

# Java Unit Test

Unit Test

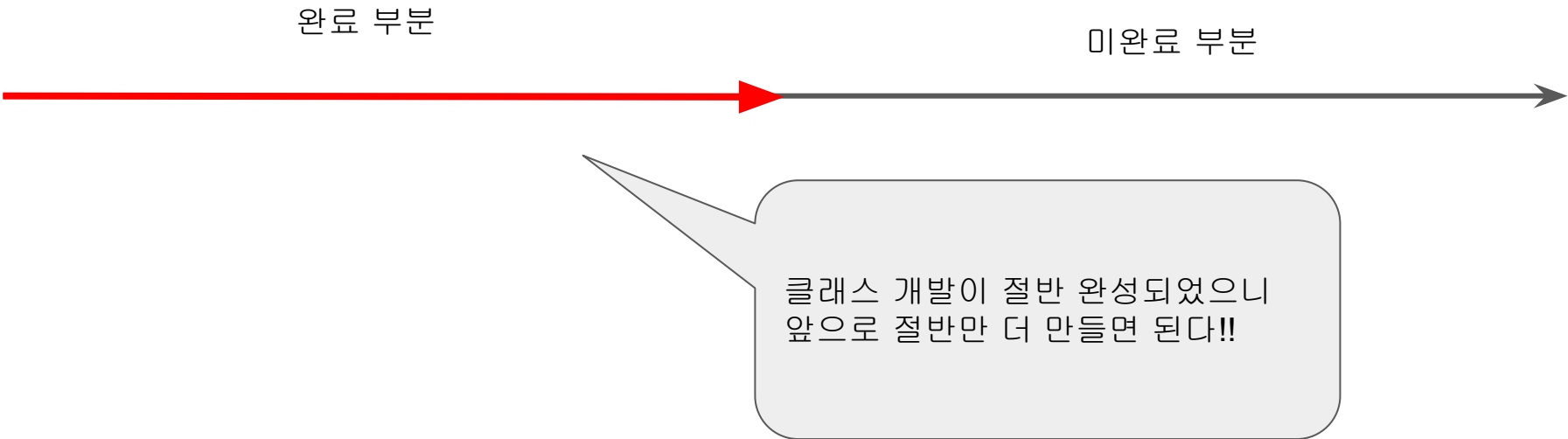


# 잘못된 개발의 완성 기준

클래스를 작성했다고 완성했다고 말할 수 있는가?

완료 부분

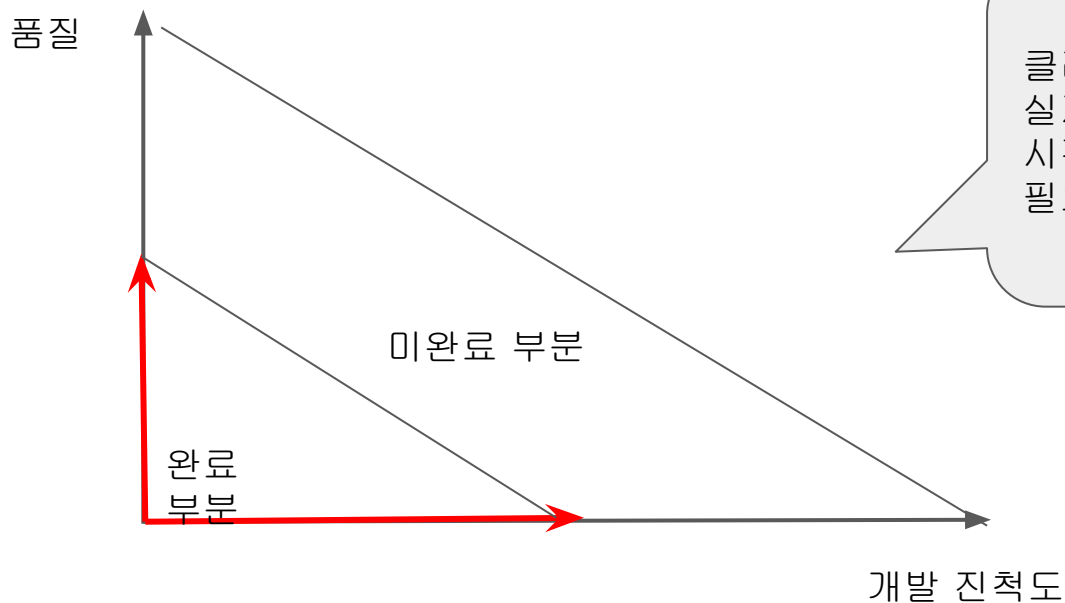
미완료 부분



클래스 개발이 절반 완성되었으니  
앞으로 절반만 더 만들면 된다!!

