# JUnit 세미나

전아름

# Contents

- 단위 테스트 vs 통합 테스트
- TDD 기본 개념
- JUnit 개념
- Mockito 기반 JUnit 테스트

# 단위 테스트 vs 통합 테스트

| | **단위 테스트 (Unit Test)** | **통합 테스트 (Integration Test)** |
|---|---|---|
| 테스트 단위 | 하나의 모듈 (어플리케이션 내 하나의 **기능** 또는 **메소드**) | 통합된 모듈 |
| 목적 | 하나의 기능이 올바르게 동작하는지 독립적인 테스트 | 통합된 모듈들이 올바르게 연계되어 동작하는지 검증 |

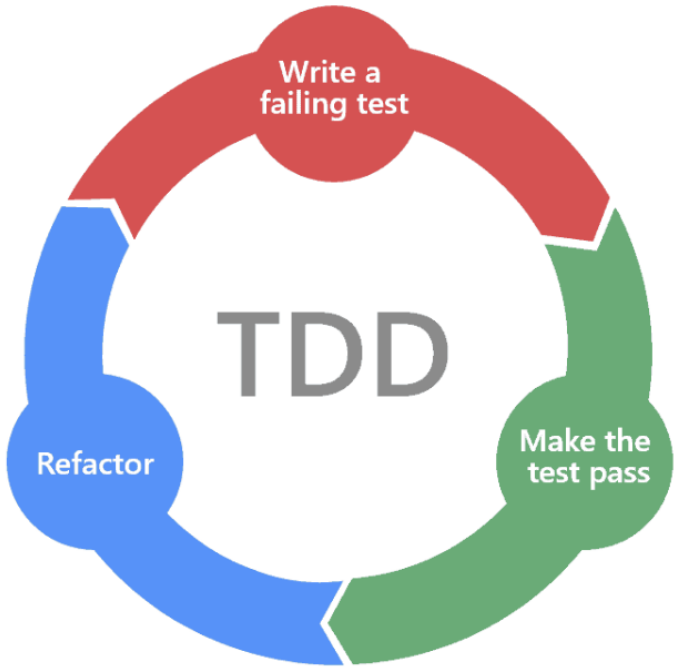
# TDD (Test-Driven Development)

