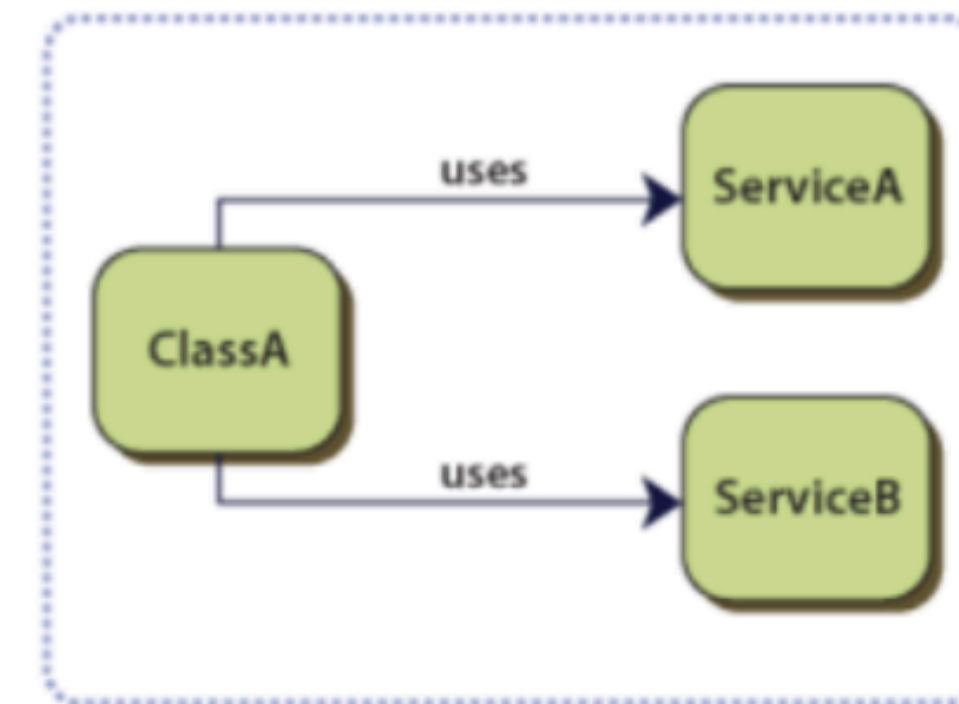


# Imports, Decorators & DI

---

## Imports, Decorators & DI



Javascript/Typescript  
Decorators & Dependency  
Injection language features  
that are central to Aurelia.

# Agenda

- Imports
- Decorators
- Dependency Injection

# Imports

- The import statement is used to import functions, objects or primitives that have been exported from an external module

```
import defaultMember from "module-name";  
import * as name from "module-name";  
import { member } from "module-name";  
import { member as alias } from "module-name";  
import { member1 , member2 } from "module-name";  
import { member1 , member2 as alias2 , [...] } from "module-name";  
import defaultMember, { member [ , [...] ] } from "module-name";  
import defaultMember, * as name from "module-name";  
import "module-name";
```

```
import defaultMember from "module-name";  
import * as name from "module-name";  
import { member } from "module-name";  
import { member as alias } from "module-name";  
import { member1 , member2 } from "module-name";  
import { member1 , member2 as alias2 , [...] } from "module-name";  
import defaultMember, { member [ , [...] ] } from "module-name";  
import defaultMember, * as name from "module-name";  
import "module-name";
```

**name**

Name of the object that will receive the imported values.

**member, memberN**

Name of the exported members to be imported.

**defaultMember**

Name of the object that will receive the default export from the module.

**alias, aliasN**

Name of the object that will receive the imported property

**module-name**

The name of the module to import. This is a file name.