# 올바른 완성의 기준

품질을 생각한 개발



클래스 개발은 절반 작성했더라도  
실제로 완성까지는 4배 정도의  
시간이  
필요할 수 있다

# 테스트를 통한 품질 향상

테스트를 하는 방법들

=> 수동 테스트 : 인간이 하는 테스트

=> 단위 테스트 : 1개 클래스를 테스트

=> 통합 테스트 : 여러개 연관된 클래스를 함께 테스트

# 단위(Unit) 테스트

단위 테스트란?

특정 모듈이 의도한 대로 잘 작동하는가를 테스트하는 것

# 단위 테스트 작성 방법

테스트할 계좌 클래스를 작성

```
public class Account {  
    private String owner;  
    private int balance;  
  
    // getter, setter  
  
    public void Account(String owner, int balance) {  
        owner = owner;  
        balance = balance;  
    }  
  
    public void transfer(Account dest, int amount) {  
        dest.setBalance(dest.getBalance() + amount);  
        balance -= amount;  
    }  
}
```

테스트를 통해 에러를 찾아 봅시다

# main 메서드를 가지는 테스트 클래스를 작성

```
public class AccountTest {  
    public static void main(String[] args) {  
        testInit();  
    }  
  
    public static void testInit() {  
        System.out.println("==== 테스트 시작 ====");  
        Account account = new Account("홍길동", 30000);  
        if (!account.equals("홍길동")) {  
            System.out.println("이름이 다름");  
        }  
        if (30000 != account.getBalance()) {  
            System.out.println("잔액이 다름");  
        }  
        System.out.println("==== 테스트 완료 ====");  
    }  
}
```

# 테스트 케이스

가능한 모든 가능성의 범위를 테스트하는 것이 좋은 테스트 케이스이다



# 테스트의 장점

- 장애에 관한 신속한 피드백
- 개발 주기에서 조기 장애 감지
- 회귀에 신경 쓸 필요 없이 코드를 최적화할 수 있도록 하는 더 안전한 코드 리팩터링
- 기술적 문제를 최소화하는 안정적인 개발 속도

# 단위(Unit) 테스트가 필요한 경우

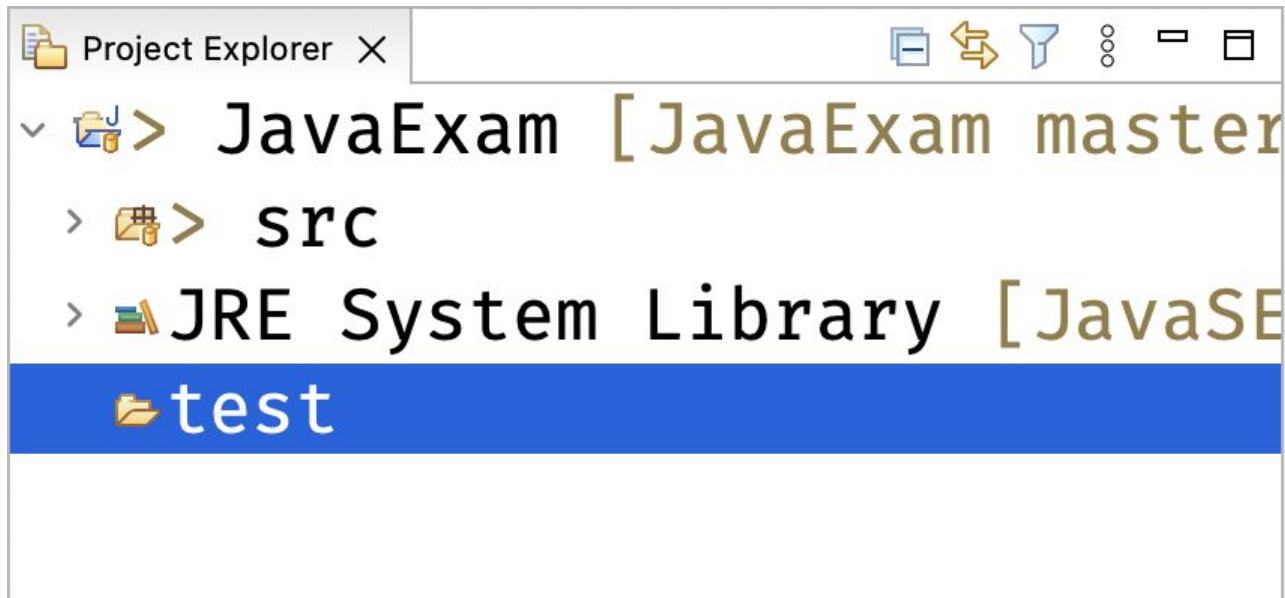
- DB
  - 스키마가 변경되는 경우
  - 모델 클래스가 변경되는 경우
- Network
  - 예측한 데이터가 제대로 들어오는지
- 데이터 검증
  - 예측한 데이터를 제대로 처리하고 있는지