: **단위 테스트** 코드를 먼저 작성하는 개발 방법론

**TDD 사상**
**- "내가 설계한 모든 잘못된 점을 수정하면 비로소 올바름이 된다."**

# TDD의 개발주기 : **Red–Green-Refactor 사이클**

**1) Red 단계** : 실패하는 테스트 케이스 작성

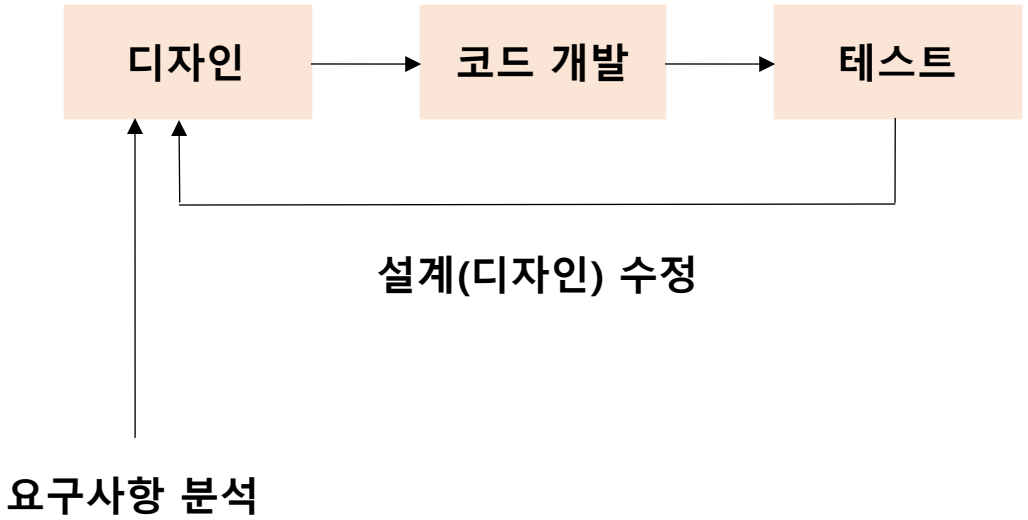



**2) Green 단계**
: 실패하는 테스트 케이스를 통과하기 위한 최소한의 코드 작성 후 테스트 통과

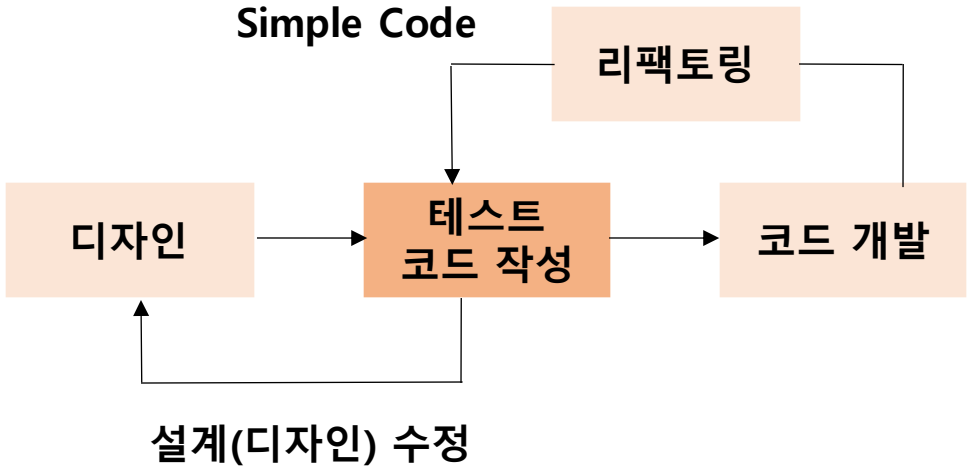
**3) Refactor 단계**
: 중복 코드 제거, 일반화 등 리팩토링

# 일반 개발 방식 vs TDD 개발 방식

## 일반 개발 방식




## TDD 개발 방식



# 좋은 단위 테스트의 특징

**1개의 테스트 함수는 1가지 개념만 테스트**

**[ First 규칙 ]**

- **CleanCode 책 -**

**(1) Fast** : 테스트는 빠르게 동작하여 자주 돌릴 수 있어야 함

**(2) Independent** : 각각의 테스트는 독립적이며 서로 의존 x

**(3) Repeatable** : 어느 환경에서든 반복 가능

**(4) Self-Validating** : 테스트는 성공 또는 실패로 Boolean 값을 결과로 내어 자체적으로 검증되어야 함

**(5) Timely** : 테스트는 테스트하려는 실제 코드를 구현하기 직전에 구현되어야 함

# TDD의 대표적인 Tool 'JUnit'

: Java에서 주로 사용되는 독립된 단위 테스트 프레임워크

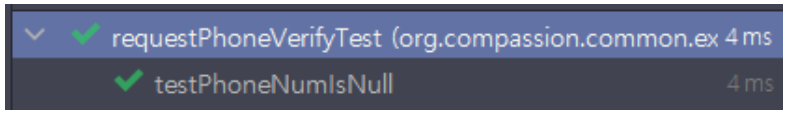
# JUnit의 특징

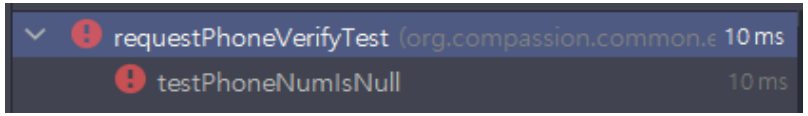
(1) 단위 테스트 Framework 중 하나로 제일 많이 사용됨 (다른 프레임워크 : TestNG)

(2) **@Test** 메서드 호출 시, 새로운 인스턴스가 생성되어 독립적인 단위 테스트 가능

(3) **단정문(Assertions)**으로 테스트케이스의 수행결과를 판별

(4) 결과는 **성공(녹색)** / **실패(붉은색)** 표시



# JUnit 기본 Annotation

(1) **@Test**

: 테스트 메소드 지정

(2) **@DisplayName**

: 테스트 클래스 또는 테스트 메서드의 이름 정의 가능 (JUnit 4에서는 사용불가)

