

ЛЕКЦИЯ I © 2021 Нет Ит

# Android: Dependency Injection

Теодор Костадинов



SOFTWARE  
ACADEMY



# Съдържание

1. Design Patterns
2. SOLID
3. Inversion of Control
4. Dependency Injection
5. Automated DI
6. Dagger



# **DESIGN PATTERNS**



# Защо да използваме шаблони?

- Използване на колективния опит за софтуерно проектиране за доказани решения на често срещани проблеми
- Поощряват reusability на кода, което води до по-качествен и лесен за поддръжка код
- Обща терминология, която помага на програмистите да се разбират лесно

# Видове шаблони

## CREATIONAL

Осигуряват начин да се създават обекти на класове, скривайки логиката по създаването им (вместо директно с оператора *new*)

## STRUCTURAL

Осигуряват различни начини за създаване на по-сложни класове чрез наследяване и композиция на по-прости класове.

## BEHAVIORAL

Свързани са с комуникацията между различните обекти в системата



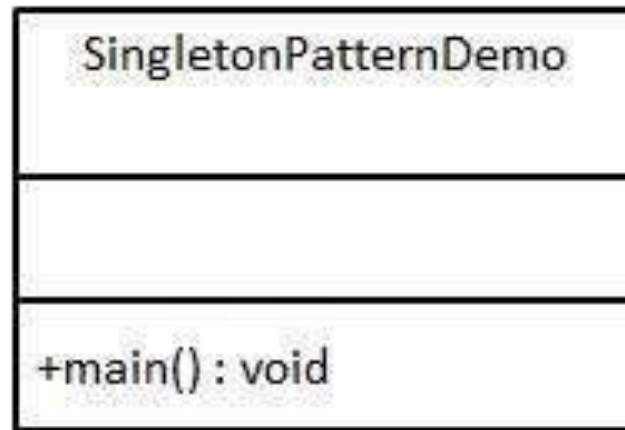


**SINGLETON**

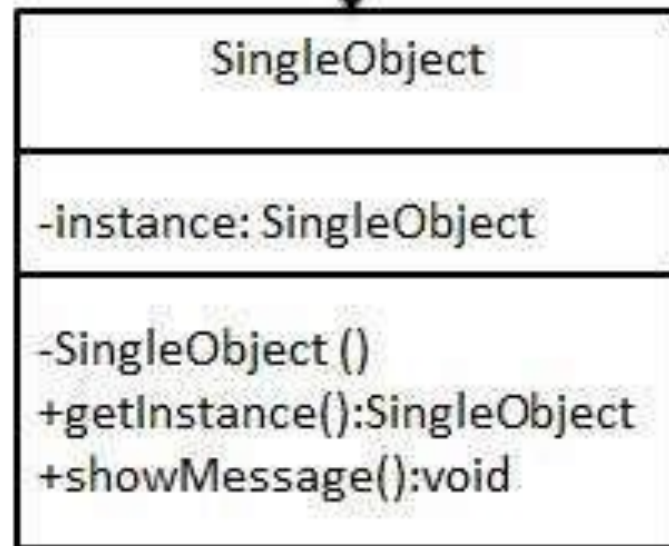
# Малко повече за Singleton pattern

- Клас, от който може да съществува най-много една инстанция
- Имплементация:
  - private конструктор
  - private static член-променлива от тип същия клас, която реферира единствената инстанция на класа
  - public static метод, който връща инстанцията на класа





