2019 MAR 29, SHENZHEN

BETTER CODING

OVERVIEW

- Paradigm
    - Procedure Oriented
    - Object Oriented
    - Functional Programming
- Design Pattern
- Coding Style

NO MATTER FRONT-END / BACK-END

- Great minds think alike
- Great patterns work alike

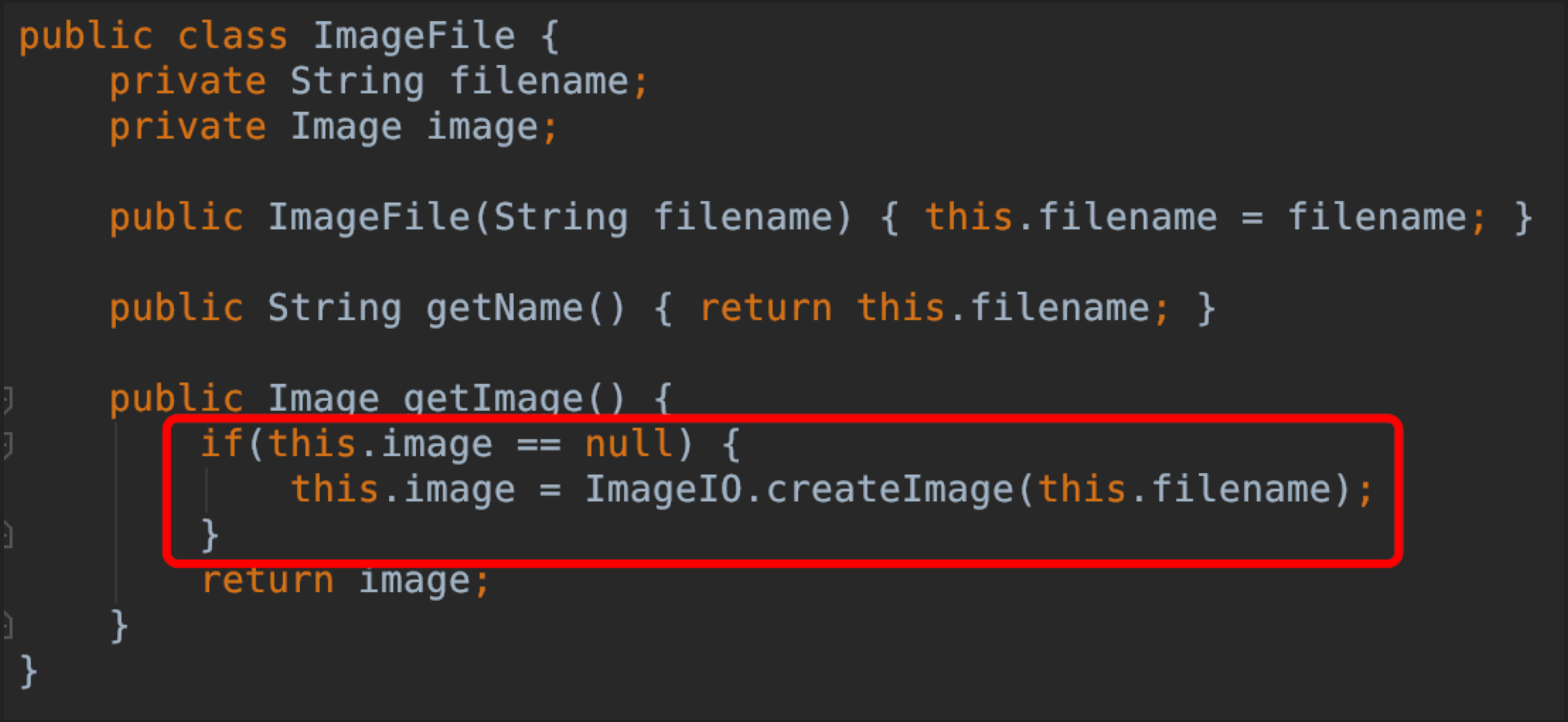
LAZY

- Scaffolding
- Lazy Evaluation
- Copy On Write
- Lazy Rendering
- Memoization

SCAFFOLDING

- Use code to generate code
  (boilerplate - whole initial project, specific logic part …)
  - Backend:
    /_sys/api
  - Frontend:
    yarn api, yarn module, yarn icon
  - Other people do:
    create-react-app, spring initialzr, CRUD generator (gii)

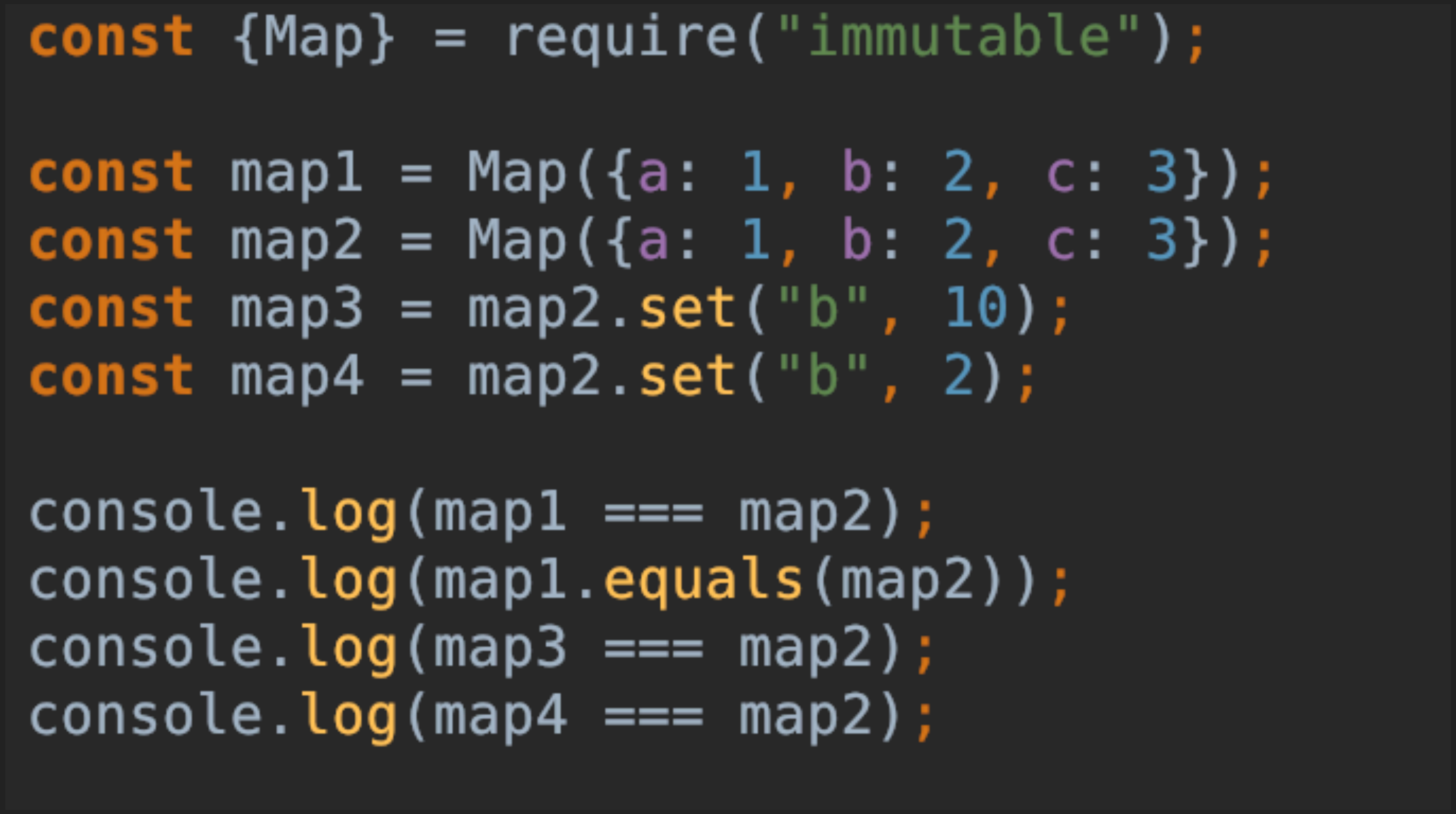
LAZY EVALUATION (JAVA)



```java
public class ImageFile {
    private String filename;
    private Image image;

    public ImageFile(String filename) { this.filename = filename; }

    public String getName() { return this.filename; }

    public Image getImage() {
        if(this.image == null) {
            this.image = ImageIO.createImage(this.filename);
        }
        return image;
    }
}
```

