# 26. 테스트 및 디버깅

## 단위 테스트 (`JUnit5`, `Mockito`)

### 1. 단위 테스트란 무엇인가?

- **정의**: 프로그램의 가장 작은 단위(메서드, 클래스 단위 등)를 독립적으로 검증하는 테스트.
- **목적**: 코드의 **정확성 보장**, 리팩토링 안정성 확보, 버그 조기 발견.
- **특징**:
  - 반복 실행 가능
  - 자동화 가능
  - 독립적 실행 (테스트끼리 의존 X)

---

### 2. JUnit5 개요

- **JUnit5**는 다음 3가지 구성요소로 이루어짐:
  - `JUnit Platform`: 테스트 실행 환경
  - `JUnit Jupiter`: JUnit5의 API (테스트 어노테이션 등)
  - `JUnit Vintage`: JUnit4/3와의 호환

---

### 3. JUnit5 주요 어노테이션

| 어노테이션 | 설명 |
|---|---|
| `@Test` | 테스트 메서드 |
| `@BeforeEach` | 각 테스트 전에 실행 |
| `@AfterEach` | 각 테스트 후에 실행 |
| `@BeforeAll` | 모든 테스트 전에 1회 실행 (static) |
| `@AfterAll` | 모든 테스트 후에 1회 실행 (static) |
| `@DisplayName` | 테스트 이름 지정 |
| `@Disabled` | 테스트 비활성화 |
| `@Nested` | 중첩 테스트 클래스 |
| `@Tag` | 테스트 분류용 태그 |
| `@RepeatedTest(n)` | 반복 테스트 |

---

## 4. JUnit5 기본 예제

```
import org.junit.jupiter.api.*;

class CalculatorTest {

    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }

    @Test
    @DisplayName("덧셈 테스트")
    void testAddition() {
        Assertions.assertEquals(5, calc.add(2, 3));
    }

    @Test
    @DisplayName("0으로 나누기 예외 테스트")
    void testDivideByZero() {
        Assertions.assertThrows(ArithmeticException.class, () -> {
            calc.divide(5, 0);
        });
    }
}
```

## 5. 주요 Assertion 메서드

| 메서드 | 설명 |
| --- | --- |
| `assertEquals(expected, actual)` | 두 값이 같은지 |
| `assertNotEquals(a, b)` | 두 값이 다른지 |
| `assertTrue(condition)` | 조건이 true인지 |
| `assertFalse(condition)` | 조건이 false인지 |
| `assertNull(obj)` / `assertNotNull(obj)` | null 여부 |
| `assertThrows(Exception.class, () -> {})` | 예외 발생 여부 확인 |

## 6. Mockito 개요

- **Mockito**: 테스트 대상 객체의 **의존 객체를 모킹(Mock)** 하여 격리된 테스트 수행.
- **모킹(mocking)**: 실제 객체의 동작을 흉내내는 가짜 객체를 생성.

---

## 7. Mockito 핵심 기능

| 기능 | 설명 |
| --- | --- |
| `mock(Class<T>)` | 가짜 객체 생성 |
| `when(...).thenReturn(...)` | 반환값 설정 |
| `verify(...)` | 메서드 호출 여부 검증 |
| `doThrow(...)` | 예외 발생 설정 |
| `@Mock`, `@InjectMocks` | 어노테이션 기반 설정 |

---

## 8. Mockito 예제

```
import org.junit.jupiter.api.*;
import org.mockito.*;

import static org.mockito.Mockito.*;

class UserServiceTest {

    @Mock
    UserRepository userRepository;

    @InjectMocks
    UserService userService;

    @BeforeEach
    void init() {
        MockitoAnnotations.openMocks(this);  // @Mock, @InjectMocks 초기화
    }

    @Test
    void testFindUserById() {
        // Given
        User dummy = new User(1L, "Alice");
        when(userRepository.findById(1L)).thenReturn(Optional.of(dummy));

        // When
        User result = userService.getUserById(1L);

        // Then
        Assertions.assertEquals("Alice", result.getName());
```

```
        verify(userRepository).findById(1L);
    }
}
```

---

## 9. Mockito + JUnit 통합 팁

- `@ExtendWith(MockitoExtension.class)` 사용 가능 (JUnit5)
- 테스트 격리를 위해 `@BeforeEach`에서 매번 초기화하거나 `MockitoAnnotations.openMocks()` 사용
- `verify`로 **정확한 호출 여부**까지 테스트 가능

