readability** - In the end, you write code. Write it well.
  - Pay attention to **names** (variables, classes, etc.)

    - **Comment at least your interfaces**/traits
- **Testing** - although this subject is about OO, we do care about tests
    - Testing is a necessity for any code
    - **A code that is not testable is probably not well designed**, so it is also a good indicator for you if you made poor design choices

We do not care about:

- Efficiency - a reasonable level of inefficiencies is disregarded, but don't make it crazy slow
- Implementation details - Does the order of arguments matter? Do we allow duplicate chars in the ASCII transformation table? We don't care about that. We care about the design (the exact opposite of BI-PA1/2)
- The type of error handling you choose - but choose some
    - Either/Option/Try… are suitable for practicing OO
    - Exceptions are fine, but sometimes heavy-duty
    - Output flags
    - …

Recommended priorities

Firstly, implement models. By models, we mean the objects that define your business area. Then, test them.

- *Example: Models for a car manufacturer are an engine, wheels, steering wheels, etc.*

Secondly, implement the necessary modules. Modules are pieces of code that solve a problem. They may use the implemented models, but they may not. Then, test them.

Thirdly, the models and modules should be orchestrated to create **processes**. Then, test them.

Lastly, implement the UI. The UI uses the models and modules to give users a good experience. The UI should also be object-oriented.

- *Making the UI "completely" OO is more challenging and time-consuming to get 100 %. If you are okay with losing some points, make some parts of the UI imperative. Parsing is usually the pain point. Primarily focus on the other steps.*

The final checklist

Before you submit the semestral project or ask for a project-scale checkup, please make sure you meet all the following points:

- Huge no-no list
    - `null` (The Billion Dollar Mistake (https://hinchman-amanda.medium.com/null-pointer-references-the-billion-dollar-mistake-1e616534d485))
    - Singleton classes (not to be confused with singleton scope in dependency injection)
    - `class` as a wrap of a bunch of functions (imperative approach)

- All objects, classes, traits, variables, methods (and others) have **short yet descriptive and intuitive naming**. For example, if your application has an object for importing images, name it `ImageImporter`. That is a short and very descriptive name. `Importer` may be too general but `ImageImportFromFileOrWhateverStreamVersion420XxX` is most certainly way too long.

- Ensure you **commented on all the traits, classes, and larger methods**. Theoretically, your whole code should be understandable from naming. Still, sometimes, this is impossible.

- Don't just `ctrl+c ctrl+v` your code. It is usually seen at importers (JPG/PNG/GIF) and their tests – yes, tests are also code and do not deserve to be mindnumbly copied. Instead, you can extract the duplicate code to (depending on the situation):

    - Common method (if in the same class)
    - Common parent
    - A new concern (module/class/service/…)
    - And more…

- Ensure you properly **analyzed and implemented data models for your domain** (=ASCII art). For example, it is very suspicious if the application does not contain at least some "Image" class. Checkout the 4th lab (102 - only in Czech, sorry) (45:00) (https://web.microsoftstream.com/video/a79e79db-2dce-4067-ae5d-2c36768d18e2?st=2700).

    - **Most certainly do not think that using only a 2D array as your data model is satisfactory** (aka draggin 2D array / BufferedImage throughout the app).
    - Make sure you **properly encapsulate your data models / classes.** For example, do not expose your data structure where you store your pixels (2D pixel array). Instead, implement a solid interface (pixel getter, pixel iterator, …) and hide your data structures. You should be able to change the internal implementation (pixel storage) without affecting anything else other than your class.
    - **Pay attention to your constructors**. Does your image accept another image in the constructor? Can your image be created only using a file or some data importer in the constructor? Then your models are probably wrong and badly testable. Think about what your image should consist of. Images probably consist of pixels and not stuff like other images, files, and so on…
    - If you return or expose a collection of your pixels, make sure it is a read-only collection (`Seq`) or a copy of your internal collection. Nobody should be able to change your private data structure (for example your 2D pixel array).
    - Make sure your data models are indeed only data models. Filters should do the filtering, converters should do the conversions, loaders should do the loading and exporters should do the exporting. **Data models should only hold data and optionally offer convenient operations and interfaces for data manipulation.**

- Even tho you have some "freedom" when it comes to implementing UI and some other areas, this does not mean you can use other programming paradigms than OOP or have permission to make hideous code. Make sure **all sections of your application are quality OOP code**.