COPY ON WRITE (JAVASCRIPT)

- Immutable JS
  https://github.com/immutable-js/immutable-js

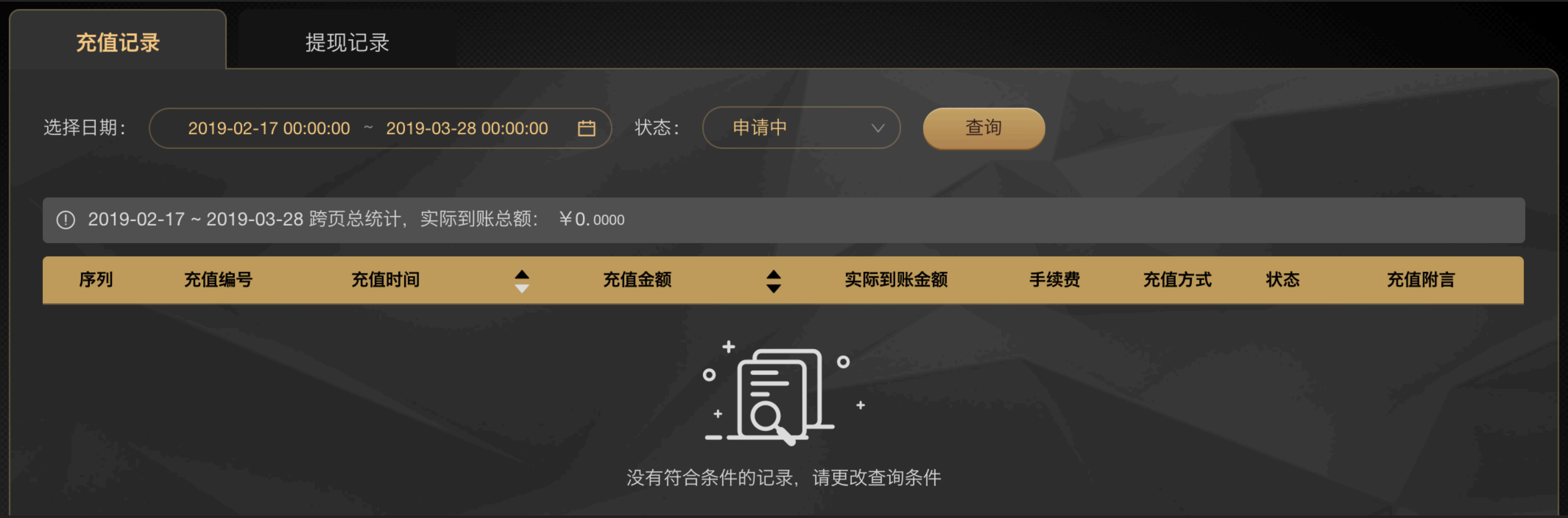


```
const {Map} = require("immutable");

const map1 = Map({a: 1, b: 2, c: 3});
const map2 = Map({a: 1, b: 2, c: 3});
const map3 = map2.set("b", 10);
const map4 = map2.set("b", 2);

console.log(map1 === map2);
console.log(map1.equals(map2));
console.log(map3 === map2);
console.log(map4 === map2);
```

PATTERN 1 - LAZY

LAZY RENDERING

- Implement <Tabs>
  - Only render the tab that is visible
  - Render all tabs (Render others as ***invisible*** nodes)



选择日期： 2019-02-17 00:00:00 ~ 2019-03-28 00:00:00

状态： 申请中

2019-02-17 ~ 2019-03-28 跨页总统计，实际到账总额： ￥0.0000

序列	充值编号	充值时间	充值金额	实际到账金额	手续费	充值方式	状态	充值附言

没有符合条件的记录，请更改查询条件

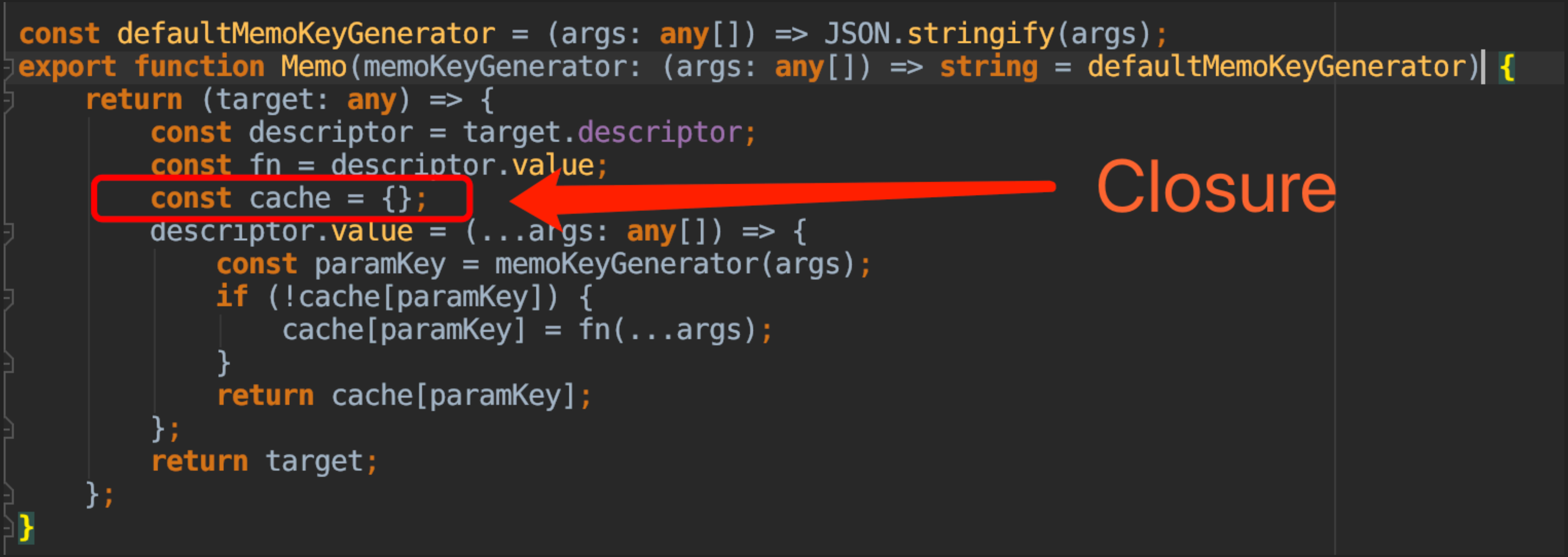
MEMOIZATION

- An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

- **Pure Function (Stateless)**
  - Same input always returns same output
  - No side effect

