6. 싱글턴패턴

학습목표

학습목표

- 싱글턴 패턴 이해하기
- 다중 스레드와 싱글턴 패턴의 관계 이해하기

리틀 싱글턴 알아보기

출처: 에릭 프리먼 외. (2023). 헤드 퍼스트 디자인 패턴

리틀 리스퍼(The Little Lisper) 스타일의 문답

1개의 개체를 만들려면 어떻게 하면 좋을까요?

```
new MyObject();
```

다른 객체에서 MyObject를 만들려면 어떻게 해야 하죠?
MyObject에 new 연산자를 다시 쓸 수 있나요?

물론 가능하죠.

클래스만 있으면 언제든지 인스턴스를 만들 수 있는 거죠?

예, public으로 선언한 거라면 별문제 없습니다.

만약 public으로 선언하지 않았다면요?

public 클래스로 선언하지 않았다면 같은 패키지에 있는 클래스에서만 인스턴스를 만들 수 있습니다. 물론 2개 이상의 인스턴스도 만들 수 있죠.

흥미롭군요.
이렇게 할 수도 있다는데 혹시 알고 계신가요?

```
public MyClass  {
        private MyClass() {}
}
```

아, 그런 건 한 번도 생각해 보지 못했습니다. 하지만 문법적으로는 전혀 문제가 없어 보이는군요.

저 코드를 설명해 주실 수 있나요?

생성자가 private으로 선언되어 있어서 인스턴스를 만들 수 없는 클래스 같군요.

---

private으로 선언된 생성자를 사용할 수 있는 객체가 과연 존재할까요?

흠… MyClass에 있는 코드에서만 호출할 수 있는 것 같은데, 객체의 인스턴스를 만들 수 없을 것 같군요.

왜 안 되죠?

생성자를 호출하려면 일단 그 클래스의 인스턴스가 있어야 하는데, 다른 클래스에서 이 클래스의 인스턴스를 만들 수 없어 불가능합니다. 닭이 먼저냐 달걀이 먼저냐라는 문제와 같다고 볼 수 있습니다. MyClass 형식의 객체에서만 private으로 선언된 생성자를 사용할 수 있고, 다른 어떤 클래스에서 'new MyClass()'라고 쓸 수 없기에 결국 인스턴스를 만들 수 없죠.

그래요. 그건 그렇고
이건 어떻게 해석할 수 있을까요?

```
public MyClass {
    public static MyClass getInstance() {
    }
}
```

MyClass에 정적 메소드가 있습니다. 그 정적 메소드는 다음과 같은 방식으로 호출할 수 있죠.

```
MyClass.getInstance();
```

왜 객체 이름을 사용하지 않고 MyClass라는 클래스 이름을 그냥 사용한 거죠?

getInstance()는 정적 메소드입니다. 클래스 메소드라고 부르기도 하죠. 정적 메소드를 지칭할 때는 클래스 이름을 써야 합니다.

신기하군요. 그럼 이렇게 합쳐 놓으면 어떨까요?
그럼 MyClass의 인스턴스를 만들 수 있지 않나요?

```
public MyClass {
    private MyClass() {}
    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

아! 그렇게 할 수도 있겠네요.

출처: 에릭 프리먼 외. (2023). 헤드 퍼스트 디자인 패턴

그러면 이제 객체를 만드는 또 다른 방법을
말해 주실 수 있을까요?

```
MyClass.getInstance();
```

6.1 프린터 관리자 만들기

코드 6-1

```
public class Printer {
  public Printer() { }

  public void print(Resource r) {
    ...
  }
}
```

그림 6-1 귀중한 프린터

부서에서 함께 사용하는 한 대의 프린터
new Printer()는 한 번만 호출되도록 구현해야 한다면?

프린터 관리자 만들기 (계속)

코드 6-2

```
public class Printer {
  private Printer() { }
  public void print(Resource r) {
    ...
  }
}
```

Printer의 생성자를 private로 하면
외부에서 생성자를 호출하지 못하게 됨

프린터 관리자 만들기 (계속)

코드 6-3

```
public class Printer {
  private static Printer printer = null;
  private Printer() { }

  public static Printer getPrinter() {
    if (printer == null)
      printer = new Printer();

    return printer;
  }

  public v
    ...
  }
}
```

정적변수 printer

*Printer 인스턴스는 일단 하나 만들어야 하므로
이와 같이 인스턴스를 하나 만들어 외부에 제공
*단 하나의 객체만 생성해서 어디서든지 참조할 수 있도록 getPrinter를 static 메소드로 선언

*static이 붙은 메소드나 변수는 인스턴스가 아닌 클래스에 속함

클라이언트 클래스

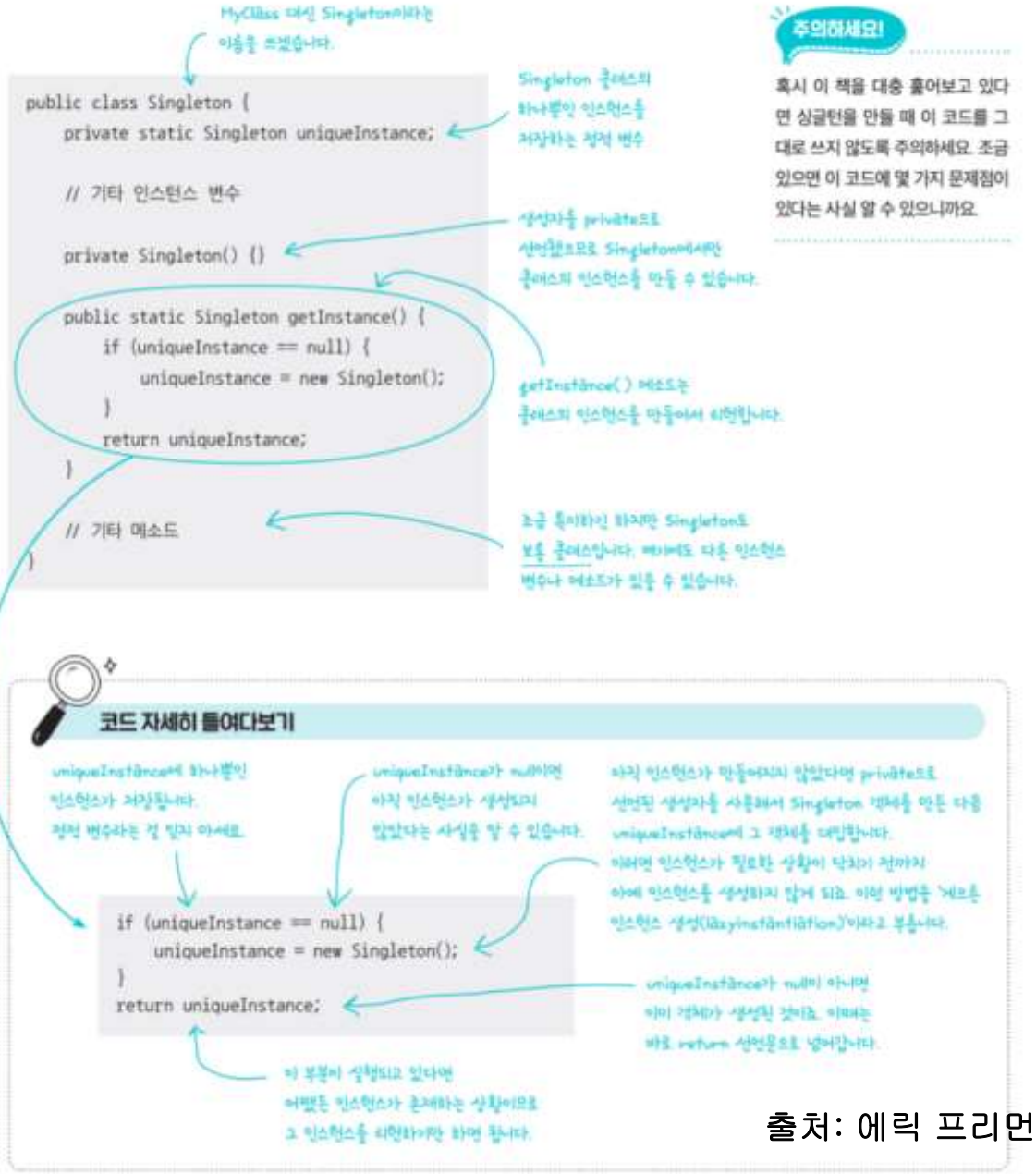
```
public class User {
    private String name;
    public User(String name) {
     this.name = name;
}
    public void print() {
     Printer printer = Printer.getPrinter();
      printer.print(this.name + " print using " + printer.toString() + ".");
 }
}
public class Printer {
   private static Printer printer = null;
    private Printer() { }
   public static Printer getPrinter() {
      if (printer = = null) {
          printer = new Printer(); // Printer 인스턴스 생성
       }
    return printer;
}
```

인스턴스 없이
클래스 이름으로
정적 메소드 접근 가능

