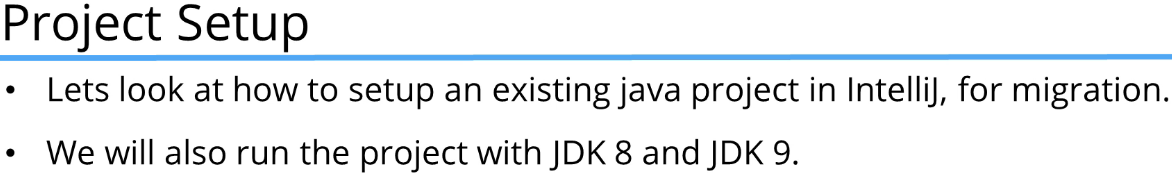
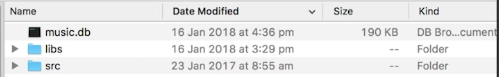
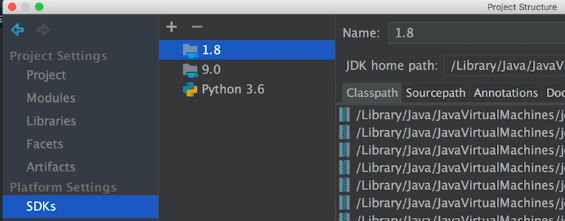
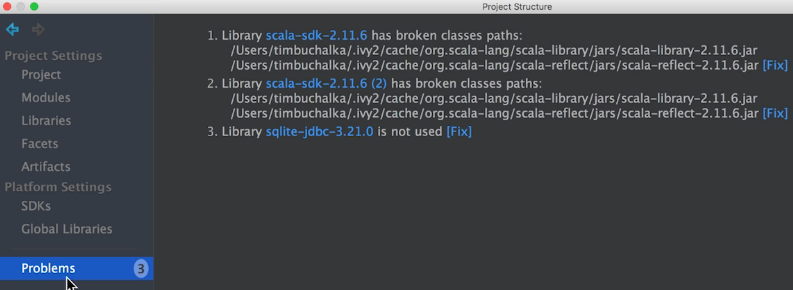
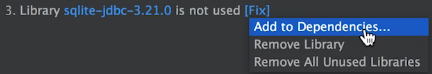
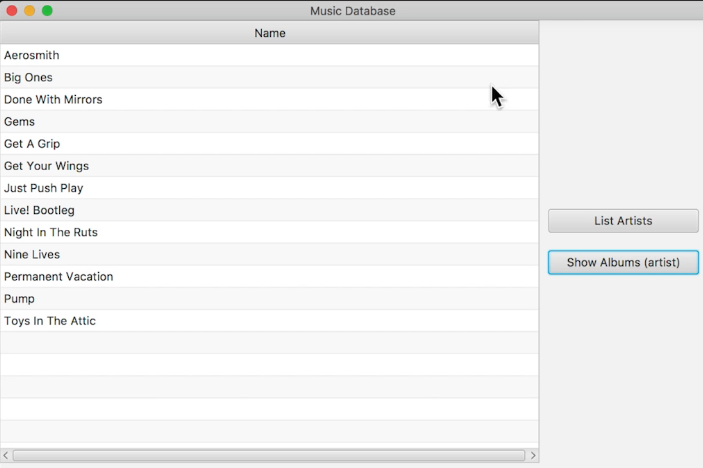
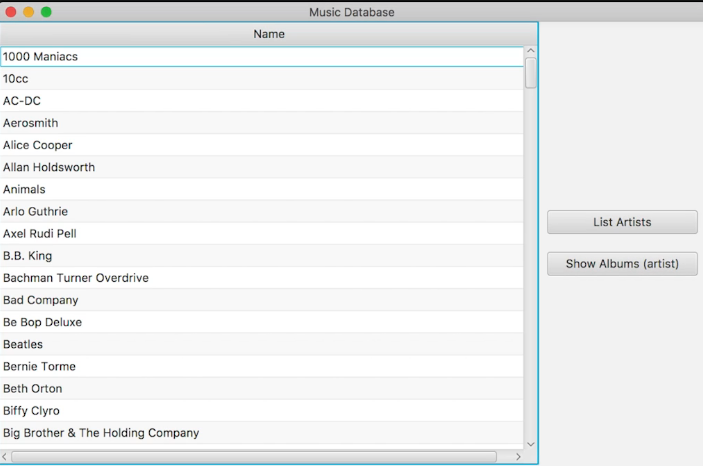
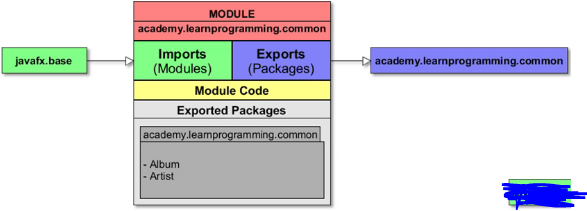
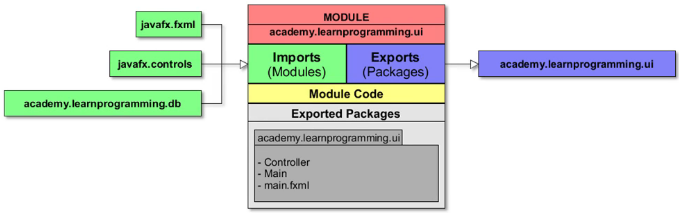
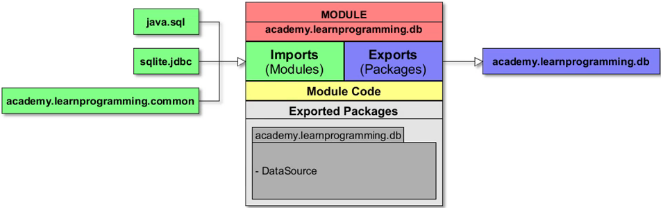
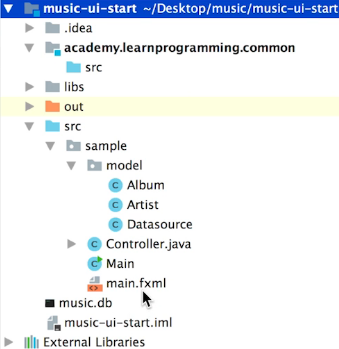
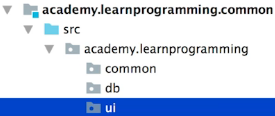
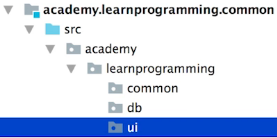
**Project Setup and Test**  
\* The main idea of this section is to show you some useful information that will help you migrating projects to Java 9.  
\* Along the way we should see some common errors that you might come across while migrating projects and also if you’re creating new projects with Java 9.  
  
