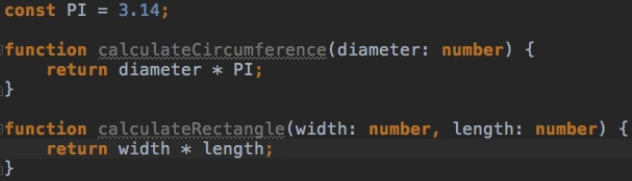
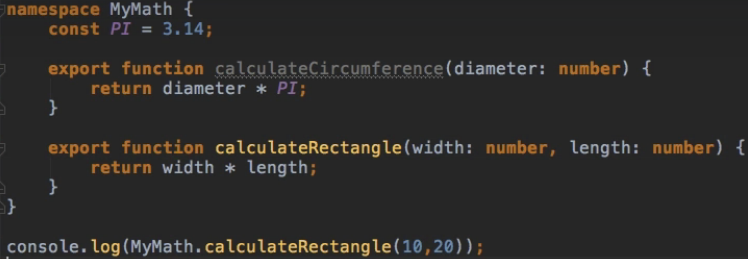
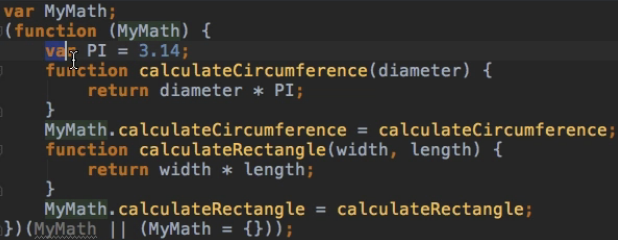
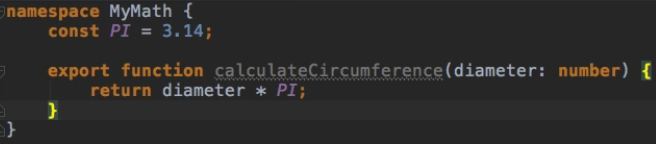
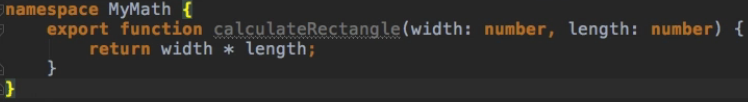
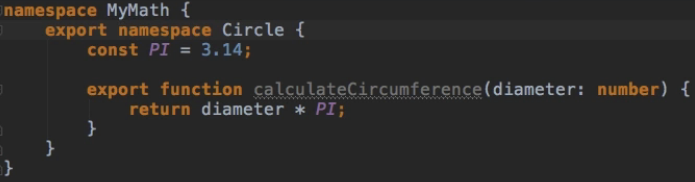
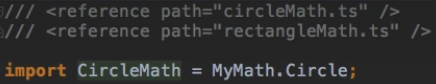
**Introduction**  
\* You might have had the case that you had a JavaScript project which really grew quite big.  
\* In the past already, we used some module loaders to modularize our JavaScript projects and split it up over multiple files.  
\* Turns out that TypeScript really supports modular code and it has 2 concepts it offers you to manage your project and split it up into different files but not only files but also logical modules:  
1) **namespaces**  
2) **modules**

**An Introduction to Namespaces**  
\* Why would we need namespaces?  
\* Why would we need to kind of improve the structure of our code?  
\* Let’s say we want to build a couple of tools we can work with regarding mathematical operations.  
  
\* We can kind of group them mentally as having them all related to math.  
\* So we might want to create a namespace where we want to group all these math things.  
\* Let’s say we wanted to do more in this file, then it could get quite messy to work with all these functions, recall them, make sure to not use duplicate names or something like that.  
**namespace**  
\* It is kind of a JavaScript object.  
**sexport**  
=> **We have to mark the things we want to be available outside of the namespace**.  
  
\* **This way we don’t pollute the global scope**.  
\* **PI is still only available within the namespace, not oustide**.  
\* **It looks like this when compiled**:  
  
\* **So we can restructure our code and make sure that that not everything is registered on our global scope which tends to get polluated quickly in JavaScript since everything is registered there by default**.

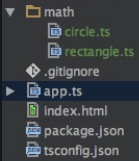
**Namespaces and Multiple Files**  
\* Imagine you have a much bigger MyMath namespace.  
\* Then you might want to split it up.  
\* You might even want to split up the namespace declaration because you might have some functions which are related to circles and other functions which are related to rectangles for eaxmple.  
\* circleMath.ts  
  
\* rectangleMath.ts  
  
\* Now we no longer have access to them because we’re only importing the app.js.  
\* We have 2 ways to solve this.  
1) **Duplicate the import**  
  
\* **The namespaces are automatically combined since they share the same names**.  
\* **The downside is that we have to import all these files**.  
2) **TypeScript compiler allows us to automatically bundle all our files together into one single file**.  
  
\* **The order might be important depending on how your code interacts with each other and what needs to get executed first**.  
  
