Monday: Dependency Injections

Dependency Injection

Dependency injection is a design pattern in which a class asks for dependencies from external sources instead of creating them itself. The reason we use dependency injection is because it separates the creation of an object from its usage, which enables us to replace dependencies without changing the class that uses them. This means that a class concentrates on fulfilling its responsibilities instead of creating the objects that help it fulfill those responsibilities.

What does this really mean? Let's do it practically. Let us create a folder outside our Goals project anywhere on your laptop and call it Car. Inside this folder, let's create 4 files, car.ts, engine.ts, wheels.ts and main.ts. We're going to familiarize ourselves with DI(Dependency Injection) with the concept of a car.

Let's write code for a class that allows us to create an instance of a car. This class would look like this:

Car/car.ts

```
import { Engine } from './engine';
import { Wheels } from './wheels';

export class Car{
  engine: Engine;
  wheels: Wheels;

  constructor(){
    this.engine = new Engine();
    this.wheels = new Wheels();
  }

  startEngine(){
    this.engine.start();
  }
}
```

At the top, we have imported the objects/dependencies we need to create a Car. Inside the car class, we have created properties engine and wheels and assigned them their types respectively. In the constructor function, we have created instances of these dependencies using the new keyword. We have then created a method startEngine() which calls another function in our engine dependency. Let's create this function in engine dependency.

Car/engine.ts

```
export class Engine{

  start(){
    console.log("Vroooooooom!");
  }
}
```

We have created a start function which simply logs a string on our console. We will not be using our wheels but let's create an empty class and assume that it's also a dependency that has its own properties and methods just like our engine.

Car/wheels.ts

```
export class Wheels{

}
```

We'll use our main.ts file to run this car app. Let's write the code to run this app in this file.

Car/main.ts

```
import { Car } from './car';
import { Engine } from './engine';
import { Wheels } from './wheels';

function main(){
  let car = new Car();

  car.startEngine();
}

main();
```

At the top, we have imported our car, engine, and wheels and then created a function main(). We have created the car instance and called the startEngine() method in the function and at the bottom, we have called our main() function so it can execute this code.

Since we have node installed, we can run this code on our terminal. Let's transpile this typescript code to javascript code using the typescript transpiler on our terminal:

```
$ tsc main.ts
```

Now let's run the transpiled javascript code directly on our terminal with node. If you recall, we installed node so we could execute javascript code directly from our terminal instead of using the browser console, so let's go ahead and do it:

```
$ node main.js
```

This command creates our car and starts the engine for us which is great, we can see the output in the terminal. If we, however, wanted to use this car in another environment that has different dependencies, it would be difficult because the car creates the dependencies that it needs for itself. For example, in the state our car class is in right now, if we want to add doors to the car, we would have to add the code to make it define doors and still define this property in the constructor function. The same would apply if we wanted to change anything that exists in the car already. Point is, our code will be hard to maintain and scale, which may also end up making it messy and prone to errors.

That's where Dependency Injection comes in handy. It relieves our Car class the responsibility of creating the dependencies it needs and makes it just consume these dependencies from external sources.

Let's implement this in our car.

Car/car.ts

```
import { Engine } from './engine';
import { Wheels } from './wheels';

export class Car{
  constructor(private engine: Engine,private wheels: Wheels){
  }

  startEngine(){
    this.engine.start();
  }
}
```

At the top, we still have the imports for our dependencies. Our constructor function no longer builds the dependencies the car class needs. This means that the car class consumes its dependencies, the engine and wheels class. We have used the private keyword to make these properties available only inside the class.

Inside the constructor function, we still call our startEngine() method. When we refactor our code like this, the Car class does not know how to create its dependencies which is amazing because we can now alter the dependencies that the class consumes more easily now. We now need to change the code in our main.ts file so we can fire up the engines again.

Car/main.ts

```
import { Car } from './car';
import { Engine } from './engine';
import { Wheels } from './wheels';

function main(){

  let engine = new Engine();
  let wheels = new Wheels()
  let car = new Car(engine,wheels);

  car.startEngine();
}

main();
```

At the top, we have still imported our classes. Inside the main function, we are now creating instances of engine, wheels, and car. When we create the car instance now, we specify that it should consume the engine and wheels dependencies that we have just instantiated. We end up by starting our engine and calling our main function.

Let's transpile this code in our terminal again to see the effect it has:

```
$ tsc main.ts
```

Now let's run our JS code on our terminal using node:

```
$ node main.js
```

Amazing, our engines still fire up! This is how Dependency Injection works. The car class does not create the dependencies it needs, it consumes them in the constructor. Now it is the *main.ts* file that creates these dependencies and also creates an instance of the car, so the car can now consume these dependencies that have been created outside it and still work as expected. This is impactful because we can create a custom engine for each car rather than having all cars create the same default engine in their constructor.

Angular has its own inbuilt framework for dependency injection which we will use and that is how it works. This DI framework enables us to create an **injector** with which we can register some classes and it figures out how to create these dependencies. Our job would be to ask the injector for the created dependencies. Dependency injection is good practice in so many languages and frameworks, not only in Angular. If you need more resources to understand the concept of dependency injection, feel free to look for them online, and also take a look at this