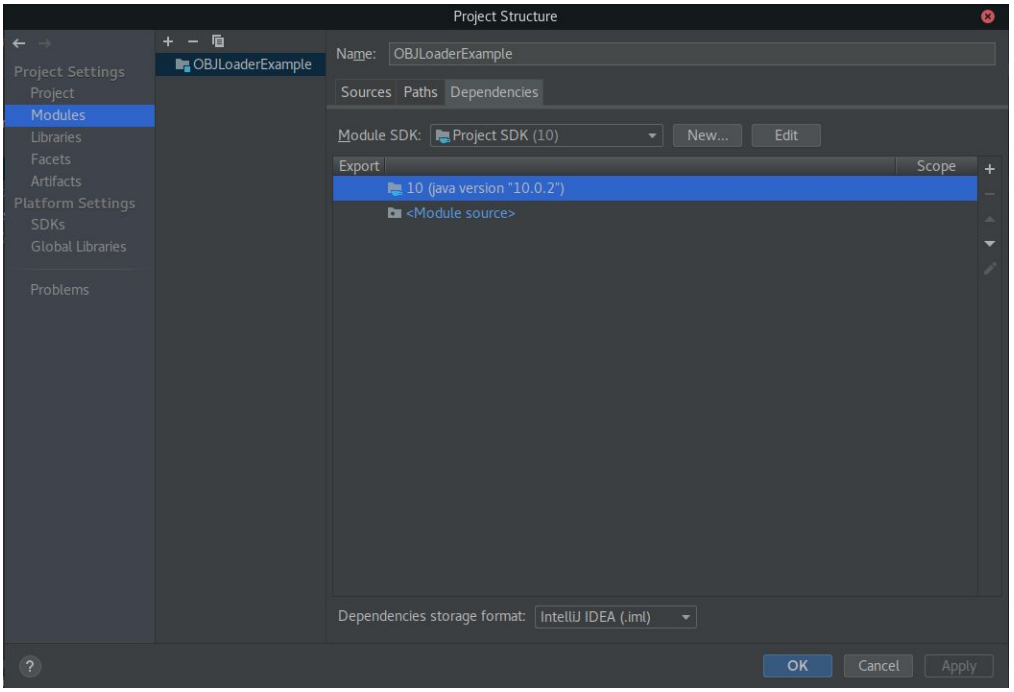
Hochschule Hamm-Lippstadt
Wavefront Loader

Getting Started | OBJLoader Version 3.0.0

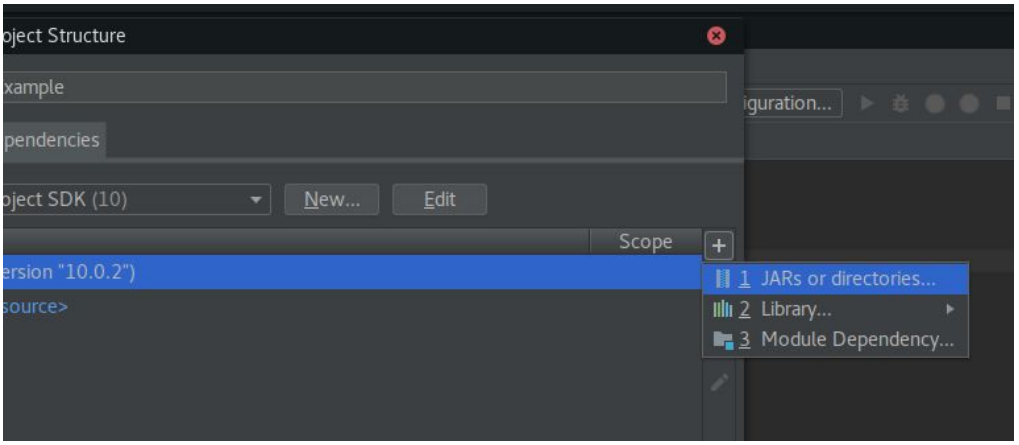
1. Adding the .jar depencency

First, setup Java to use the OBJLoader.jar as a library.
This guide displays how to specify the OBJLoader.jar dependency in IntelliJ IDEA.

Open “File > Project Structure …”, navigate to “Modules”, and open “Dependencies”.



Click the plus button on the right and select “JARs or directories…“



Finally, select the “OBJLoader.jar” file.
Also, please add the JavaDoc to the project so that IntelliJ can show documentation of code suggestions in-place.