\* I’m going to use the music project we used earlier in this course.  
=> Import Project => navigate to the music folder => Open => Create project from existing sources => Next => Next => Next => Next => Next => Select the JDK option - select 1.8 if you have it installed on your system or JDK 9 if you haven’t => Finish.  
  
\* Before we run the project, we need to do 2 things:  
1) Right click on the project => Open Module Settings => SDKs to check what SDK I have - you see I have JDK 1.8 and 9.0.  
  
=> I’m going to try running it with 1.8 first.  
2) Right click on the project => Open Module Settings => Project => make sure the Project language level is set to 8.  
\* In the Open Module Settings we can see Problems:  
  
\* We don’t use scala so we don’t need to worry about that.  
\* The sqlite-jdbc:  
  
OR  
=> Modules => add the jar file manually. It’s probably easier to use the FIX.

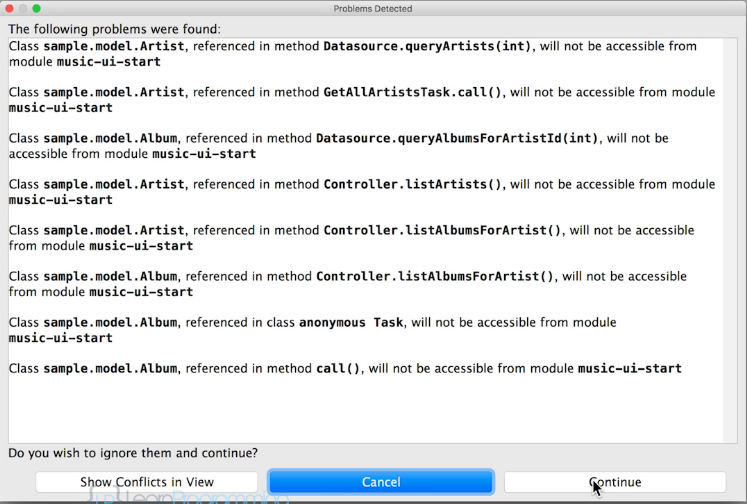
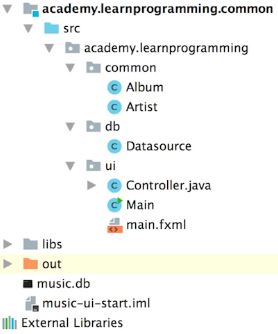
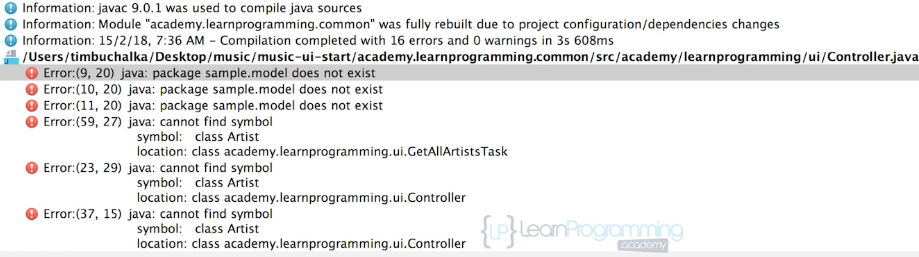
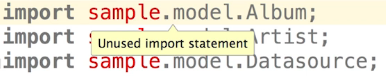
\* Let’s now run the Main.java:  
  
\* As you can see it works fine.  
\* **Now let’s get this running on JDK 9**.  
=> File => Project Structure… => Project => Project SDK: 9.0 => Project language level: 9  
\* So we’re telling IntelliJ to use JDK 9.  
\* The first time you use the JDK 9 it may take some time to start. But once it has run once you’ll find that it’ll be significantly faster.  
\* Let’s now run the Main:  
  
\* It’s all working nicely as well.  
=> So as you can see, a project written in the JDK 8 works without any problems when using JDK 9.  
\* That’s only the start though, in the next video, **let’s start discussing how to split the project into multiple modules**.