\* The downside is that we need to list all the files we want to bundle together.

**Namespace Imports**  
\* I’m not talking about imports like you might know them from the ES6 syntax.  
\* So we won’t do `import from something`.  
\* We’ll talk about that when we cover modules.  
\* **TypeScript has its own import syntax for namespaces**.  
**/// <reference path=”” />**  
  
**tsc app.ts --outFile app.js**

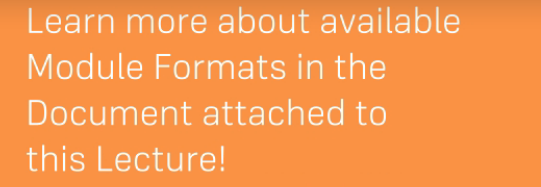
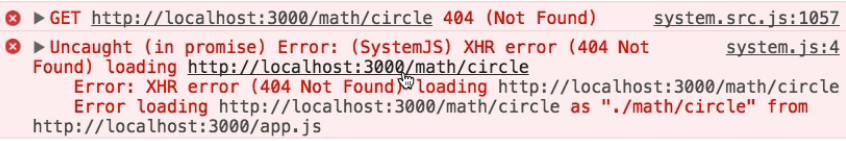
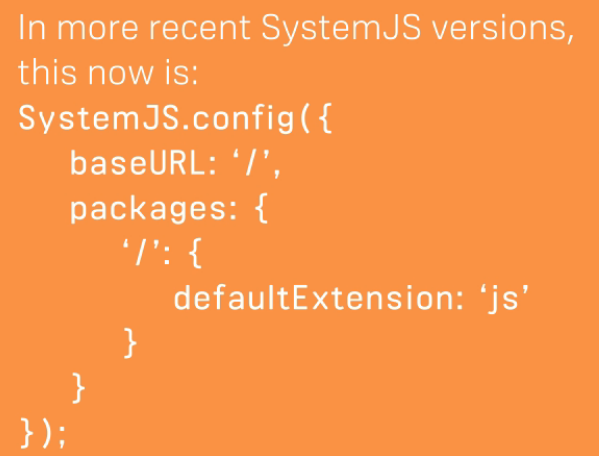
**More on Namespaces**  
\* **You can also have namespaces in namespaces**.  
  


\* Of course don’t overdo it.  
\* If you then want to still access everything without nested . in the app.ts file for example, one thing you could do is add a special keyword `**import**`.  
  
  
\* With that you’re setting up an alias here where you use the nested namespace and assign it into one single alias in the end.

**Limitations of Namespaces**  
\* Our little example is quite manageable.  
\* But we already have to watch out that we’re not forgetting anything and that we’re keeping the right order.  
\* The problem is that our files are not very declarative about what they need.  
\* Let’s say the circleMath would need another namespace, then we would need to add the import either in circleMath or app too but since we can add it anywhere we want as long as we make sure that everything gets bundled into 1 file, it can be kind of hard to clearly see which file has which dependencies and it’s not really that manageable for bigger projects.  
\* So it has some disadvantages compared to the ES6 modules which are very declarative about which file or module has which dependency and which also makes sure that each ES6 module has its own scope and doesn’t pollute the global scope.  
\* So while namespaces here a great tool for smaller applications, you probably want to fallback to ES6 modules or modules in general whn working on medium-size to large projects.

**Modules**  
\* Let’s create a `math` folder.  
  
\* TypeScript will automatically look for .ts .tsx .tts files.  