2. Loading an OBJ File

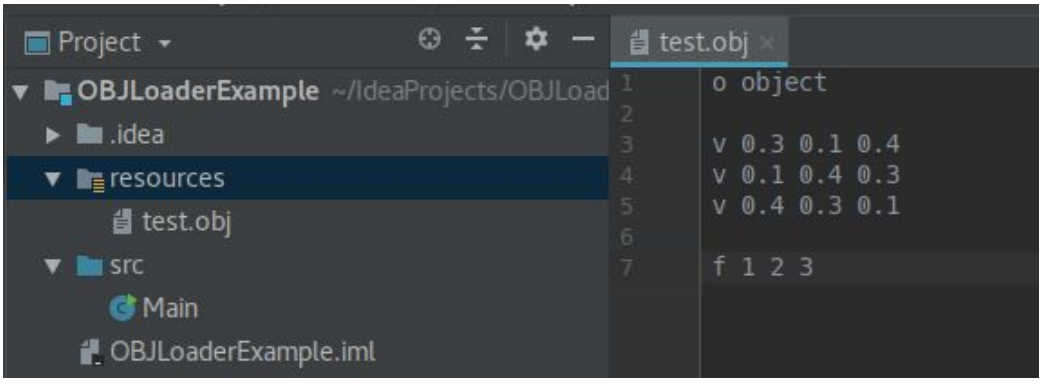
In this example, we want to load the following OBJ file:

```
v 0.3 0.1 0.4
v 0.1 0.4 0.3
v 0.4 0.3 0.1
f 1 2 3
```

This file contains only a single triangle, made up of three vertices. Save these five lines into a file on your computer. You can rename the file to “triangle.obj” or something similar.

Add a class called “Main” to your project and add the main method to it, as always:

```
public class Main {
	public static void main(String[] args) {
		// our loader code will go here
	}
}
```



The next thing we need to do is to create a loader object. The loader object can be configured to fit our specific needs. That way, we can tell it to ignore the unimportant parts of the file and give us only the content we’re really interested in.

To use the loader class, we need to import it:

```
import de.hshl.obj.loader.OBJLoader;
```

Now we can finally create a loader object inside our main method:

```
import de.hshl.obj.loader.OBJLoader;
public class Main {
	public static void main(String[] args) {
		OBJLoader loader = new OBJLoader();
	}
}
```

The next thing we should do is to configure our `OBJLoader` instance, called `loader` in this example.

Configuring the Loader

Vertex Data

By default, the `loader` will load only vertex positions, but neither texture coordinates nor surface normals. Because our file does not contain any data but vertex positions, this is fine. If you have surface normals or texture coordinates in your file, you would have to add the following lines to your main method:

```
loader.setLoadTextureCoordinates(true);
loader.setLoadNormals(true);
```

Vertex Indexing

By default, the `loader` will put the vertex data of our file into a single array. However, it also supports separating the data into two arrays: A vertex array, containing every vertex only once, and a second array containing indices to these vertices, which allows us to reuse the same vertex multiple times. This is called “indexed vertices”. Loading all data into a single array is a easier to handle and may be faster to load, but will most of the time consume more memory than two separate arrays, because the vertices will need to be repeated for each triangle.

With our simple example file, we don’t need the performance benefits of indexed vertices. If you want indexed vertices, configure the `loader` by adding the following line to the main method:

```
loader.setGenerateIndexedMeshes(true);
```

Have a look at the JavaDoc for more information.

Loading the contents from the file

Up to this point, no file has been loaded, but only has the `OBJLoader` been customized to fit our needs.

Now it's finally time to load our OBJ file using our perfectly configured `loader` object. It is important to first configure the `loader` and after that load the file. Otherwise, the configuration of the `loader` will not affect the loading of the file.

First, we need to remember where our example OBJ file is located. On my computer, I have put the file into a folder inside the java project called "resources". This is a relative path, because our file is placed inside the project folder and will possibly be contained in a jar file of your own project. We will use the `Resource` class of the OBJ library, which enables us to both load files from absolute file paths but also from local resources in your projects jar. Because our file is in our project folder, it is a local file. That's why we will use `Resource.bundled(path)` to tell the `loader` where to find our files.

Add the following bold lines to your project:

```
import java.nio.file.Paths;
import de.hshl.obj.loader.Resource;
...
Resource file = Resource.bundled(Paths.get("triangle.obj"));
loader.loadMesh(file);
```

If you don't use the "resource" folder to include your files in your JAR file, you can use a plain old file path instead, using `Resource.file(path)`. On Windows, this may look like the following code:

```
Resource file = Resource.file(Paths.get("C:/data/triangle.obj"));
```

Using the Path class enables us to use the forward slash on any operating system, even though paths on Windows internally use the backwards slash instead.

This loading method will put all data from the file into a single array. However, in total, there are four different loading methods for different purposes:

1. Load all data from the file into a single array:
   `loader.loadMesh(file)`

2. Load the separate polygon objects from the file into multiple arrays:
   `loader.loadMeshObjects(file)`

3. Load the all data from the file into a single array, and also load materials:
   `loader.loadSurfaces(file)`

4. Load the separate polygon objects and their materials from the file:
   `loader.loadSurfaceObjects(file)`

Have a look at the JavaDoc for more information.