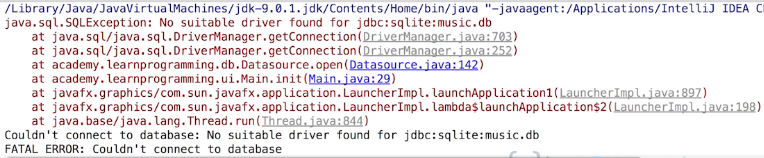
**Structuring the new project**  


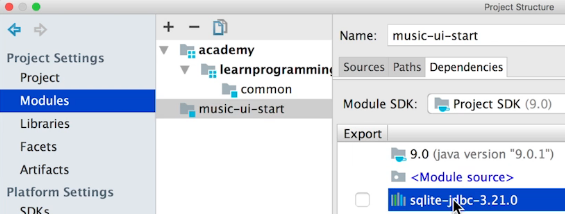
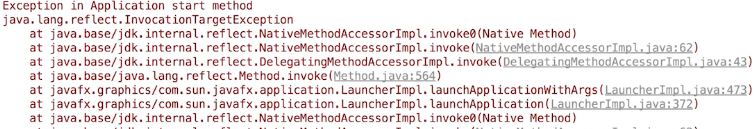
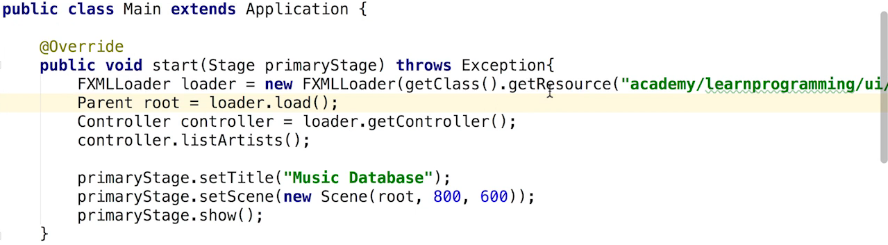
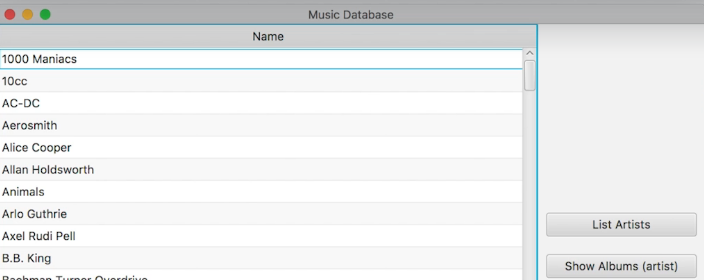


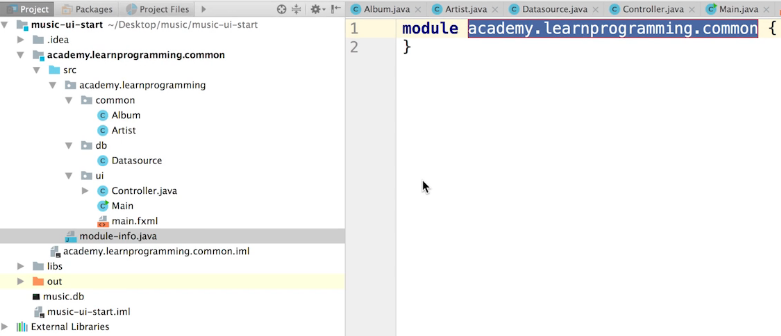
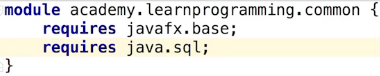
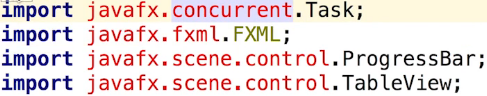
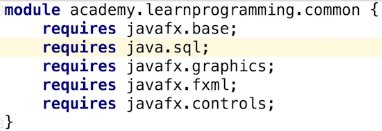
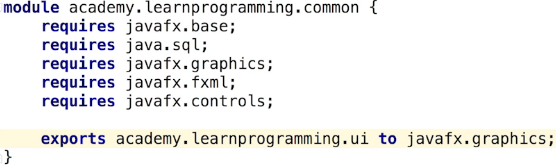
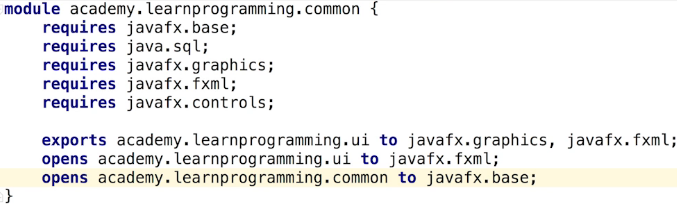
  
=> **Common** for the common code.  
=> **UI** for the user interface  
=> **DB** for the database  
\* I prefixed them with academy.learnprogramming to make a unique module name for each.  
\* The common was anything that didn’t fit into UI or DB.  
\* There are other solutions, you can easily separate it into more modules. But since the project is pretty simple we’re running a 5 classes, so 3 modules is okay at this stage.  
\* And we don’t have any internal packages for any of the modules.  
\* **I call one of the modules COMMON because it will contain common classes for the other 2 modules**.  
=> **So another good name instead of common could be core**.  
\* Common has Album and Artist - we use them to read data from the database and display it on the user interface.  
\* The common module doesn’t know anything about the database module. It doesn’t care how the Artist or Album will be saved - will it be a database, xml or something else.  
\* But the database module requires the common module to be able to convert the result from the database to an artist or an album.  
\* The UI module requires the database module because the results from the database are displayed on the user interface. It contains the Controller, Main and main.fxml.  
\* Keep in mind and remember that modules can contain source code but also resources like images and in our case an fxml file.