```
public void print(String str) {
    System.out.println(str);
                }
}

public class Main {
   private static final int User_NUM = 5;
   public static void main(String[] args) {
   User[] user = new User[User_NUM];
   for (int i = 0; i < User_NUM; i++) {
     user[i] = new User((i + 1) + "-user"); // User 인스턴스 생성
   user[i].print();
    }
  }
}
```

```
1-user print using Printer@27973e9b.
2-user print using Printer@27973e9b.
3-user print using Printer@27973e9b.
4-user print using Printer@27973e9b.
5-user print using Printer@27973e9b.
```

고전적인 싱글턴 패턴 구현법



출처: 에릭 프리먼 외. (2023). 헤드 퍼스트 디자인 패턴

6.2 문제점

❖ 다중 스레드를 사용한다면?

1. Printer 인스턴스가 아직 생성되지 않았을 때 스레드 1이 getPrinter 메서드의 if문을 실행해 이미 인스턴스가 생성되었는지 확인한다. 현재 printer 변수는 null인 상태다.
2. 만약 스레드 1이 생성자를 호출해 인스턴스를 만들기 전 스레드 2가 if문을 실행해 printer 변수가 null인지 확인한다. 현재 null이므로 인스턴스를 생성하는 코드, 즉 생성자를 호출하는 코드를 실행하게 된다.
3. 스레드 1도 스레드 2와 마찬가지로 인스턴스를 생성하는 코드를 실행하게 되면 결과적으로 Printer 클래스의 인스턴스가 2개 생성된다.

경합

기존의 User대신 UserThread로 구현

```
public class UserThread extends Thread {
  public UserThread(String name) {
    super(name);
  }

  public void run() {
    Printer printer = printer.getPrinter();
    printer.print(Thread.currentThread().getName() +
      " print using " + printer.toString() + ".");

    }
  }
```

전부 다 다른 Printer 인스턴스를 사용하여 결과 출력

```
2-thread print using Printer@2cfa930d.
4-thread print using Printer@76cc518c.
5-thread print using Printer@5ffdfb42.
3-thread print using Printer@1b7adb4a.
1-thread print using Printer@1ed2e55e.
```

```
public class Printer {
  private static Printer printer = null;
  private Printer() { }

  public static Printer getPrinter() {
    if (printer == null) {
      try {
        Thread.sleep(1);
      }
      catch (InterruptedException e) { }
      printer = new Printer();
    }
    return printer;
  }

  public void print(String str) {
    System.out.println(str);
  }
}

public class Client {
  private static final int THREAD_NUM = 5;
  public static void main(String[] args) {
    UserThread[] user = new UserThread[THREAD_NUM];
    for (int i = 0; i < THREAD_NUM; i++) {
      user[i] = new UserThread((i + 1) + "-thread");
      user[i].start();
    }
  }
}
```

```
3-thread print using Printer@61b11265.
1-thread print using Printer@2e631b45.
2-thread print using Printer@2831d06c.
4-thread print using Printer@8198da8.
5-thread print using Printer@3541c883.
```

다중 스레드 문제

코드 6-6

```
public class Printer {
  private static Printer printer = null;
  private int counter = 0;
  private Printer() { }

  public static Printer getPrinter() {
    if (printer == null) { // Printer 인스턴스가 생성되었는지 검사
      try {
        Thread.sleep(1);
      }
      catch (InterruptedException e) { }
      printer = new Printer(); // Printer 인스턴스 생성
    }
    return printer;
  }

  public void print(String str) {
    counter++; // 카운터 값 증가
    System.out.println(str+counter);
  }
}
```

그런데 만약
counter변수와 같은 값을
인스턴스가 유지해야 한다면?

인스턴스마다 counter변수를
각각 만들어 유지하므로
생각과는 다른 결과가 발생

```
5-thread print using Printer@46c0b080.2
1-thread print using Printer@321937df.2
2-thread print using Printer@321937df.1
4-thread print using Printer@46c0b080.1
3-thread print using Printer@46c0b080.3
```

해결책

- 사실 프린터 관리자가 다중 스레드 애플리케이션이 아니라면 아무 제가 되지 않음
- 다중 스레드 애플리케이션에서 발생하는 문제를 해결하려면?
  - 정적 변수에 인스턴스를 만들어 바로 초기화하는 방법
  - 인스턴스를 만드는 메서드에 동기화하는 방법

방법 1

정적 변수 printer에 Printer
인스턴스를 만들어 초기화
->초기화 한 번만 실행됨
*프로그램 시작부터 종료까지 없어지지 않고 메모리에 계속 상주함
*클래스에서 생성된 모든 객체에서 참조 가능

코드 6-7

```
public class Printer {
  private static Printer printer = new Printer();

  private Printer() { }

  public static Printer getPrinter() {
    return printer;
  }

  public void print(String str) {

    System.out.println(str);
  }
}
```

not null

if (printer==null) 검사구문 제거됨

```
5-thread print using Printer@9751485.
3-thread print using Printer@9751485.
4-thread print using Printer@9751485.
2-thread print using Printer@9751485.
1-thread print using Printer@9751485.
```

방법 2

Printer 클래스의 객체를 얻는 getPrinter 메소드를 동기화
-> 다중 스레드 환경에서 동시에 여러 스레드가 getPrinter를 소유하는 객체에 접근하는 것을 방지
*결과적으로 Printer 클래스의 인스턴스가 한 개의 인스턴스만 생성

코드 6-8