PATTERN 1 - LAZY

MEMOIZATION



```
const defaultMemoKeyGenerator = (args: any[]) => JSON.stringify(args);
export function Memo(memoKeyGenerator: (args: any[]) => string = defaultMemoKeyGenerator) {
    return (target: any) => {
        const descriptor = target.descriptor;
        const fn = descriptor.value;
        const cache = {};
        descriptor.value = (...args: any[]) => {
            const paramKey = memoKeyGenerator(args);
            if (!cache[paramKey]) {
                cache[paramKey] = fn(...args);
            }
            return cache[paramKey];
        };
        return target;
    };
}
```

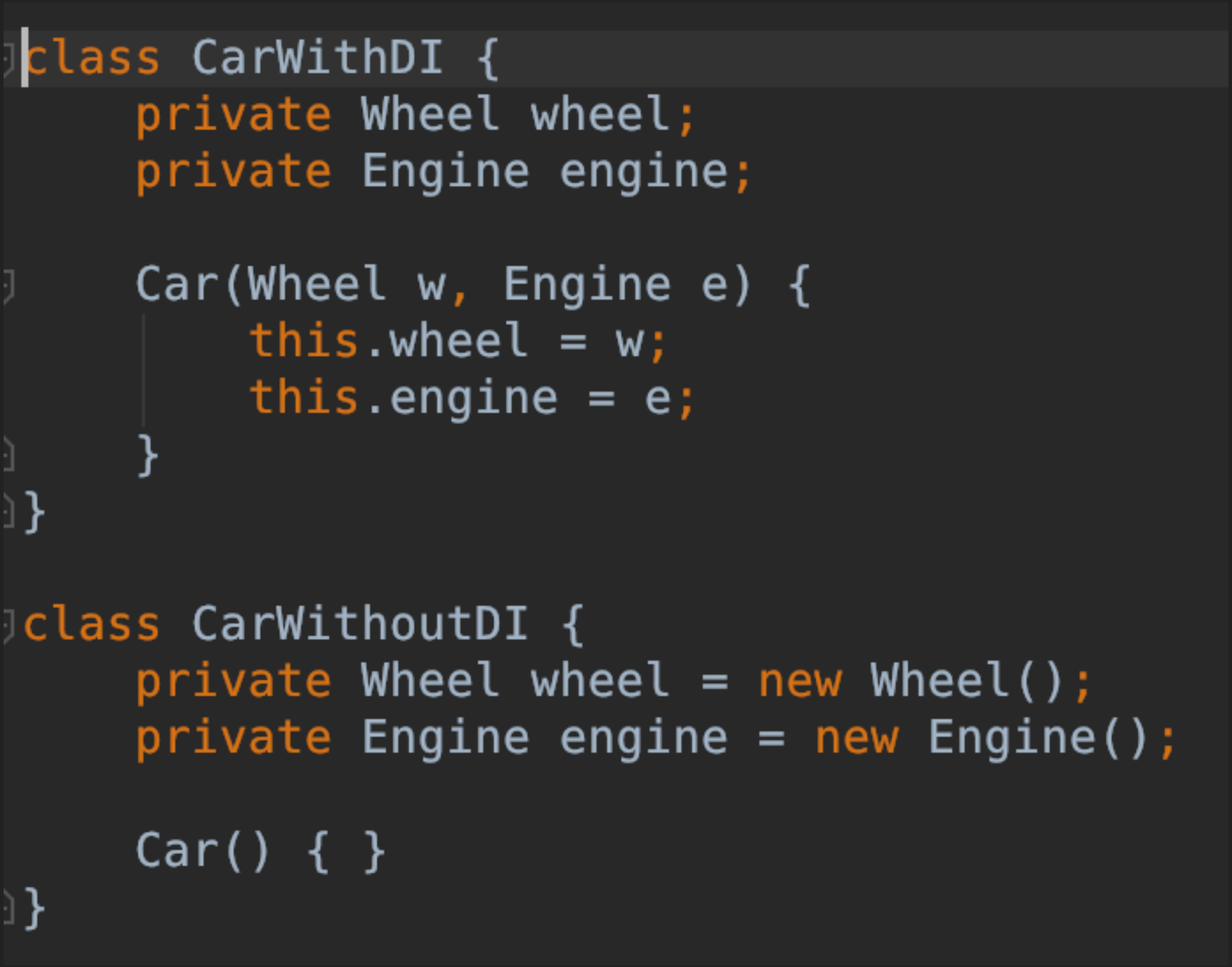
Closure

DEPENDENCY INJECTION

A technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

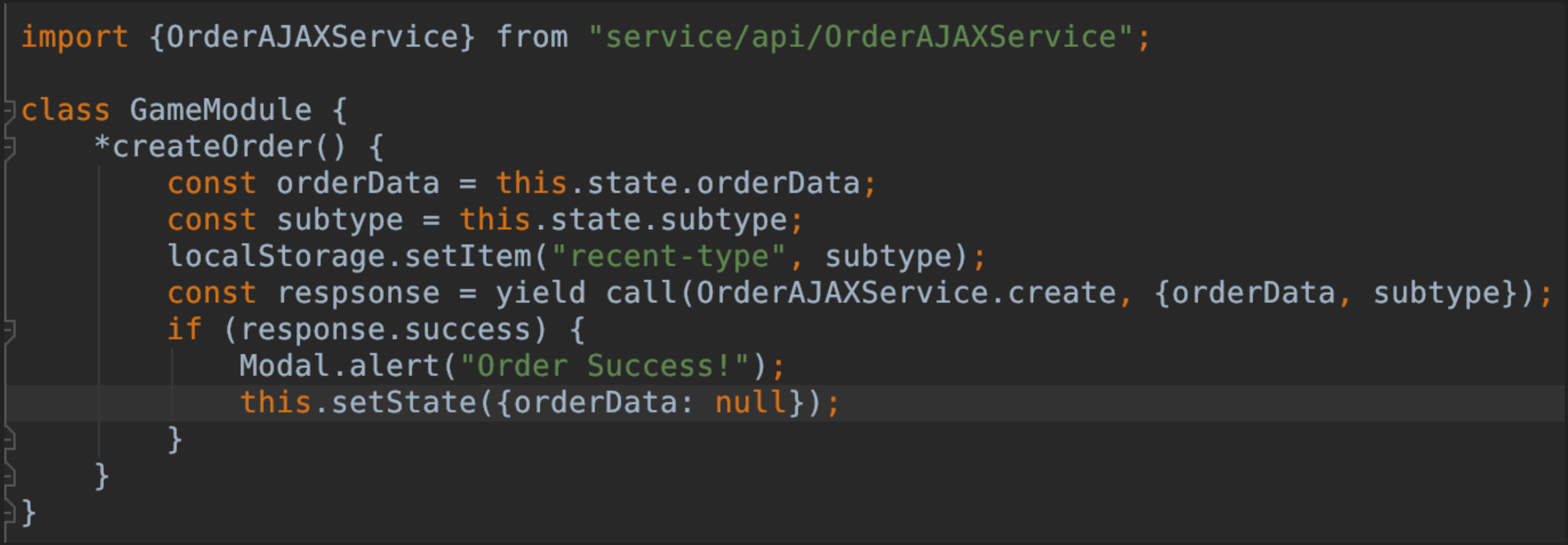
- ***Java SpringMVC***
- ***TypeScript Angular***