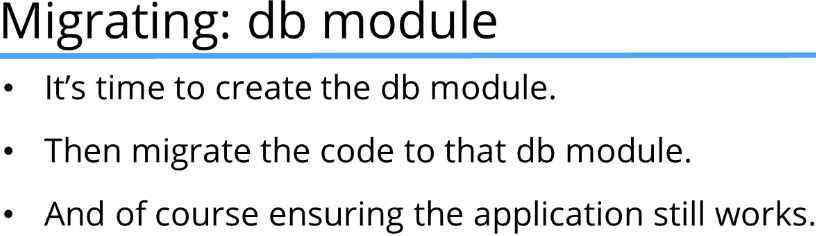
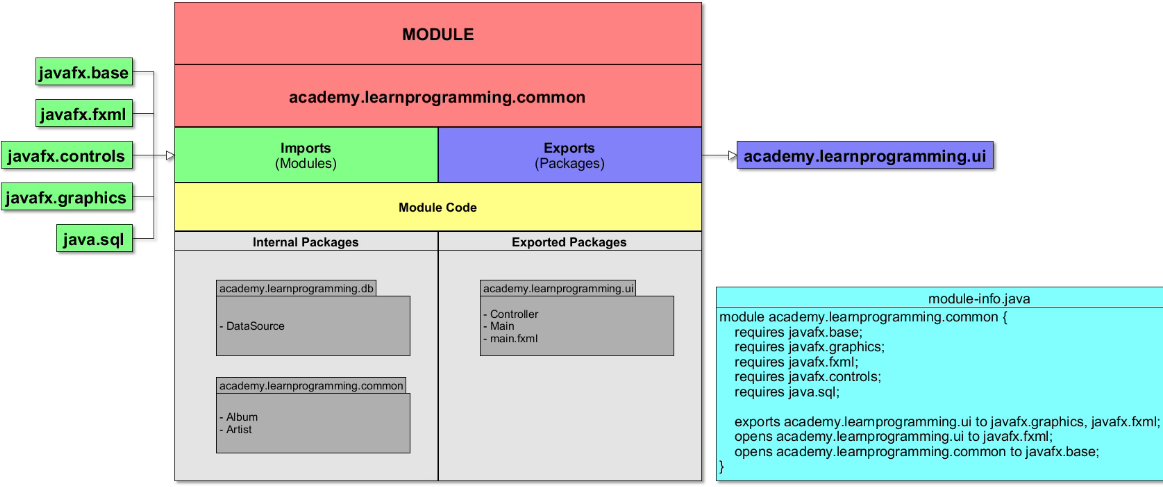
**Creating the first module (Common)**  
\* Here I’m using IntelliJ Community Edition and I think it’s important because there are several sort of renaming or refactoring pieces of functionality in the Ultimate Edition where it doesn’t happen in the Community Edition.  
\* **First we need to change the project structure**.  
=> We have to craete the module in the root folder.  
\* Confusingly in IntelliJ there’s also a concept of a module but that’s got nothing to do with the JDK 9 module, that’s an IntelliJ module (for example you can see the Open Module Settings in project). What the people behind IntelliJ Jetbrains have done is decided that 1 IntelliJ module will represent 1 Java 9 module, then that way we only need to add the one module descriptor to that module.  
\* Once you’ve done this a few times I think it will be pretty straightforward.  
**=> File => New => Module… => Java => Module SDK: (9.0) => Next => name it => Finish**.  
\* **This is our current project structure**:  
  
\* One approach is to move all classes into this module initially but in their relevant packages and then later move those packages to the other modules as we create them.  
=> src (in the module) => New => Package => name it. (do this for our 3 packages)  
  
\* This can be a bit confusing - you can click the GEAR icon => Hide Empty Middle Packages and turn it off:  
  
\* **Personally I prefer to hide the empty package but it’s really up to you**.

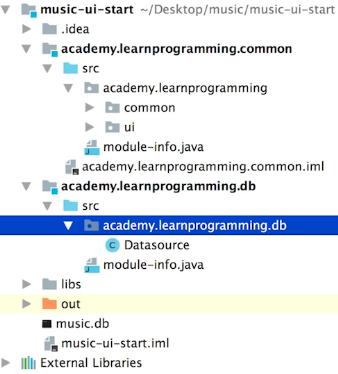
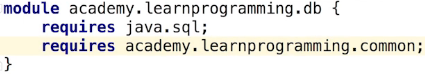
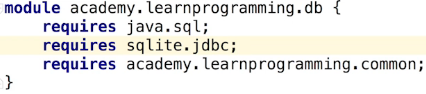
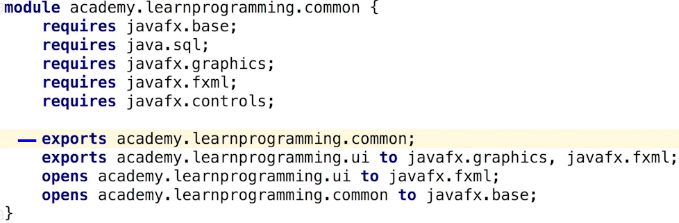
\* When you move the files, you will see:  
  
=> The Ultimate Edition it automatically fixes this for you.  
=> **In the Community Edition we’re going to have to fix this later**.  
  
\* **We can delete the old src folder once we’ve moved everything**.  
\* If we run it:  
  
=> This is the other thing you need to fix in the Community Edition.  
  
=> Let’s delete those.