(3) **@Test(timeout=1)**

: 해당 메서드가 결과를 반환하는 데에 1 밀리초를 넘기면 테스트 실패

(4) **@Test(expected=RuntimeException.class)**

: 해당 메소드가 RuntimeException이 발생해야 테스트 성공

# JUnit 기본 Annotation

(5) **@BeforeClass, @AfterClass → @BeforeAll, @AfterAll**

: 클래스 안에 정의된 모든 @Test 메서드들이 수행되기 전과 후에 **한 번씩만** 호출

: 해당 메서드는 **static** 이어야 함

(6) **@Before, @After → @BeforeAll, @AfterAll**

: 각 **@Test** 메서드들이 호출되기 직전과 직후에 실행

: **@Test** 메서드들의 성공/실패 여부와는 상관없이 실행

: **특정 상태로 미리 세팅하기 위한 공동 코드**를 뽑아내는 목적으로 많이 사용

(7) **@Ignore ->@Disabled** : 테스트 클래스 또는 메서드를 비활성화

# JUnit 사용법 : given/when/then 패턴

1) **given (준비)** : 테스트를 위한 준비과정 (데이터 준비)

2) **when (실행)** : 테스트 실행 (테스트하고자 하는 내용)

3) **then (검증)** : 테스트 검증 (단정문 Assertions 사용)

# 단정문 (Assertions) 메서드

JUnit Jupiter에서 제공하는 어설션 기능이 많은 테스트 시나리오에 충분하더라도 더 많은 성능과 *일치기와* 같은 추가 기능이 필요하거나 필요한 경우가 있습니다. 이러한 경우 JUnit 팀은 AssertJ , Hamcrest , Truth 등과 같은 타사 주장 라이브러리를 사용할 것을 권장합니다. 따라서 개발자는 원하는 주장 라이브러리를 자유롭게 사용할 수 있습니다.

# 단정문 (Assertions) 메서드

**JUnit 에서 기본적으로 제공하는 단정문 (JUnit Jupiter)**

- assertTrue() / assertFalse()
- assertNull() / assertNotNull()
- assertEquals() / assertNotEquals()

→ 오류 발생할 경우 바로 즉시 모든 테스트 멈추고 에러메시지 발생

# 단정문 (Assertions) 메서드

**AssertJ 라이브러리에서 제공하는 Assertions**

- **assertThat()**

  → 메서드 체이닝이 가능하여 표현이 명확하며 사용 용이

# TDD 방식으로 테스트 코드 작성하는 법

## 규칙

- **쉬운 경우 -> 어려운 경우**
  : 어려운 경우로 테스트 코드 작성 시작 시, 한 번에 구현해야 하는 코드 많아짐

- **예외적인 경우 -> 정상적인 경우**
  : 예외상황을 고려하지 않고 코드 작성할 경우, 나중에 코드를 뒤집거나 복잡하게 만드는 경우 발생

