Dependency Injection

Our Goal

- In this lesson we will learn more thoroughly about Angular’s DI
- We will learn about how the DI works

What is DI?

- DI is a way to create objects that depend on other objects
- We supply the DI information about the dependent object (dependencies) and the DI creates the instance
- We can inject our services in different areas of our app
- Advantages of DI:
  - Easier to manage dependencies between objects
  - Easier to manage object caching
  - Improving performance
  - Easier testing and mocking services
  - Code more modular and reusable

How Angular DI Works

Keeping things simple, this is how you use DI in Angular:

- You register your services
  - providers array in modules
  - providers array in components
  - as of ng6 you can supply the register information in the **@Injectable** decorator
- You ask for the services in the constructor
  - by specifying the service type
  - if using DI in services we must decorate the service with **@Injectable**
  - **tsconfig** has to have **emitDecoratorsMetadata**

How Angular DI Works

- In practice for each component with **providers** array in the component metadata, angular creates an **Injector** service (can also be asked in the constructor)
- When you register a service in the providers array you register the service with an **Injector**
- the component injector inherits the providers from the parent **Injector**
- this create a tree hierarchy of injectors
- at the root of the tree there is the **root injector**
- service registered is a singleton with lifespan same as the component

Service Asking for Service

- Unlike the components, a service will ask for other services from the root injector
- You have to decorate a service with the **@Injectable** decorator otherwise the DI can’t get the types of the arguments in the constructor

Providers

- Providers array get's a recipe how to create the service in the following format:
  - **providers: [ClassName]**
  - **providers: [{provide: ClassName, useClass: SomeOtherClass}]**
  - **providers: [{provide: ClassName, useValue: some instance or object}]**
  - **providers: [{provide: ClassName, deps: [Service1, Service2] ,useFactory: SomeOtherClass}]**
  - **providers: [{provide: ClassName, useExisting: OtherClass}]**

- Let's create a service **TaskService** which talks to our server but the service is created at runtime with server url set from environment

Aliasing

- suppose we are creating a new task service: **NewTaskService** with similar api as the previous **TaskService**
- we want to gradually test the new service with new components but the old components that use the old service should remain the same
- we can register our new service

  providers: [NewTaskService]
- after a while we want the **TaskService** to be replaced

  {provide: TaskService, useClass: NewTaskService} // this will create new instance
- after a while we want aliasing and that a new instance won't create and that when we ask for **TaskService** we get **NewTaskService** (aliasing)

  {provide: TaskService, useExisting: NewTaskService}

forwardRef

- We use **forwardRef** if we are looking to inject a class which is not yet defined
- We place it in the **@Inject** decorator
- **forwardRef** gets a function which returns the class and will call this function after the class is defined

InjectionToken

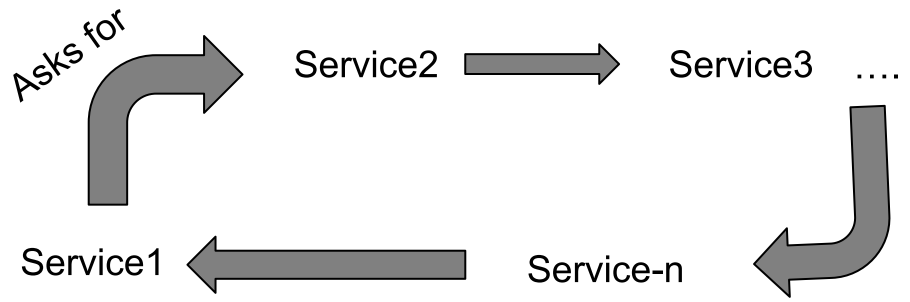
- the **Injector** can map the service we want based on the class type
- but do we only need to inject classes?
- what if we want to inject a primitive or an object or an array
- for example what if we want to inject the server url from our environments file
- we can use **InjectionToken** to create a token
- **export const ServerUrl: InjectionToken<string> = new InjectionToken('token');**
- we can then register in the providers array

  providers: [{provide: ServerUrl, useValue: environment.serverUrl}]
- we can inject in the constructor but we need to use the **@Inject** decorator

  constructor(@Inject(ServerUrl) private serverUrl: string)

Cyclic dependency

- Cyclic dependency happens when:



Cyclic dependency

- let's try to simulate Cyclic dependency and try to solve the problem
- we will create the following services

