INFO0062 - Object-Oriented Programming

Presentation of the project

Jean-François Grailet

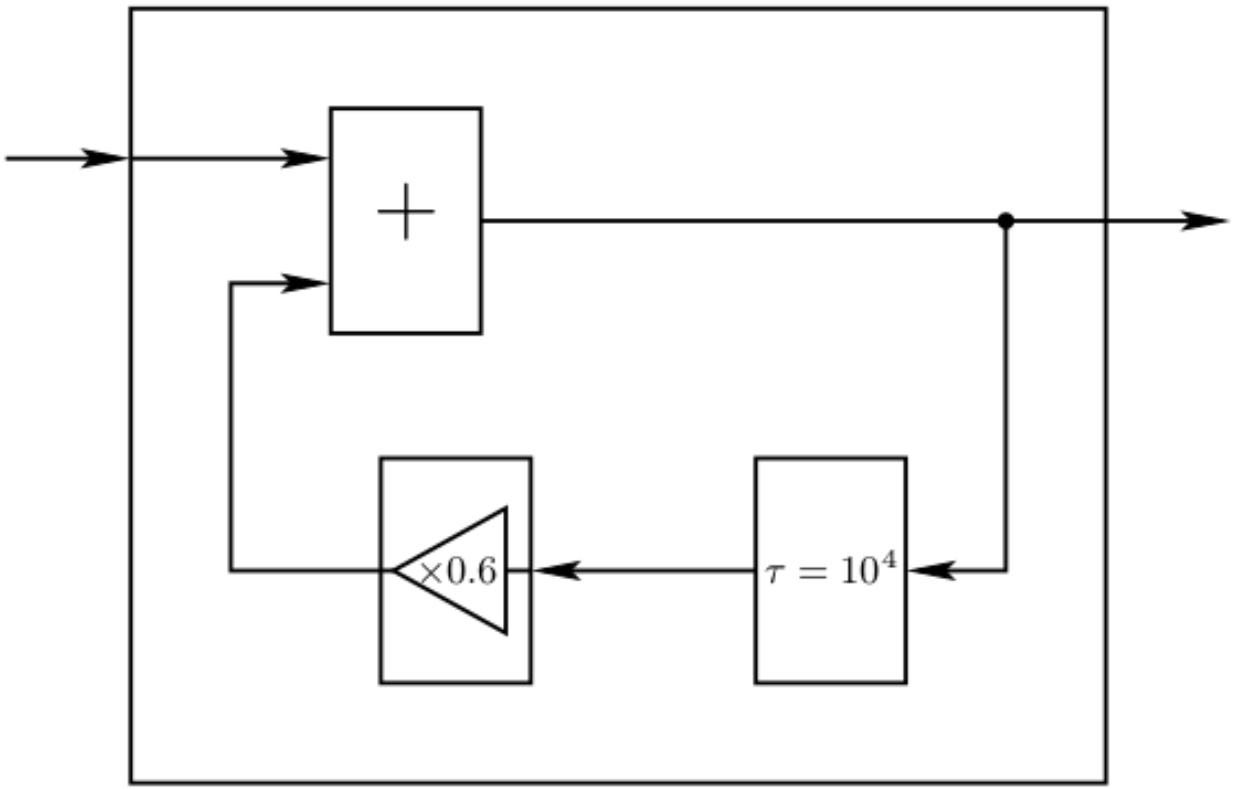
University of Liège

Faculty of Applied Sciences

Academic Year 2019 - 2020

Project

Audio filtering library



Statement

- This project can be done alone or with a classmate.
- Your task is to create a library of classes in Java to manage *digital filters*. [1]
- Such filters can be pictured with *block diagrams*.
  - Assemblies of interconnected blocks.
  - Each block has one or several input(s); same goes for output(s).
  - Each block implements a simple operation.
  - A block produces one (set of) output(s) for each (set of) input(s).
- See previous slide for an example of block diagram.
  - Pictures an *echo filter* in the context of audio filtering.

[1] https://en.wikipedia.org/wiki/Digital_filter

Statement (II)

- You will do this project in two steps.
  - **First step:** blocks for elementary operations.
  - **Second step:** class to manage block diagrams.
- **Elementary blocks**
  - Must implement the `Filter` interface (see statement).
  - Must have specific names and operations.
- **Block diagrams**
  - Must be created via a `CompositeFilter` class.
  - Such a class must provide the interface described in the statement.

Audio filtering

- In order to test your library, you will filter audio sequences.
  - Audio sequences here are large sequences of *samples*.
  - *Samples* are values approximating the original audio signal (when in sequence).
  - Audio sequences will come as WAV files.
    - Uncompressed sequences of samples.
- As a demo, you will use your library to implement an echo filter.
  - This filter is depicted in the statement and at the start of these slides.
- Note that your library could be used for other kinds of filtering.