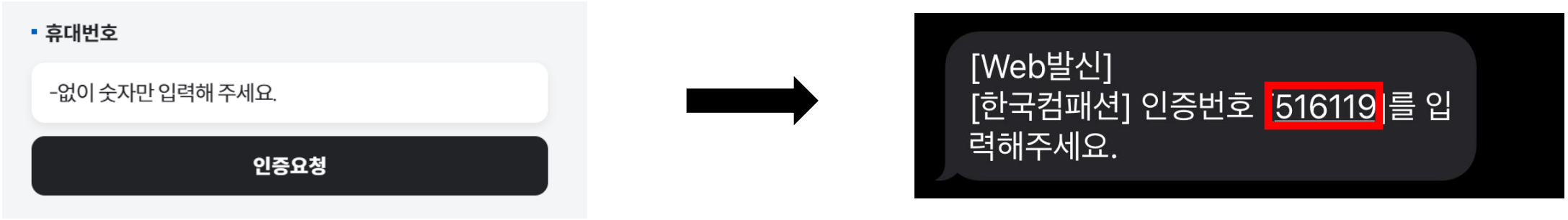
# TDD 방식으로 테스트 코드 작성하는 법

**예시**

- 회원가입 : **동일 이메일 존재 O** VS 동일 이메일 존재 X → 예외적인 경우

- 만료일 계산 : 1월 31일에서 한 달 뒤 VS **12월 31일에서 한 달 뒤** → 쉬운 경우

# 예시 : 휴대폰 인증번호 문자 요청 API




**[ 해당 API 프로세스 ]**

1) 휴대폰 번호 validation 체크

2) 인증번호 6자리 생성

3) DB에 데이터 저장

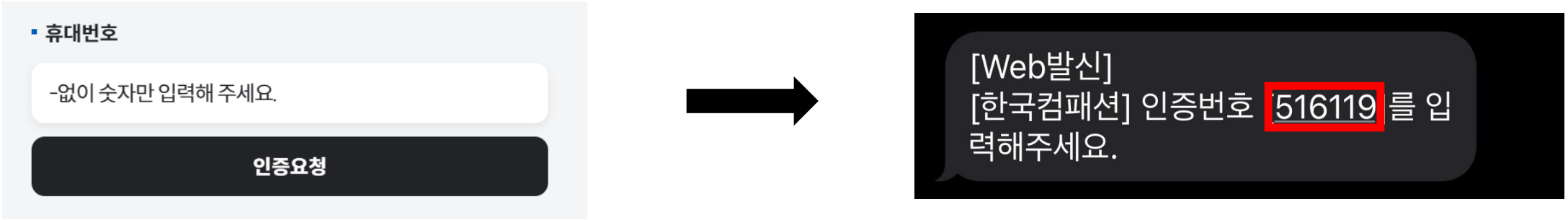
4) 문자 전송

# 예시 : 휴대폰 인증번호 문자 요청 API

**휴대폰 번호 validation 체크 기능**

- 핸드폰 번호가 입력이 안된 경우 (null값인 경우) → 예외적이며 쉬운 경우
- 핸드폰 번호가 11자리가 아닌 경우 → 예외적인 경우
- 핸드폰 번호가 정상적으로 들어오는 경우 → 쉬운 경우
- 핸드폰 번호가 정상적인 형태가 아닌 경우 ("010"으로 시작하지 않는 경우)

# 예시 : 휴대폰 인증번호 문자 요청 API




**[ 해당 API 프로세스 ]**

1) 휴대폰 번호 validation 체크

2) 인증번호 6자리 생성

**3) DB에 데이터 저장**

4) 문자 전송

# **Mockito**를 이용한 단위 테스트

# Mockito


- 단위 테스트를 위한 **가짜 객체(Mock)**를 지원하는 테스트 프레임워크

- Mock을 쉽게 만들고 Mock의 행동을 정하는 **Stubbing**, 정상적으로 작동하는지 verify() 등 다양한 기능 제공

# Mock 객체란?

: "모의, 가짜의"

: 테스트할 때 필요한 실제 객체와 동일한 **가짜 객체**

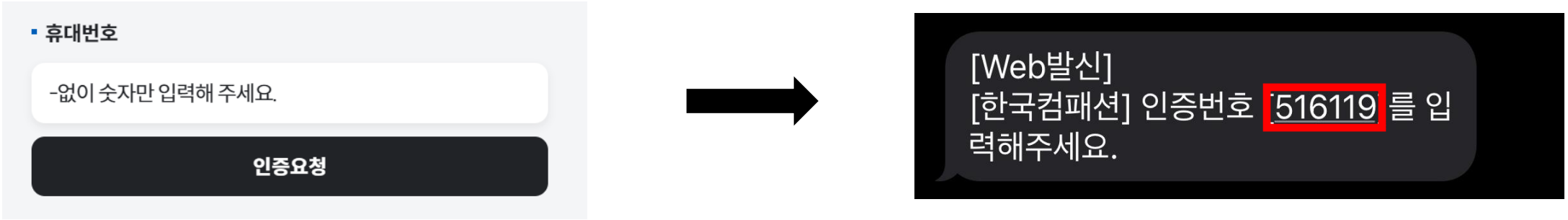
: Mock 객체에 원하는 결과를 반환하고 싶은 경우 **Stub** 해야함