---

## 10. 테스트 코드 구조 및 전략

### 계층적 테스트

- **단위 테스트(Unit Test)**: 메서드, 클래스 단위
- **통합 테스트(Integration Test)**: 모듈 간 상호작용
- **엔드투엔드 테스트(E2E)**: 전체 흐름 테스트

### 테스트 3단계 패턴 (AAA)

```
Arrange (준비) → Act (실행) → Assert (검증)
```

### 명명 규칙 예시

```
void add_whenTwoPositiveNumbers_thenReturnsSum()
```

---

## 11. Maven/Gradle 의존성

### Maven

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>5.12.0</version>
  <scope>test</scope>
</dependency>
```

### Gradle

```
testImplementation 'org.junit.jupiter:junit-jupiter:5.10.0'
testImplementation 'org.mockito:mockito-core:5.12.0'
```

---

## 12. 고급 기능

- `ArgumentCaptor`: 전달된 인자 값 검증
- `spy(...)`: 실제 객체 감시 (부분만 모킹)
- `@Captor`: 인자 캡처 자동화
- `@Disabled`: 특정 조건에서 테스트 제외

---

## 13. 실무 팁

- 테스트는 **독립적이며 반복 가능하게 작성**해야 한다.
- **테스트 더블(Mock, Stub, Spy, Fake, Dummy)**의 역할을 구분하라.
- CI 도구 (GitHub Actions, Jenkins 등)와 통합해 자동화하라.
- 커버리지 도구 (JaCoCo 등)와 함께 사용하는 것이 좋다.

# 애플리케이션 로깅 (`Log4j`, `SLF4J`, `java.util.logging`)

## 1. 로깅이란 무엇인가?

**로깅(logging)**은 애플리케이션의 동작 상태, 오류, 성능 등의 정보를 **파일/콘솔에 기록**하는 것을 말한다.
단순 `System.out.println()`을 넘어 다음과 같은 특성이 요구된다:

- 로그 레벨 분류 (INFO, DEBUG, ERROR 등)
- 로그 파일 저장, 회전(rotation)
- 포맷팅, 출력 대상 설정 (콘솔, 파일, 원격)
- 비동기 처리 및 필터링
- 시스템 운영 및 디버깅 지원

---

## 2. 대표적인 Java 로깅 프레임워크

| 로깅 API | 설명 | 장점 |
| --- | --- | --- |
| `java.util.logging` | Java 표준 API | JDK 기본 내장 |
| `Log4j` | Apache의 강력한 로깅 프레임워크 | 유연한 설정, 고성능 |
| `SLF4J` | 추상화 API (Facade) | 구현체(Log4j, Logback 등)와 분리 가능 |

---

## 3. 로그 레벨(Level)의 의미

| 레벨 | 설명 |
| --- | --- |
| `TRACE` | 가장 상세한 로그 (진단, 디버깅용) |
| `DEBUG` | 개발자용 디버깅 로그 |
| `INFO` | 일반적인 정보 로그 |
| `WARN` | 경고 상황 (하지만 치명적이지 않음) |
| `ERROR` | 에러 발생 (처리됨) |
| `FATAL` | 치명적 오류 (시스템 중단 가능성) |

---

## 4. `java.util.logging` (JUL)

### 특징

- JDK에 기본 포함
- 설정 파일: `logging.properties`
- 비교적 단순하지만 확장성은 제한적

### 사용 예시

```
import java.util.logging.*;

public class JULExample {
    private static final Logger logger = Logger.getLogger(JULExample.class.getName());

    public static void main(String[] args) {
        logger.info("정보 로그입니다.");
        logger.warning("경고 로그입니다.");
        logger.severe("심각한 에러 로그입니다.");
    }
}
```

### 설정 (`logging.properties`)

```
.level=INFO
handlers= java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

---

## 5. Log4j 2 (Apache)

### 특징

- 고성능, 유연한 설정(XML, JSON, YAML, Properties)
- AsyncAppender, RollingFileAppender 등 제공
- **Log4j 2** 권장 (Log4j 1은 보안 이슈로 종료됨)

