Effective Java

Item 15 : 클래스와 멤버의 접근 권한을 최소화하라
Item 16 : public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라
Item 17 : 변경 가능성을 최소화하라
Extra : BigDecimal

Item 15

클래스와 멤버의 접근 권한을 최소화하라

접근 제한자

클래스, 인터페이스

public

default

필드, 생성자, 메서드
중첩 클래스, 중첩 인터페이스

public

protected

default

private

접근 제한자

public	모두 공개
protected	같은 패키지 또는 자식 클래스
default	같은 패키지
private	모두 비공개

기본 원칙 : 최대한 접근성을 좁히자

클래스, 인터페이스

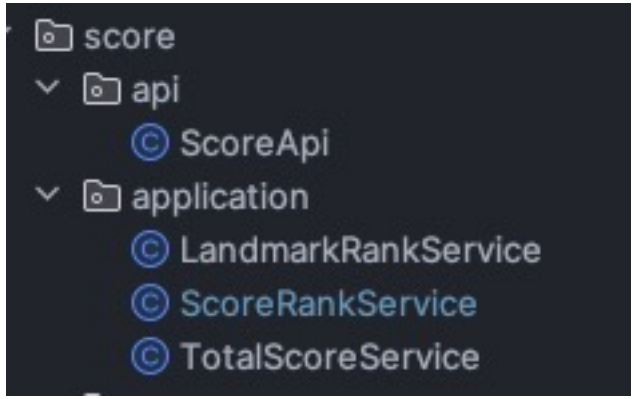
public : 공개 API

