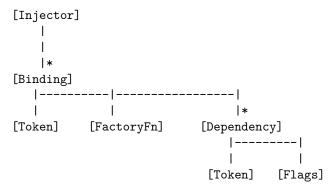
# Dependency Injection (DI): Documentation

This document describes in detail how the DI module works in Angular.

### Core Abstractions

The library is built on top of the following core abstractions: Injector, Binding, and Dependency.

- An injector is created from a set of bindings.
- An injector resolves dependencies and creates objects.
- A binding maps a token, such as a string or class, to a factory function and a list of dependencies. So a binding defines how to create an object.
- A dependency points to a token and contains extra information on how the object corresponding to that token should be injected.



### Example

```
class Engine {
}

class Car {
   constructor(@Inject(Engine) engine) {
   }
}

var inj = Injector.resolveAndCreate([
   bind(Car).toClass(Car),
   bind(Engine).toClass(Engine)
]);

var car = inj.get(Car);
```

In this example we create two bindings: one for Car and one for Engine. @Inject(Engine) declares a dependency on Engine.

# Injector

An injector instantiates objects lazily, only when asked for, and then caches them.

```
Compare
```

```
var car = inj.get(Car); //instantiates both an Engine and a Car
with
var engine = inj.get(Engine); //instantiates an Engine
var car = inj.get(Car); //instantiates a Car (reuses Engine)
and with
var car = inj.get(Car); //instantiates both an Engine and a Car
var engine = inj.get(Engine); //reads the Engine from the cache
```

To avoid bugs make sure the registered objects have side-effect-free constructors. In this case, an injector acts like a hash map, where the order in which the objects got created does not matter.

# Child Injectors and Dependencies

Injectors are hierarchical.

```
var parent = Injector.resolveAndCreate([
 bind(Engine).toClass(TurboEngine)
]);
var child = parent.resolveAndCreateChild([Car]);
var car = child.get(Car); // uses the Car binding from the child injector and Engine from the
Injectors form a tree.
  GrandParentInjector
Parent1Injector Parent2Injector
ChildInjector
The dependency resolution algorithm works as follows:
// this is pseudocode.
var inj = this;
while (inj) {
 if (inj.hasKey(requestedKey)) {
    return inj.get(requestedKey);
  } else {
    inj = inj.parent;
```

```
}
throw new NoProviderError(requestedKey);
So in the following example
class Car {
  constructor(e: Engine){}
}
```

DI will start resolving Engine in the same injector where the Car binding is defined. It will check whether that injector has the Engine binding. If it is the case, it will return that instance. If not, the injector will ask its parent whether it has an instance of Engine. The process continues until either an instance of Engine has been found, or we have reached the root of the injector tree.

#### Constraints

You can put upper and lower bound constraints on a dependency. For instance, the @Self decorator tells DI to look for Engine only in the same injector where Car is defined. So it will not walk up the tree.

```
class Car {
  constructor(@Self() e: Engine){}
}
```

A more realistic example is having two bindings that have to be provided together (e.g., NgModel and NgRequiredValidator.)

The @Host decorator tells DI to look for Engine in this injector, its parent, until it reaches a host (see the section on hosts.)

```
class Car {
  constructor(@Host() e: Engine){}
}
```

The @SkipSelf decorator tells DI to look for Engine in the whole tree starting from the parent injector.

```
class Car {
  constructor(@SkipSelf() e: Engine){}
}
```

### DI Does Not Walk Down

Dependency resolution only walks up the tree. The following will throw because DI will look for an instance of Engine starting from parent.

```
var parent = Injector.resolveAndCreate([Car]);
var child = parent.resolveAndCreateChild([
   bind(Engine).toClass(TurboEngine)
]);
```

```
parent.get(Car); // will throw NoProviderError
```

# **Bindings**

You can bind to a class, a value, or a factory. It is also possible to alias existing bindings.

```
var inj = Injector.resolveAndCreate([
 bind(Car).toClass(Car),
 bind(Engine).toClass(Engine)
]);
var inj = Injector.resolveAndCreate([
 Car, // syntax sugar for bind(Car).toClass(Car)
 Engine
]);
var inj = Injector.resolveAndCreate([
 bind(Car).toValue(new Car(new Engine()))
]);
var inj = Injector.resolveAndCreate([
  bind(Car).toFactory((e) => new Car(e), [Engine]),
 bind(Engine).toFactory(() => new Engine())
]);
You can bind any token.
var inj = Injector.resolveAndCreate([
 bind(Car).toFactory((e) => new Car(), ["engine!"]),
 bind("engine!").toClass(Engine)
]);
If you want to alias an existing binding, you can do so using toAlias:
var inj = Injector.resolveAndCreate([
 bind(Engine).toClass(Engine),
 bind("engine!").toAlias(Engine)
]);
which implies inj.get(Engine) === inj.get("engine!").
Note that tokens and factory functions are decoupled.
bind("some token").toFactory(someFactory);
The someFactory function does not have to know that it creates an object for
```

### Resolved Bindings

When DI receives bind(Car).toClass(Car), it needs to do a few things before it can create an instance of Car:

- It needs to reflect on Car to create a factory function.
- It needs to normalize the dependencies (e.g., calculate lower and upper bounds).

The result of these two operations is a ResolvedBinding.

The resolveAndCreate and resolveAndCreateChild functions resolve passed-in bindings before creating an injector. But you can resolve bindings yourself using Injector.resolve([bind(Car).toClass(Car)]). Creating an injector from pre-resolved bindings is faster, and may be needed for performance sensitive areas.

You can create an injector using a list of resolved bindings.

```
var listOfResolvingProviders = Injector.resolve([Provider11, Provider2]);
var inj = Injector.fromResolvedProviders(listOfResolvingProviders);
inj.createChildFromResolvedProviders(listOfResolvedProviders);
```

### Transient Dependencies

An injector has only one instance created by each registered binding.

```
inj.get(MyClass) === inj.get(MyClass); //always holds
```

If we need a transient dependency, something that we want a new instance of every single time, we have two options.

We can create a child injector for each new instance:

var child = inj.resolveAndCreateChild([MyClass]);

```
child.get(MyClass);
Or we can register a factory function:
var inj = Injector.resolveAndCreate([
   bind('MyClassFactory').toFactory(dep => () => new MyClass(dep), [SomeDependency])
]);
var factory = inj.get('MyClassFactory');
var instance1 = factory(), instance2 = factory();
// Depends on the implementation of MyClass, but generally holds.
expect(instance1).not.toBe(instance2);
```