Summarizing the Base Code

Our file now looks like this:

```
import de.hshl.obj.loader.OBJLoader;
import de.hshl.obj.loader.Resource;
import java.nio.file.Paths;

public class Main {
	public static void main(String[] args) {
		OBJLoader loader = new OBJLoader();
		Resource file = Resource.bundled(Paths.get(“triangle.obj”));
		loader.loadMesh(file);
	}
}
```

Which can be simplified to the following code:

```
import de.hshl.obj.loader.OBJLoader;
import de.hshl.obj.loader.Resource;
import java.nio.file.Paths;

public class Main {
	public static void main(String[] args) {
		new OBJLoader()
			.loadMesh(Resource.bundled(Paths.get(“triangle.obj”)));
	}
}
```

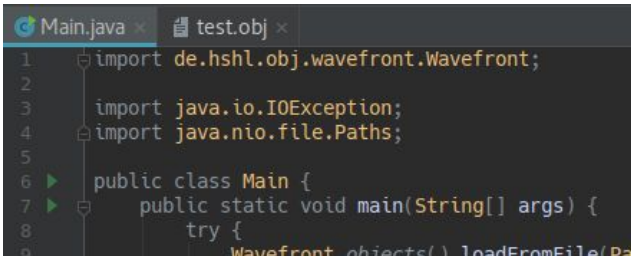
Handling Exceptions

But we can't run the program yet. Java wants us to explicitly react to any exceptional behaviour. So the next thing we need to do is to handle any errors that could occur while loading the file. We can do that using a try-catch-statement, which handles any `IOExceptions`. The following code will react to all loading errors by printing the error to the java console. Add the bold lines to your Main class:

```
import java.io.IOException;
...

// inside our main method
try {
      // load the file as explained above
}
catch (IOException exception){
      exception.printStackTrace(); // print the exception to the terminal
      System.exit(1); // stop the java application since there was an
error
}
```

Running the program



Our IDE will finally allow us to run the project using the green arrow next to our class definition, found at line 6 in this screenshot. Click the arrow and choose *Run Main.main()*.

This will execute our main method. If nothing went wrong, you should see something like `Process finished with exit code 0`. The file was loaded, but we did not do anything with the loaded contents of our OBJ file.

3. Looking at the contents of the file

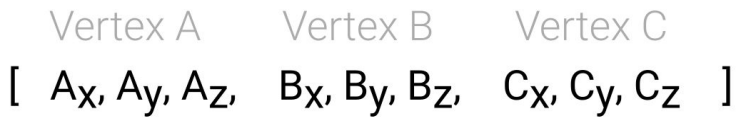
Finally, we can load our OBJ file. But of course, we would want to take a look the loaded contents. The object returned by `loadMesh(...)` is an object containing the vertex array (and optionally index array) of the whole OBJ file. Using `getVertices()`, we can extract the vertex data from the mesh:

```
import de.hshl.obj.loader.objects.Mesh;
...
Mesh mesh = loader.loadMesh(...);
float[] vertices = mesh.getVertices();
```

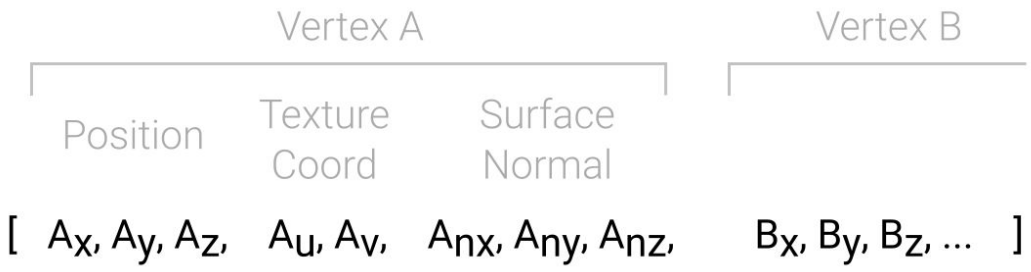
Vertices Format

Vertices are stored in a commonly used interleaved format, where the x, y, and z coordinates of all vertex positions are written one after each other. That is the exact format that is typically used when loading vertex data into OpenGL.

If only vertex positions were loaded, the array looks like depicted:



If texture coordinates and normals were loaded, they would appear in that array in the following order:



You can check whether the Mesh contains normals and texture coordinates by looking at `mesh.containsTextureCoordinates()` and `mesh.containsNormals()`. Have a look at the JavaDoc for more information.

Summarizing the example

The java file:

```
import de.hshl.obj.loader.OBJLoader;
import de.hshl.obj.loader.Resource;
import de.hshl.obj.loader.objects.Mesh;
import java.nio.file.Paths;

public class Main {
	public static void main(String[] args) {
		try {
			Mesh mesh = new OBJLoader().loadMesh(
				Resource.bundled(Paths.get(“triangle.obj”))
			);

			System.out.println(mesh);
		}
		catch (IOException exception){
			exception.printStackTrace();
			System.exit(1);
		}
	}
}
```

The OBJ file “triangle.obj” located in the resource folder inside the Java project:

```
v 0.3 0.1 0.4
v 0.1 0.4 0.3
v 0.4 0.3 0.1
f 1 2 3
```

Should print the following text:

```
Mesh {
	indices = null,
	contains normals = false,
	contains texture coordinates = false,
	vertices = [
		0.3, 0.1, 0.4,
		0.1, 0.4, 0.3,
		0.4, 0.3, 0.1
	]
}
```