CAR EXAMPLE



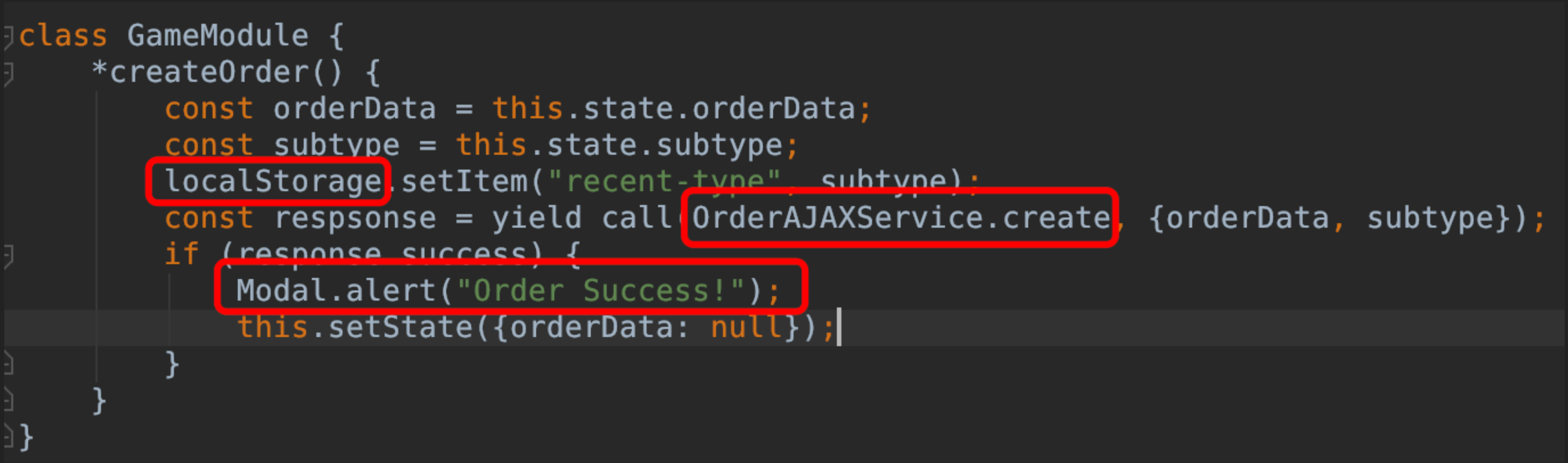
```
class CarWithDI {
    private Wheel wheel;
    private Engine engine;

    Car(Wheel w, Engine e) {
        this.wheel = w;
        this.engine = e;
    }
}

class CarWithoutDI {
    private Wheel wheel = new Wheel();
    private Engine engine = new Engine();

    Car() { }
}
```

FRONT-END CASE



```
import {OrderAJAXService} from "service/api/OrderAJAXService";

class GameModule {
    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        localStorage.setItem("recent-type", subtype);
        const respsonse = yield call(OrderAJAXService.create, {orderData, subtype});
        if (response.success) {
            Modal.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

FRONT-END CASE (CONT.)



```
class GameModule {
    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        localStorage.setItem("recent-type", subtype);
        const respsonse = yield call(OrderAJAXService.create, {orderData, subtype});
        if (response.success) {
            Modal.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

Hard To Test:

1, In test environment (Jest), **no localStorage**

2, We have to wait for real API call, **if no network**, we cannot perform test

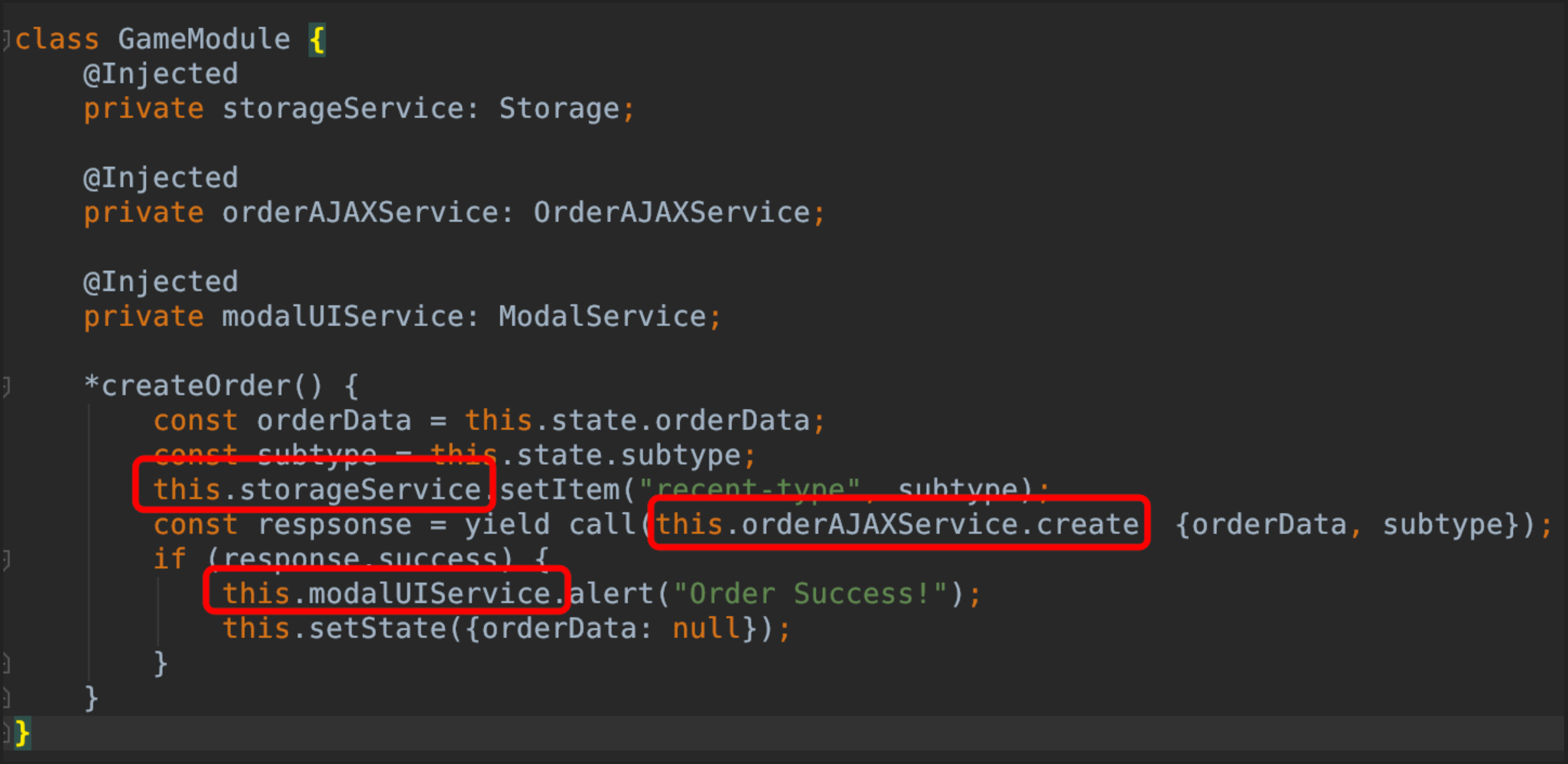
3, Modal.alert has even more **dependencies (AntD, React, Browser DOM etc.)**
So it cannot work in test environment (Jest),

WHY?