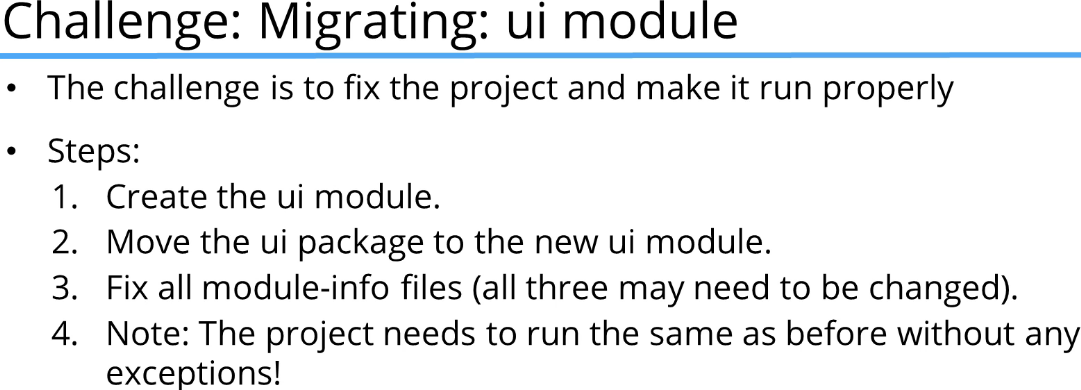
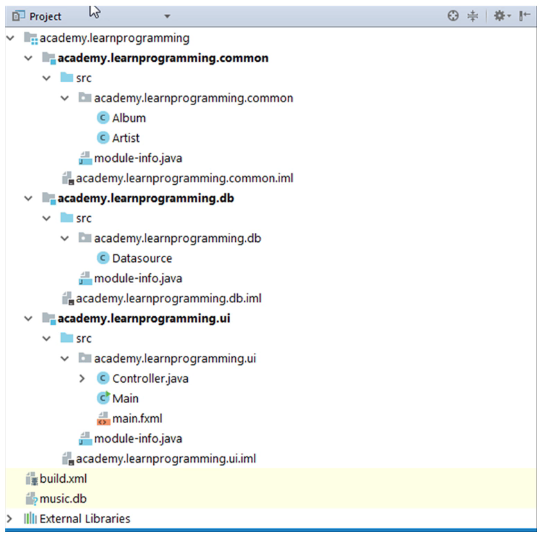
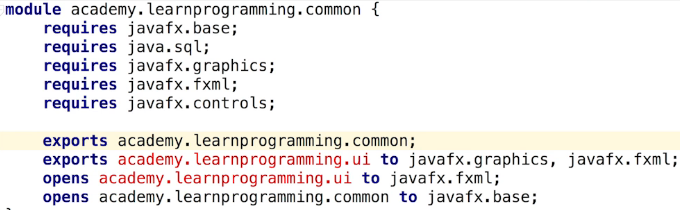
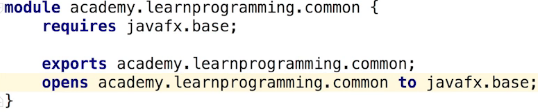
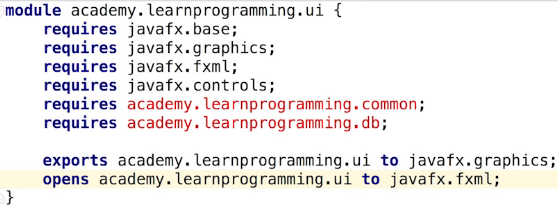
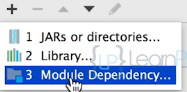
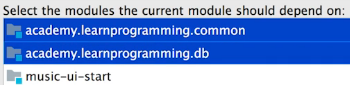
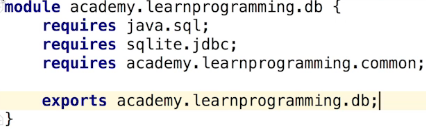
  
=> **Because I have the auto sync of imports enabled, it automatically fixed those for me**.  
\* After fixing the imports and running it:  
  
=> We can’t connect to the database. The module can’t find the SQLite driver to connect to the database.  
\* **The reason why it can’t find it is because we didn’t add this driver jar file for the SQLite database, we didn’t add it as a module dependency. It is added as a project dependency but now the root project doesn’t have any code and for that reason the dependency no longer does anything and it’s not actually finding it here**.

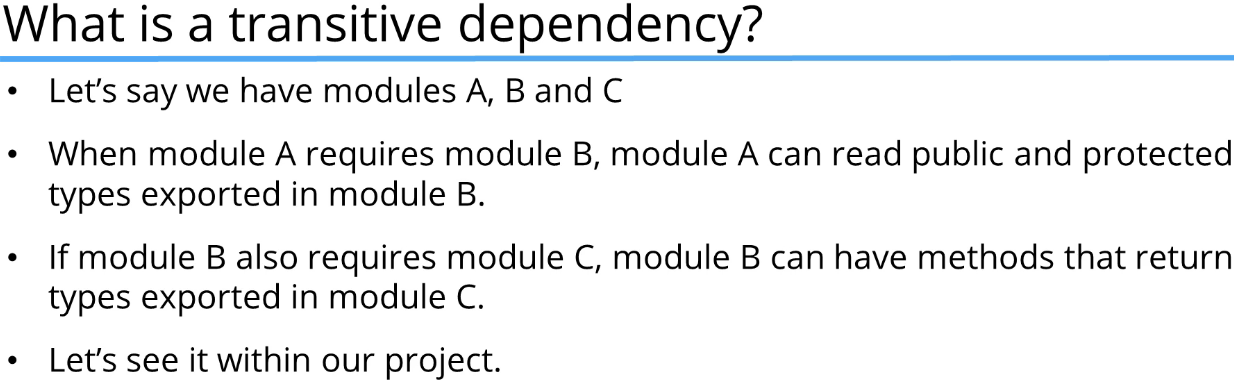
**=> File => Project Structure… => Modules => remove the sqlite-jdbc from the music-ui-start => academy learnprogramming common => + => JARs or directories… => navigate to the new folder that we’ve put this project in - into the libs folder and select the sqlite-jdbc.jar => Open**  
  
\* Now if we run it:  
  
  
\* If we click on this blue link, we see that the problem is here:  
  
=> The issue here is that previously it was using a relative path to our main.fxml. We had a sample path and the system was able to find that. But now we’re actually using a module.  
  
=> So what we need to do is specify the path the leading / so that the path is found at the start - at the root level.  
  