```
public class Printer {
  private static Printer printer = null;
  private Printer() { }

  public synchronized static Printer getPrinter() { // 메서드 동기화
    if (printer == null) {
      printer = new Printer();
    }
    return printer;
  }

  public void print(String str) {
    System.out.println(str);
  }
}
```

```
4-thread print using Printer@1a6ca720.
1-thread print using Printer@1a6ca720.
2-thread print using Printer@1a6ca720.
3-thread print using Printer@1a6ca720.
5-thread print using Printer@1a6ca720.
```

문제

코드 6-9

```
public class Printer {
  private static Printer printer = null;
  private int counter = 0;
  private Printer() { }

  public synchronized static Printer getPrinter() {
    if (printer == null) {
      printer = new Printer();
    }
    return printer;
  }

  public void print(String str) {
    counter++;
    System.out.println(str+counter);
  }
}
```

변수 counter에 여러 스레드가 동시 접근해서 갱신하므로 아래와 같은 결과가 발생

```
1-thread print using Printer@5ffdfb42.2
3-thread print using Printer@5ffdfb42.5
2-thread print using Printer@5ffdfb42.4
5-thread print using Printer@5ffdfb42.3
4-thread print using Printer@5ffdfb42.2
```

해결

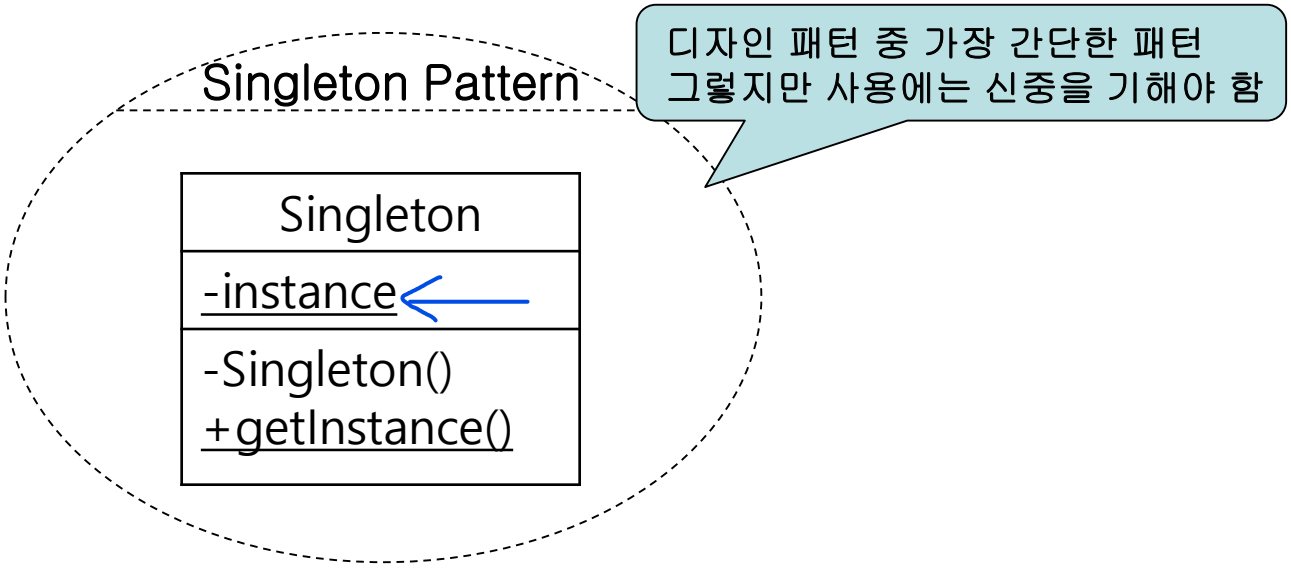
코드 6-10

```
public class Printer {
  private static Printer printer = null;
  private int counter = 0;
  private Printer() { }

  public synchronized static Printer getPrinter() {
    if (printer == null) {
      printer = new Printer();
    }
    return printer;
  }

  public void print(String str) {
    synchronized(this) { // 오직 하나의 스레드만 접근을 허용함
      counter++;
      System.out.println(str+ counter);
    }
  }
}
```