Your tools

- To get started with the project, download `project_basis.zip`.[2]
- After unzipping, this archive provides the following content:
  - `audio.jar`: an archive providing the following classes
    - `Filter` interface
    - `FilterException` exception class (checked exception; cf. Chapter 6)
    - `TestAudioFilter` class
    - All are part of a package `be.uliege.montefiore.oop.audio`
  - `README.md`: instructions to include `audio.jar` while compiling/running your project
  - Basic project architecture
    - `bin/`: empty folder where you can put your `.class` (compiled) files
    - `src/`: source folder with an `Example.java` file

[2] Download it at `http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062.php`

Your tools (II)

- `audio.jar` is meant to help you filter WAV files.
  - `applyFilter()` class method from `TestAudioFilter`
  - Processes a given source WAV file with an object implementing the `Filter` interface
  - Example of use shown in `Example.java`
- An example of filter will be reviewed in a few slides.
  - How you can compile it will be reviewed as well.

Your tools (III)

- The exercise sessions webpage [3] will provide you several useful resources.
  - Examples of WAV files you can toy with
  - Examples of audio filters you can try to implement
  - A `DummyFilter` class; example of class implementing the `Filter` interface
    - This class will also be presented in next slides.
- Always keep an eye on the statement while doing your project.
  - Stick to the provided class/method names for explicitely requested classes.
  - Pay attention to **all** details, including submission guidelines.

[3] http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062_proj_19-20.php

Getting started with `audio.jar`

A simple use of `audio.jar`

- Next slides will review an example of program using `audio.jar`.
  - Includes an example of class implementing the `Filter` interface.
  - You will already be able to process a WAV file with it.
- Next slides also review how you can compile and run it.
  - With CLI
  - With Eclipse IDE

Quick reminder: interfaces

- Any filter in this project should implement the `Filter` interface.
- Interfaces are collections of signatures of public methods.
  - Cf. Chapter 5 (pp. 131-133)
- A class implementing an interface must provide a body for each of its methods.
- Implementing an interface is the programming equivalent of signing a contract.
- Especially useful to dialog with classes whose implementation is not known.
  - In this context, you are unaware of how a WAV file is extracted and processed.
  - But if you implement the `Filter` interface, you can still filter one.
  - Indeed, classes of `audio.jar` invoke methods of this interface when filtering.

A very simple filter

- We are going to process a WAV file in a simple (and stupid) way.
  - Keep the first `x` seconds of the audio sequence untouched.
  - Cut the sound for the next `x` seconds.
  - Then keep the next `x` seconds of the audio sequence.
  - Then cut sound again for `x` seconds, etc.
- Seconds can be easily translated into an amount of samples.
  - In this context, one second = 44100 samples.
    - Our example WAV files are all sampled at 44,1 kHz.
  - Samples to let pass/to cut for `x` seconds = 44100 * `x`.
- Keep in mind that there is no *block diagram* here.
  - We are only going to create a single *block*.

A very simple filter (II)

- Our filter will be modeled by a `DummyFilter` class.
- The code of this class will be placed in a file `DummyFilter.java`.
- It will consist of a public class implementing the `Filter` interface.
  - `Filter` must be imported first (see below).
  - Use the keyword `implements` to announce commitment to `Filter`.

```
import be.uliege.montefiore.oop.audio.Filter;

public class DummyFilter implements Filter
{
    // ... code of the DummyFilter class
}
```

A very simple filter (III)

- We will need some instance variables for our filter.
  - `duration`: amount of samples to let pass/to cut
  - `count`: current amount of samples that passed/were cut
  - `cutting`: boolean set to true if sound is cut off

```
// ...
private int duration, count;
private boolean cutting;

public DummyFilter(int duration)
{
  this.duration = duration;
  count = 0;
  cutting = false;
}
// ...
```

A very simple filter (IV)

- We start by implementing `nbInputs()` and `nbOutputs()`.
  - Trivial: our filter has one input and one output.
  - Don't forget to keep the same method signatures as in `Filter`.

```
// ...
public int nbInputs()
{
  return 1;
}

public int nbOutputs()
{
  return 1;
}
// ...
```

A very simple filter (V)

- Let's now implement `computeOneStep()`, the main operation.
- In the case of our filter, we will
  - increment `count`,
  - check if we reached the `duration`,
  - flip [4] `cutting` if yes (and reset `count`), using the `!` operator,
  - produce our output depending on `cutting`.
- Note that the exceptions (cf. Chapter 6) thrown by the method can be changed.
  - In this case, no particular exception needs to be thrown.
  - You can change this behaviour if you wish.
  - E.g., to throw a `FilterException` if there's more than one input.

[4] I.e., `false` becomes `true` and vice versa

A very simple filter (VI)

```
// ...
public double[] computeOneStep(double[] input)
{
  count++;
  if(count == duration)
  {
     count = 0;
     cutting = !cutting;
  }

  double[] output = new double[1];
  output[0] = 0;
  if(!cutting)
     output[0] = input[0];
  return output;
}
// ...
```

A very simple filter (VII)

- Let's not forget the `reset()` method.
  - Used in practice by classes from `audio.jar` to deal with stereo sound.
  - If you need to maintain a state (like here), it must reset this state.

```
// ...
public void reset()
{
  count = 0;
  cutting = false;
}
// ...
```

A very simple filter (VIII)

- To complete our program, we just have to update `Example.java`.
- We will instiantiate a `DummyFilter` object (named `df`).
- Note that you can use two types for `df` here.
  - `Filter` (because `DummyFilter` implements `Filter`)
  - `DummyFilter`

```
// ...
Filter df = new DummyFilter(44100 * 3); // 3 seconds

TestAudioFilter.applyFilter(df, "Source.wav", "Filtered.wav");
// ...
```

How do we compile all this ?