# JUnit

Java 용 Unit Test 용 라이브러리

# 테스트 환경 준비

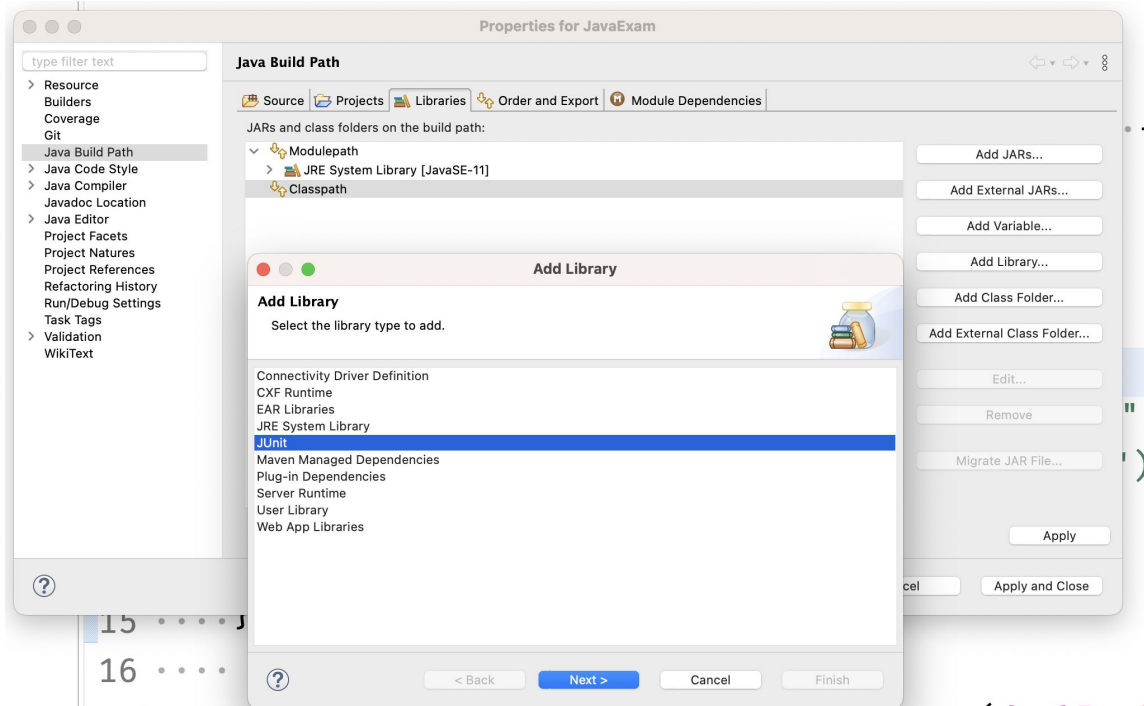
## 테스트 폴더 작성



# 테스트 환경 준비

project > properties > Java Build Path > Libraries 탭 > Classpath > Add Library >

JUnit 추가 (JUnit4)



테스트할 클래스 우클릭 New > Other

com.survivalcoding

Main.java

JRE System

JUnit 4

test

New

Go Into

Open Type Hierarchy

F4

Show In

⌘ W

Open

F3

Open With

Copy

⌘ C

Copy Qualified Name

Paste

⌘ V

Delete

⌘ X

Build Path

Source

⌘ S

Refactor

⌘ T

Import...

Project...