asks



returns



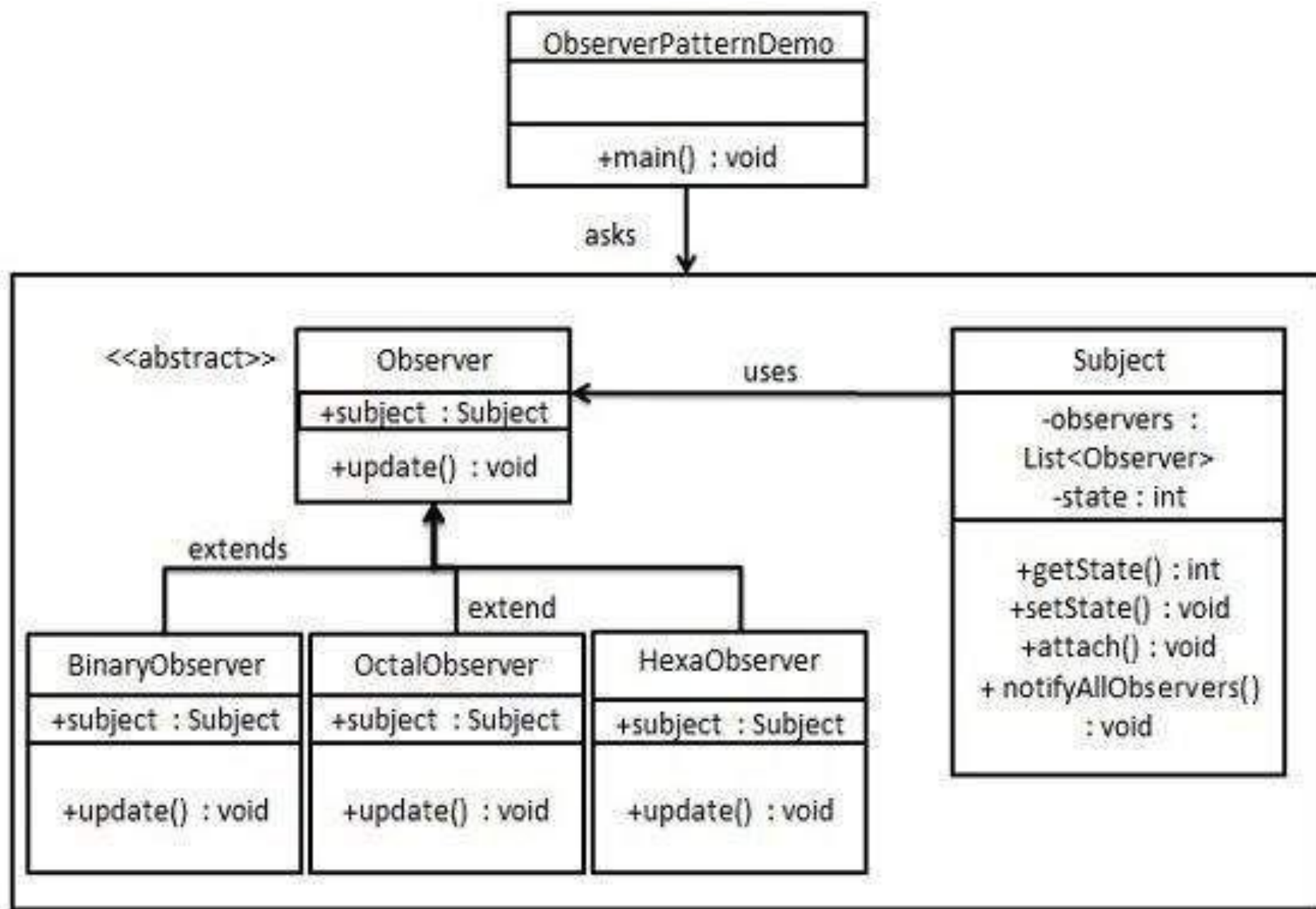


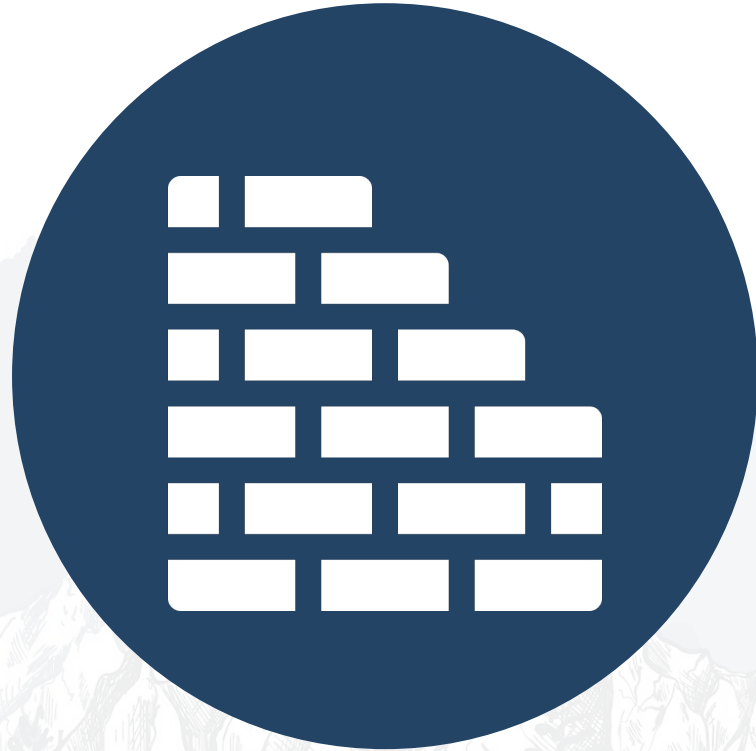
**OBSERVER**

# Малко повече за Observer pattern

- Удобен е, когато се интересуваме от състоянието на даден обект и искаме да бъдем нотифицирани, когато има промяна в състоянието
- Обектът, който наблюдава за промяна на състоянието на друг обект, се нарича *Observer*, а наблюдаваният обект се нарича *Subject*







**S.O.L.I.D.**





SINGLE-RESPONSIBILITY PRINCIPLE

OPEN/CLOSED PRINCIPLE

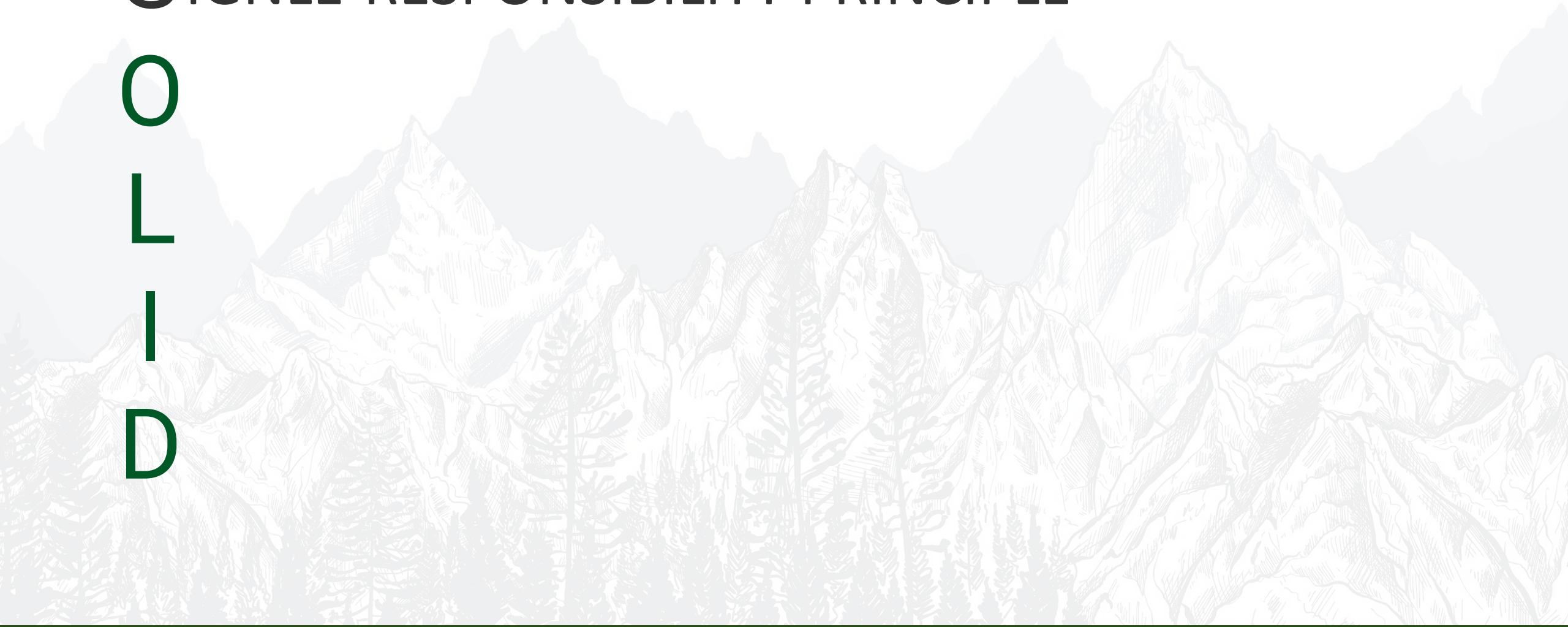
LISKOV SUBSTITUTION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