- How you will compile your program depends on your preferred approach.
  - I.e., whether you are using CLI or an IDE to program with Java.
- Next slides show how to compile and run with CLI (any OS).
- Subsequent slides describe how to do the same under Eclipse IDE.

Compiling with audio.jar and CLI

- First of all, you must ensure all your files are at the right places.
- You can inspire yourself from project_basis.zip after unzipping it.
  - Empty bin/ folder (will contain .class files)
  - src/ folder with your edited Example.java and DummyFilter.java
  - audio.jar located in the parent folder of both bin/ and src/
- In fact, this is also what should appear in your final submission (see statement).
- To complete this, add a WAV file in the same folder as audio.jar.
  - Download one of the WAV files available on the exercise sessions webpage. [5]
  - To match the original code of Example.java, rename it Source.wav.
  - Or better: modify Example.java.

[5] http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062_proj_19-20.php

Compiling with audio.jar and CLI (II)

- Using your terminal/command prompt, go to the directory where `audio.jar` is.
- **Compilation**

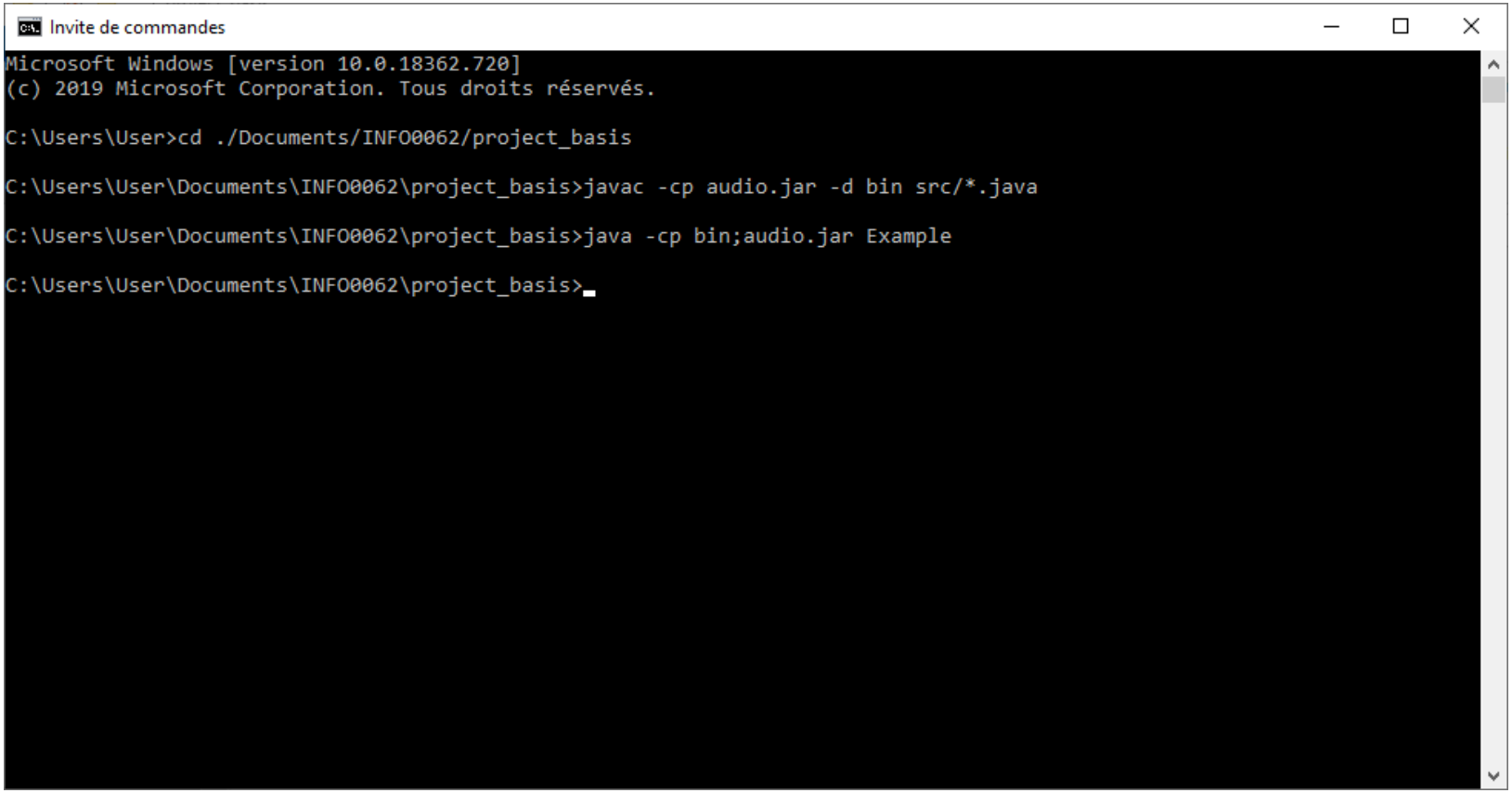
```
javac -d bin -cp audio.jar src/*.java
```