## Examples (1)

Import an entire module's contents. This inserts `myModule` into the current scope, containing all the exported bindings from "my-module.js".

```
1 | import * as myModule from "my-module";
```

Import a single member of a module. This inserts `myMember` into the current scope.

```
1 | import {myMember} from "my-module";
```

Import multiple members of a module. This inserts both `foo` and `bar` into the current scope.

```
1 | import {foo, bar} from "my-module";
```

## Examples (2)

Import a member with a more convenient alias. This inserts `shortName` into the current scope.

```
1 | import {reallyReallyLongModuleMemberName as shortName} from "my-module";
```

Import an entire module for side effects only, without importing any bindings.

```
1 | import "my-module";
```

Import multiple members of a module with convenient aliases.

```
1 | import {reallyReallyLongModuleMemberName as shortName, anotherLongModuleName as short} from "my-module";
```

## Complete Example

```
// --file.js--  
functiongetJSON(url, callback) {  
  let xhr = new XMLHttpRequest();  
  xhr.onload = function () {  
    callback(this.responseText)  
  };  
  xhr.open("GET", url, true);  
  xhr.send();  
}  
  
export functiongetUsefulContents(url, callback) {  
  getJSON(url, data => callback(JSON.parse(data)));  
}  
  
// --main.js--  
import { getUsefulContents } from "file";  
getUsefulContents("http://www.example.com", data => {  
  doSomethingUseful(data);  
});
```

# Decorators

- Similar to Java Annotations
- Make it possible to annotate and modify classes and properties at design time.
- Libraries can compose their own decorators

```
class Person {  
    @readonly  
    name() { return `${this.first} ${this.last}` }  
}
```



# Method & Class Decorators

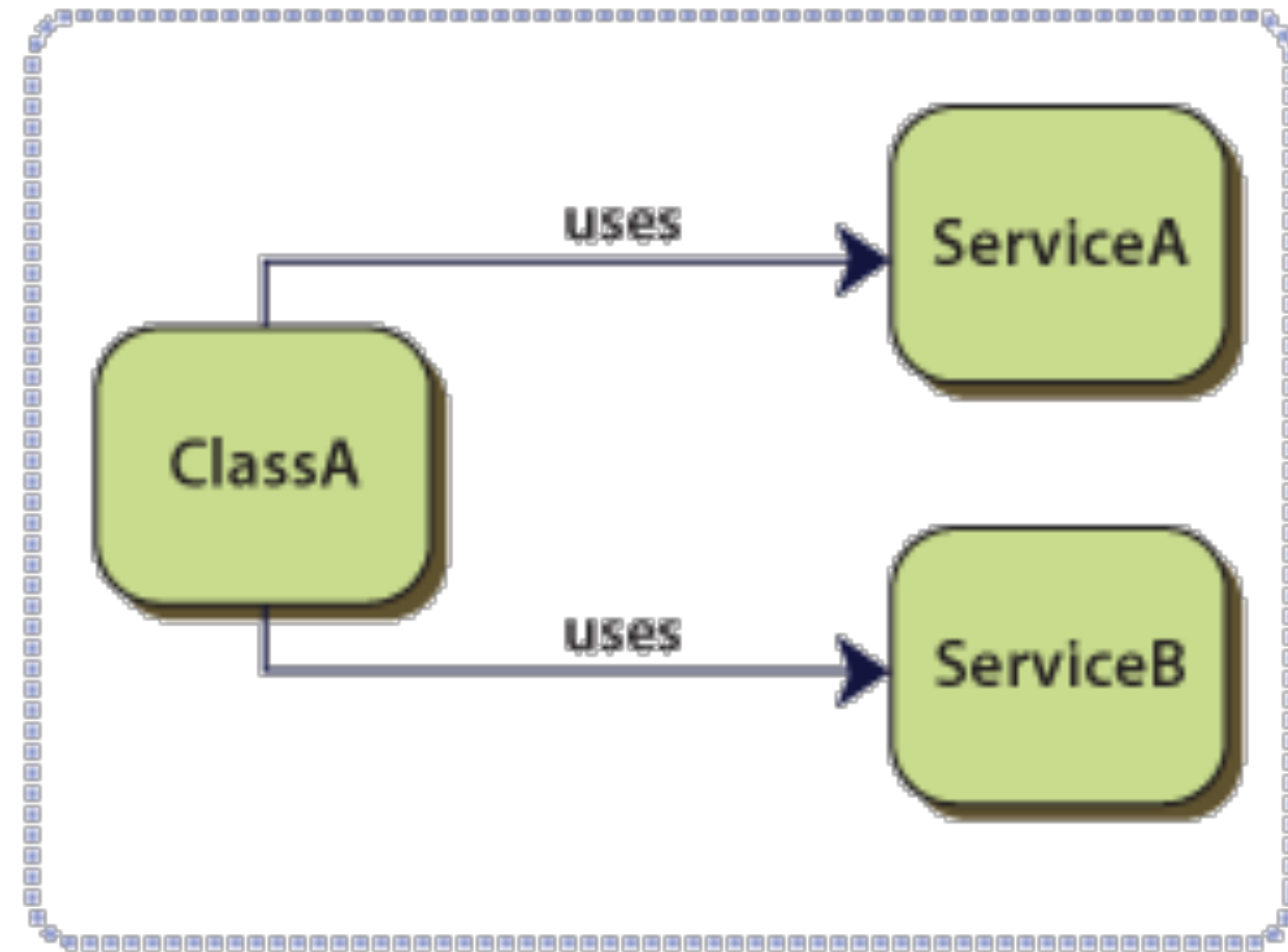
- Can be applied to classes and functions.
- Broad range of potential uses
- Provide a mechanism for providing meta-data about a class or function

```
class Person {  
    @readonly  
    name() { return `${this.first} ${this.last}` }  
}
```

```
@isTestable(true)  
class MyClass { }  
  
function isTestable(value) {  
    return function decorator(target) {  
        target.isTestable = value;  
    }  
}
```