# SINGLE-RESPONSIBILITY PRINCIPLE

OLD





```
public class Book {  
  
    private String name;  
    private String author;  
    private String text;  
  
    //constructor, getters and setters  
  
    // methods that directly relate to the book properties  
    public String replaceWordInText(String word){  
        return text.replaceAll(word, text);  
    }  
  
    public boolean isWordInText(String word){  
        return text.contains(word);  
    }  
  
    void printTextToConsole(){  
        // our code for formatting and printing the text  
    }  
  
}
```

```
public class Book {  
  
    private String name;  
    private String author;  
    private String text;  
  
    //constructor, getters and setters  
  
    //methods that directly relate to the  
    //book properties  
    public String replaceWordInText(String word){  
        return text.replaceAll(word, text);  
    }  
  
    public boolean isWordInText(String word){  
        return text.contains(word);  
    }  
}
```

```
public class BookPrinter {  
  
    // methods for outputting text  
    void printTextToConsole(String text){  
        //our code for formatting and printing  
    }  
  
    void printTextToAnotherMedium(String text){  
        // code for writing to any other location  
    }  
}
```



# Защо SINGLE-RESPONSIBILITY?

1. **Тестване** – клас с една отговорност ще има много по-малко случая за проверка
2. **По-малък coupling** – по-малко функционалност в даден клас означава по-малко dependency-та
3. **Организация** – по-малки, добре организирани класове са по-лесни за поддържане и търсене