- **Execution**

```
java -cp bin:audio.jar Example
```

- **Remarks**
  - Under Windows, the `:` in the execution command must be replaced with `;`.
  - `*` is called a *wildcard*. `src/*.java` means "*all `.java` files in `src/`*".
  - `-d bin` tells `javac` to put the result `.class` files in the `bin/` folder.
- Now, a new WAV file should appear in your project folder. You can listen to it !

Compiling with audio.jar and CLI (III)

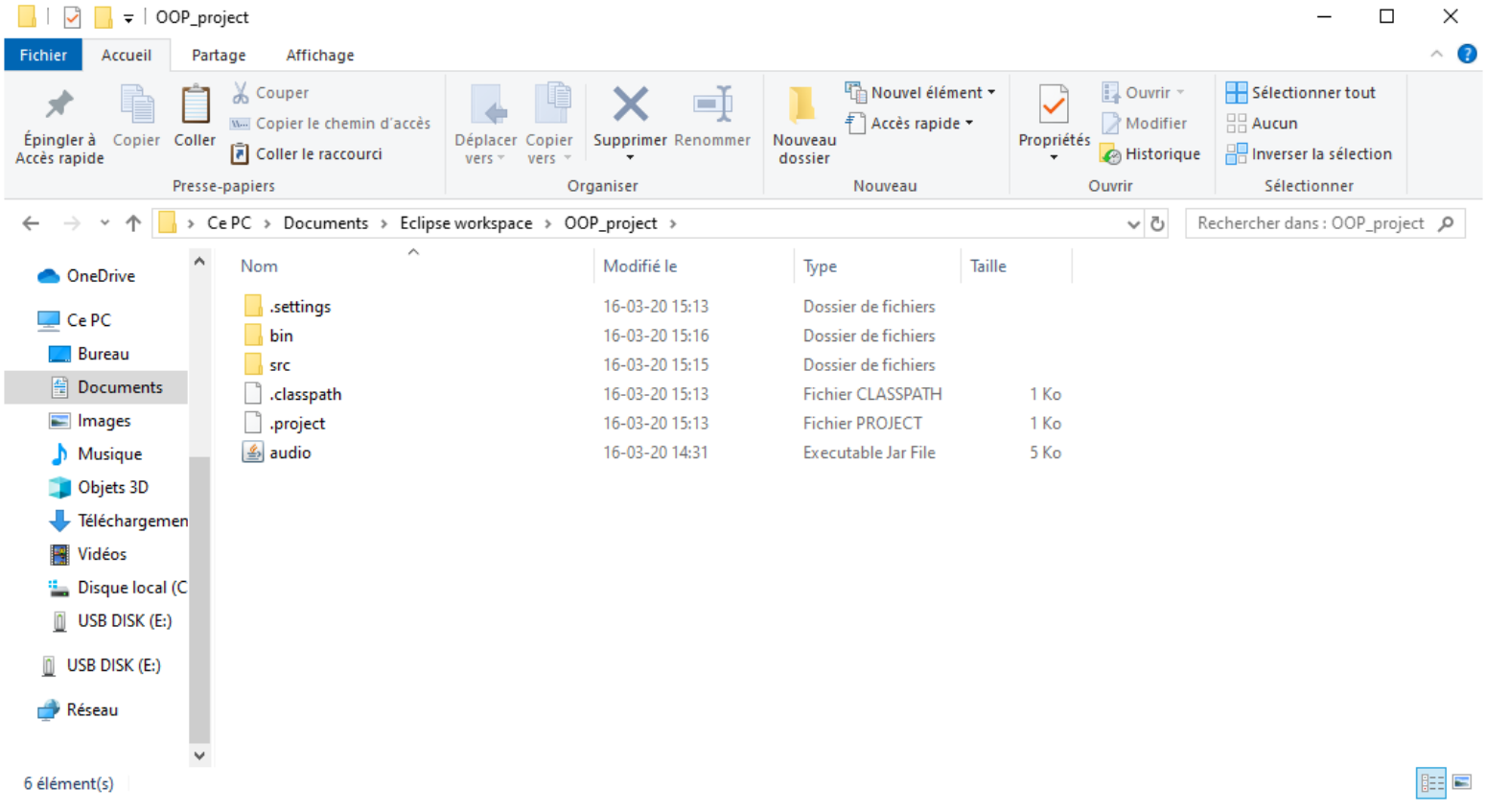


Compiling with audio.jar and Eclipse IDE

- Open Eclipse IDE and click on “Create a new Java project”.
- Give a name to the project (e.g.: OOP_project) and click on “Finish”.
- When asked to create a module, click on “Don’t create”.
- Now, copy audio.jar into the root folder of your project.
  - You will find audio.jar after unzipping project_basis.zip.
  - The root folder of your project should be in your *Eclipse workspace*.
  - I.e., the folder Eclipse IDE requests at start-up.