- Unit Test
  - We only need that interface, no exact behavior
  - Especially that dependency has interaction with outer environment
- Car ->
  GameModule
- Engine/Wheel ->
  API Service/Storage Service/Modal Service

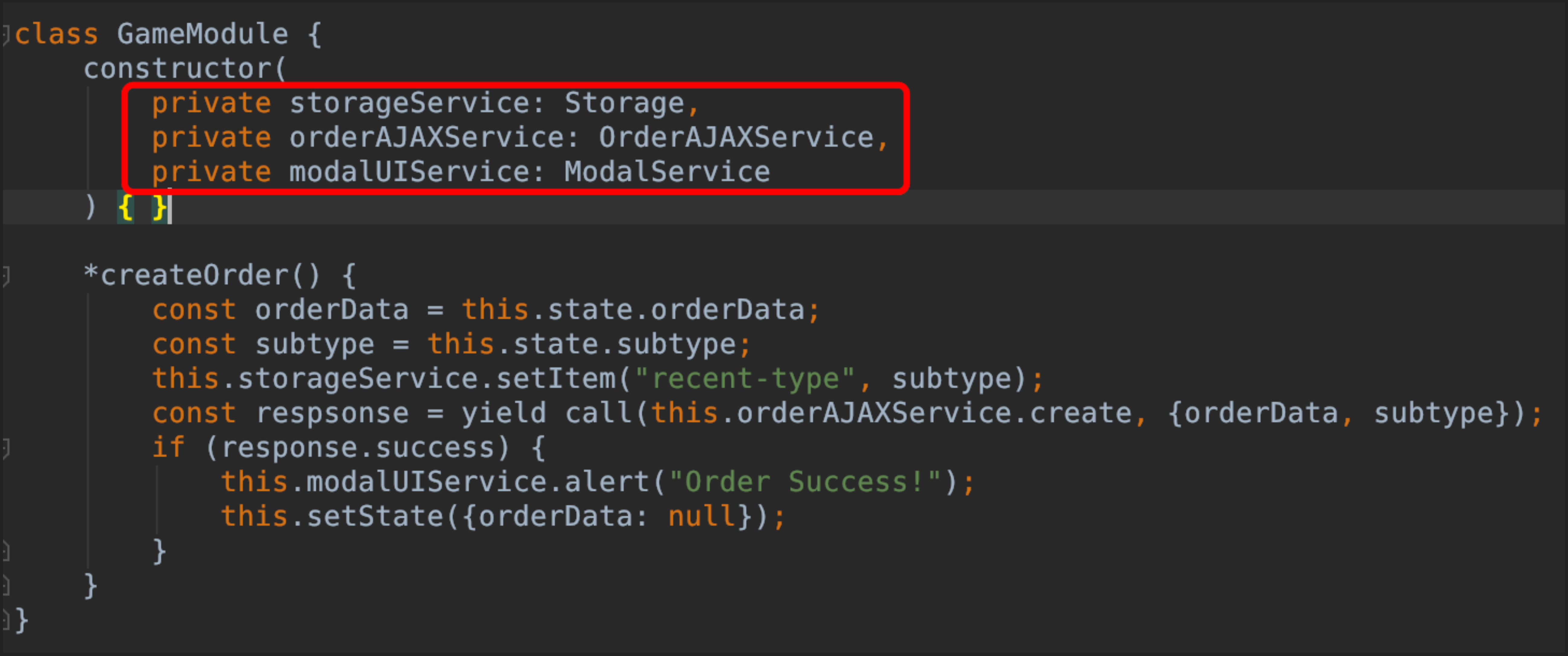
PATTERN 2 - DI

WITH INJECTION (JAVA CORE-NG STYLE)



```
class GameModule {
    @Injected
    private storageService: Storage;

    @Injected
    private orderAJAXService: OrderAJAXService;

    @Injected
    private modalUIService: ModalService;

    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        this.storageService.setItem("recent-type", subtype);
        const respsonse = yield call(this.orderAJAXService.create, {orderData, subtype});
        if (response.success) {
            this.modalUIService.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

WITH INJECTION (ANGULAR - CONSTRUCTOR INJECTION)



```
class GameModule {
    constructor(
        private storageService: Storage,
        private orderAJAXService: OrderAJAXService,
        private modalUIService: ModalService
    ) { }

    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        this.storageService.setItem("recent-type", subtype);
        const respsonse = yield call(this.orderAJAXService.create, {orderData, subtype});
        if (response.success) {
            this.modalUIService.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

WE CAN TEST GAME-MODULE NOW

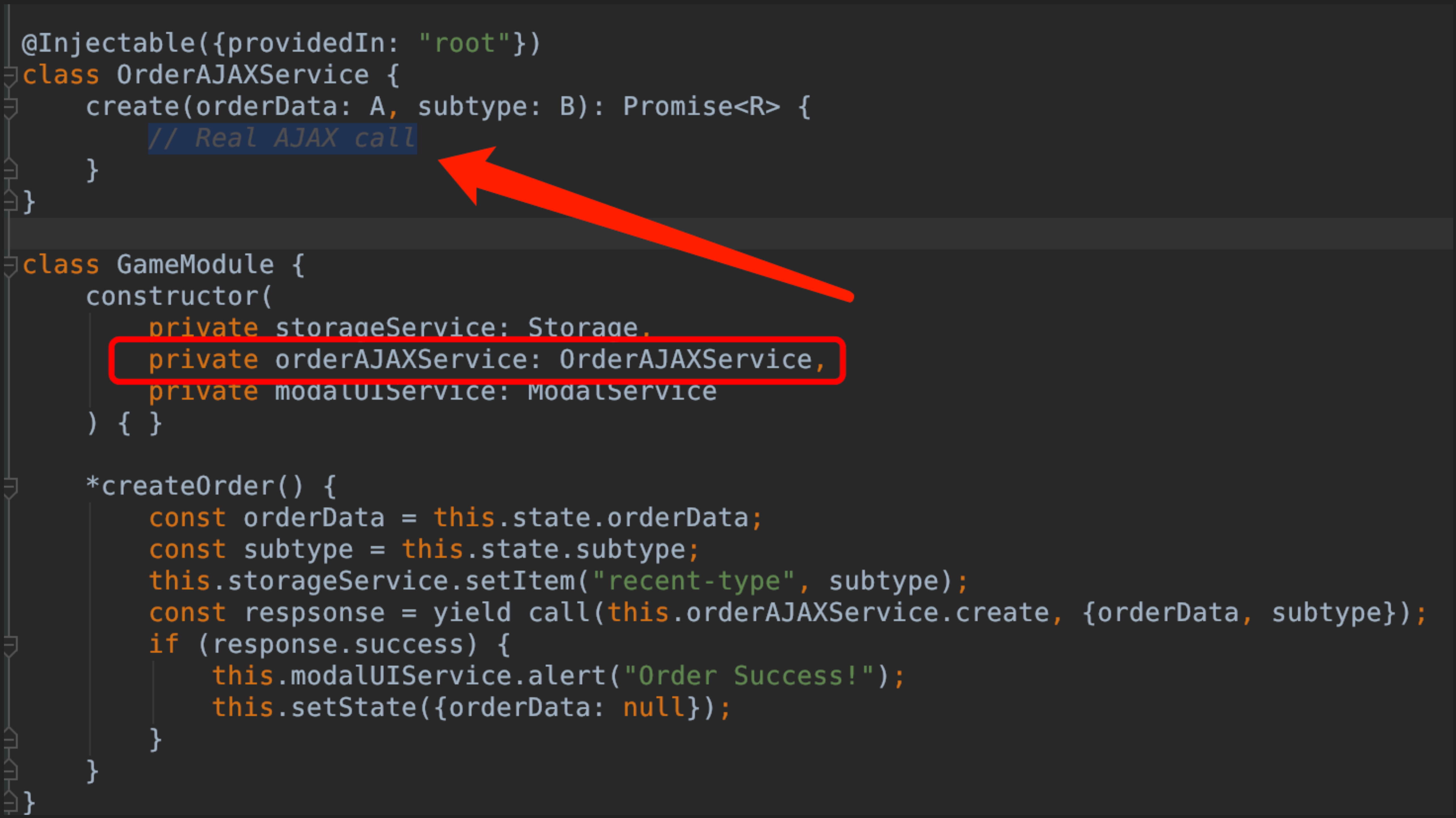
-
  ```
  new GameModule(
      new MockStorage(),
      new MockOrderAJAXService(),
      new MockModalService()
  );
  ```
- As long as each mock has **the same interface** with real one.