```
3-thread print using Printer@700abc1c.1
4-thread print using Printer@700abc1c.2
1-thread print using Printer@700abc1c.3
2-thread print using Printer@700abc1c.4
5-thread print using Printer@700abc1c.5
```

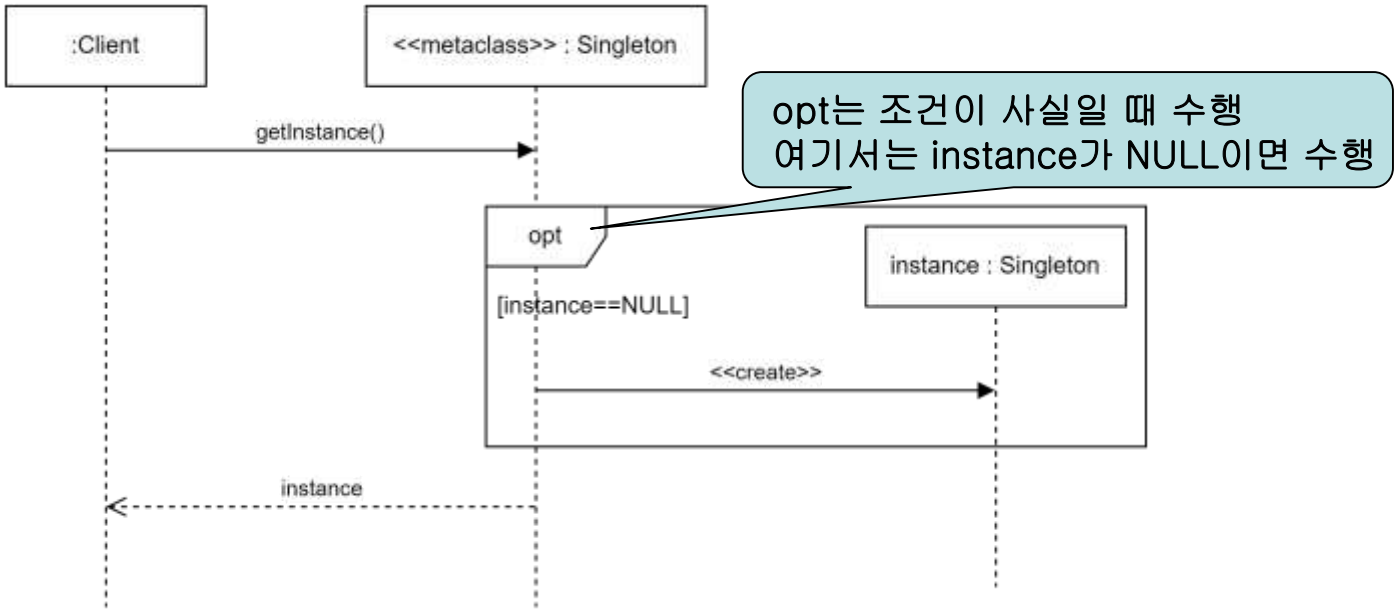
싱글턴(Singleton) 패턴

- 싱글턴 패턴(Singleton Pattern)은 인스턴스가 오직 하나만 생성되는 것을 보장하고 어디서든 이 인스턴스에 접근할 수 있도록 하는 디자인 패턴



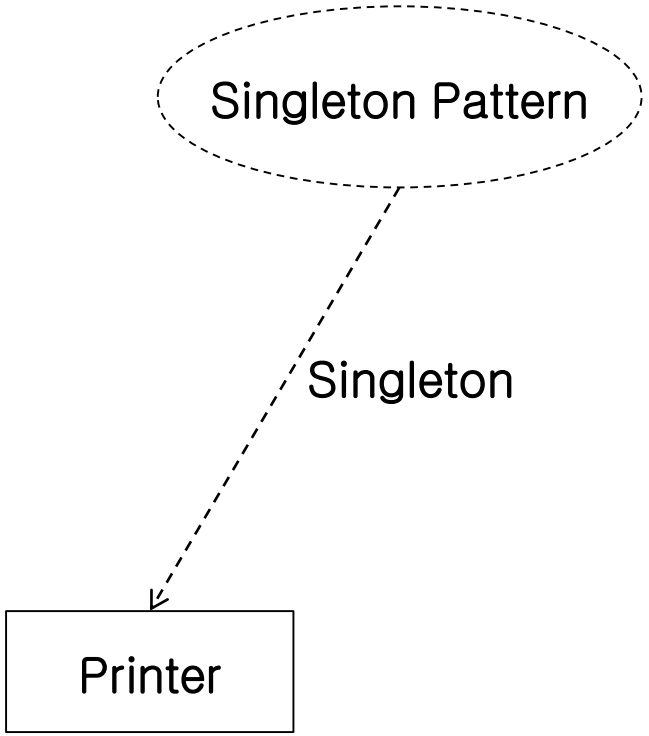
*Singleton: 하나의 인스턴스만을 생성하는 책임이 있으며 getInstance 메소드를 통해 모든 클라이언트에게 동일한 인스턴스를 반환하는 작업을 수행

싱글턴 패턴의 순차 다이어그램



클라이언트가 싱글턴 클래스에 getInstance 메소드를 통해 객체 생성을 요청하면
이미 객체가 있는 경우에는 객체 반환
처음 생성 시에는 생성자를 호출해서 객체를 생성하고 반환

싱글턴 패턴과 프린터 관리자 예제



싱글턴 패턴과 정적 클래스

- 싱글턴 패턴을 사용하지 않고 정적 메소드로만 이루어진 정적 클래스를 사용해도 동일한 효과를 얻을 수 있음

```java
public class Printer {  4 usages
    private static int counter=0;  2 usages
    public synchronized static void print(String str){
        counter++;
        System.out.println(str+counter);
    }
}
```

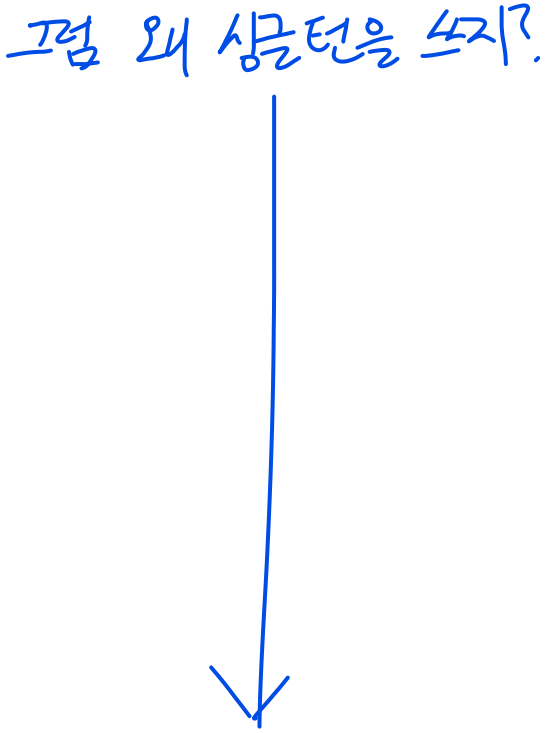
```java
public class UserThread extends Thread{  3 usages
    public UserThread(String name){  1 usage
        super(name);
    }

    public void run(){
        Printer.print(Thread.currentThread().getName() + " print using.");

    }
}
```

```java
public class Main {
    private static final int THREAD_NUM = 5;  2 usages
    public static void main(String[] args) {
        UserThread[] user = new UserThread[THREAD_NUM];
        for (int i = 0; i < THREAD_NUM; i++) {
            user[i] = new UserThread( name: (i + 1) + "-thread");
            user[i].start();
        }
    }
}
```

```
1-thread print using.1
2-thread print using.2
4-thread print using.3
5-thread print using.4
3-thread print using.5
```

싱글턴 패턴과 정적 클래스

- 정적 클래스 이용과 싱글턴 패턴의 차이는 객체를 전혀 생성하지 않고 메소드를 사용한다는 점임
- 정적 메소드를 사용할 때는 실제로 컴파일 시에 바인딩되는 인스턴스 메소드를 사용하는 것보다 성능 면에서 우수함



정적 클래스를 사용할 수 없는 경우

- 인터페이스를 구현해야 할 때
  - 정적 메소드는 인터페이스에서 사용할 수 없음
  - 인터페이스를 사용하는 주된 이유는 대체 구현이 필요한 경우임. 모의 객체를 사용하여 단위 테스트 수행 시 매우 중요

```
public interface Printer {  1 usage
    public static void print(String str);
}
```

허용되지 않음

```
public class RealPrinter315 implements Printer{  no usages
    public synchronized static void print(String str){