Compiling with audio.jar and Eclipse IDE (II)

Root folder of a newly created OOP_project project, with audio.jar

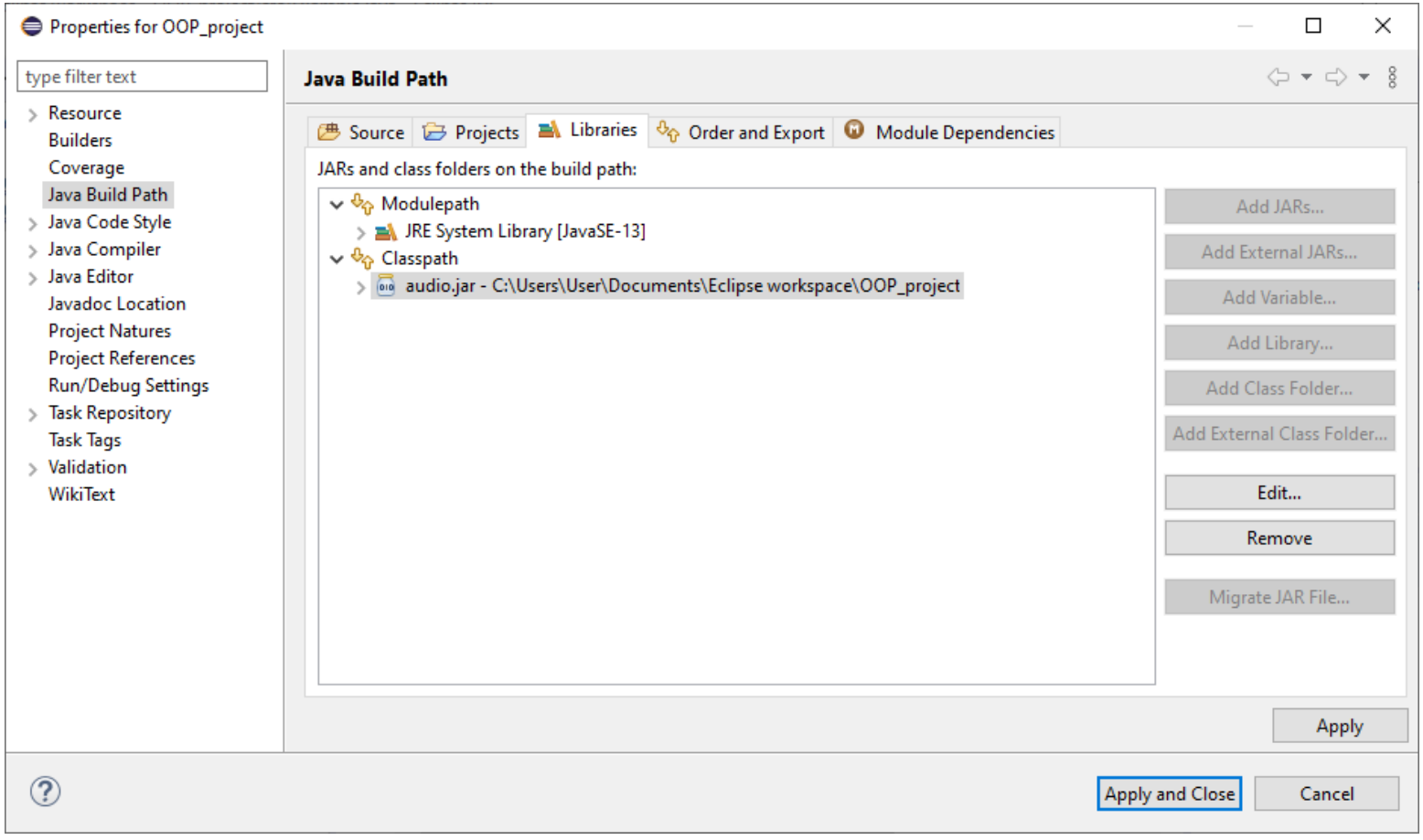


Compiling with audio.jar and Eclipse IDE (III)

- In Eclipse IDE, right-click on your project.
- Select “Build Path” and click on “Configure Build Path”.
- In the new window, go to the “Libraries” tab.
- Select “Classpath” by left-clicking it.
- Click on the “Add External JARs...” button.
- Go to the root folder of your project and select audio.jar.
- Click on “Apply and Close”.