File

Folder

SQL File

Annotation

Class

Enum

Interface

Package

Record

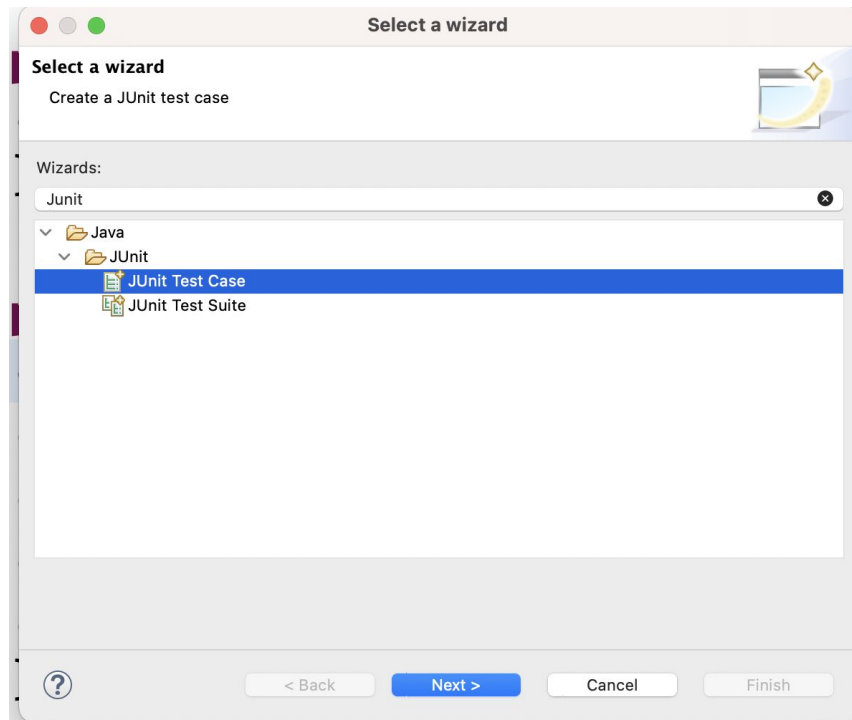
Source Folder

Example...

Other...

⌘ N

## JUnit Test Case 선택 > Next



# Package 이름 수정

테스트 코드가 본 코드와 섞이지 않도록

New JUnit Test Case

JUnit Test Case

Type already exists.

☐ New JUnit 3 test ☒ New JUnit 4 test ☐ New JUnit Jupiter test

Source folder: JavaExam/src Browse...

Package: com.survivalcoding.test Browse...

Name: MainTest

Superclass: java.lang.Object Browse...

Which method stubs would you like to create?

☐ @BeforeClass setUpBeforeClass() ☒ @AfterClass tearDownAfterClass()

☐ @Before setUp() ☐ @After tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test: com.survivalcoding.Main Browse...

? < Back Next > Cancel Finish



# Test 코드 작성 예

assertXXX 코드로 예측 값과 실제 값을 체크

```
import static org.junit.Assert.*;
```

```
public class MainTest {
```

```
    @Test
```

```
    public void testEmailStringStringString() {
```

```
        String result = Main.solution2(10, 20);
```

```
        assertEquals("10.20", result);
```

```
    }
```

```
}
```

특정 테스트가 특정 예외가 발생되어야 하는 것을  
테스트

```
@Test(expected = IllegalArgumentException.class)
public void throwsExceptionWithTwoCharName() {
    ...
}
```