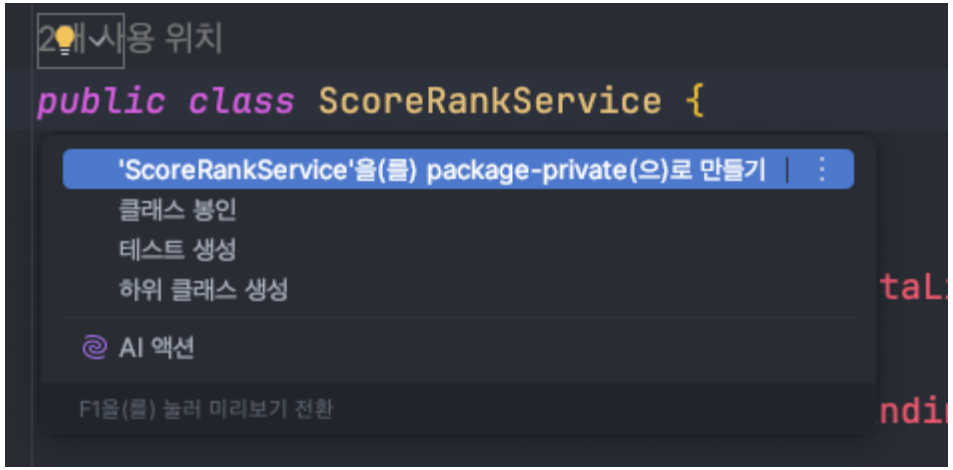
default : 내부 구현

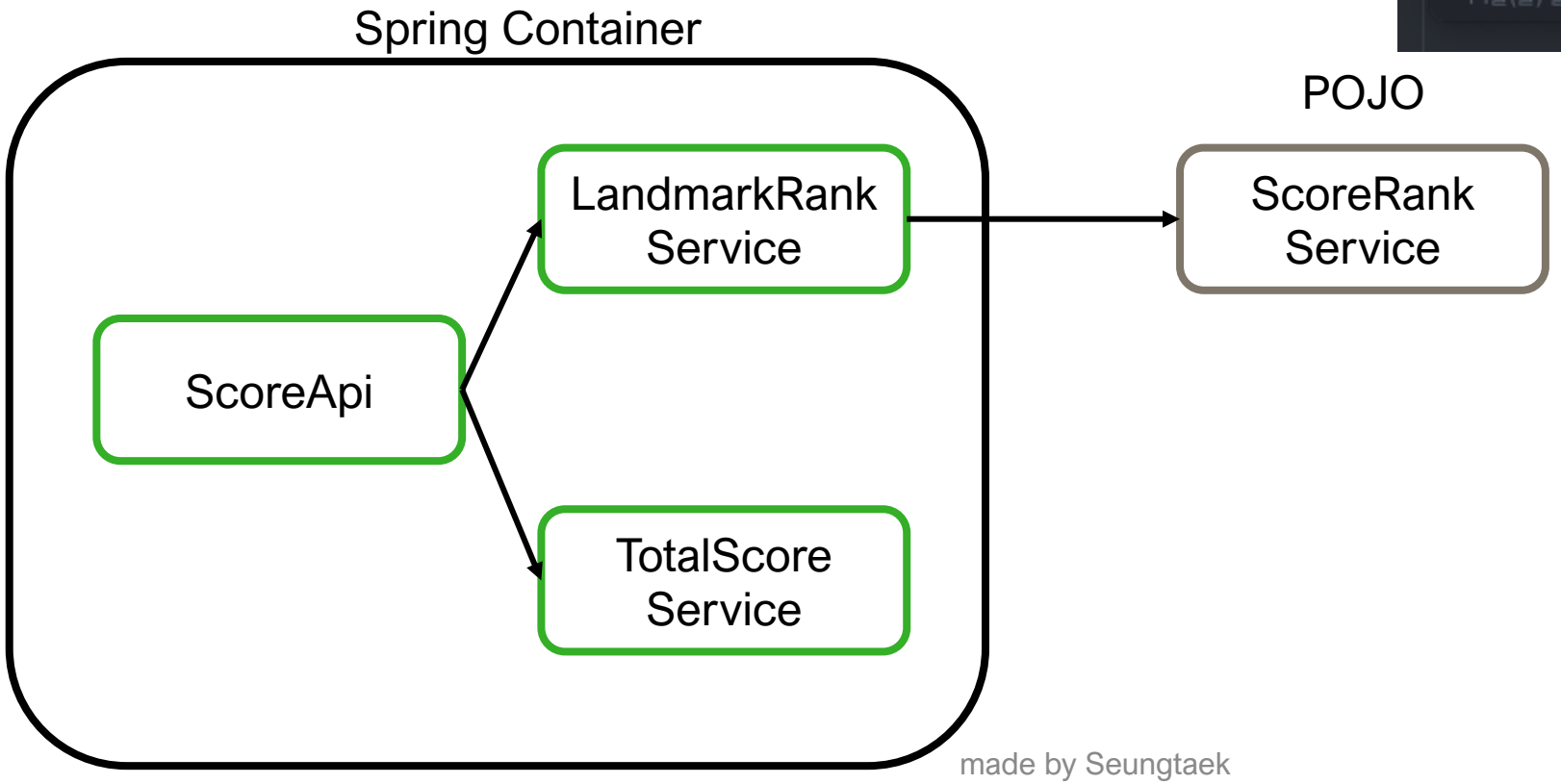
심지어 한 클래스에서만 사용한다면 private static 중첩클래스로..

public일 필요가 없는 클래스는 default로 바꾸자!

클래스, 인터페이스


score
api
ScoreApi
application
LandmarkRankService
ScoreRankService
TotalScoreService


2개 사용 위치
public class ScoreRankService {
'ScoreRankService'을(를) package-private(으)로 만들기
클래스 봉인
테스트 생성
하위 클래스 생성
AI 액션
F1을(를) 눌러 미리보기 전환


Spring Container
POJO
LandmarkRank Service
ScoreRank Service
ScoreApi
TotalScore Service