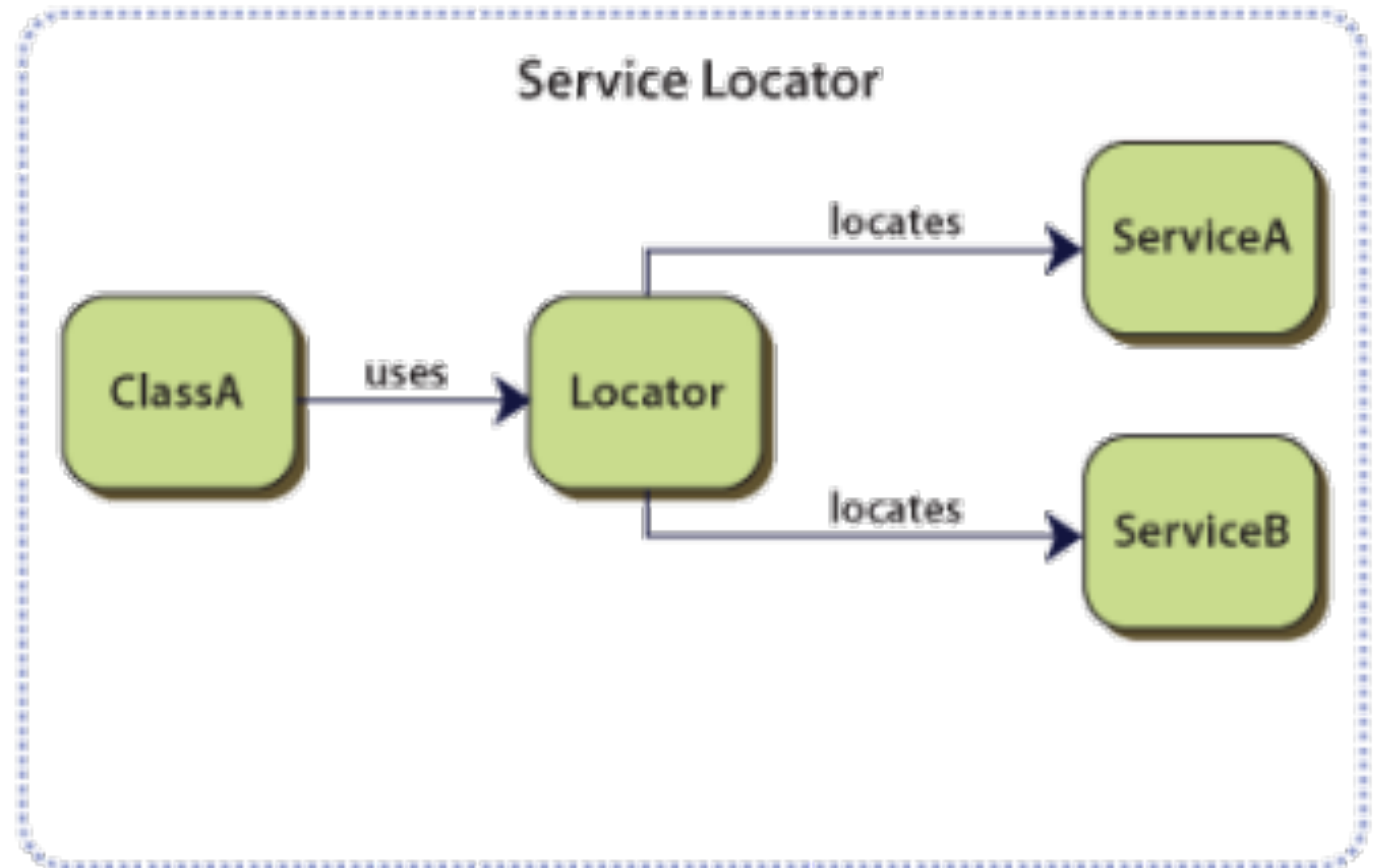
# Dependency Injection (DI)

