The Dependency Injection pattern

Dependency injection is an important application design pattern. It's used so widely that almost everyone just calls it *DI*.

Angular has its own dependency injection framework, and you really can't build an Angular application without it.

This page covers what DI is and why it's useful.

When you've learned the general pattern, you're ready to turn to the Angular Dependency Injection guide to see how it works in an Angular app.

{@a why-di }

Why dependency injection?

To understand why dependency injection is so important, consider an example without it. Imagine writing the following code:

The `Car` class creates everything it needs inside its constructor. What's the problem? The problem is that the `Car` class is brittle, inflexible, and hard to test.

This `Car` needs an engine and tires. Instead of asking for them, the `Car` constructor instantiates its own copies from the very specific classes `Engine` and `Tires`.

What if the `Engine` class evolves and its constructor requires a parameter? That would break the `Car` class and it would stay broken until you rewrote it along the lines of `this.engine = new Engine(theNewParameter)`. The `Engine` constructor parameters weren't even a consideration when you first wrote `Car`. You may not anticipate them even now. But you'll *have* to start caring because when the definition of `Engine` changes, the `Car` class must change. That makes `Car` brittle.

What if you want to put a different brand of tires on your `Car`? Too bad. You're locked into whatever brand the `Tires` class creates. That makes the `Car` class inflexible.

Right now each new car gets its own `engine`. It can't share an `engine` with other cars. While that makes sense for an automobile engine, surely you can think of other dependencies that should be shared, such as the onboard wireless connection to the manufacturer's service center. This `Car` lacks the flexibility to share

services that have been created previously for other consumers.

When you write tests for `Car` you're at the mercy of its hidden dependencies. Is it even possible to create a new `Engine` in a test environment? What does `Engine` depend upon? What does that dependency depend on? Will a new instance of `Engine` make an asynchronous call to the server? You certainly don't want that going on during tests.

What if the `Car` should flash a warning signal when tire pressure is low? How do you confirm that it actually does flash a warning if you can't swap in low-pressure tires during the test?

You have no control over the car's hidden dependencies. When you can't control the dependencies, a class becomes difficult to test.

How can you make `Car` more robust, flexible, and testable?

{@a ctor-injection} That's super easy. Change the `Car` constructor to a version with DI:

See what happened? The definition of the dependencies are now in the constructor. The `Car` class no longer creates an `engine` or `tires`. It just consumes them.

This example leverages TypeScript's constructor syntax for declaring parameters and properties simultaneously.

Now you can create a car by passing the engine and tires to the constructor.

How cool is that? The definition of the `engine` and `tire` dependencies are decoupled from the `Car` class. You can pass in any kind of `engine` or `tires` you like, as long as they conform to the general API requirements of an `engine` or `tires`.

Now, if someone extends the `Engine` class, that is not `Car`'s problem.

The _consumer_ of `Car` has the problem. The consumer must update the car creation code to something like this: The critical point is this: the `Car` class did not have to change. You'll take care of the consumer's problem shortly.

The `Car` class is much easier to test now because you are in complete control of its dependencies. You can pass mocks to the constructor that do exactly what you want them to do during each test:

You just learned what dependency injection is.

It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Cool! But what about that poor consumer? Anyone who wants a `Car` must now create all three parts: the

`Car`, `Engine`, and `Tires`. The `Car` class shed its problems at the consumer's expense. You need something that takes care of assembling these parts.

You *could* write a giant class to do that:

It's not so bad now with only three creation methods. But maintaining it will be hairy as the application grows. This factory is going to become a huge spiderweb of interdependent factory methods!

Wouldn't it be nice if you could simply list the things you want to build without having to define which dependency gets injected into what?

This is where the dependency injection framework comes into play. Imagine the framework had something called an *injector*. You register some classes with this injector, and it figures out how to create them.

When you need a `Car`, you simply ask the injector to get it for you and you're good to go.

Everyone wins. The `Car` knows nothing about creating an `Engine` or `Tires`. The consumer knows nothing about creating a `Car`. You don't have a gigantic factory class to maintain. Both `Car` and consumer simply ask for what they need and the injector delivers.

This is what a **dependency injection framework** is all about.

Now that you know what dependency injection is and appreciate its benefits, turn to the Angular Dependency Injection guide to see how it is implemented in Angular.