    }
}
```

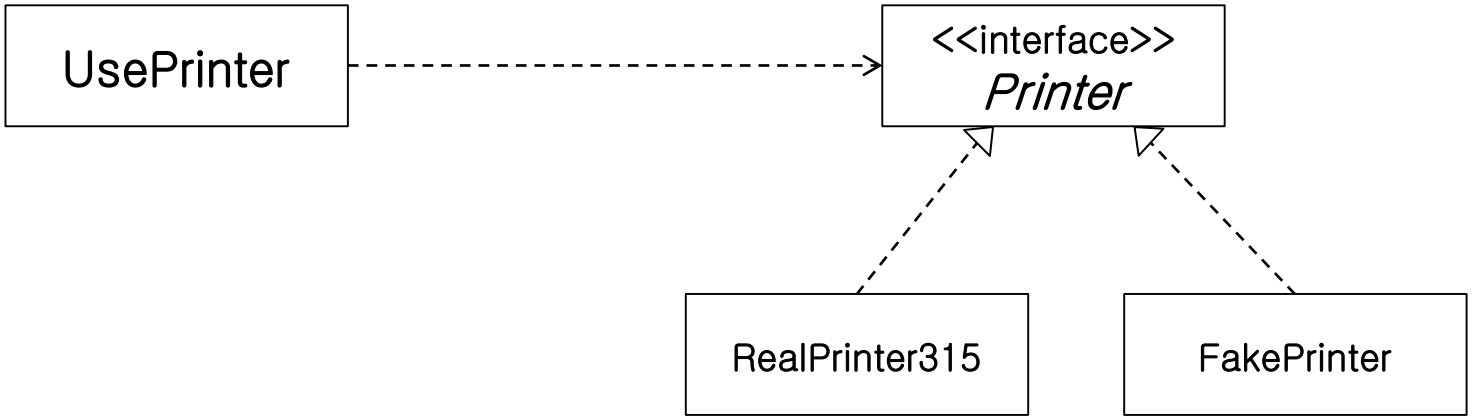
프린터 구현 후 테스트를 하려면

- 다시 프린터 예제로 돌아와서, 실제로 출력을 해야 할 프린터가 아직 준비되지 않았거나 준비가 되었다고 하더라도 테스트 시 결과가 올바른지 확인하기 위해 매번 출력물을 검사하는 것은 번거로운 일
- 테스트 실행 시간에 병목 현상이 나타날 수도 있음
  - 단위 테스트는 빠르게 실행되어야 함
- UsePrinter클래스가 다음과 같이 주어졌을 때, 단위 테스트를 실행할 때는 실제 프린터를 테스트용 가짜 프린터 객체로 대체

UsePrinter

```
public class UsePrinter{
  public void doSomething(){
    String str;
    ...
    str = ...;

    RealPrinter315.print(str);
    ...
  }
}

public class RealPrinter315 {
  public synchronized static void print(String str) {
    ...
  }
}
```

UsePrinter의 변경된 설계



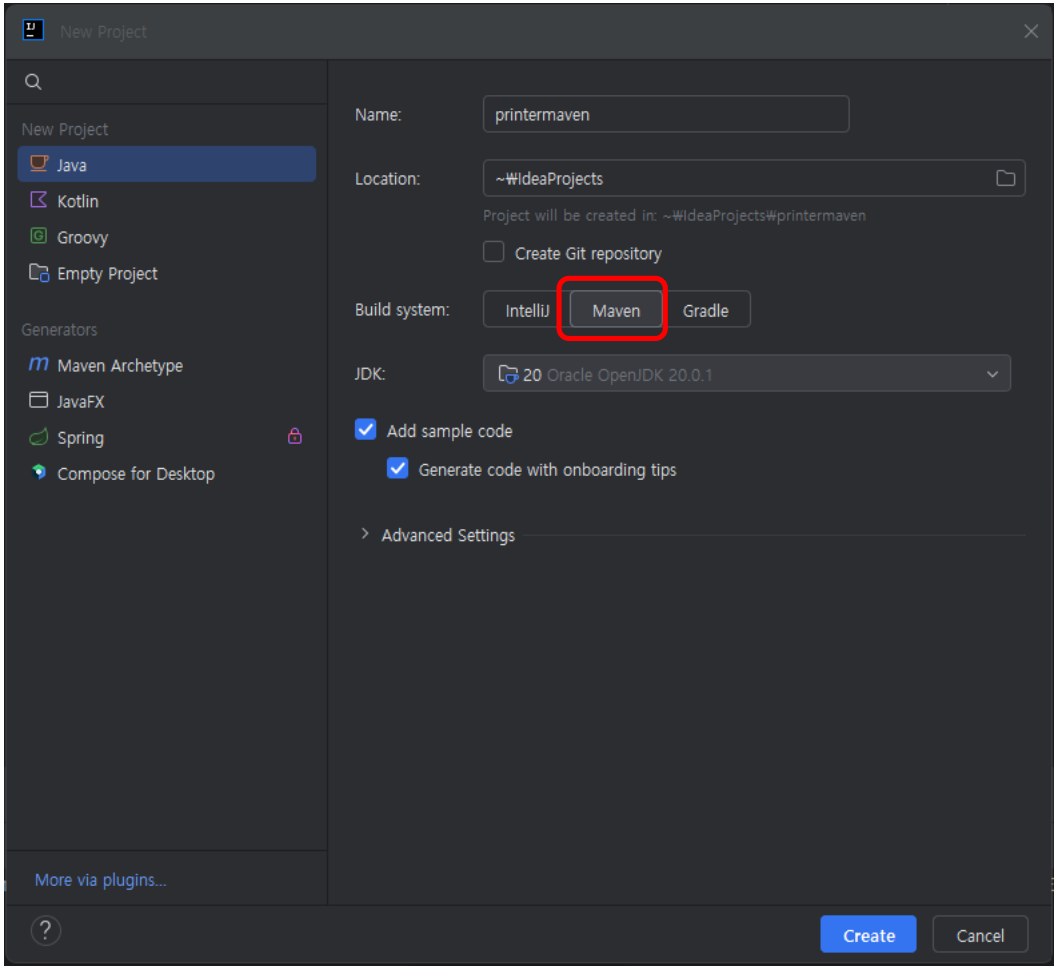
변경된 코드

```java
public class UsePrinter {  no usages
    public void doSomething(Printer printer){
        String str;

        str="테스트 페이지 입니다.";

        printer.print(str);
    }

}
```

```java
public class RealPrinter315 implements Printer{  1 usag
    private static Printer printer=null;  3 usages
    private RealPrinter315(){}  1 usage
    public synchronized static Printer getPrinter(){
        if (printer==null){
            printer=new RealPrinter315();
        }
        return printer;
    }
    @Override  2 usages
    public void print(String str) {
        //실제 프린터 하드웨어 조작하는 코드
    }
}
```

```java
public class FakePrinter implements Printer{
    private String str;  2 usages
    @Override  2 usages
    public void print(String str) {
        this.str=str;
    }