- **Double-check your export/import interfaces. Are they truly universal?** For example, if your highest trait / abstract class for importing requires you to input a file path / some source, your implementation does not allow

other import approaches (generator, …) and consequently is not very well expandable. Make your traits/interfaces as general as possible. You can gradually inherit more and more concrete traits/classes.

    - Do you see importing and generating as two different things/modules? Are you sure about that?
    - Do you have a magic `(RGB)FileImporter` that imports *all* file formats? Reconsider that. It obviously can't import *all* of the current and future image formats. What if I want to add other formats? Where do I put the importing logic? You should probably specialize/specify individual importers.

- The conversion process goes something like **Rgb -> Greyscale -> Ascii -> string**. Your application should make these steps **independently** and align with the high cohesion principle.
- The **high cohesion** is not to be taken lightly. If one of your classes/modules does two things simultaneously, it is a severe violation of OOP. For example, if your exporter translates to ASCII and then exports, that is wrong. Double-check all your classes. One class/module for one purpose, no more.
- Make sure that the **application logic (ASCII transformation) and UI (console related code) are completely and unmistakenly separated**. UI should have a dependency on app logic, but not the other way around.
- The DadJokeDB project from labs has an interactive UI. The ASCII art project does not. It is a simple argument-based application. If you want to get "heavily inspired" by the MVC pattern and style used in the DadJokeDB project, think twice about what you should change and implement. It may not be that similar. A good indicator that you are doing it wrong is having a stateful controller.
    - You do not need an MVC pattern anyway. Feel free to implement some simple module for processing your arguments, as it will probably be much simpler and less prone to design flaws.
- Take into consideration that **ASCII conversion is a vital operation**, that is expected to be worked on in the future (by other programmers). For example, different conversion tables and conversion methods should be added easily. A simple fixed conversion "table" in a non-abstracted standalone function is, therefore, obviously not sufficient.
- Make sure you are not parsing user input in (for example) filters instead in the UI layer. Accepting (for example) degrees as a string, when degrees are obviously an integer, can mean only two things – you are probably violating app logic / UI separation or you are just using an incredibly wrong data type.
- Do you have any `Factory` classes? Please highly consider if your design is correct or if you really do need a factory. **The chances are, you do not need a factory.** If you have a factory or multiple factories, please consider watching a short educational video (https://youtu.be/FEv3NC_YqYk)
- Testing in short: **Make sure your tests cover various situations and as many edge cases as you can imagine.** If you implement just one simple test on each functionality, it will most probably not be sufficient. **Testing has a 50 % impact on the resulting points, do not underestimate it.** Be sure to cover the following points:
    - Each functionality should have multiple tested situations. For example, test rotation filter for 90°, 180°, 270° with square and rectangle images.
    - Each functionality should have all possible edge cases covered. For example, test rotation filter for 0°, 360°, -90°, … and rotation for images such as 0×0, 0×n, n×0 (if those dimensions are allowed) and so

on…

  - Each test case scenario should have its own test case (I know, shocking). If you test for example rotation filter for 90°, 180° and 270°, make three test cases, do not stuff it in only one!
  - Name your test cases well. The names should reflect on what they test. For example, instead of names like “Rotate1”, “Rotate2”, … Name them “Rotate +90°”, “Rotate -90°”, “Rotate invalid image 0×0”, …
  - Separate your unit tests into files (1 set = 1 file). Try to structure those files the same way as the code of your program (same packaging/folders). It is much easier to navigate.
  - Unit tests should be deterministic, not randomized. Random elements should be removed using mocking and other tools.
  - Keep unit tests simple. Their power is in numbers, not in huge chunks of code. It is possible to have long unit tests in terms of line-count (long image definitions and asserting), but the test should be still pretty simple. Test small parts, but test a lot.

- Do not get distracted by test coverage. Although it is a good-to-know metric, it does not measure the quality and correctness of your tests. It could, however, be a good hint that your code is under-tested. For example, zero percent test coverage usually indicates that you should do some testing.

- Make sure you are using types as general as possible. For example, do you really need to accept/return `List`, or will simple `Seq` suffice?

- Do you need a value/parameter such as `axis`? Make sure you use the correct data type. String or char is inappropriate for values such as `axis`.

- `object` is a valuable keyword. However, it should be used with care and caution. It is, for example, useful when in need of a static environment or singleton instances. However, you should avoid using `object` everywhere your constructor does not have parameters and so on. Formally check that your `class` does not indeed require any constructor parameters now and in the future. Do not just “hand out” this keyword to anyone.

ASCII art
projects/ASCII-art.md, last change c0e4b689 (2023-12-21 at 14:34, Martin Drozdík)