Compiling with audio.jar and Eclipse IDE (IV)

Libraries of OOP_project after successfully adding audio.jar

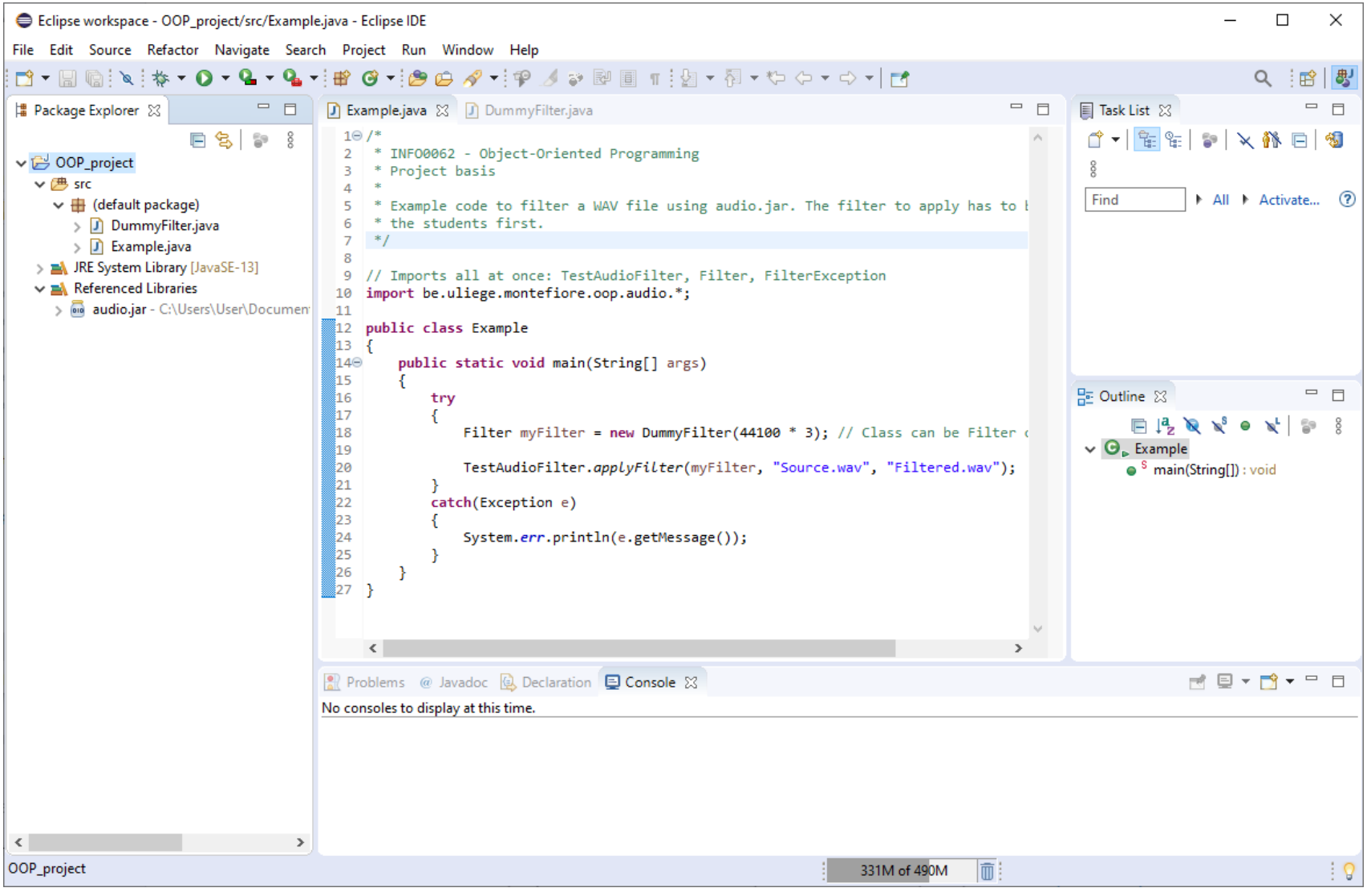


Compiling with audio.jar and Eclipse IDE (V)

- Now, copy your edited Example.java and DummyFilter.java in your project.
- To complete this, add a WAV file in the same folder as audio.jar.
  - Download one of the WAV files available on the exercise sessions webpage. [6]
  - To match the original code of Example.java, rename it Source.wav.
  - Or better: modify Example.java.
- Click on the green arrow to compile and run.
- You should have a display similar to what you see on the next slide.
- A new WAV file should have appeared in your project folder. You can listen to it !

[6] http://www.run.montefiore.ulg.ac.be/~grailet/INFO0062_proj_19-20.php

Compiling with audio.jar and Eclipse IDE (VI)



Tips for your project

Tips for your project

- **Summary**
  - General advice
  - Designing a solution for `CompositeFilter`
  - Remarks on delay filters (`DelayFilter`)
  - Command-line parameters
  - Useful classes from the Java library

General advice

- In such a context, it is important to proceed **step by step**.
  - For instance, start by creating your `DelayFilter` class.
  - Test it by delaying the start of a music piece by 5 seconds.
  - Proceed with other filters only if this first step worked.
- If you are working with a classmate, coordinate yourselves.
  - When designing something, discuss together before implementing anything.
  - Decide who will work on which part of the project.
  - Agree on interfaces if working on a same part.

Designing a solution for `CompositeFilter`

- The output(s) of one *block* can be computed only if input(s) are all available.
- This is, in fact, the main challenge to tackle when designing `CompositeFilter`.
  - How can a block know if all its inputs are available ?
    - This notion of "available inputs" doesn't appear in the `Filter` interface.
    - I.e., this is something you have to handle yourself.
  - How do you handle an output when it's an input for several separate blocks ?
  - How can you test if a block diagram is complete and consistent ?
  - If the block diagram is inconsistent, how can it be signaled ?
- You might want to use one or several auxiliary classes for this.
  - What will each of these classes model ?
  - Is inheritance relevant in this context ?