\* Now the application runs:  
  
\* This controller you may find it’s not updated (it is for me):  

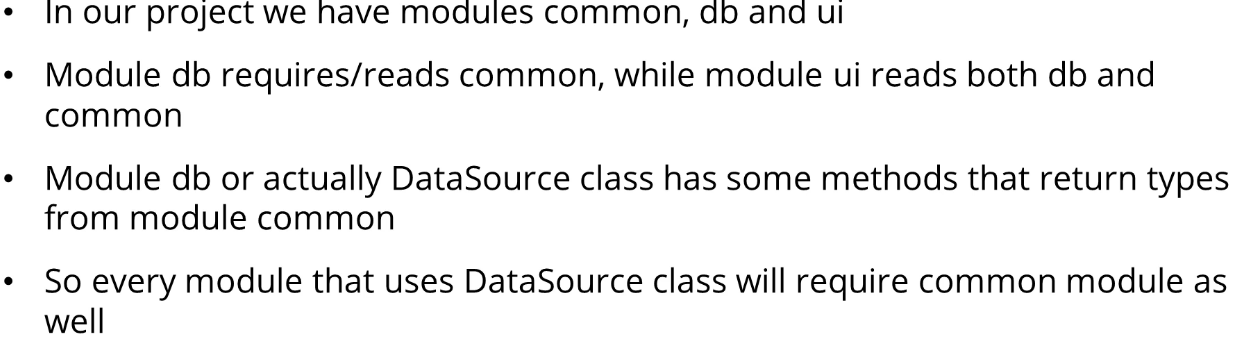

**Creating the Module Descriptor File**  
\* Let’s now think about our module descriptor.  
\* In IntelliJ we can easily add a module descriptor just by pointing or right-clicking on the src folder.  
**=> right click src => New => module-info.java  
module**  
  
\* If we do nothing else, it’s going to result in a lot of compilation errors.  
=> The problem is now we’re using classes from JDK modules and we have to require those modules, go through and add that keyword to get things to work.  
\* For example:  
  
\* That means we’re using the package from the javafx base module but we’re not reading that module.  
=> **To read a module or to use a module in our modules, we have to declare it with a requires statement**.  
\* **We could add the requires statements manually or we can get the IntelliJ to help us with these errors. To get the help, click on the red part of the scroll area**:  
  
=> Then you can see the red bulb appear:  
(or you can click on the last part of the red text in import)   
  
\* So what actually happened to the module-info.java file after clicking this?  
**requires**  
  
\* **Now let’s try doing that manually for the Datasource.java file**.  
  
=> We just type it manually:  
  
\* Let’s now fix the controller:  
  
  
\* Let’s try running it again:  
\* We still get an error, if we scroll down  
  
  
=> That means that the javafx launcher can’t launch our application because we didn’t export our package to the javafx.graphics module. If we don’t export any package classes in the module they can’t be accessed.  
\* **To export a package, we need to use the export statement**.  
**exports X to X**  
\* By exporting a package to the javafx.graphics module, javafx launchers should be able to access our package and therefore run the application.  
  
\* **Using another export statement for the same package leads to compilation errors**.  
  
  
=> We have to open the package UI to the javafx.fxml module.   
=> The reason for that is because the fxml loader needs to set fields annotated with the @fxml annotation. Without opening that package, the fxml loader can’t load load the fxml and inject or set those controller fields.  
**opens X to X**  
  
\* You can see that the process of converting to module is fairly intuitive if you take the time to read the exceptions, it’s a logical process to follow. And there are a few things that need to be done but ultimately you can say the main work was actually making sure the module-info - the module descriptor file had the right options set in there.  
\* Currently now we’ve got our entire code base inside a single module.  
\* Our current module-info file has some redundant lines.  
=> This is because some modules that we require depend on other modules but we’ll talk about it later.

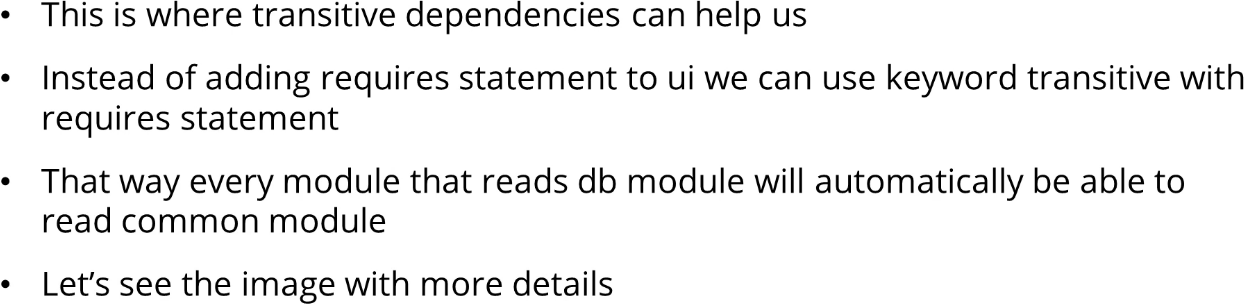
**Creating the 2nd module (Database)**  
  
\* Our current status:  
  
\* UI is exported so that we could get our graphical user interface working.