Distinguishing the Objects in an OBJ File

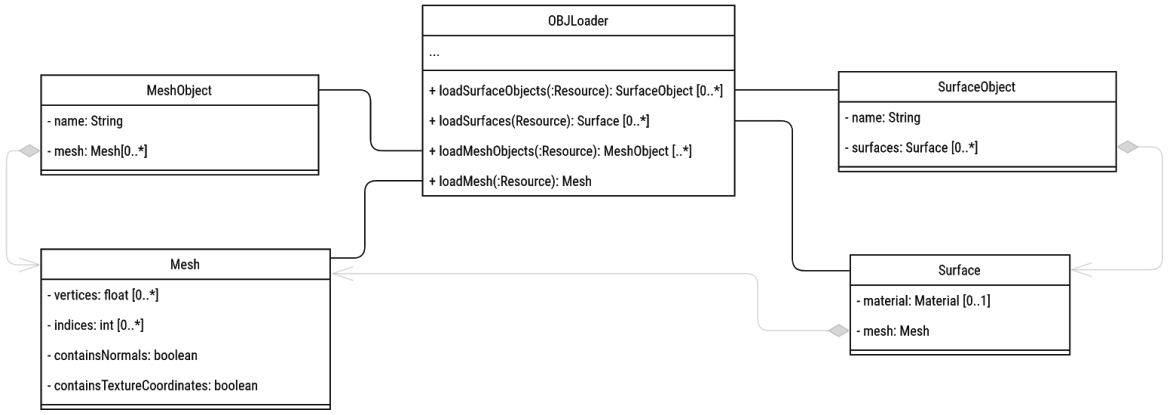
As mentioned earlier, the loader has four different methods for loading different types of contents.

The simplest one is `loadMesh`. As you can see, it returns a `Mesh`, which just contains the plain vertex data arrays, and nothing else.

The next method would be `loadMeshObjects`, which still only loads geometry, but distinguishes the individual objects and groups of an OBJ file by loading each of those into a separate `MeshObject`. Each of these objects contains its own name and mesh.

If you want to load materials from an OBJ file however, you must use one of the methods that load `Surfaces`. The `loadSurfaces` method, for example, loads all contents of the file without distinguishing objects, but also includes `Materials`. It returns a separate `Surface` for each `Material` in the file.

The most detailed method is `loadSurfaceObjects`, which divides the OBJ file into its objects, and then divides those objects into `Surfaces`, for each `Material` in that object. The following diagram should clarify that architecture:



Further Loader Configuration

`loader.setLoadedStructureKind(StructureKind.Objects / StructureKind.Groups)`
By default, when using either “loadMeshObjects” or “loadSurfaceObjects”, the loader will load only content inside OBJ objects, but not OBJ groups. This can be changed by setting the structure kind to “groups”.

`loader.setPrintWarnings(true / false)`
Supress warnings about unsupported features.

`loader.ignoreThirdTextureCoordinate(true / false)`
Some programs export 3D texture coordinates. If you cannot disable that behaviour in your program, this configuration will ignore the third coordinate and load only the first two texture coordinates.

`loader.withTriangulator(new CustomTriangulator())`
The OBJLoader library allows you to load OBJ files using your own triangulation algorithm. The default triangulation algorithm will split polygons in a simple manner. If your project required more complex triangulation algorithms, you can use this method. Your custom triangulation algorithm could for example look at vertex positions and surface normal to achieve a better triangulation.

Have a look at the JavaDoc for more information.

4. Supported OBJ Features

Supported

- vertex positions, texture coordinates, surface normals
- faces, including non-triangular polygons
- material libraries
- objects, groups

Ignored OBJ Features

- smoothing groups, bevel, point clouds, line strips
- color interpolation, dissolve interpolation
- level of detail
- shadow objects, ray tracing objects

Note that you can of course manually use level of detail meshes using multiple objects in a scene or multiple OBJ files.

You can support these features yourself by extending the `de.hshl.obj.wavefront.obj.TriangularOBJBuilder` class.