    public String get(){  no usages
        return str;
    }
}
```

Test Project 생성(Maven)



JUnit 설치하기

- https://mvnrepository.com/search?q=junit
- 맨 위에 있는 JUnit Jupiter API 선택
- 5.11.0-RC1 선택(다른 버전 선택해도 무방함)

MVN Repository에서 Buildr 복사

Home » org.junit.jupiter » junit-jupiter-api » 5.11.0-RC1

JUnit Jupiter API » 5.11.0-RC1

JUnit Jupiter is the API for writing tests using JUnit 5.

License	EPL 2.0
Categories	Testing Frameworks & Tools
Tags	quality junit testing api
HomePage	https://junit.org/junit5/
Date	Jul 31, 2024
Files	pom (3 KB) jar (211 KB) View All
Repositories	Central
Ranking	#22 in MvnRepository (See Top Artifacts) #3 in Testing Frameworks & Tools
Used By	16,462 artifacts

Note: There is a new version for this artifact

New Version	5.11.1

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

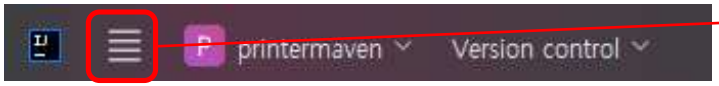
```
# https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api
'org.junit.jupiter:junit-jupiter-api:jar:5.11.0-RC1'