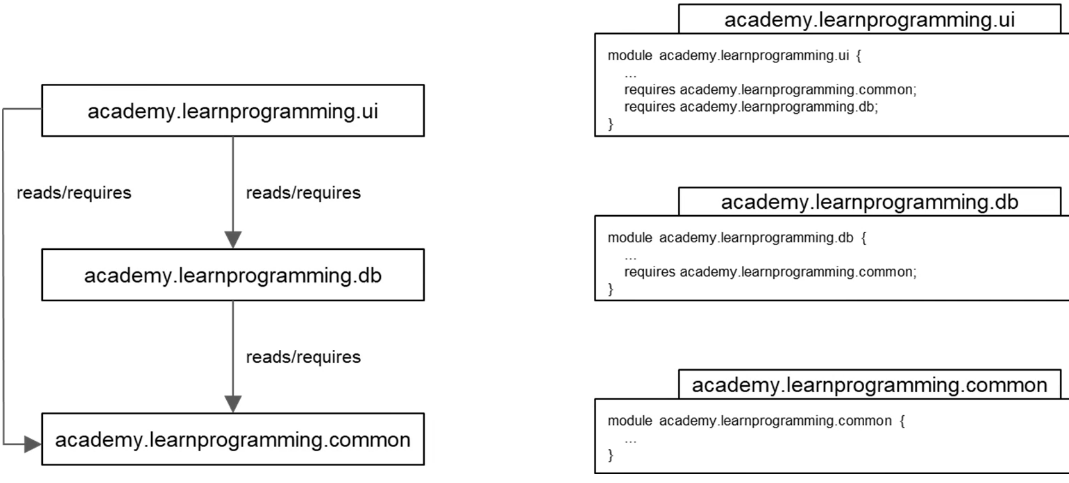
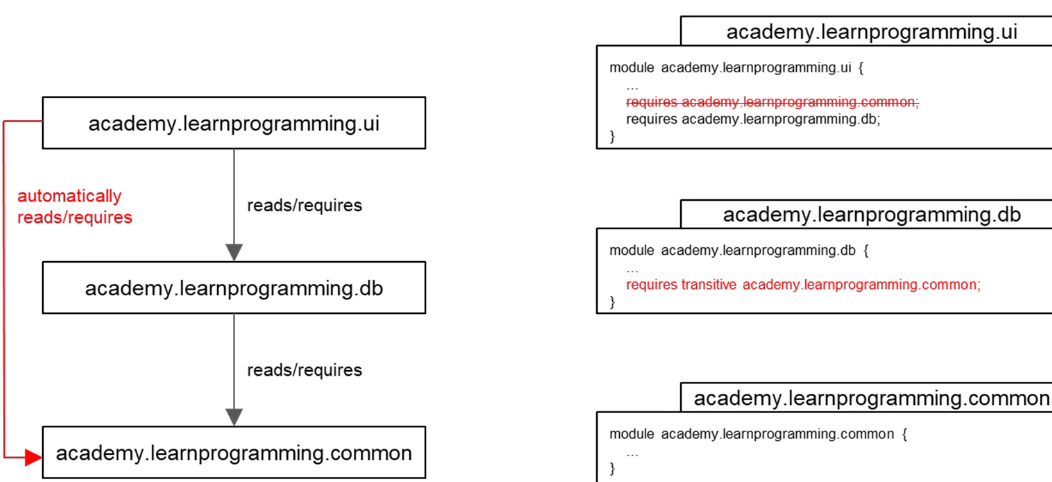
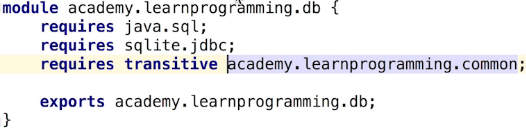
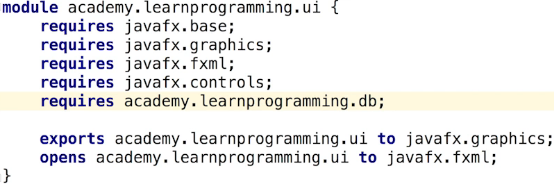
\* Let’s create the new module.  
\* Let’s create the module-info.java file.  
\* Let’s move the DB package into the new module.  
=> right click our new src => New => Package => name: academy.learnprogramming  
=> **Notice I didn’t do the .db there**.  
\* Now let’s move the db package from the common module to the db module.  
  
\* It has the .db now.  
\* Since the db module uses the classes Album and Artist but also some classes from the sql module, you probably know by now that we need to add the requires statements into the module-info.java.  
\* If we add this manually, we get an error:  
  
=> We wouldn’t get this error if we used the red bulb to do it for us.  
=> That’s because the module common is not in the dependencies in IntelliJ and we have to add the module as a dependency.  
=> The easiest way to do that is to use the red bulb.  
  
\* **Since now only the db mobule requires sql, what we’re going to have to do is remove the sqlite from common and add it to the db module**.  
\* But also we need to add the common module to the db module dependencies.  
=> The easiest way to do that is to go back to Project Structure… => Modules => db => + => jars => libs => sqlite. And let’s also remove the sqlite-jdbc from the commom module.  
\* **Now we need to add a dependency for sqlite-jdbc**.  
=> **That’s an automatic Java 9 module**.  
=> **Remember, automatic modules are modules that are created when the JAR is added to the module path.** (we had to add the sqlite dependency to the db module which we just did)  
**\* Previously with our common module we didn’t need to add the requires statement for the sqlite-jdbc module and that’s because we only had that one module**.  
\* **The name of an automatic module is the name of the jar file without the version**.  
  
(sqlite-jdbc-version => sqlite.jdbc) => that’s the general rule when you’re adding a JAR file and you want to add an automatic object as a requirement for your module.  
  
=> **We have to export the common package from the common module so that our db module can use those classes from the common module. Without exporting packages we don’t get access to classes from another module**.  
\* **In the common module**:  
  
=> **There are 2 different ways to export**.  
1) **Export to a specific package using the `to` keyword**.  
2) **Export without specifying the module**.  
\* Another problem here is that the Controller and Main classes also use the Datasource class.  
  