- When building applications, it's often necessary to take a "divide and conquer" approach by breaking down complex problems into a series of simpler problems.
- This translates to breaking down complex objects into a series of smaller objects, each focusing on a single concern, and collaborating with the others to form a complex system.



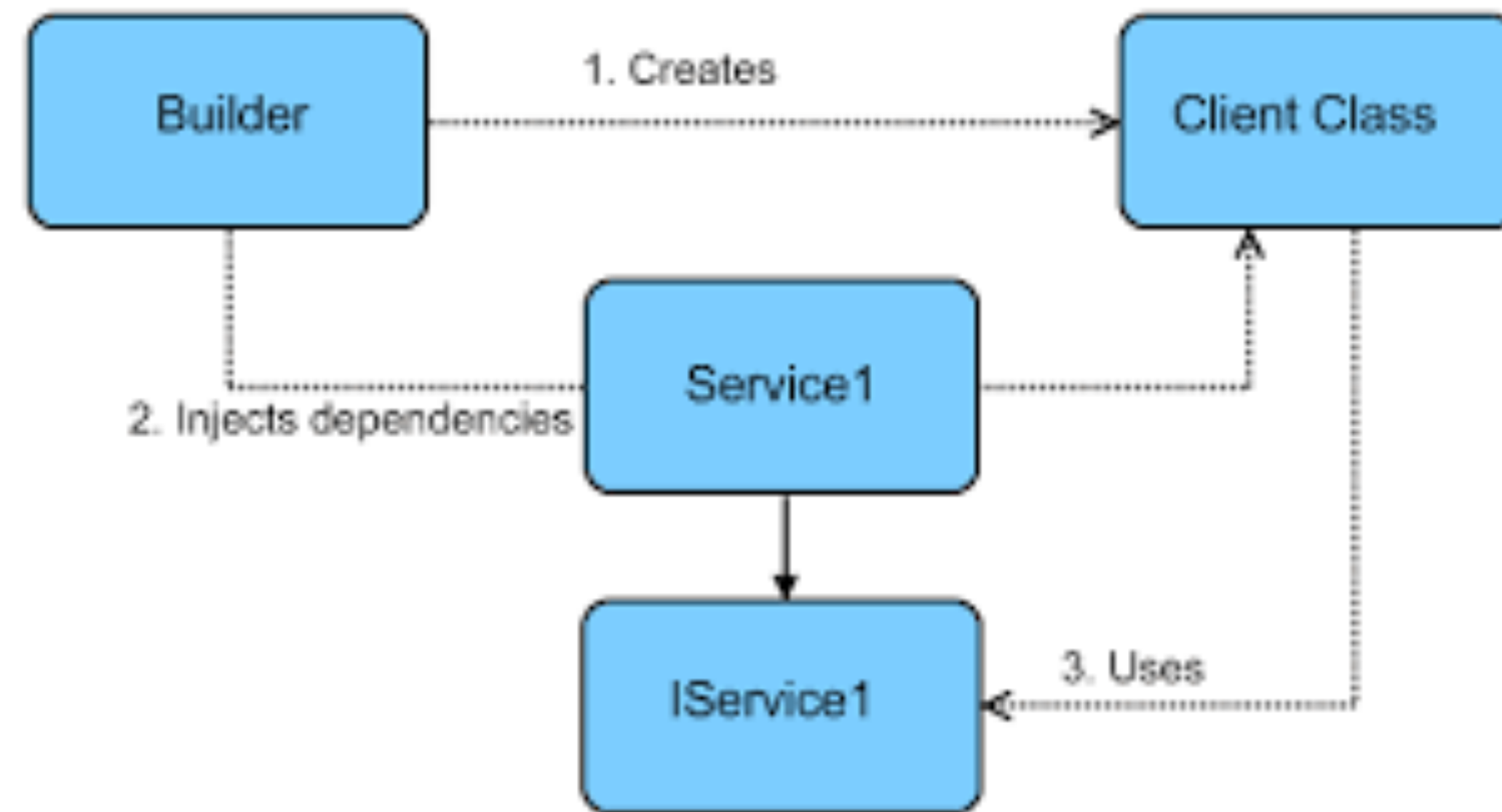
# Dependency Injection (DI)