HOW ABOUT REAL CODE?

- Usually, we do not construct **new GameModule(…)** ourselves, it is done by our framework.
- That is why, Dependency Injection, is a **framework-level** matter.
- Framework responsibility:
  - DI Container (Provider)
    > Angular: **Tree** Structure
    > Java: **Map** Structure

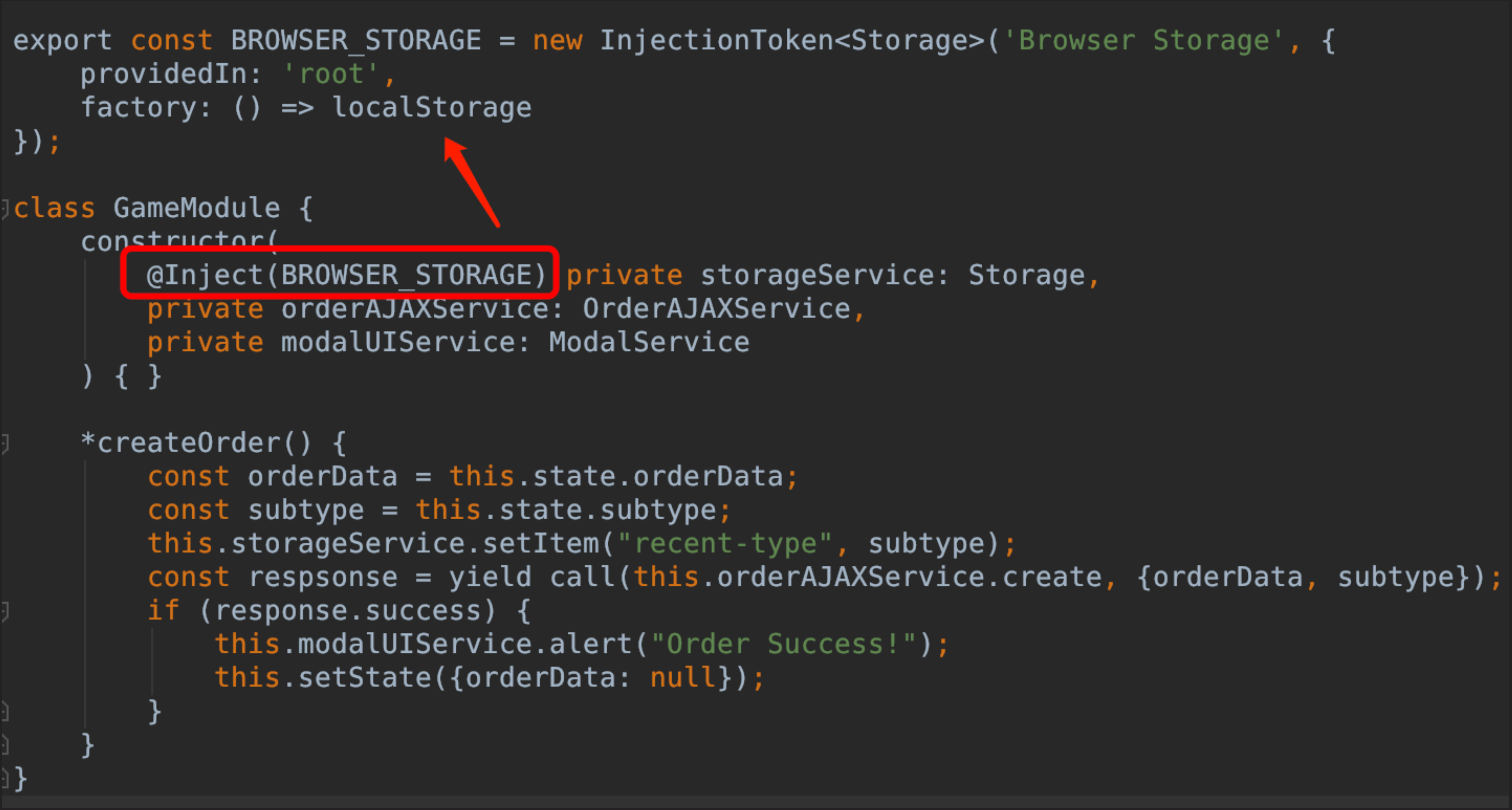
PATTERN 2 - DI

ROOT PROVIDER INSTANCE



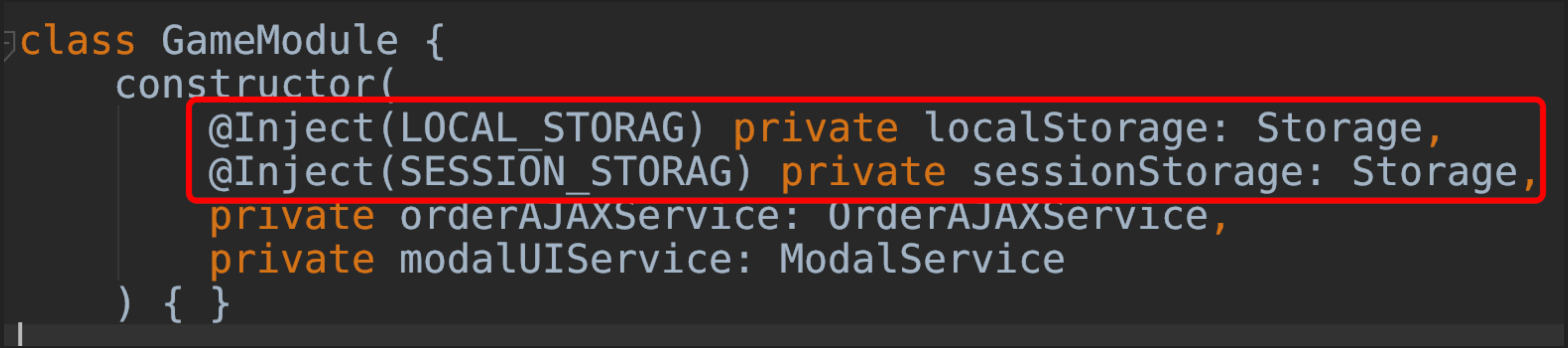
```
@Injectable({providedIn: "root"})
class OrderAJAXService {
    create(orderData: A, subtype: B): Promise<R> {
        // Real AJAX call
    }
}

class GameModule {
    constructor(
        private storageService: Storage,
        private orderAJAXService: OrderAJAXService,
        private modalUIService: ModalService
    ) { }

    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        this.storageService.setItem("recent-type", subtype);
        const respsonse = yield call(this.orderAJAXService.create, {orderData, subtype});
        if (response.success) {
            this.modalUIService.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

RE-USE BUILT-IN PROVIDER



```
export const BROWSER_STORAGE = new InjectionToken<Storage>('Browser Storage', {
    providedIn: 'root',
    factory: () => localStorage
});

class GameModule {
    constructor(
        @Inject(BROWSER_STORAGE) private storageService: Storage,
        private orderAJAXService: OrderAJAXService,
        private modalUIService: ModalService
    ) { }

    *createOrder() {
        const orderData = this.state.orderData;
        const subtype = this.state.subtype;
        this.storageService.setItem("recent-type", subtype);
        const respsonse = yield call(this.orderAJAXService.create, {orderData, subtype});
        if (response.success) {
            this.modalUIService.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