```

두번째 줄의 org부터 RC-1까지 복사
(따옴표 제외)

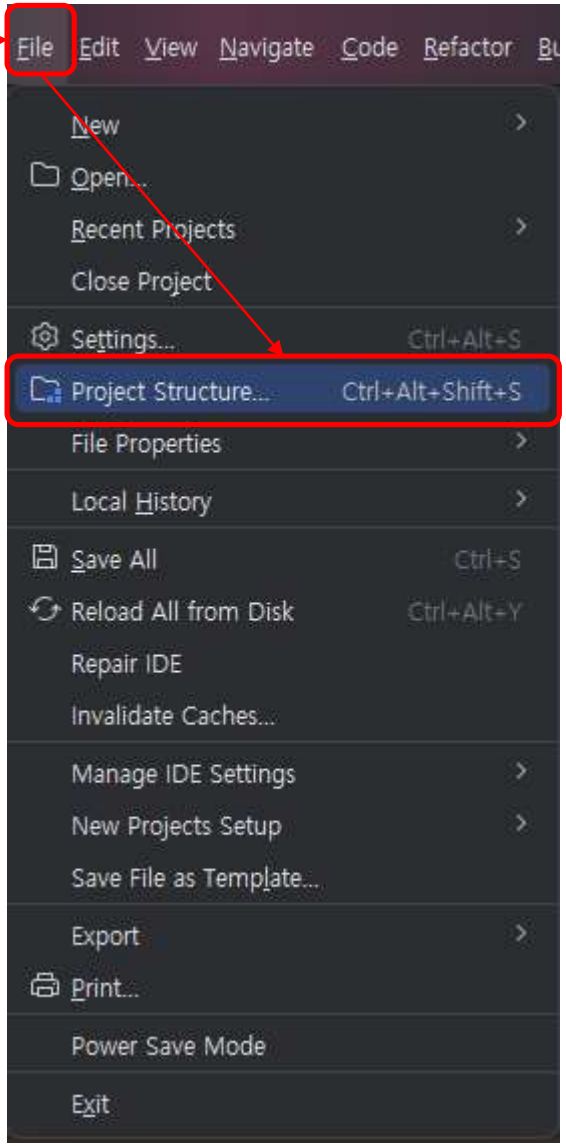
Include comment with link to declaration

[File]–[Project Structure]

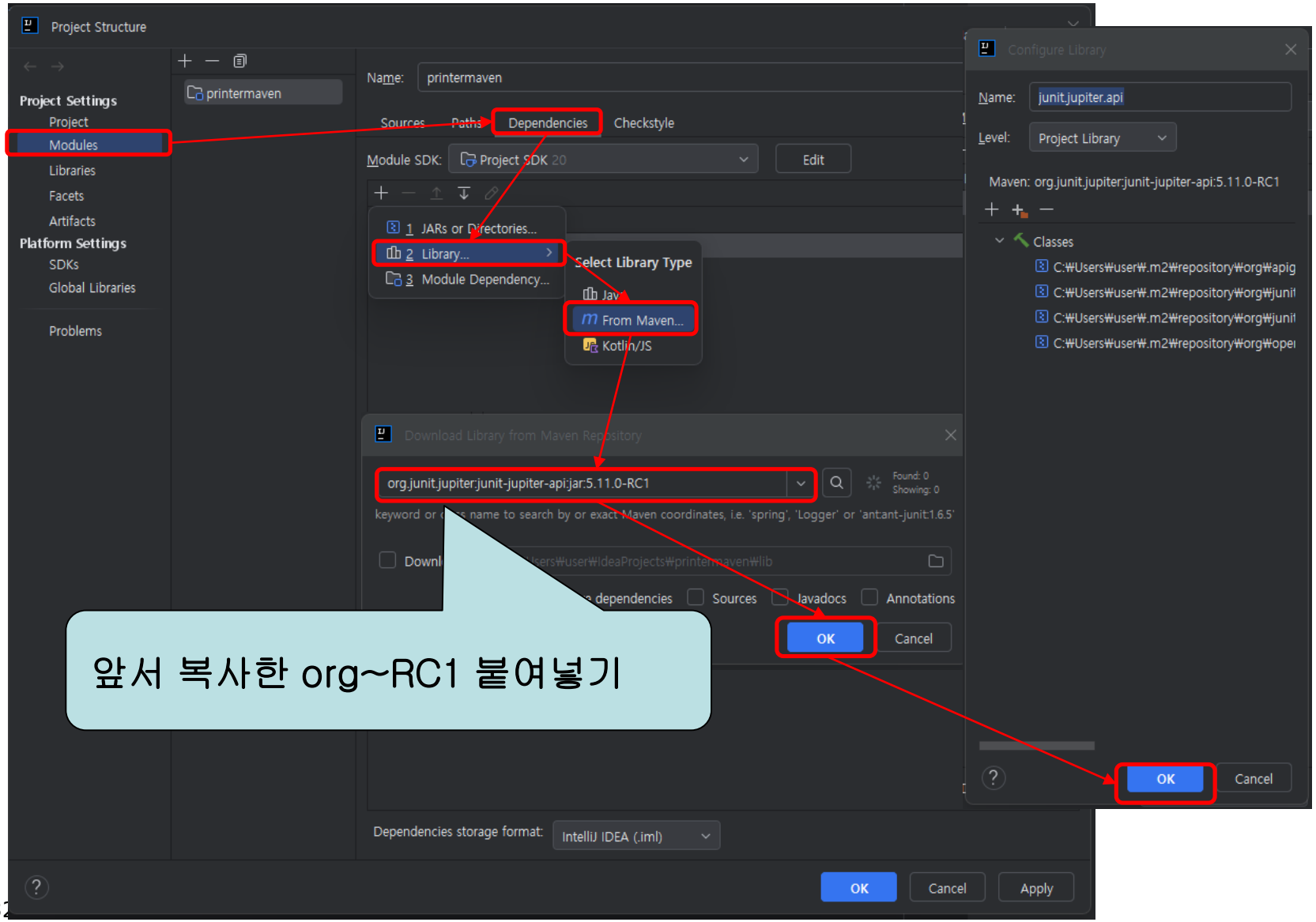
❖ 왼쪽 상단의



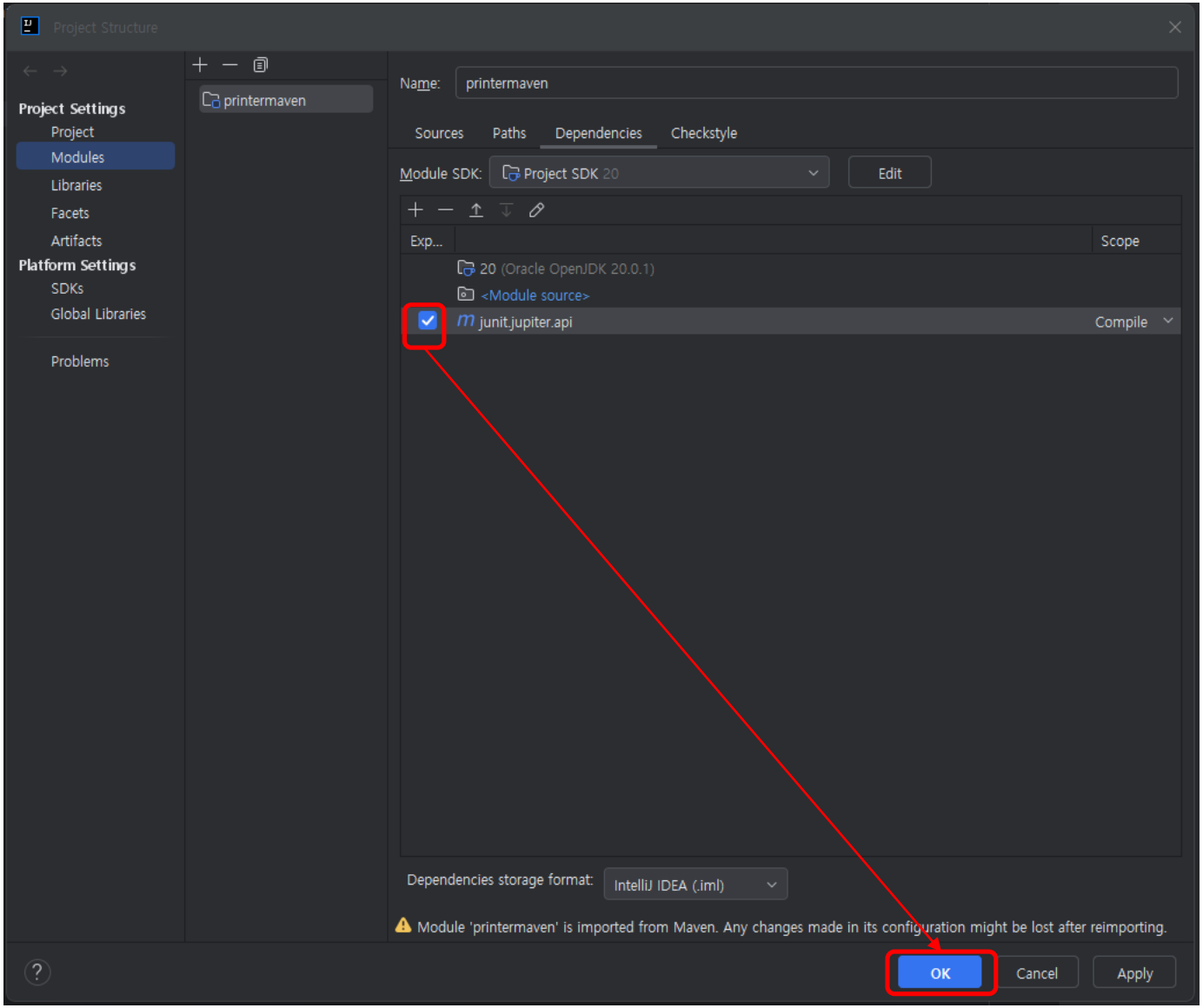
– 아이콘 클릭



Dependencies에서 JUnit 관련 체크



다시 OK 클릭


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
printermaven
Name: printermaven
Sources
Paths
Dependencies
Checkstyle
Module SDK: Project SDK 20
Edit
Exp...
Scope
20 (Oracle OpenJDK 20.0.1)
<Module source>
junit.jupiter.api
Compile
Dependencies storage format: IntelliJ IDEA (.iml)
Module 'printermaven' is imported from Maven. Any changes made in its configuration might be lost after reimporting.
OK
Cancel
Apply

코드 입력

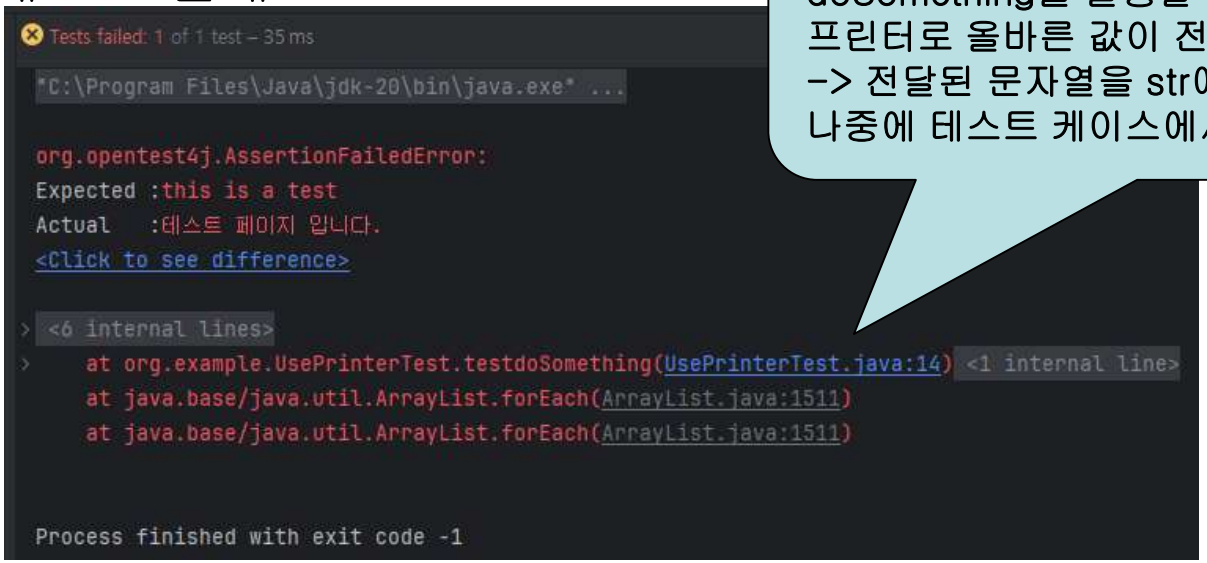
- 슬라이드 24의 코드 전체와 아래 코드 입력
- 맨 윗줄의 package org.example;은 실행환경에 따라 없을 수도 있으며 패키지 명에 따라 바뀔 수도 있음

Main이 아님에도
JUnit으로 인해
이 부분만 실행 가능