S

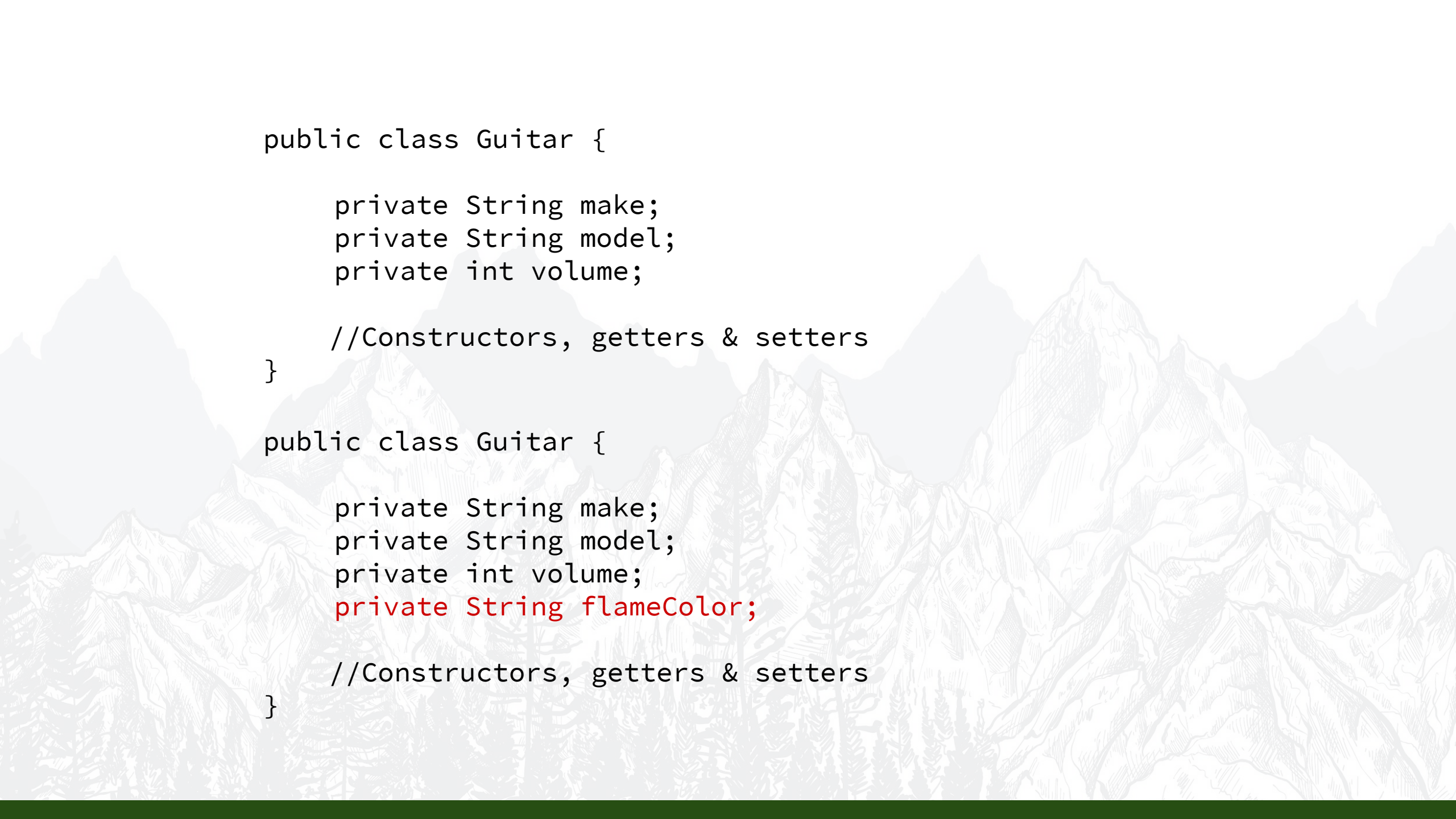
OPEN/CLOSED PRINCIPLE

L

I

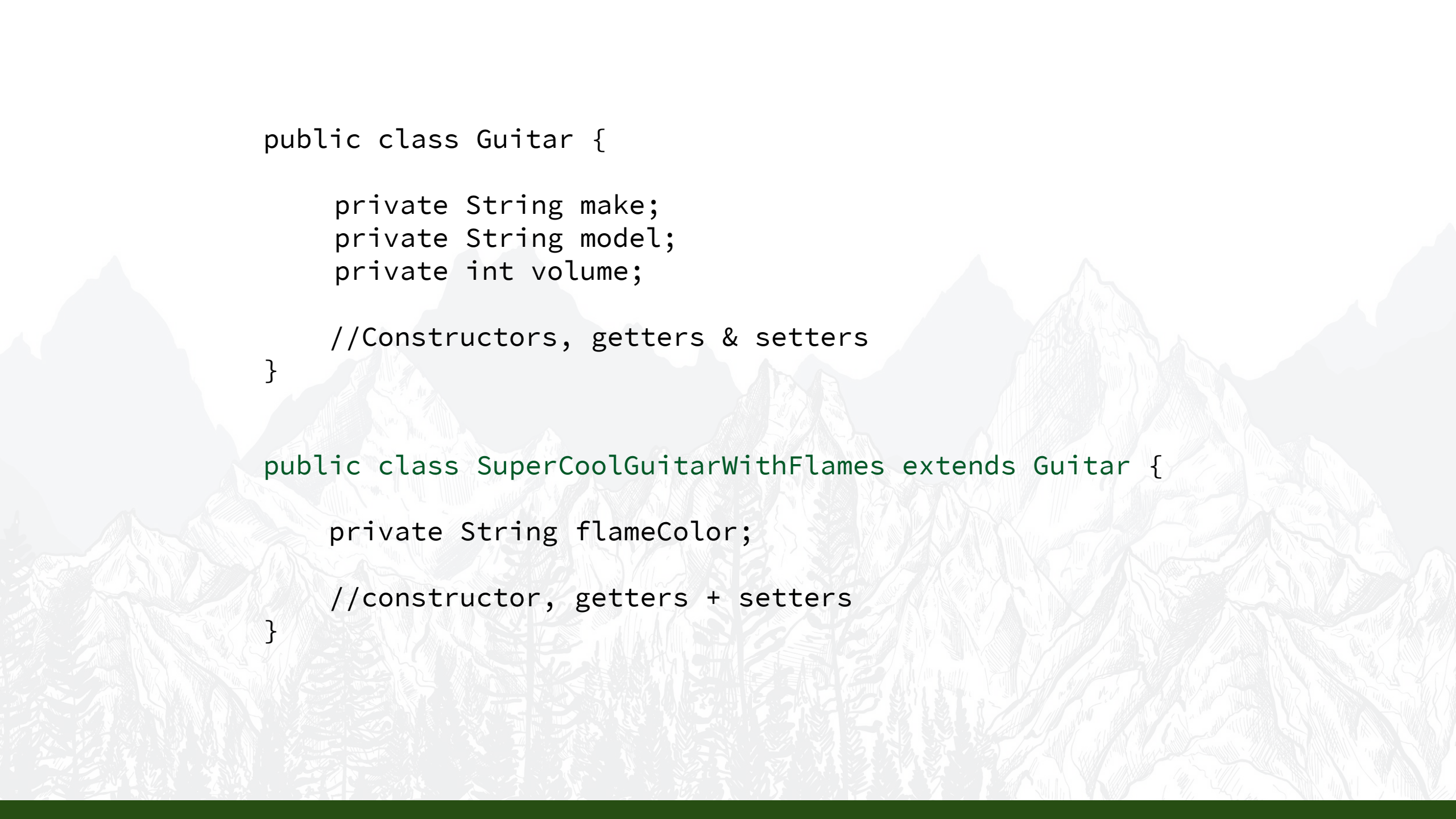
D





```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
  
    //Constructors, getters & setters  
}
```

```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
    private String flameColor;  
  
    //Constructors, getters & setters  
}
```



```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
  
    //Constructors, getters & setters  
}  
  
public class SuperCoolGuitarWithFlames extends Guitar {  
  
    private String flameColor;  
  
    //constructor, getters + setters  
}
```



# Защо OPEN-CLOSED?

1. **Намаляване на бъговете** – тъй като не променяме вече *работещ* код, шансът за възникване на проблеми в тези секции намалява
2. **Тестване** – не променяме вече написаните от нас тестове, а само пишем нови за новата функционалност



S

O

LISKOV SUBSTITUTION PRINCIPLE

I

D



```
public interface Car {  
  
    void turnOnEngine();  
    void accelerate();  
}
```

```
public class MotorCar implements Car {
```

```
    private Engine engine;
```

```
    //Constructors, getters + setters
```

```
    public void turnOnEngine() {  
        //turn on the engine!  
        engine.on();  
    }
```

```
    public void accelerate() {  
        //move forward!  
        engine.powerOn(1000);  
    }
```

```
}
```

```
public class ElectricCar implements Car {
```

```
    public void turnOnEngine() {  
        throw new AssertionError("No engine");  
    }
```

```
    public void accelerate() {  
        //this acceleration is crazy!  
    }  
}
```

# Защо LISKOV SUBSTITUTION?

1. **Консистентност** – гарантираме, че приложението има предсказуемо поведение
2. **Взаимна заменяемост** – можем лесно да заменяме класовете наследници и да си гарантираме, че приложението ще работи