TWO DEPENDENCIES WITH SAME INTERFACE



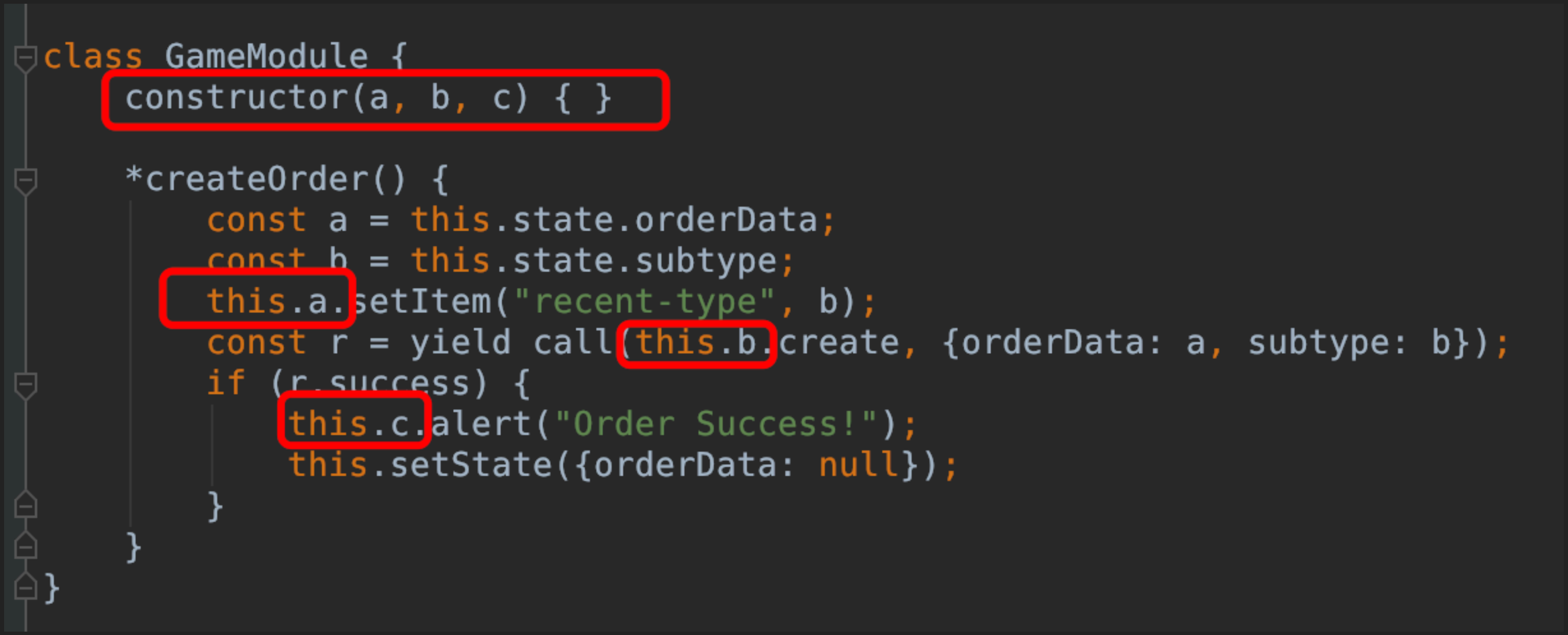
```
class GameModule {
    constructor(
        @Inject(LOCAL_STORAG) private localStorage: Storage,
        @Inject(SESSION_STORAG) private sessionStorage: Storage,
        private orderAJAXService: OrderAJAXService,
        private modalUIService: ModalService
    ) { }
```

IMPLEMENTATION (JAVA)

- Get the injected object type (**Reflection**)
  storageService.class
- Get the injected token
  Usually with a default token (**Lazy Singleton**)
- Retrieve the instance in a framework map
  By **class + token**

IMPLEMENTATION (ANGULAR)

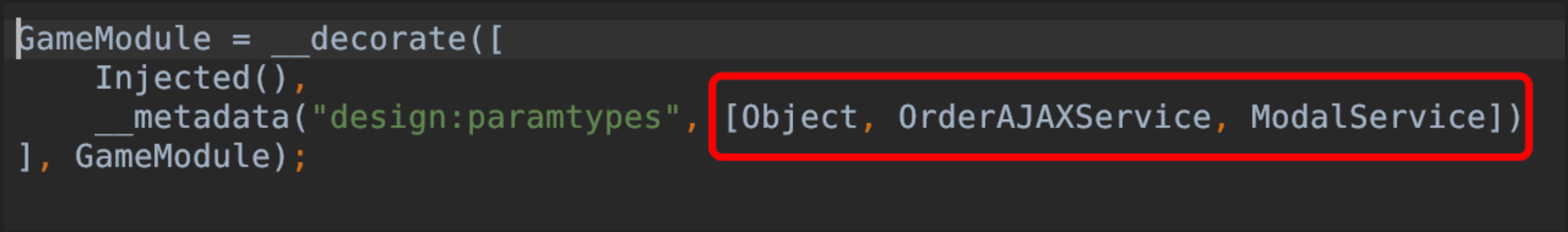
- JavaScript has no type!
  typeof storageService === “object”
  typeof orderAJAXService === “object”
- TypeScript has type, but it only exists at compilation!



```
class GameModule {
    constructor(a, b, c) { }

    *createOrder() {
        const a = this.state.orderData;
        const b = this.state.subtype;
        this.a.setItem("recent-type", b);
        const r = yield call(this.b.create, {orderData: a, subtype: b});
        if (r.success) {
            this.c.alert("Order Success!");
            this.setState({orderData: null});
        }
    }
}
```

IMPLEMENTATION (ANGULAR)

- TypeScript supports type reflection, but disabled by default
  {emitDecoratorMetadata: true}