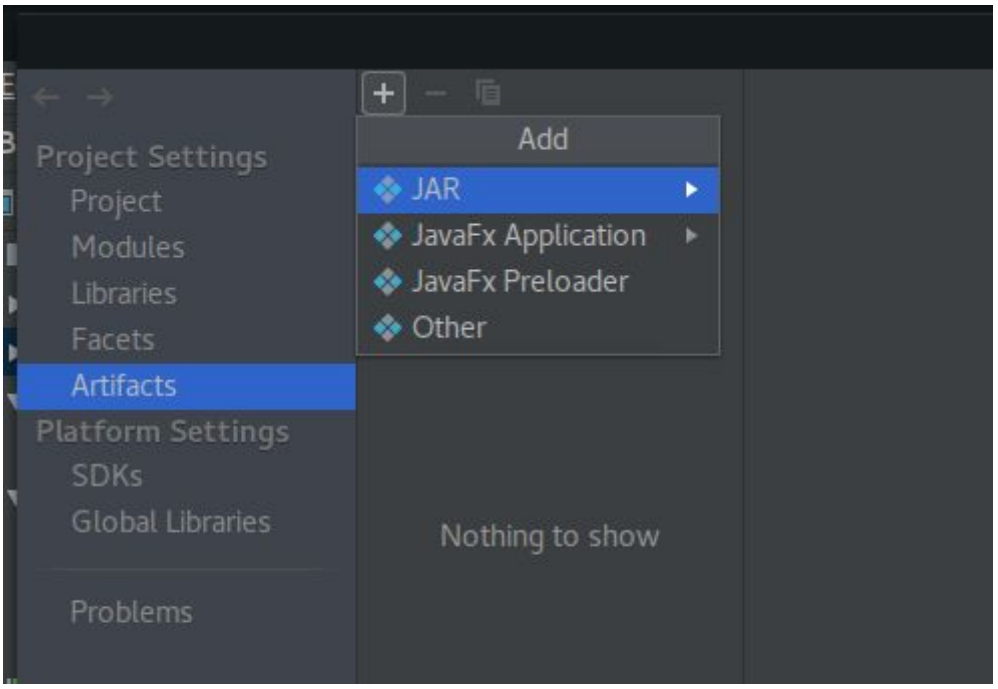
Unsupported OBJ Features

- free form surfaces and curves, basis matrices, step size, trimming loops
- executing unix commands
- linked obj files

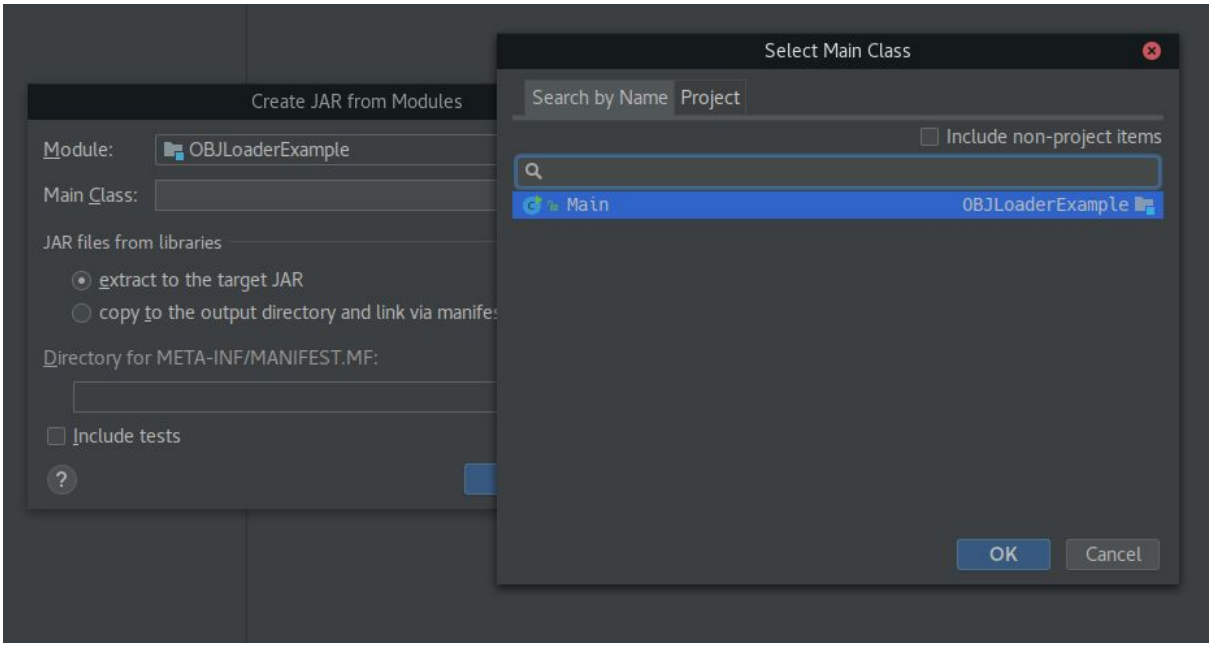
You can support these features yourself by extending the `de.hshl.obj.wavefront.obj.OBJParser and de.hshl.obj.wavefront.obj.TriangularOBJBuilder` class.

5. Building a .jar file

To build your project into a .jar, go to “File” > “Project Structure…”, then choose “Artifacts”.
Click the “+” button at the top.



Choose “JAR” > “From modules with dependencies…” and where it says “Main Class” select your class that contains the main method.

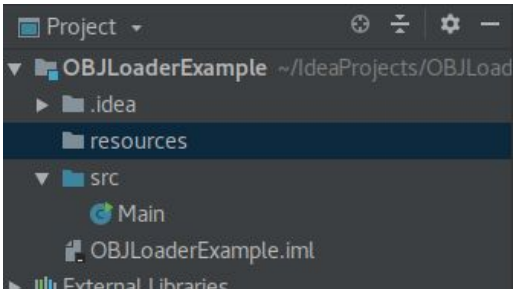


To finally generate the .jar file, choose “Build” > “Build Artifacts…” and then “Build”. The built .jar file will be written into your projects folder in “out/artifacts/YourProject_jar/YourProject.jar”.

6. Packing your OBJ files with your .jar

This can be useful in order to load a file in your project without specifying the absolute path. Also, if you export your project as .jar file, you can load files relative to that .jar file using this technique.