Remarks on delay filters (`DelayFilter`)

- `DelayFilter` objects will play a very specific role here.
  - Whenever a loop appears in a block diagram, a `DelayFilter` is part of it.
  - The output of a `DelayFilter` is its input from a previous step.
  - If no sample has been fully delayed, a `DelayFilter` outputs 0.
- **Problem:** waiting for the output of a `DelayFilter` can induce loops.
- **Tip:** consider the output of a `DelayFilter` is *always available*.
  - I.e., if this output enters another block, this block doesn't have to wait for it.
- In other words, you have to decouple two operations:
  - reading the (previous) output of a `DelayFilter`,
  - updating the `DelayFilter`.
- How can you include this in your design for `CompositeFilter` ?

Command-line arguments

- For you `Demo` program, you will have to handle command-line arguments.
  - Let's say we want to add an echo on `Virtual_Insanity_1m.wav`.
  - We want the output file to be named `Echo.wav`.
  - The command (under Linux/macOS) to do this should look like this.

```
java -cp bin:audio.jar Demo Virtual_Insanity_1m.wav Echo.wav
```

- In Java, arguments are provided as `String` objects via the `args` array.
  - In this example, `args[0]` contains the string "Virtual_Insanity_1m.wav".
  - On the other hand, `args[1]` contains the string "Echo.wav".
- A more complete example of a program using `args` is shown next slide.
- You can check `args.length` to verify the number of arguments.

Command-line arguments (II)

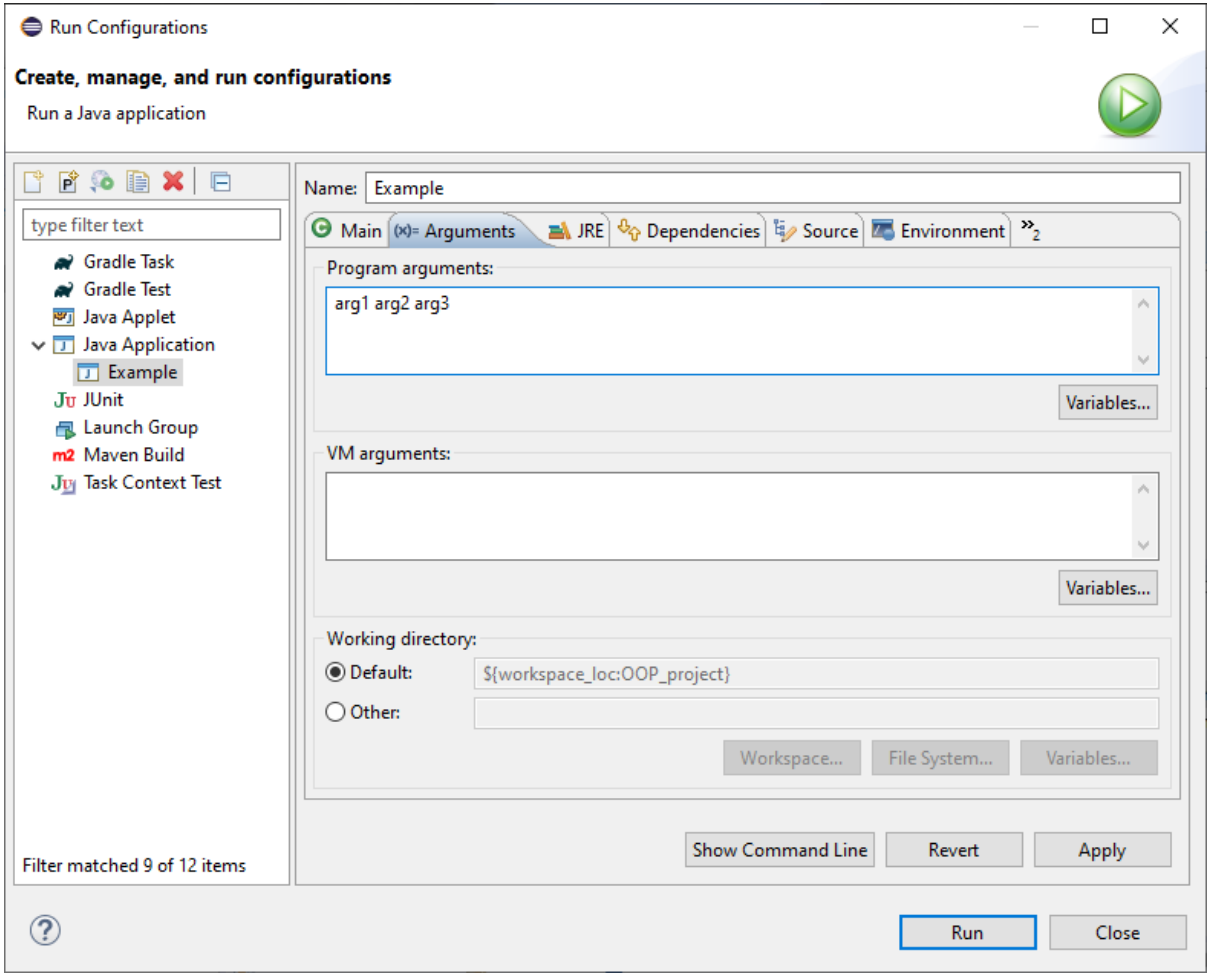
```
import be.uliege.montefiore.oop.audio.*;

public class Demo
{
    public static void main(String[] args)
    {
        if(args.length != 2)
        {
            // Error message: bad amount of arguments
            return;
        }

        String inputFile = args[0];
        String outputFile = args[1];

        // ... (rest of the program)
    }
}
```