S  
O  
L  
|  
D

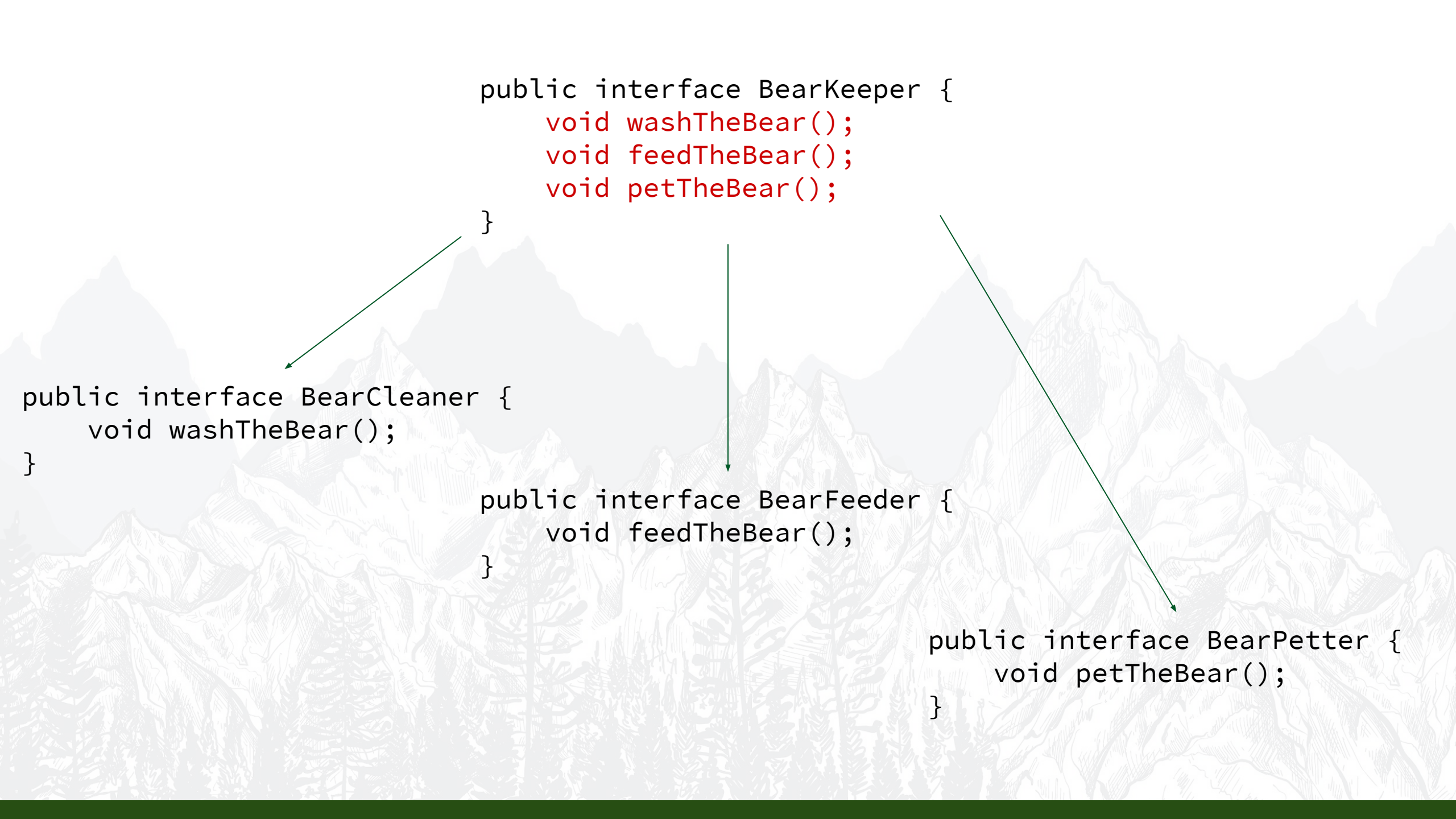
INTERFACE SEGREGATION PRINCIPLE

```
public interface BearKeeper {  
    void washTheBear();  
    void feedTheBear();  
    void petTheBear();  
}
```

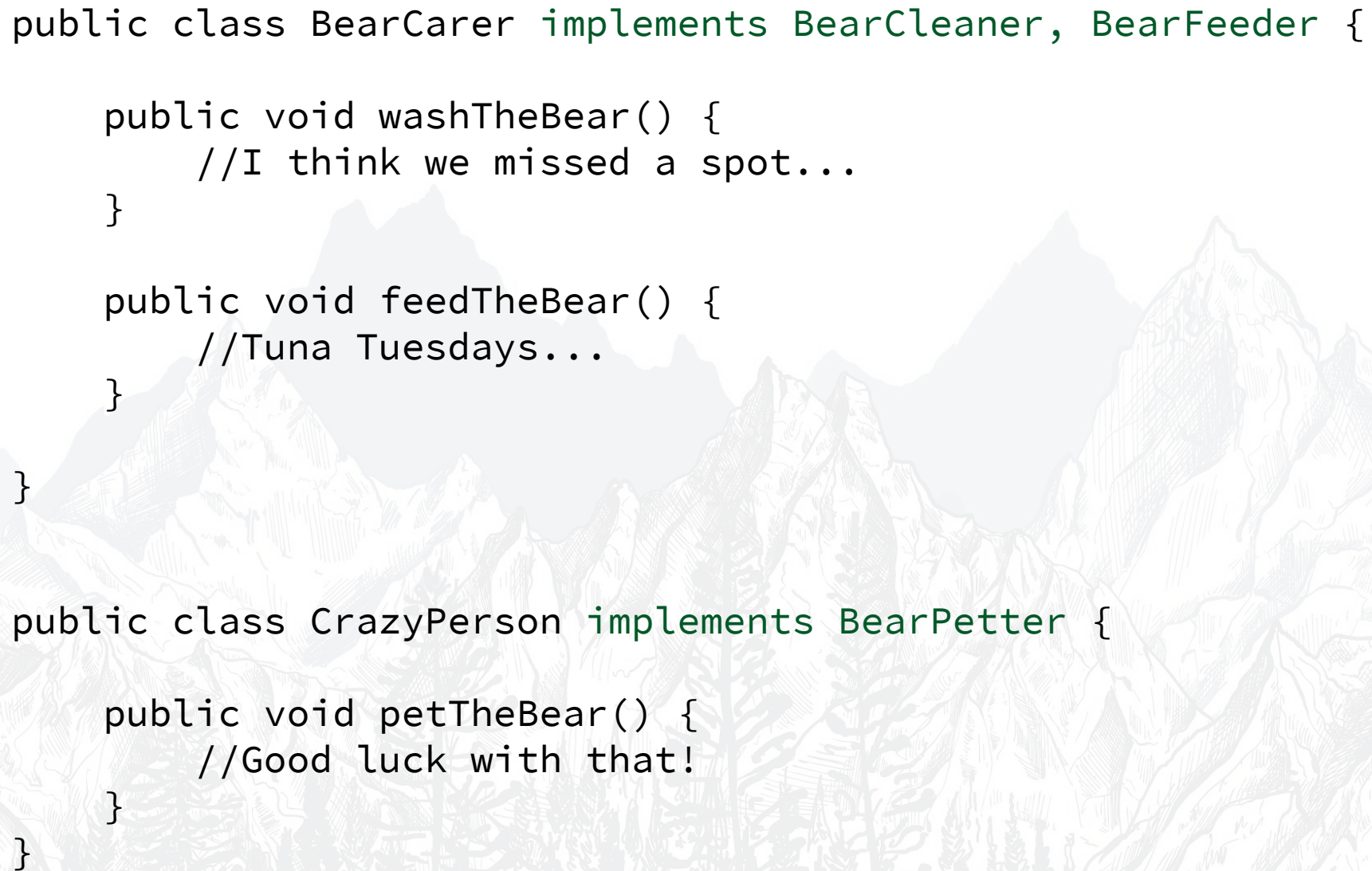
```
public interface BearCleaner {  
    void washTheBear();  
}
```

```
public interface BearFeeder {  
    void feedTheBear();  
}
```

```
public interface BearPetter {  
    void petTheBear();  
}
```







```
public class BearCarer implements BearCleaner, BearFeeder {  
  
    public void washTheBear() {  
        //I think we missed a spot...  
    }  
  
    public void feedTheBear() {  
        //Tuna Tuesdays...  
    }  
}  
  
public class CrazyPerson implements BearPetter {  
  
    public void petTheBear() {  
        //Good luck with that!  
    }  
}
```