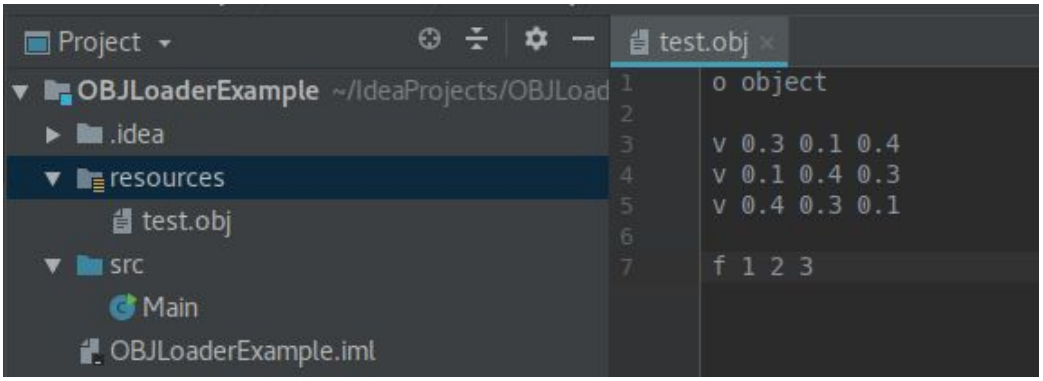
Add a resources folder to your project. Right click on it and choose “Mark directory as…” > “Resources Root”.



Add a new obj file to the resources

```
o triangle
v 0.3 0.1 0.4
v 0.1 0.4 0.3
v 0.4 0.3 0.1
f 1 2 3
```

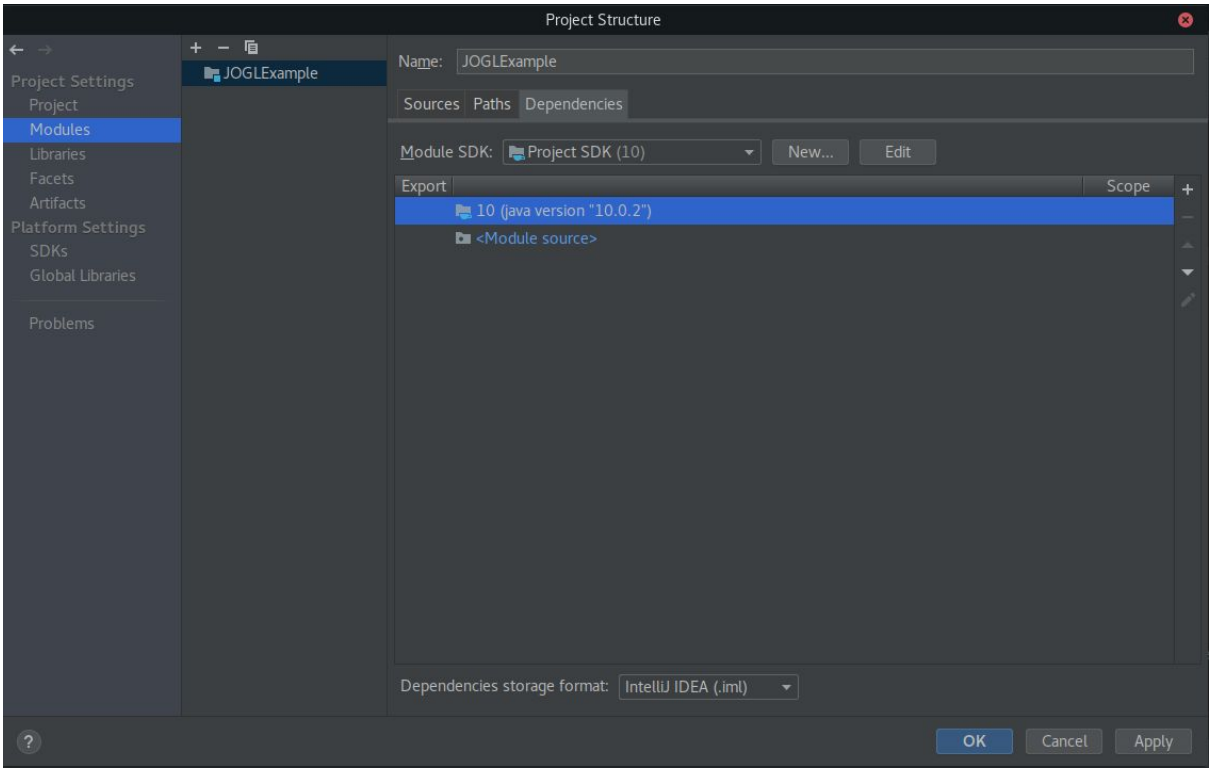
This file will contain a single triangle with only positions (without surface normals, without texture coordinates, and without materials).