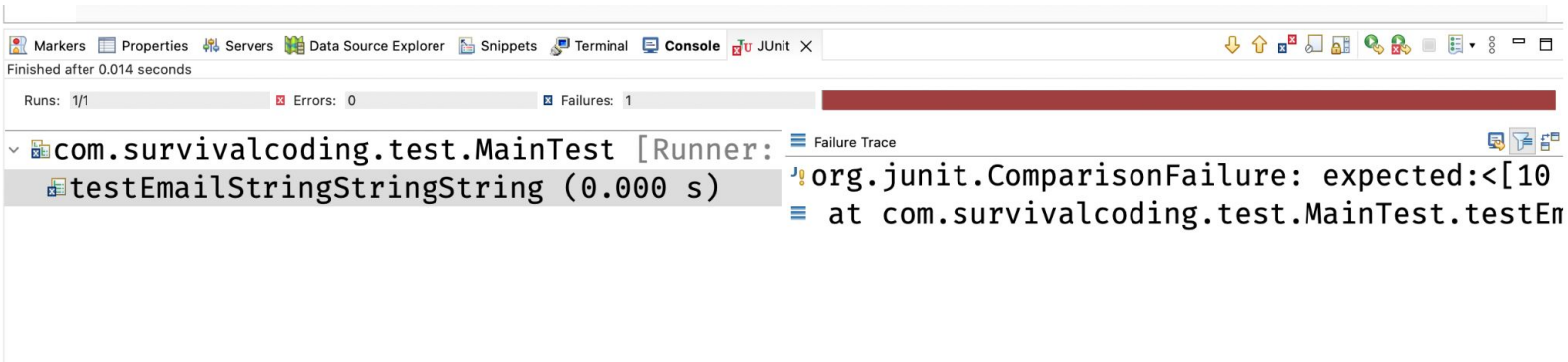
# Text 코드 실행

우클릭 &gt; Run As &gt; JUnit Test

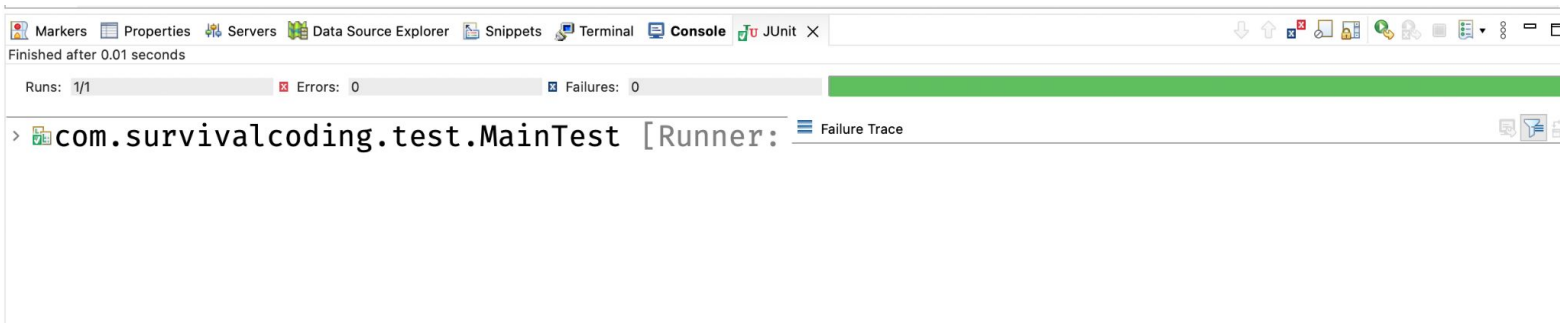
The screenshot shows an IDE with a C# code editor. The code defines a `PackageComSurvivalCoding.Test` namespace containing a `Test` class with a `Test` method. The `Test` method calls `StringStringStrir.Main.solution2(10, "20", result);`. A context menu is open over the code, listing various actions like "Undo Typing", "Save", "Open Declaration", "Quick Outline", "Cut", "Copy", "Paste", "Quick Fix", "Source", "Refactor", "Local History", "References", "Declarations", "Add to Snippets...", "Coverage As", "Run As", "Debug As", and "Profile As". The "Run As" option is highlighted, and a sub-menu is open showing "JUnit Test" and "Run Configurations...". The "Run Configurations..." option is selected.

# 결과 확인

## 실패



## 성공



# Mock 객체 활용

- Interface를 활용하여 가짜 객체를 작성하여 테스트

# 연습문제 1

다음 코드를 검사하는 JUnit 테스트 클래스 **BankTest** 클래스를 작성하시오.

```
public class Bank {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        if (name.length() <= 3) {  
            throw new IllegalArgumentException("이름이 잘못 되었음");  
        }  
        this.name = name;  
    }  
}
```

## 연습문제 2

다음과 같은 Counter 클래스가 있습니다. 이 클래스는 1씩 증가하는 기능이 필요할 때 쉽게 활용할 수 있습니다.

1씩 감소하는 카운터가 필요할 때를 위해, 1씩 감소하는 기능을 가지는 클래스 DownCouner를 추가하시오.

```
class Counter {  
    private int count = 0;  
  
    // getter, setter  
  
    public void increment() {  
        count++;  
    }  
}
```

## 연습문제 3

Counter 클래스의 Unit 테스트 코드를 작성합니다.

테스트는 public 메소드 전부를 테스트 합니다.



## 연습문제 4

카운터는 항상 카운트를 하는 동작 하나를 제공합니다.

`Counter`와 `DownCounter` 의 동작 메소드 `increment()` 나 `decrement()` 는 적절한 이름이 아닌 것 같습니다. 공통의 카운트 느낌을 가지는 `count()` 이름으로 수정하는 것이 좋아 보입니다. 수정 하시오.

## 연습문제 5

공통의 메소드를 가지는 2개의 클래스는 하나의 공통 인터페이스를 구현할 수 있습니다.

**Counter** 인터페이스를 구현하면 `int count()` 메소드를 가지고 있으며 카운트 동작을 수행한 후에 현재의 카운트 값을 리턴합니다.

**Counter** 클래스는 적절한 이름으로 수정하고 대신 **Counter** 인터페이스를 구현하도록 수정하시오.

**DownDounter** 또한 **Counter** 인터페이스를 구현하도록 수정하시오.