### 주요 구성 요소

- **Logger**: 로그 기록
- **Appender**: 출력 대상 (콘솔, 파일 등)
- **Layout**: 로그 포맷 형식
- **Filter**: 로그 필터링

### 설정 파일 예 (log4j2.xml)

```
<Configuration status="INFO">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger - %msg%n"/>
    </Console>
    <File name="FileLogger" fileName="logs/app.log">
      <PatternLayout pattern="%d %p %c [%t] %m%n"/>
    </File>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Console"/>
      <AppenderRef ref="FileLogger"/>
    </Root>
  </Loggers>
</Configuration>
```

### 사용 예제

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4jExample {
    private static final Logger logger = LogManager.getLogger(Log4jExample.class);

    public static void main(String[] args) {
        logger.info("정보 출력");
        logger.error("에러 발생");
    }
}
```

---

## 6. SLF4J (Simple Logging Facade for Java)

### 특징

- **로깅 인터페이스(API)**를 제공
- 실제 로깅 구현체(Logback, Log4j2 등)와 분리
- `LoggerFactory.getLogger(...)` 를 통해 사용
- 개발 시 SLF4J 의존성만으로 다양한 구현체를 교체 가능

### 설정 흐름

```
[ SLF4J API ] → [ 로깅 구현체 (Log4j2, Logback 등) ]
```

### 사용 예제 (Log4j2를 백엔드로 사용)

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Slf4jExample {
    private static final Logger log = LoggerFactory.getLogger(Slf4jExample.class);

    public static void main(String[] args) {
        log.debug("디버그 메시지");
        log.info("정보 메시지");
        log.warn("경고 메시지");
        log.error("에러 메시지");
    }
}
```

---

## 7. Maven 의존성 예

### SLF4J + Log4j2 구성

```
<dependencies>
  <!-- SLF4J API -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.13</version>
  </dependency>

  <!-- Log4j2 Backend -->
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j2-impl</artifactId>
    <version>2.23.1</version>
  </dependency>
</dependencies>
```

## 8. 로그 출력 예시

```
14:33:12.123 [main] INFO  com.example.Main - 서버가 시작되었습니다.
14:33:12.456 [main] ERROR com.example.Main - 데이터베이스 연결 실패
```

---

## 9. 비교 요약

| 항목 | `java.util.logging` | `Log4j2` | `SLF4J` |
|---|---|---|---|
| 타입 | 로깅 구현체 | 로깅 구현체 | 로깅 추상화 |
| 성능 | 중 | 매우 높음 | 구현체에 따라 다름 |
| 유연성 | 낮음 | 매우 높음 | 높음 (구현체 교체 가능) |
| 설정 | `logging.properties` | XML, JSON, YAML | 구현체 설정에 따름 |
| 추천 | 간단한 경우 | 실무/대규모 | 모든 상황에서 추상화용 |

---

## 10. 실무 팁

- 대부분의 프로젝트에서는 **SLF4J + Logback 또는 Log4j2**를 함께 사용한다.
- 로그는 **레벨 기반 필터링**으로 운영/개발 로그 분리 관리가 필수.
- 민감한 정보 (비밀번호, 인증키 등)는 로그로 남기지 말 것.
- `Thread`, `Exception`, `Request Context` 포함한 포맷으로 남기자.
- 로그 파일은 **logrotate** 혹은 자체 회전(RollingFileAppender)으로 관리하자.

# 디버깅 기법 (`breakpoint`, `watch`, `stack trace`)

## 1. 디버깅(Debugging)이란?

**디버깅**은 코드의 논리적 오류, 예외 상황, 비정상적인 동작 등을 추적하고 수정하는 과정이다.
단순 로그 출력만으로는 부족한 경우, **IDE의 디버거**를 사용해 코드의 실행 흐름을 제어하고 상태를 분석한다.

---

## 2. 주요 디버깅 도구

| 도구 | 설명 |
|---|---|
| **IDE 내장 디버거** | IntelliJ, Eclipse, VSCode 등 |
| **JVM 옵션 기반 디버깅** | `-Xdebug`, `-agentlib:jdwp=...` |
| **원격 디버깅** | 원격 서버에서 디버거 연결 |
| **jdb** | Java Debugger CLI 도구 |

| 도구 | 설명 |
| --- | --- |
| **jvisualvm** | 스레드/메모리 시각화 도구 |

---

## 3. 기본 디버깅 개념

### ✅ 3.1 Breakpoint (중단점)

- 특정 라인에서 코드 실행을 일시 중단
- 조건부 중단 가능 (`if count > 10` 등)
- 메서드 진입 전, 예외 발생 시 등도 설정 가능