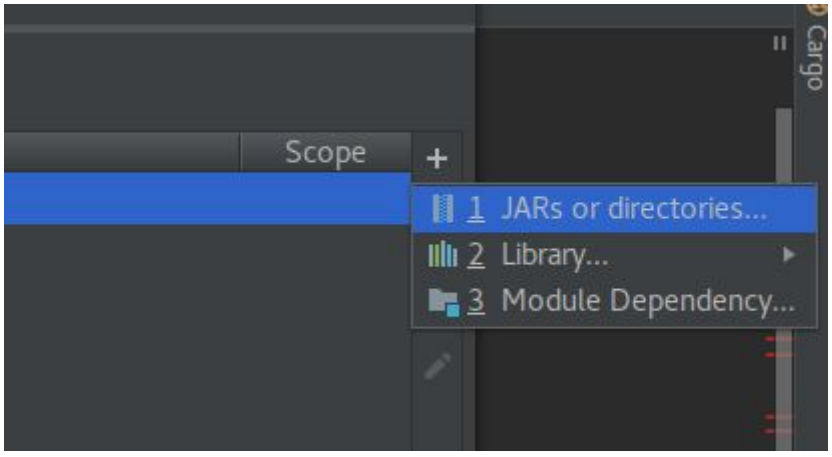
To load local files from your project folder, use `Resource.bundled(Paths.get(“resources”, “test.obj”))` instead of `Resources.file(...)`.
This **will also work when you have built your application into a .jar file**, because IntelliJ will put anything from your resources folder into that .jar file.

7. Including JOGL in your Project

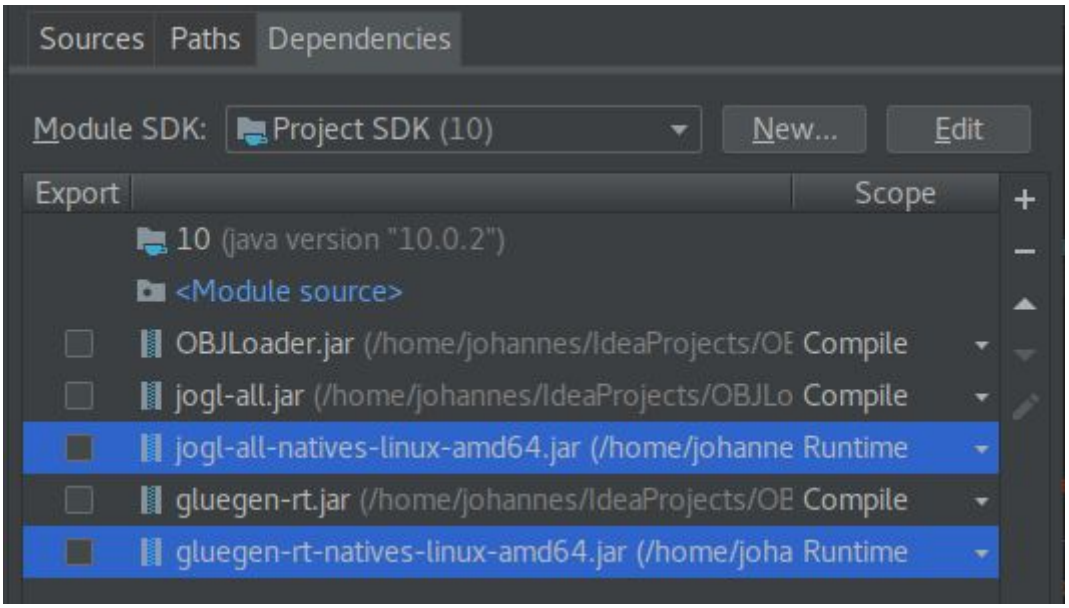
To use the OBJLoader with JOGL, we need to import all the .jar libraries first.
Go to “File” > “Project Structure…” and then choose “Modules” and navigate to “Dependencies”.



Click the “+” button to the right, and choose “JARs or directories…”. Select the .jars of the JOGL library and the OBJLoader.jar.



Set the scope to the right of the native .jars to “Runtime”.



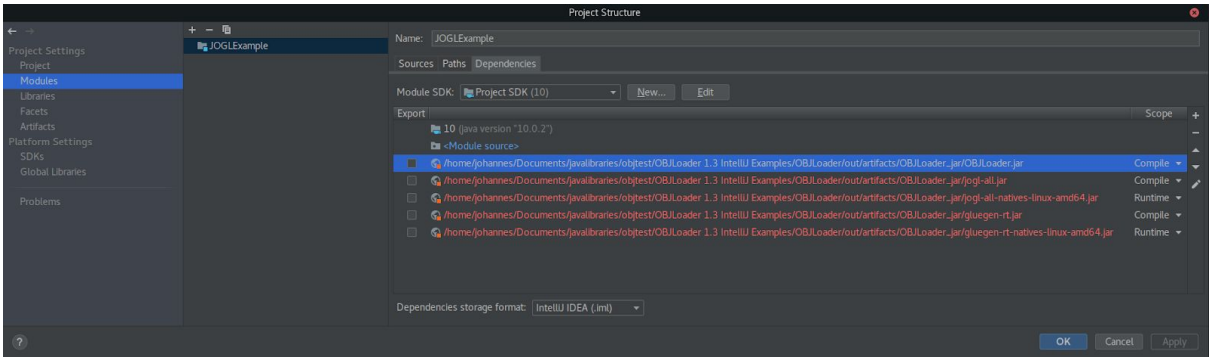
These jars will be included in your own application when you build your artifact.