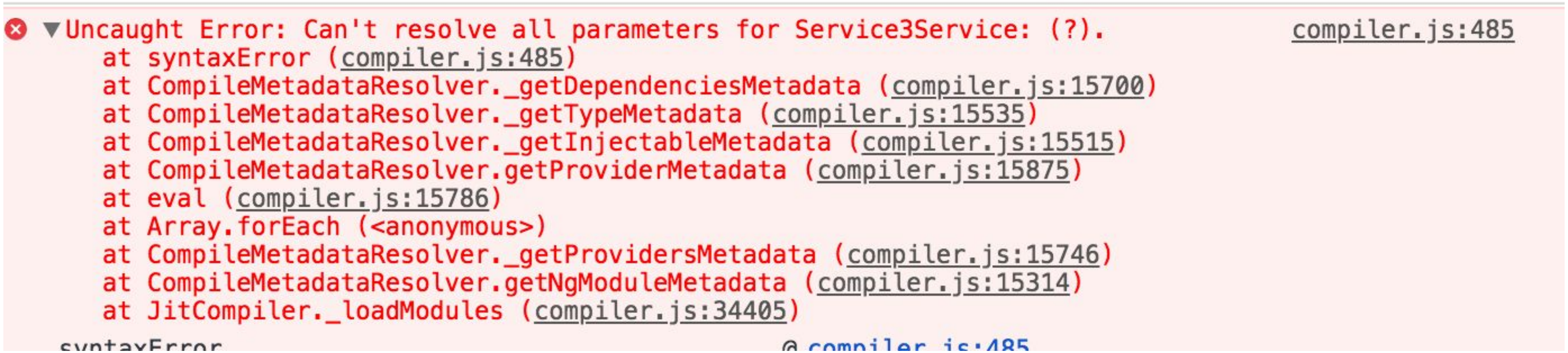
Service1 -> Service2 -> Service3 -> Service1

```
@Injectable()
export class Service1Service {
 constructor(private _service2: Service2Service) { }
 public stam1 () {
   console.log('stam1');
   this._service2.stam2();
 }
}
```

```
@Injectable()
export class Service2Service {
 constructor(private _service3: Service3Service) { }
 public stam2() {
   console.log('stam2');
   this._service3.stam3();
 }
}
```

```
@Injectable()
export class Service3Service {
 constructor(private _service1: Service1Service) { }
 public stam3() {
   console.log('stam3');
 }
 public justToUserService1() {
   this._service1.stam1();
 }
}
```

Cyclic Dependency



```
Uncaught Error: Can't resolve all parameters for Service3Service: (?).          compiler.js:485
    at syntaxError (compiler.js:485)
    at CompileMetadataResolver._getDependenciesMetadata (compiler.js:15700)
    at CompileMetadataResolver._getTypeMetadata (compiler.js:15535)
    at CompileMetadataResolver._getInjectableMetadata (compiler.js:15515)
    at CompileMetadataResolver.getProviderMetadata (compiler.js:15875)
    at eval (compiler.js:15786)
    at Array.forEach (<anonymous>)
    at CompileMetadataResolver._getProvidersMetadata (compiler.js:15746)
    at CompileMetadataResolver.getNgModuleMetadata (compiler.js:15314)
    at JitCompiler._loadModules (compiler.js:34405)
  syntaxError                                     @ compiler.js:485
```

How can we solve the problem?

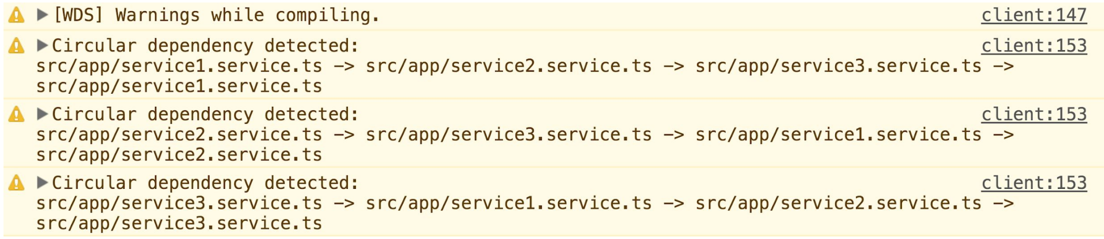
- one way to solve the problem is arrange our code differently by creating for example:

$$\begin{tikzcd}
 & \text{Service1} & \\
\text{Service2} \arrow[ur] & & \text{Service3} \arrow[ul]
\end{tikzcd}$$

How can we solve the problem?

- another way we can solve it is injecting the Injector in service3 and manually injecting Service1 when needed

```
@Injectable()
export class Service3Service {
 constructor(private _injector: Injector) { }
 public stam3() {
   console.log('stam3');
 }
 public justToUserService1() {
   this._injector.get(Service1Service).stam1();
 }
}
```



```
▶[WDS] Warnings while compiling.                                          client:147
▶Circular dependency detected:                                            client:153
src/app/service1.service.ts -> src/app/service2.service.ts -> src/app/service3.service.ts ->
src/app/service1.service.ts
▶Circular dependency detected:                                            client:153
src/app/service2.service.ts -> src/app/service3.service.ts -> src/app/service1.service.ts ->
src/app/service2.service.ts
▶Circular dependency detected:                                            client:153
src/app/service3.service.ts -> src/app/service1.service.ts -> src/app/service2.service.ts ->
src/app/service3.service.ts
```

- you still have a problem and it's not a recommended solution but your code will run

Seperation of concerns

- In practice the way to solve circular dependency is by following `Separation of Concerns (SoP)`
- SoP means separating a program into distinct sections
- each section addresses a separate concern
- a concern is a set of information that affects the code of a computer program
- it helps thinking of services from bottom up
- at the bottom level we have angular services
- at the level above we create services that can only inject angular services but not our services
- at the level above that we can use the services at the bottom level only and so forth
- Keeping this design principle in mind and thinking bottom up regarding our services will help you avoid circular dependencies

Summary

- Understand how the DI works will help us get the benefits from using this patters
- We can now understand where to register our services
- We can now use the knowledge to split our modules to **SharedModule** which can help us provide common services to all our modules