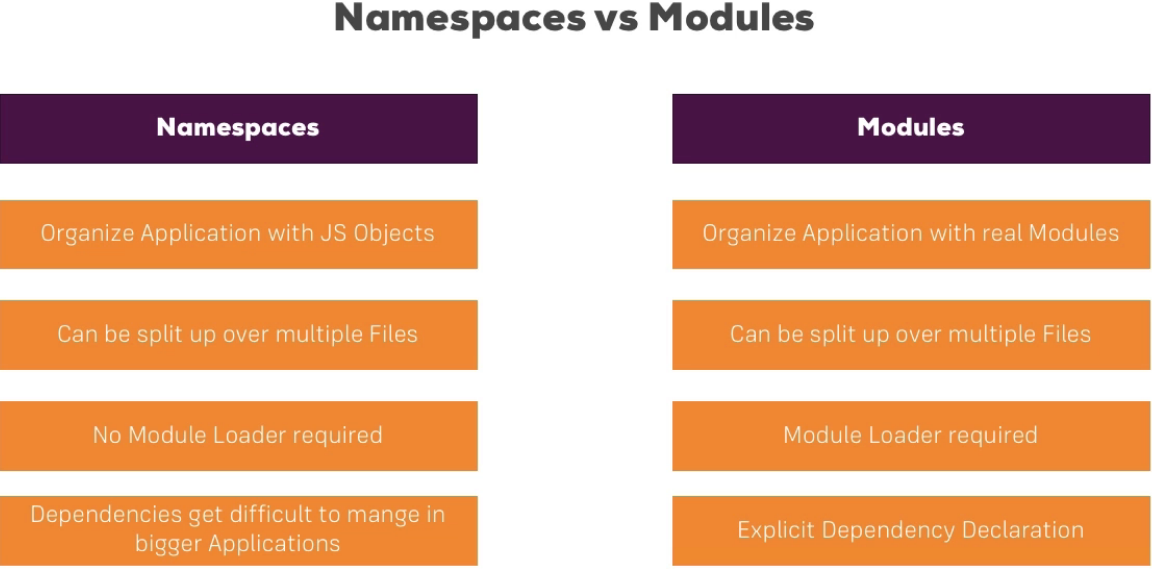

**Use SystemJS 0.x**  
\* In the next lecture, we'll install a package that helps us with loading modules.  
\* Make sure to install SystemJS (that's the package we'll use) < 2.  
\* With a command like this one:  
**npm install --save systemjs@0.21.5**  
\* Execute this command instead of just npm install --save systemjs@

**Loading Modules**  
\* If we try **tsc --outFile app.js**, we get:  
  
\* That refers to how TypeScript should compile this to JavaScript because as you might be aware, native JavaScript doesn’t support modules, native JavaScript doesn’t know the `import` `export` syntax. We can’t import JavaScript files into JavaScript files.  
\* **We can with ES6**.  
  
  
(ts-modules.pdf)  
=> The imports are compiled to meet the commonjs module standard.  
=> It means that it is trying to fulfill the commonjs way of structuring your code, that would be with the require keyword which is added as an import. And we can see this if we just run **tsc** without bundling.  
  
\* **That’s the same syntax as NodeJS**.  
\* **That syntax doesn’t work in the browser, at least not without a module loader which is what we need for that**.  
\* The error message before told us that we can compile everything into one file, if we set this to **amd** for example  
  
**tsc --outFile app.js**  
\* Now it didn’t give us an error in the console.  
\* If we check it in the browser, we get:  
  
\* The reason stays the same - we need a **Module Loader**.  
=> **Module Loader is JavaScript itself but it adds the logic our browser needs to be able to work with these commands that it doesn’t know by default, like `require` for example**.  
**npm install --save systemjs@0.21.5**  
  
  
**tsc**  
  
\* We have to tell SystemJS that our base extension is JavaSscript.  
\* And in theSystemJS.config, we can easily fix this by adding:  
  
\* Now it all works, it’s just some initial setup and it weren’t that many steps.  
\* It was just difficult to understand why we need this Module Loader and that it won’t work without one because the different module formats which evolved over the last years are all not supported by native JavaScript, we need to have a Module Loader.  
\* It’s totally up to you which Module Format you want to use.  
\* **Available / Useable Module Formats**  
You can compile your code against a couple of common module formats. This determines, to which way of JS code splitting your TS code is compiled.  
**The following article gives a great overview and explanation**:  
<https://auth0.com/blog/javascript-module-systems-showdown/>   
**More on TS modules can be learned here**:  
<https://www.typescriptlang.org/docs/handbook/modules.html#code-generation-formodules>

**Importing & Exporting Modules**  
\* **Regarding the styles of imports and exports or how you can use these imports and exports, pretty much all the things you can learn about ES6 imports applies**.  
  


**default**  
  
\* **We can use any name we want for default exports.**

**Module Resolution**  
\* There’s another important thing when talking about modules and module imports.  
\* **The way TypeScript resolves the files**.  
\* As I already said, we don’t set the file extension as TypeScript will figure that out automatically, but actually that is not all that it will figure out automatically - there are 2 other things it will figure out for us.  
**1) We can have different paths of imports.**\* Here we’re setting a relative path:  
  
\* TypeScript is very clever when resolving imports.  
\* It sees if you’re using a relative path or an absolute path.  
\* The absolute path will always be locked up in the node\_modules folder by default or if you have an ambient typings definitions file, something to which I’ll come back in a future module, it will fetch it from there.  
**2) You can set up global exports which kind of are available in your whole project and which can be imported like this**:  
  
\* Or if you want to call it like that, Local Exports which have a clear position like in this file which you import by pointing to that file.  
  
\* So you have these 2 different import syntaxes and you don’t really have to worry about it, TypeScript will figure out everything for you but it’s important to understand what’s happening behind the scenes.

**Namespaces vs Modules - Wrap Up**  


**Module Summary**  
\* You are now really prepared to write bigger TypeScript applications.  
\* You now know how to use Namespaces and Modules and when to use what.  
\* You are prepared to set up your project in a way that you split up your code over multiple files, as well as logical modules, each containing their own code or piece of your application.  
\* You then also learned how to put it all back together and make it run in the browser or in the application.

**Resources**  
The following article gives a great overview and explanation:  
<https://auth0.com/blog/javascript-module-systems-showdown/>   
More on TS modules can be learned here:  
<https://www.typescriptlang.org/docs/handbook/modules.html#code-generation-formodules>