# ???

```
public class BookPrinter {  
  
    // methods for outputting text  
    void printTextToConsole(String text){  
        //our code for formatting and printing  
    }  
  
    void printTextToAnotherMedium(String text){  
        // code for writing to any other location  
    }  
}
```



# Защо INTERFACE SEPARATION?


1. **Точност** – имплементираме интерфейси само с функционалността, която ни е нужна
2. **Прецизност** – избягваме един клас да има повече от една отговорност, с което не нарушаваме *Single responsibility principle*



S  
O  
L  
I

DEPENDENCY INVERSION PRINCIPLE





```
public class Windows98Machine {  
  
    private final StandardKeyboard keyboard;  
    private final Monitor monitor;  
  
    public Windows98Machine() {  
        monitor = new Monitor();  
        keyboard = new StandardKeyboard();  
    }  
}
```

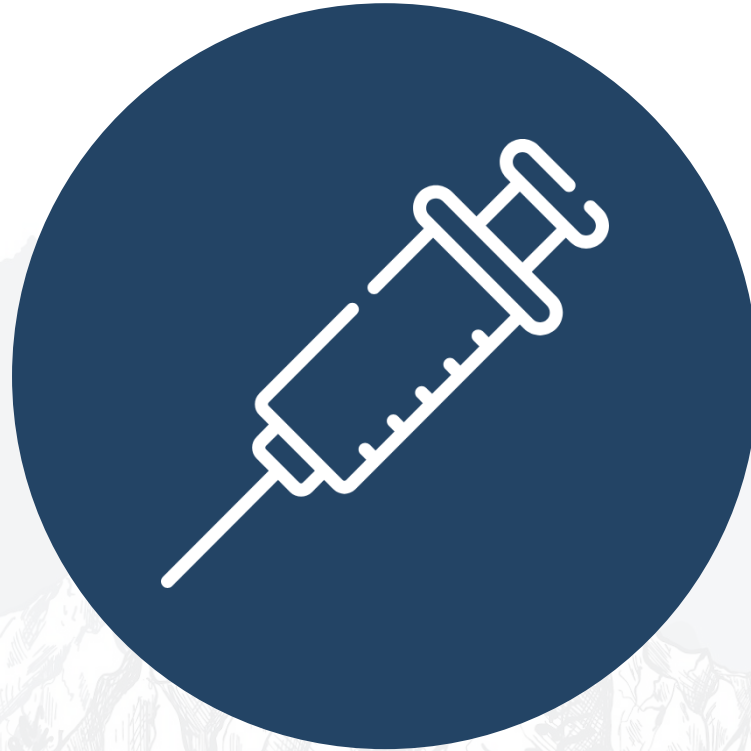
```
public class Windows98Machine{  
  
    private final Keyboard keyboard;  
    private final Monitor monitor;  
  
    public Windows98Machine(Keyboard keyboard, Monitor monitor) {  
        this.keyboard = keyboard;  
        this.monitor = monitor;  
    }  
}
```

```
public class StandardKeyboard implements Keyboard { }
```



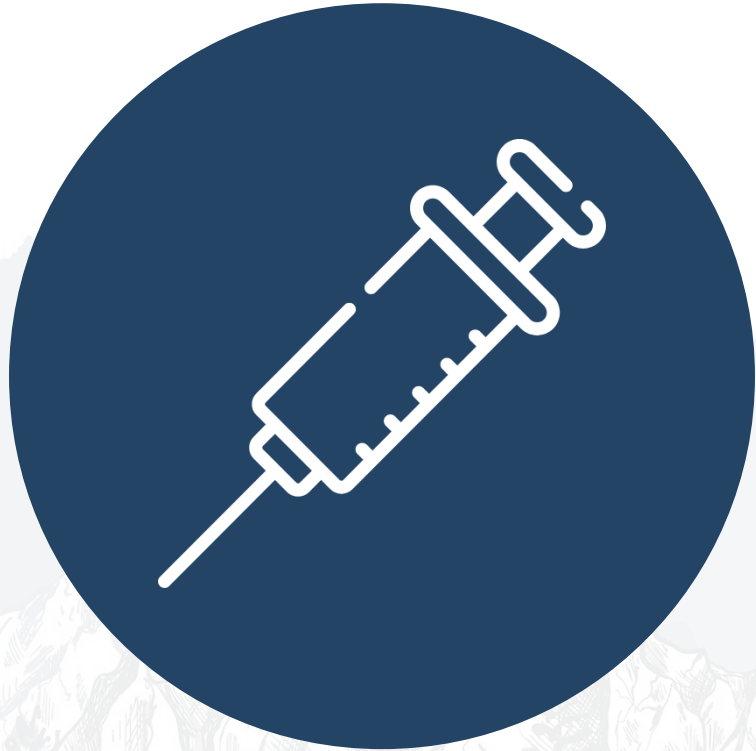
# Защо DEPENDENCY INVERSION?