=> We can see that the issue really is that the import for the Datasource can’t be resolved.  
\* This is not as easy as you may think to solve - that’s because adding the module db as a dependency to the common module would result in a cyclic dependency. We would have 2 modules where both depend on each other.  
\* **Cyclic dependencies are not a good practice and Java 9 doesn’t actually allow these cyclic module dependencies**.  
\* The problem is that our current project won’t compile and we can’t run it yet.  
=> One way to solve this issue is to use **Transitive Dependencies** but we’re going to be talking more about these Transitive Dependencies later, so we won’t get into too much detail about that now.  
=> **The goal now to get around this is to actually move the UI package to the UI module and the process of doing that should resolve some of these other problems**.

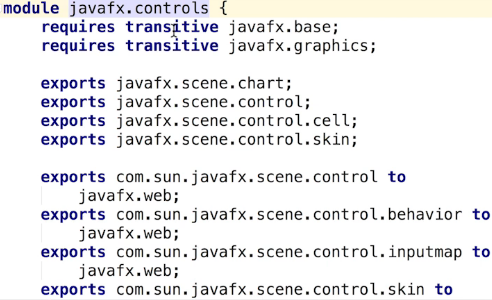
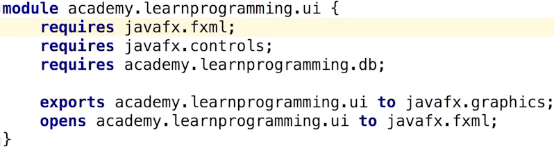
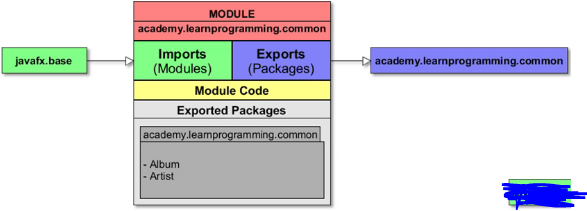
**Challenge - Create the final module (UI)**  
  
\* Just a small tip - the UI module will need the common module, not just db.  
  
  
  
  
  
  
**Final project structure**  
  
\* Let’s create the ui module.  
\* Let’s create the module-info file.  
\* Let’s create a new package called academy.learnprogramming.  
\* Let’s move the ui package from the common module to our ui module.  
\* Since the UI module uses the classes Album, Artist, Datasource and also some classes from the javafx module, we have to add the requires statements for them.  
\* **In common module we can clean up the module-info**:  
  
**to**:  
  
\* **Now in ui module-info module descriptor file**:  
  
\* We need to add the common and db modules as dependencies.  
=> File => Project Structure… => ui => + => Module Dependency…  
  
  
\* That fixes a lot of the issues.  
\* Now the Datasource:  
  
=> **our db module is not exporting the package db so other modules can’t use them now**.  
\* **In db module-info**:  
  
\* That fixes the error.  
\* **Let’s now run the application. It’s working**.  
\* **We managed to have our application working and we’ve migrated it from a single project structure to a 3-module structure**.  
\* We’ve got some extra statements that we don’t really need in our module-info module descriptor files but we’re going to fix those in the next video and we’re going to do that by understanding a way around that by using Transitive Dependencies.

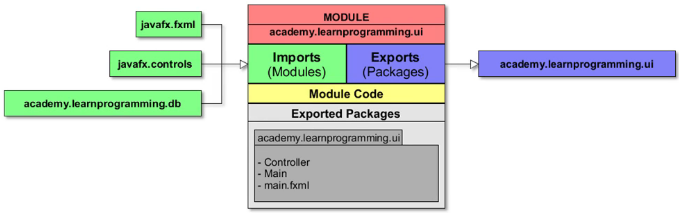
**Transitive Dependencies**  
**Transitive Dependencies**  
**Module Graphs**  


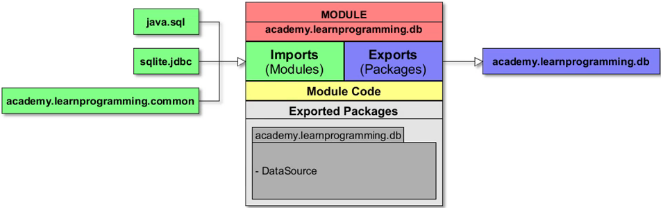




**Module Graph**  
  
  
\* **The result of Java modules depending on each other is that we can now draw a complete graph of dependencies with the modules as nodes and the relationship between nodes as module dependencies**.  
\* This graph only shows our modules, not other modules that we’re using like javafx.base.  
\* What does this have to do with transitive dependencies?  
=> You can see how our UI module depends on both the common and DB modules. Remember that the Datasource class returns some types from the common module and the UI module is using the Datasource class to get results from the database.  
=> **So instead of requiring the common module in the UI module, we can declare it as a Transitive Dependency in the DB module**.  
=> That way whatever modules use the DB module would also have access to the common module.  
  
=> **This way all modules that require the DB module now have access to packages that are exported by the common module**.  
**transitive**  
  
\* And we can remove the requires for it in the UI module descriptor file:  
  
\* So with Transitive Dependencies we don’t have to require every single module.  
\* What other requires statements can be removed?  
\* Let’s look at the UI first.  
\* You can CTRL + CLICK on the module name (e.g. javafx.base) and there you can see what packages are exported by default.  
  
=> You can see here that it requires java.desktop.

\* If we click on the javafx.controls:  
  
=> It requires javafx.base and javafx.graphics so you can see that we don’t have to depend on the javafx.base module because we’re already depending on the javafx.controls which is using the transitive keyword to require the javafx.base module and the javafx.graphics module.  
=> So we can remove the javafx.base and javafx.graphics requires from the UI module-info.  
  
\* **This now represents our final project structure**:  




  
\* As you can see now at this point, the common module only depends on javafx.base module.  
\* The db module depends on java.sql, sqlite.jdbc and our common module.  
\* The ui module depends on javafx.fxml, javafx.controls and our db module.

\* Everything works fine.