- The work of destructuring a system can introduce a new complexity of "re-assembling" the smaller parts again at runtime.
- This is what a dependency injection aims to simplify - using simple declarative hints.



## Benefits of DI

- Dependency injection separates the creation of a client's dependencies from the client's behaviour, which allows program designs to be loosely coupled



[https://en.wikipedia.org/wiki/Dependency\\_injection](https://en.wikipedia.org/wiki/Dependency_injection)



# DI Example

- A CustomerEditScreen needs to load a Customer entity by ID from a web service.
- We wouldn't want to place all the details of our AJAX implementation inside our CustomerEditScreen class.
- Instead, we would want to factor that into a CustomerService class that our CustomerEditScreen, or any other class, can use when it needs to load a Customer.

```
import {CustomerService} from 'backend/customer-service';
import {inject} from 'aurelia-framework';


@inject(CustomerService)
export class CustomerEditScreen {
  constructor(customerService) {
    this.customerService = customerService;
    this.customer = null;
  }

  activate(params) {
    return this.customerService.getCustomerById(params.customerId)
      .then(customer => this.customer = customer);
  }
}
```

# DI Example

```
import {CustomerService} from 'backend/customer-service';
import {inject} from 'aurelia-framework';

@Inject(CustomerService)
export class CustomerEditScreen {
  constructor(customerService) {
    this.customerService = customerService;
    this.customer = null;
  }
}
```



- The inject decorator and the constructor signature match.

- This tells the DI that any time it wants to create an instance of CustomerEditScreen it must first obtain an instance of CustomerService which it can inject into the constructor of CustomerEditScreen during instantiation

# Lifetime of Injected Objects

## Singleton

- A singleton class, A, is instantiated when it is first needed by the DI container.
- The container then holds a reference to class A's instance so that even if no other objects reference it, the container will keep it in memory.
- When any other class needs to inject A, the container will return the exact same instance. Thus, the instance of A has its lifetime connected to the container instance.

## Transient

- These instances are created each time they are needed. The container holds no references to them and always creates a new instance for each request.