1. **Няма coupling** - отделните слоеве не дори не съдържат header-ите на останалите
2. **Гъвкавост** – свободно да правим промени; можем да *mix-and-match*-ваме различни слоеве
3. **Пестим време** – промените са много по-лесни и не отнемат ценно време



**Почивка до 21:10**





# Dependency Injection

# Dependency Injection

**Техника за писане на код, при която нужните за работата на един клас полета не се създават в него, а се подават отвън**



# Защо DI?

- Преизползваемост на кода
- Лесно тестване
- Лесно рефакториране

# Как работи DI?

- Някои класове използват вътрешно инстанции на други класове - dependencies
- Един клас може да си набави dependency като:
  - си го създаде сам
  - го получи отвън (dependency injection)



# Пример:

Имаме клас Person. Всеки човек има застраховка => в Person има инстанция на класа Insurance.

Как можем да получим Insurance в Person?

# Вариант 1:

Можем директно в Person да създадем инстанция на Insurance:

```
class Person {  
    private Insurance insurance = new Insurance();  
}  
  
class MyApp {  
    public static void main(String[] args) {  
        Person person = new Person();  
    }  
}
```



# Вариант 1

Какви са проблемите на този подход?

- Person и Insurance са прекалено тясно обвързани - Person използва конкретна имплементация на Insurance и не може да използва други нейни варианти.
- Не можем да използваме различни Insurance обекти за различни тестови случаи

# Вариант 2:

Можем да подадем Insurance през конструктора:

```
class Person {  
    private Insurance insurance;  
    public Person(Insurance insurance){  
        this.insurance = insurance;  
    }  
}
```

```
class MyApp {  
    public static void main(String[] args) {  
        Insurance insurance = new Insurance()  
        Person person = new Person(insurance);  
    }  
}
```



# Вариант 2

Как са решени проблемите от вариант 1?

- Можем да преизползваме Person - да създаваме много хора с различни видове Insurance, без да се налага да променяме Person класа
- Можем да тестваме как се държи Person в различни случаи - с различни застраховки



**Automated DI**



# Automated DI

- Когато имаме само няколко dependencies, не е трудно да използваме manual DI
- Когато обаче те станат повече, става трудно да правим всичко ръчно
- Ръчната обработка на dependencies води до два сериозни проблема:
  - boilerplate code
  - exceptions

# Boilerplate code

- Свързване на dependencies в големи приложения
- При многослойни приложения - всеки слой се нуждае от dependencies, които пък от своя страна имат нужда от такива.



# Exceptions

- Ако не успеем да създадем някое dependency преди да го подадем, ще получим exception
- Трябва да си създадем custom container или dependency graph, който да се грижи за подобни проблеми

# Библиотеки:

- Dagger
- Dagger 2
- Hilt
- Koin





**Dagger (2)**

# Dagger

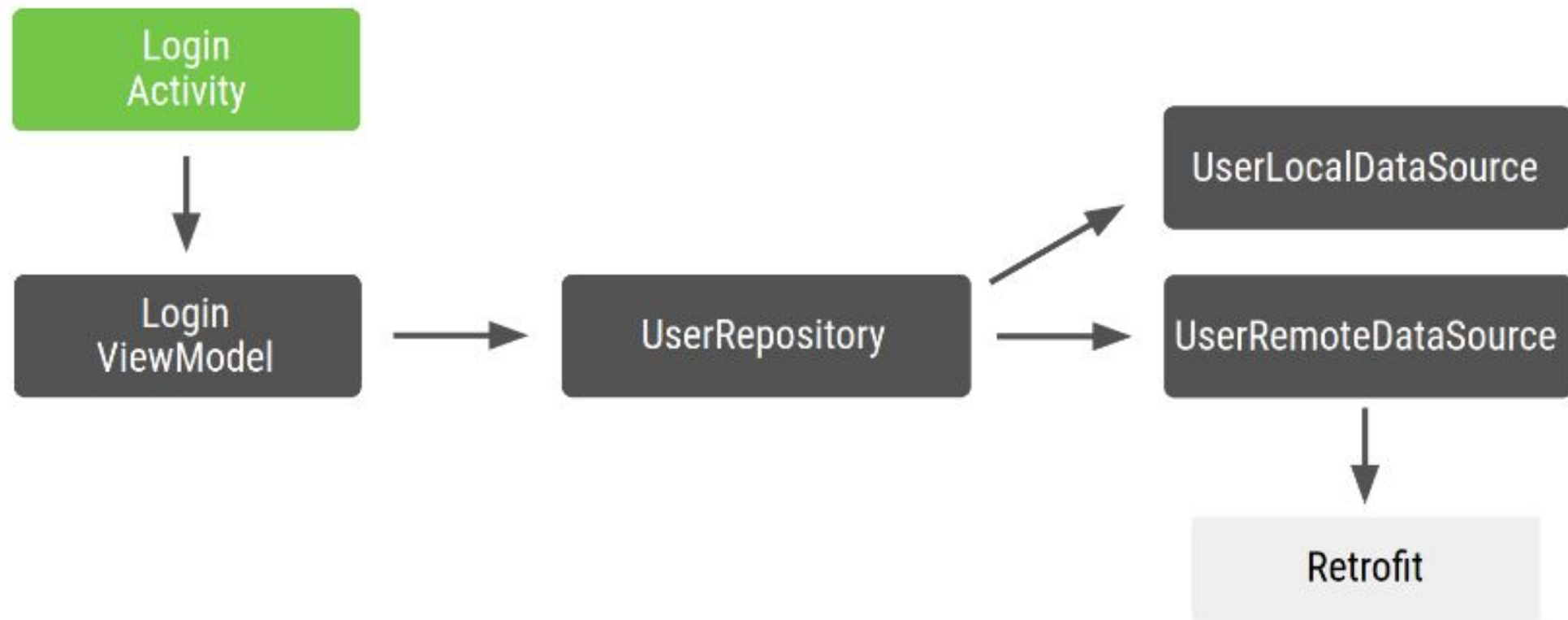
- Dependency Injection Framework
- Maintained by Google
- Java, Kotlin, Android