# Mock 객체의 **Stub**란?

: 실제로 준비되어 있지 않지만 원활한 테스트를 위해 메서드 호출에
**미리 정해진 답을 반환하는 구현체**

```
when().thenReturn();
```

# 예시 : 휴대폰 인증번호 문자 요청 API




**[ 해당 API 프로세스 ]**

1) 휴대폰 번호 validation 체크

2) 인증번호 6자리 생성

**3) DB에 데이터 저장** **Mock 객체로 만들 수 있는 대상**

4) 문자 전송

# Mockito Annotation

의존성 주입을 위한 Annotation

- **@ExtendWith(MockitoExtension.class)** : JUnit5와 Mockito 연동
  **@RunWith(MockitoJUnitRunner.class)** : JUnit4와 Mockito 연동

- **@Mock** : 가짜 객체를 만들어 반환

- **@InjectMocks** : **@Mock**으로 생성된 가짜 객체를 자동으로 주입

# 예시 : 휴대폰 인증번호 문자 요청 API

**@InjectMocks**

| CommonController |
| --- |
| requestPhoneVerificationCode() |

1) 휴대폰 번호 validation 체크

2) 인증번호 6자리 생성

**3) DB에 데이터 저장**

4) 문자 전송

**@Mock**

| CommonService |
| --- |
| insertSmsAuthCode() |

**해당 메서드가 특정 리턴값을 던지도록 Stubbing**

**When(commonService.insertSmsAuthCode(anyObject())).thenReturn(0);**

# 코드 - 핸드폰 인증번호 validation 체크

```
@Slf4j
@ExtendWith(MockitoExtension.class)
public class RequestPhoneVerifyTest {


/******** 핸드폰 번호 validation 체크 기능 ********/

//핸드폰 번호가 null 인 경우
@Test(expected = RuntimeException.class)
public void testPhoneNumIsNull() {
String phoneNumber = null;

assertFalse(checkPhoneNumber(phoneNumber));

}


//핸드폰 번호가 11자리가 아닌 경우
@Test(expected=RuntimeException.class)
public void testPhoneNumIsNotCorrect() {
String phoneNumber = "0107777";

assertFalse(checkPhoneNumber(phoneNumber));

}

//정상적인 핸드폰 번호
@Test
public void testPhoneNumIsCorrect() {
String phoneNumber = "01085695138";
assertTrue(checkPhoneNumber(phoneNumber));;
}
```

```
// 이동통신 전화번호가 아닌 경우 (010~ 으로 시작하지 않는 경우)
@Test(expected = RuntimeException.class)
public void testPhoneNumIsNotCorrect2() {
String phoneNum = "02085695138";

assertTrue(checkPhoneNumber(phoneNum));

}


private boolean checkPhoneNumber(String phoneNumber) {

if (phoneNumber == null || phoneNumber.length() != 11) {
throw new RuntimeException("핸드폰 번호 입력 안됨");
}

String phoneCode = phoneNumber.substring(0, 3);

if (!("010".equals(phoneCode))) {
throw new RuntimeException("핸드폰 번호 비정상");
}

return true;


}

}
```

# 코드 - Mock 객체를 이용한 API 테스트

```
@Slf4j
//@RunWith(MockitoJUnitRunner.class)
public class CommonControllerTest extends TestConfig {

private MockMvc mockMvc;

@Mock
private CommonServiceImpl commonService;

@InjectMocks
CommonController commonController;


@Before

public void setUp2() throws Exception {

MockitoAnnotations.initMocks(this);
mockMvc = MockMvcBuilders.standaloneSetup(commonController).build();

}
```

```
// Mock 객체를 이용한 핸드폰 인증번호 요청 API 테스트
@Test
public void requestPhoneVerificationTest() throws Exception {

// given
String phoneNumber = "01022222222";
when(commonService.insertSmsAuthCode(anyObject())).thenReturn(2);
// Mock 객체에 특정값 stubbing


// when
ResultActions result = mockMvc.perform(post("/common/requestPhoneVerificationCode")
.param("phoneNumber", phoneNumber));


// then
result.andExpect(status().isOk())
.andExpect(jsonPath("$.resultCode", equalTo(1)));

    }
}
```