Command-line arguments (III)

- You can also use arguments with Eclipse IDE.
- To do so, unfold the “Run” menu.
- Click on “Run Configurations...”.
- Go to the “Arguments” tab.
- Fill the text area “Program arguments” with your own arguments.
- Click on “Run” to run your program with your arguments.

Command-line arguments (IV)

Giving arguments *arg1* (`args[0]`), *arg2* (`args[1]`) and *arg3* (`args[2]`) in Eclipse



Useful classes from the Java library

- You can use classes from the Java library to implement your project.
- Useful classes for this project include notably
  - `java.util.Vector` 
  - `java.util.ArrayList` 
  - `java.util.HashMap` 

Coding style and documentation

About coding style

- Use meaningful variable, method and class names.
- For instance, compare the readability of the two following methods:

```
public static int a(int b) {
  if (b <= 0)
    return 1;

  return b * a(b - 1);
}
```

```
public static int factorial(int input) {
  if (input <= 0)
    return 1;

  return input * factorial(input - 1);
}
```

About coding style (II)

- Convention for variable/method names is to use lowercase[7] words.
- Starting from the second word, the first letter is uppercase[8].
  - E.g. `priceWithTaxes`.
- Alternatively, you can use lowercase words separated by "_" (underscore).
  - E.g. `price_with_taxes`.
- For constants, the convention is to use uppercase words separated by "_".
  - E.g. `TVA_IN_BELGIUM`.
- For classes and interfaces, lowercase words that begin with an uppercase letter.
  - E.g. `TaxesCalculator`.

[7] FR: en lettre minuscule
[8] FR: en lettre majuscule

About coding style (III)

- Two conventions for curly braces related to blocks (choose one):

```
while (true) {

}
```

```
while (true)
{

}
```

- Indentation must be coherent and strongly respected:

```
public class MyClass {
      public static void m1() {
      instruction1;
    instruction2;
    }

public static void m2() {
 instruction1;
     instruction2;
}
    }
```

```
public class MyClass {
  public static void m1() {
    instruction1;
    instruction2;
  }

  public static void m2() {
    instruction1;
    instruction2;
  }
}
```

About coding style (IV)

- You can insert spaces or empty lines in your code to improve readability.

```
public class Probability{
  public static double arrange(int n,int k){
    return (double)factorial(n)/factorial(n-k);
  }
  public static int factorial(int input){
    if(input<=0)return 1; return input*factorial(input-1);
  }
}
```

```
public class Probability {
  public static double arrange(int n, int k) {
    return (double) factorial(n) / factorial(n - k);
  }

  public static int factorial(int input) {
    if (input <= 0)
      return 1;
    return input * factorial(input - 1);
  }
}
```

About coding style (V)

- Choose a maximal number of characters per line of code.
- Common convention: 80 columns rule.
- But you can also use 100 columns if you prefer.
- **The most important is to make consistent choices and to respect them.**

Documentation

- You can document your code using comments.
- It is useful to remember what you did, but also to inform other programmers.
- Typically, you should at least describe the role of a class.

```
/*
 * This class offers a set of static methods to perform various
 * calculations relative to the probability theory.
 */

public class Probability {
   ...
}
```

Documentation (II)

- You can describe the purpose of a method by detailing
  - its parameter(s) (if any) and returned value (if any),
  - the instantiation context of its exception(s) (if any).
- You can go as far as using Javadoc  (optional).

```
/*
 * This method tests whether the input parameter is odd and
 * returns a boolean to confirm it. In the case where the input
 * parameter is negative, a MyException exception is thrown.
 */

public static boolean isOdd(int input) throws MyException {
  if (input < 0)
    throw new MyException();
  return (input % 2) == 1;
}
```

About language(s)

- You can choose English or French for your documentation.
- Prefer English for the names of variables, methods and classes.
- However, once you chose a language, **stick with it**.

```
/**
 * Cette méthode teste si un entier positif est impair.
 *
 * @param  input       L'entier à tester.
 * @return boolean     Vrai si l'entier est impair, faux sinon.
 * @throws MyException Lancée quand un entier négatif est donné.
 */

public static boolean isOdd(int input) throws MyException {
  if (input < 0)
    throw new MyException();
  return (input % 2) == 1;
}
```