```java
package org.example;


import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class UsePrinterTest{
    @Test
    public void testdoSomething(){
        FakePrinter fake = new FakePrinter();
        UsePrinter u = new UsePrinter();
        u.doSomething(fake);
        assertEquals( expected: "this is a test", fake.get());
    }
}
```

결과

❖ 테스트 실패



FakePrinter는 실제 출력이 아닌
doSomething을 실행할 때
프린터로 올바른 값이 전달되었는지 확인해야 함
-> 전달된 문자열을 str에 저장하고
나중에 테스트 케이스에서 get 메소드를 통해 확인

❖ 테스트 성공

1. assertEquals의 “this is a test”를
“테스트 페이지 입니다.”로 바꾸거나
2. doSomething의 str을
“this is a test”로 바꾸면 테스트 성공

static setter 메소드를 사용하는 방법(1)

```java
package org.example;

public class FakePrinter extends Printer{ 3 usages
    private String str; 1 usage
    @Override 1 usage
    public void print(String str) {
        System.out.println("Fake print: " + str);
    }

    public String get(){ 1 usage
        return str;
    }
}
```

```java
package org.example;

public class UsePrinter { 2 usages
    public void doSomething(){ 1 usage
        String str;
        str="";
        PrinterFactory.getPrinterFactory().getPrinter().print(str);
    }

    public void print(String str){ no usages

    }
}
```

```java
public class FakePrinterFactory extends PrinterFactory{ no usage
    public Printer getPrinter(){ 1 usage
        return new FakePrinter();
    }
}
```

```java
package org.example;

public class Printer { 7 usages 1 inherit
    public Printer(){} 3 usages

    public void print(String str) {
        System.out.println(str);
    }
}
```

static setter 메소드를 사용하는 방법(2)

```java
package org.example;

public class PrinterFactory {  7 usages  1 inheritor
    private static PrinterFactory printerFactory=null;  4 usages
    private Printer printer;  1 usage
    protected PrinterFactory(){  2 usages
        this.printer = new Printer();
    }

    public synchronized static PrinterFactory getPrinterFactory(){  1 usage
        if (printerFactory==null){
            printerFactory=new PrinterFactory();
        }
        return printerFactory;
    }

    public static void setPrinterFactory(PrinterFactory p){//정적 setter 메소드
        printerFactory=p;
    }



    public static void setPrinter(Printer printer) {  1 usage
        printer = printer; // 사용자 정의 프린터 설정
        printer.print("check fake or not "); //printer=printer 제대로 되나 확인
    }

    public Printer getPrinter(){  1 usage  1 override
        return new Printer();
    }
}
```

```java
package org.example;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class DoSomethingTest {
    @Test
    public void testdoSomething() {
        FakePrinter fakePrinter = new FakePrinter();
        PrinterFactory.setPrinter(fakePrinter); // FakePrinter 설정
        UsePrinter u = new UsePrinter();
        u.doSomething();
        //assertEquals("this is a test", fakePrinter.get());
    }

}
```

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Fake print: check fake or not


Process finished with exit code 0
```

싱글턴 패턴의 여러 형태

- Eager Initialization
- Static block initialization
- Lazy initialization
- Thread safe initialization
- Double-Checked Locking
- Bill Pugh Solution
- Enum 이용

출처 : 싱글톤(Singleton) 패턴 - 꼼꼼하게 알아보자, https://url.kr/fmb75g

싱글턴 패턴의 장점

- 객체 생성 비용 절약 가능
  - 메모리 낭비를 방지
- 전역변수를 쓰지 않아도 접근성이 좋은 코드 생성 가능(전역 상태 유지)
- 데이터 일관성을 보장함

싱글턴 패턴의 단점

- 클래스 사이에 의존성이 높아짐(결합도가 높아짐)
  - 싱글턴 내부 인스턴스 변경 시 해당 인스턴스 참조하는 모든 클래스 수정 필요
- 생성자가 private이라 상속이 어렵고 이는 다형성을 적용하지 못하게 함
- 단위 테스트가 어려움
  - 독립적인 테스트를 위해서는 전역에서 상태를 공유하는 인스턴스 상태를 매번 초기화 해야 함
- 멀티스레드 환경에서 싱글턴 패턴 사용 시 성능에 영향을 미칠 수 있음
- 한 클래스가 여러 책임 담당 시 단일 책임 원칙(SRP)을 위배할 수 있음. 결과적으로 개방 폐쇄 원칙(OCP)도 위배되는 상황을 만들어 내기도 함
- 객체 생성 / 소멸 시점에 대한 유연성이 좋지 않음
- 의존성 주입 패턴(결합도를 낮추는 패턴) 사용과 호환되지 않음

싱글턴 패턴을 사용하려 한다면?

- 편리함이 목적이 아닌 클래스의 인스턴스가 언제나 단 한 개만 존재하도록 보장하는 것을 목적으로 해야 함
- 제한적으로 사용해야 하는 패턴이다
- 불필요하게 싱글턴 패턴을 많이 사용하면 나중에 리팩토링 지옥에 빠질 수 있음

생각해봅시다

```java
public class Theme {
    private static Theme instance;
    private String themeColor;

    private Theme() {
        this.themeColor = "light"; // Default theme
    }

    public static Theme getInstance() {
        if (instance == null) {
            instance = new Theme();
        }
        return instance;
    }

    public String getThemeColor() {
        return themeColor;
    }
    public void setThemeColor(String themeColor) {
        this.themeColor = themeColor;
    }
}
```

```java
public class Button {
    private String label;

    public Button(String label) {
        this.label = label;
    }

    public void display() {
        String themeColor = Theme.getInstance().getThemeColor();
        System.out.println(
            "Button [" + label + "] displayed in " + themeColor + " theme."
        );
    }
}

public class TextField {
    private String text;

    public TextField(String text) {
        this.text = text;
    }

    public void display() {
        String themeColor = Theme.getInstance().getThemeColor();
        System.out.println(
            "TextField [" + text + "] displayed in " + themeColor + " theme."
        );
    }
}

public class Label {
    private String text;

    public Label(String text) {
        this.text = text;
    }

    public void display() {
        String themeColor = Theme.getInstance().getThemeColor();
        System.out.println(
            "Label [" + text + "] displayed in " + themeColor + " theme."
        );
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Button button = new Button("Submit");
        TextField textField = new TextField("Enter your name");
        Label label = new Label("Username");

        button.display();
        textField.display();
        label.display();

        Theme.getInstance().setThemeColor("dark");

        button.display();
        textField.display();
        label.display();
    }
}
```

위의 코드를
멀티쓰레드 환경에서
실행 가능하도록
수정해보세요.

출처 : 얄팍한 코딩사전, https://www.youtube.com/watch?v=1c_wLi0_B1A

만들어봅시다

- 스피커 클래스를 만들되, 단 하나의 객체만 생성하도록 구현하려고 합니다. (안 그러면 컴퓨터에서도, 휴대폰에서도 스피커랑 연결을 해서 여기저기서 노래를 틀어댈 거에요)
- 고려해야 할 사항이 무엇인지 생각해보고 싱글턴 패턴을 이용하여 구현해보세요.
- 한발 더 나아가 멀티쓰레드 환경에서 실행되도록 구현해보세요.

과제 1

- 코딩라운지(공학관 B동 115호)에 방문하여 상주하고 있는 코치에게 평소에 공부하다가 궁금했던 것 질문하고 ①질의응답내용 작성하여 제출
  - 운영시간은 월~금 오후 2시~6시
- 방문했다는 것을 확인할 수 있는 ②인증사진 첨부
  - 얼굴이 다 나오지 않아도 되며 눈만 나오도록 해서 촬영해도 됨
    - B동 115호임을 확인할 수 있게 사진 촬영(필수)