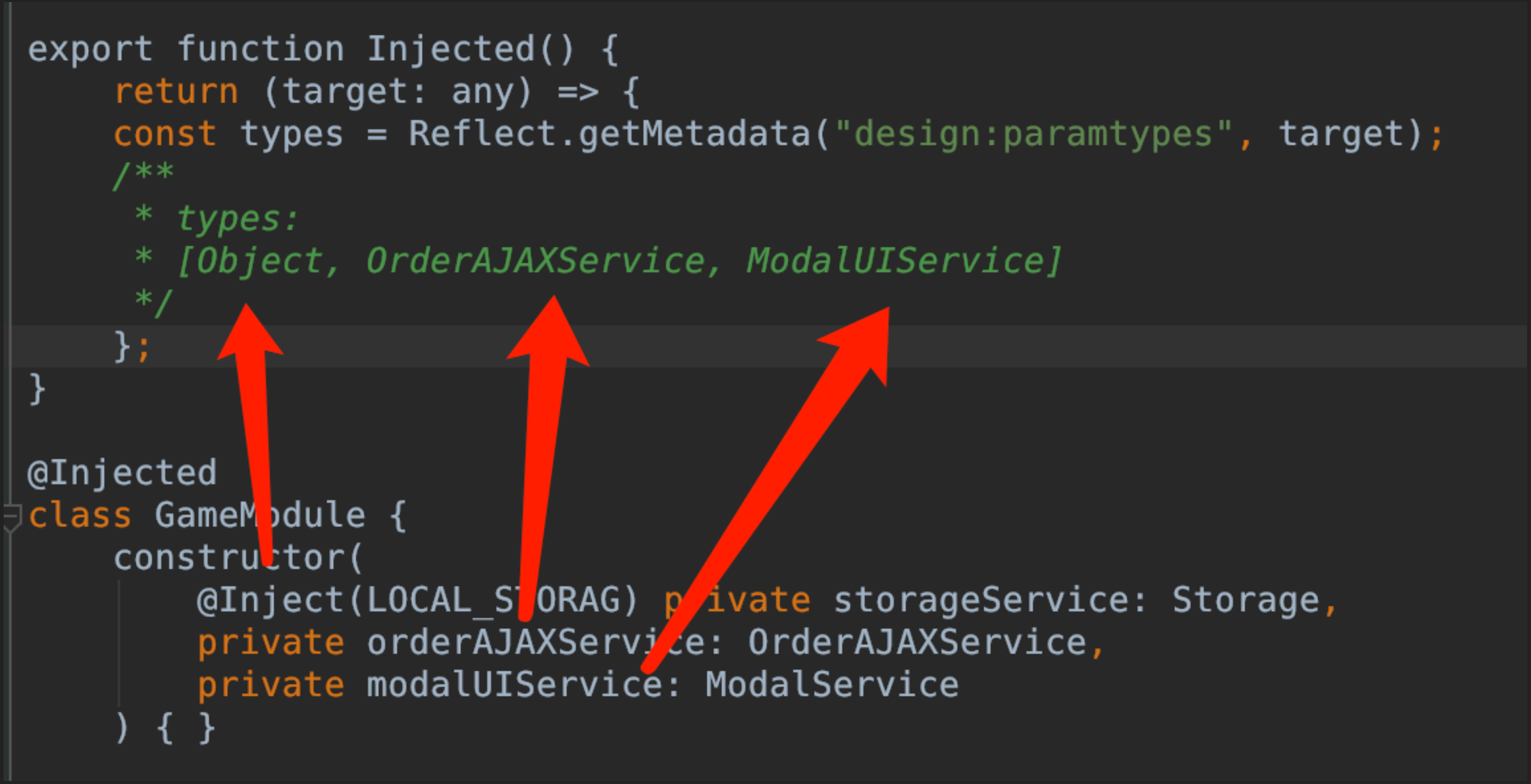


```
GameModule = __decorate([
    Injected(),
    __metadata("design:paramtypes", [Object, OrderAJAXService, ModalService])
], GameModule);
```

- Then use Reflect API (not in ES standard yet)

- Ref:
  https://www.zhihu.com/question/265773703/answer/299346644
  https://zhuanlan.zhihu.com/p/59771686

PATTERN 2 - DI



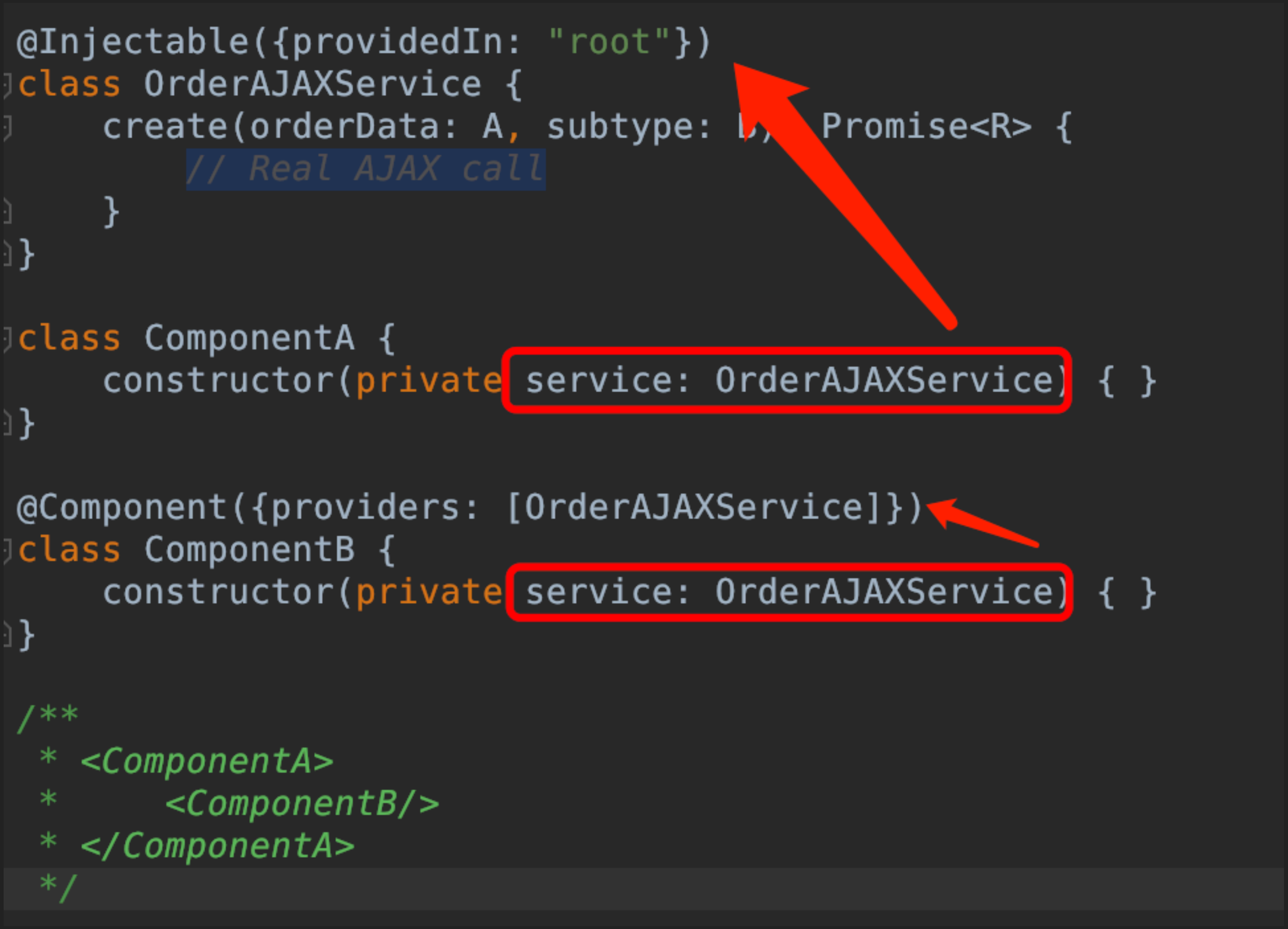
```
export function Injected() {
    return (target: any) => {
    const types = Reflect.getMetadata("design:paramtypes", target);
    /**
     * types:
     * [Object, OrderAJAXService, ModalUIService]
     */
    };
}

@Injected
class GameModule {
    constructor(
        @Inject(LOCAL_STORAG) private storageService: Storage,
        private orderAJAXService: OrderAJAXService,
        private modalUIService: ModalService
    ) { }
```

SESSION STORAGE ISSUE

- **SessionStorage: Storage** is just an interface
  Meta data of its type is marked as **Object**
- That's why we need **InjectToken**, for such interface-only dependencies.
  Other two, are real object prototypes.

  That's how DI works in JavaScript, without runtime type info.

ANGULAR INJECTION HIERARCHY



```
@Injectable({providedIn: "root"})
class OrderAJAXService {
    create(orderData: A, subtype: B): Promise<R> {
        // Real AJAX call
    }
}

class ComponentA {
    constructor(private service: OrderAJAXService) { }
}

@Component({providers: [OrderAJAXService]})
class ComponentB {
    constructor(private service: OrderAJAXService) { }
}

/**
 * <ComponentA>
 *     <ComponentB/>
 * </ComponentA>
 */
```

2 instances here:

(1) **Root provider**

(2) **ComponentB provider**

Only exists while ComponentB is mounted

What about <ComponentA> inside <ComponentB> ?

THANK YOU