> 📍 IDE 사용 시: 라인 번호 클릭 → 빨간 점 표시

---

### ✅ 3.2 Watch (감시 표현식)

- 특정 변수나 표현식의 값을 **계속 추적**
- 현재 값이 어떻게 변하는지 시각적으로 확인 가능

예시:

```
user.getName().toUpperCase()
order.getTotalPrice()
```

---

### ✅ 3.3 Stack Trace (스택 추적)

- 예외가 발생했을 때 **메서드 호출 경로**를 출력
- 위쪽부터 차례대로 호출된 메서드를 보여줌

예시 출력:

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.OrderService.process(OrderService.java:42)
    at com.example.Main.main(Main.java:12)
```

> 스택 트레이스를 통해 **오류가 발생한 코드 위치**, **원인 메서드**, **호출 순서**를 역추적 가능

---

### ✅ 3.4 Step Into / Step Over / Step Out

| 명령 | 설명 |
| --- | --- |
| **Step Into (F7)** | 메서드 내부로 진입 |
| **Step Over (F8)** | 현재 라인 실행 후 다음 라인으로 이동 |
| **Step Out (Shift+F8)** | 현재 메서드를 빠져나옴 |

---

## 4. 실습 예시 코드

```
public class DebugExample {
    public static void main(String[] args) {
        int x = 10;
        int y = 0;
        int result = divide(x, y);
        System.out.println("결과: " + result);
    }

    public static int divide(int a, int b) {
        return a / b; // 여기에 Breakpoint 설정
    }
}
```

> 여기서 `b = 0`이므로 `ArithmeticException` 발생.
> 중단점에서 `a`, `b` 값을 감시하고, 예외 발생 시 스택 트레이스를 분석하자.

---

## 5. IntelliJ 디버깅 기능 요약

| 기능 | 단축키 | 설명 |
|---|---|---|
| Debug 실행 | Shift + F9 | 디버깅 모드로 시작 |
| Breakpoint 설정/해제 | Ctrl + F8 | 중단점 추가 |
| Step Over | F8 | 현재 라인 실행 |
| Step Into | F7 | 메서드 진입 |
| Step Out | Shift + F8 | 메서드 빠져나오기 |
| Evaluate Expression | Alt + F8 | 감시/계산창 |
| View Variables | Alt + 1 → Variables | 현재 스택 프레임의 변수 확인 |

---

## 6. 조건부 브레이크포인트

```
// 조건이 참일 때만 멈춤
while (i < 100) {
    sum += i;
    i++;
}
```

👉 조건: `i == 42`

> 중단점에 마우스 우클릭 → "조건 설정"

---

## 7. Evaluate Expression (표현식 계산)

- 런타임 중에 임의의 표현식을 평가할 수 있음
- 예: `user.getName().startsWith("J")`

---

## 8. 예외 중단 설정

- `RuntimeException` 또는 `NullPointerException` 등 특정 예외가 발생했을 때 자동 중단
- IntelliJ 기준:
  - Run → View Breakpoints → Add Java Exception Breakpoint

---

## 9. 원격 디버깅

서버에서 동작 중인 Java 프로세스에 디버거 연결

### JVM 옵션 예시

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

### IDE 연결 설정

- Host: 서버 IP
- Port: 5005
- 디버그 모드로 애플리케이션 시작

---

## 10. 로그 디버깅과 병행하기

- 로깅과 디버깅은 함께 사용하는 게 좋음
- 개발 환경에서는 `DEBUG` 로그 수준을 사용하여 디버깅 힌트를 남길 수 있음
- 예외 발생 시 스택 트레이스 로그 기록 필수

---

## 11. 디버깅 팁

- 코드 로직을 **작게 나누고 순차 디버깅**하자.
- **데이터 흐름**이 의심될 때는 `Evaluate Expression`과 `Watch`를 적극 활용하자.
- **반복문**이나 **재귀함수**는 조건부 중단점을 사용하자.
- `Thread.sleep()`은 디버깅 시 정확한 타이밍을 볼 수 있어 유용할 수 있음.