# Setup:

```
dependencies {  
    implementation 'com.google.dagger:dagger:2.x'  
    annotationProcessor 'com.google.dagger:dagger-compiler:2.x'  
}
```

# Dependency graph:





# @Module

- A class, annotated with @Module
- Начин да енкапсулираме логика за предоставяне на класове
- Пример: NetworkModule - тук ще имаме логика за предоставяне на класове, свързани с network

# @Provides

- Чрез @Provides анотацията можем да предоставяме dependencies
- Така казваме на dagger как да създаде обекти от типа на функцията, която е анотирана с @Provides
- Използва се за обекти, които нашето приложение не притежава



# Пример:

```
@Module
public class NetworkModule {

    @Provides
    public LoginRetrofitService provideLoginRetrofitService() {
        return new Retrofit.Builder()
            .baseUrl("https://example.com")
            .build()
            .create(LoginService.class);
    }
}
```

Казваме на Dagger как да създаде инстанции от типа, която тази функция връща

Когато искаме от Dagger да ни даде инстанция на LoginRetrofitService, кодът от тази функция се изпълнява

# @Component

- Dagger създава граф на dependencies, от който търси тези dependencies, които са ни нужни
- За да създаде този граф, ние трябва да създадем специален interface, аотиран с @Component



# Пример:

Казваме на Dagger кои модули да включи в създаването на графа

```
@Component(modules = NetworkModule.class)
public interface ApplicationComponent {
    ...
}
```

# Инстанциране на компонент:

В най-простия начин за сетъп на Dagger, Component-а трябва да се създаде на такова място, че всички Activity и Fragments да имат достъп до него, че да могат да комуникират с него, когато имат нужда да бъдат инжектирани.

Такова подходящо място е Application класа. Той може да инициализира Component-а и да създаде getter с който активититата и фрагментите да го достъпват.

```
DaggerAppComponent appComponent = DaggerAppComponent.builder()  
    .networkModule(new NetworkModule())  
    .appModule(new AppModule(this))  
    .build();
```



# Методи в компонента

```
@Singleton
@Component(modules = {AppModule.class})
public interface AppComponent {
    void inject(MainActivity mainActivity);
}
```

За всяко Активити и Фрагмент трябва да създадем метод тук. Съответното Активити/Фрагмент трябва да извика този метод при създаването си.

# Модул с КОНТЕКСТ

```
@Module
public class AppModule {
    Application application;
    public AppModule(Application application) {
        this.application = application;
    }
    @Provides
    Application providesApplication() {
        return application;
    }
    @Provides
    @Singleton
    public SharedPreferences providePreferences() {
        return application.getSharedPreferences(DATA_STORE, Context.MODE_PRIVATE);
    }
}
```



# @Inject

- Когато аотираме конструктор с @Inject, казваме на Dagger как да създава инстанции от съответния клас
- Когато аотираме поле с @Inject, казваме на Dagger, че трябва да създаде инстанция на съответния клас

# Пример:

```
public class UserRepository {  
  
    private final UserLocalDataSource userLocalDataSource;  
    private final UserRemoteDataSource userRemoteDataSource;  
  
    @Inject  
    public UserRepository(UserLocalDataSource userLocalDataSource,  
                          UserRemoteDataSource userRemoteDataSource) {  
        this.userLocalDataSource = userLocalDataSource;  
        this.userRemoteDataSource = userRemoteDataSource;  
    }  
}
```

Казваме на Dagger как  
да създаде инстанции  
на UserRepository



# Пример:

```
public class UserLocalDataSource {  
    @Inject  
    public UserLocalDataSource() { }  
}  
  
public class UserRemoteDataSource {  
    @Inject  
    public UserRemoteDataSource() { }  
}
```

# Пример за инжектиране на Dagger в Activity

```
public class LoginActivity extends Activity {
```

```
    @Inject  
    LoginViewModel loginViewModel;
```

Казваме на Dagger да  
предостави инстанция  
на LoginViewModel

```
    @Override  
    protected void onCreate(Bundle savedInstanceState) {
```

Казваме на Dagger да  
инстанцира @Inject  
полетата в LoginActivity

```
        ((MyApplication) getApplicationContext()).appComponent.inject(this);  
        super.onCreate(savedInstanceState);
```

```
    }
```

```
}
```



# Пример:

```
public class LoginViewModel {  
  
    private final UserRepository userRepository;  
  
    @Inject  
    public LoginViewModel(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
}
```

@Inject казва на Dagger  
как да  
създаде инстанция  
на LoginViewModel

# Полезни линкове:

ContributesAndroidInjector:

<https://proandroiddev.com/dagger-2-annotations-binds-contributesandroidinjector-a09e6a57758f?qi=2b36f73e535>

Android Developers:

<https://developer.android.com/training/dependency-injection/dagger-basics>

Component Builder:

<https://proandroiddev.com/dagger-2-component-builder-1f2b91237856>

Official Dagger Documentation:

<https://dagger.dev/>



ВЪПРОСИ?





© 2020 Нет Ит

# БЛАГОДАРЯ ЗА ВНИМАНИЕТО!



SOFTWARE  
ACADEMY