8. Opening the Example Projects

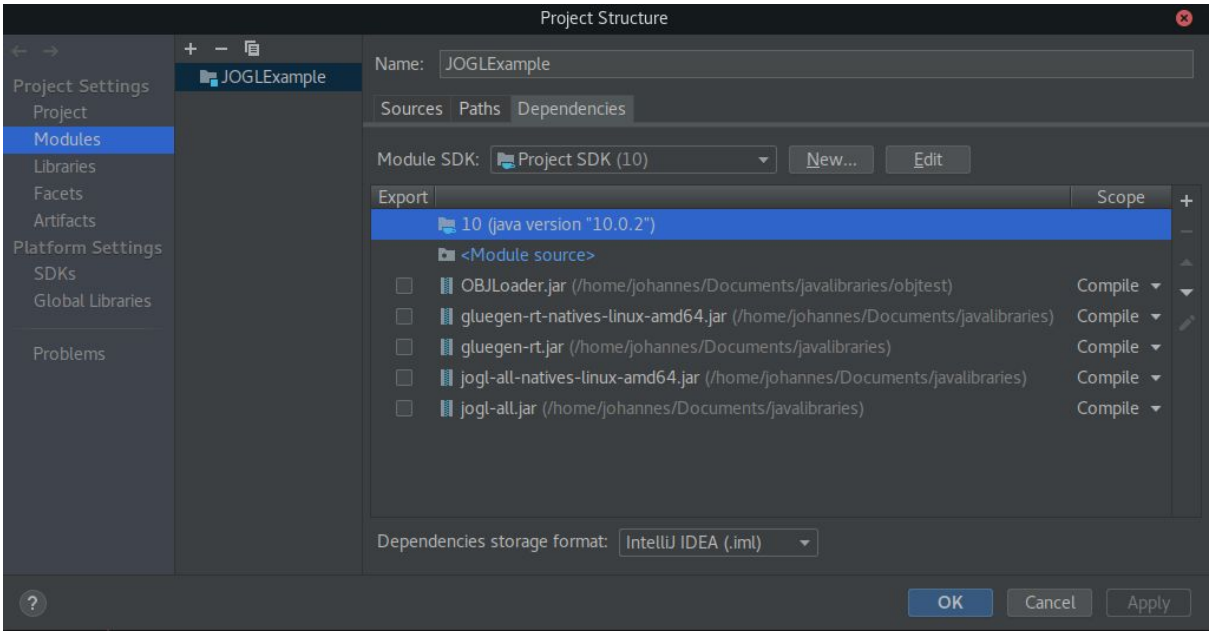
Opening “JOGLExample”

To open the JOGL example, not only will we have to update the path to the OBJLoader.jar, but also to the paths to the various JOGL libraries which are required to run JOGL.

After opening the project just like before, again navigate to “Project Structure…” > “Modules” > “Dependencies”.



This time, there are multiple outdated library paths. Remove each red entry from the list using the “-” button and then add your own library paths instead. For JOGL to work on a 64bit Linux computer with an AMD processor, you will need the “jogl-all.jar”, “jogl-all-natives-linux-amd64.jar”, “gluegen-rt.jar” and “gluegen-rt-natives-linux-amd64.jar”.



This it what it should look like, if it worked.

9. Troubleshooting

Problem: Nothing can be seen.

Try:

- Make sure the OBJ loader library returned valid content
    a. Make sure no exceptions have been thrown
    b. See the next question in this document
- Make sure your OpenGL code works
    a. Use “glGetError()” to find problems in your code
    b. Make sure the camera is pointed at the object
    c. Make sure the shaders are valid

Problem: The loader returns nothing.

Try:

- Check if your modeling software has exported any objects. If it has written only the plain vertex data without any object names, use `loader.loadMesh(...)` or `loader.loadSurfaces(...)` but not `loader.loadMeshObjects(...)` or `loader.loadSurfacesObjects(...)` to load the whole file into a single array, instead of extracting only the data contained within objects.
- If your modeling software has not exported any materials, use `loader.loadMesh(...)` or `loader.loadMeshObjects(...)` but not `loader.loadSurfaces(...)` or `loader.loadSurfacesObjects(...)`.
- If your software has exported groups but not objects, use `loader.setLoadedStructureKind(StructureKind.Groups)` to ignore OBJ object declarations and load OBJ groups instead.

Problem: The vertices are not connected to faces properly in OpenGL.

Try: If a Mesh contains indexed vertices, the vertex data cannot represent the mesh on its own, but must always be used in combination with the index array. For simplified usage without an index array, use `loader.generateIndexedMeshes(false)` to load all data into a single array.

Problem: Only triangles are visible, quads and other polygons are not.

Try:

- For best results and predictable outcomes, use your 3D software to export only triangles. If you use blender, just make sure the checkbox called “triangulate” is checked while exporting your models to OBJ.
- Make sure to use GL_TRIANGLES but not GL_TRIANGLE_STRIPS and not GL_QUADS.
- Disable back face culling in OpenGL.
- Provide your own triangulation algorithm. Per default, all faces will be triangulated using a rather primitive algorithm.