필드, 메서드, 중첩 클래스, 중첩 인터페이스

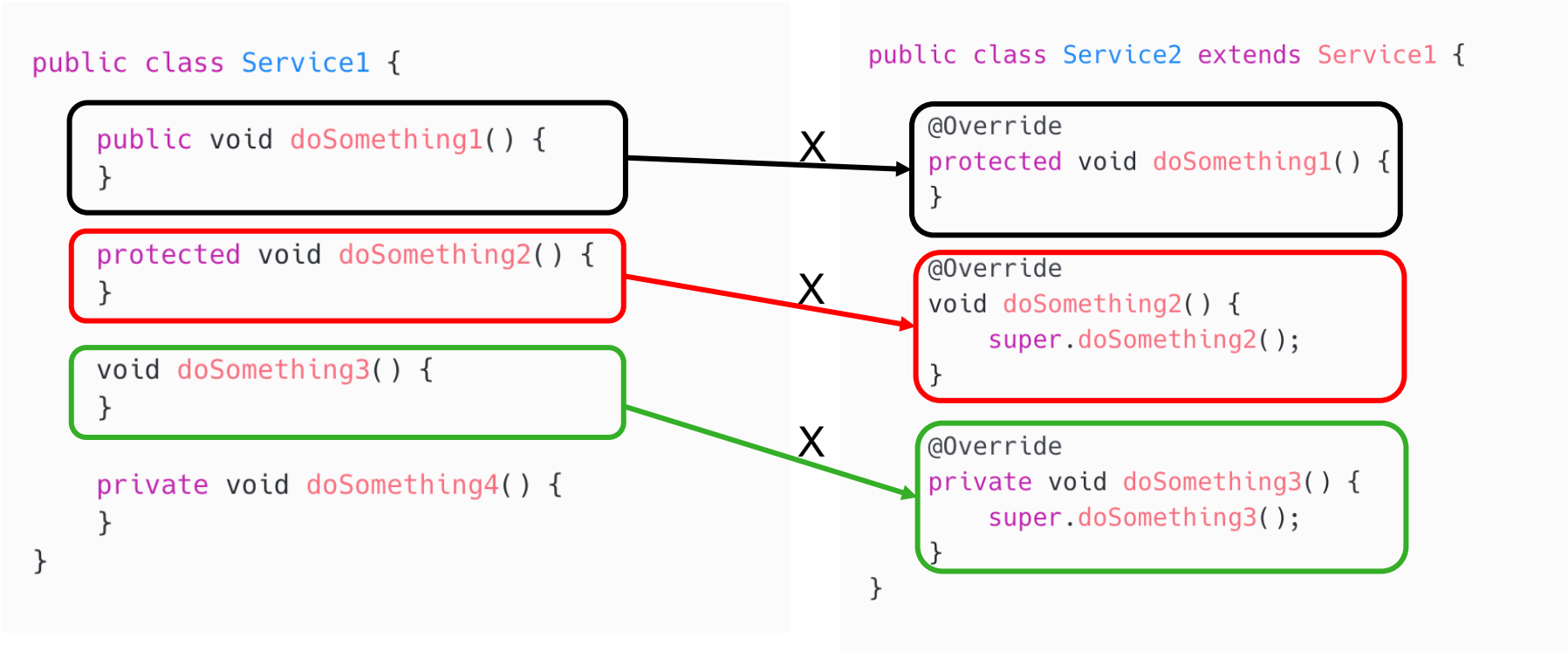
private, default, protected, public

1. 모든 멤버를 private으로 만들기
2. 조금씩 default로 풀어주기

   - 권한을 너무 자주 풀어준다면 컴포넌트 분리를 고민하기
3. protected부터는 접근 대상이 엄청나게 넓어지므로 주의하기

필드, 메서드, 중첩 클래스, 중첩 인터페이스

메서드 Override 시에는 상위 클래스에서보다 좁게 설정할 수 없다.



cf) 인터페이스의 멤버는 기본값이 public이다. 구현 클래스도 모든 메서드를 public으로 선언해야 한다.
cf2) 단위 테스트를 원한다면 default까지 풀어주는거는 허용. 같은 패키지에서 테스트하면 됨

public 클래스

public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다. [item 16]

단, 상수값은 public static final 필드로 공개해도 좋다.

관례상 상수 이름은 대문자, 각 단어 사이에 밑줄(_)을 넣는다.

반드시 기본 타입이나 불변 객체를 참조해야 한다.

```
public class Dice {

    public static final int MAX_VALUE = 6;
    public static final int MIN_VALUE = 1;

}
```

public 클래스 내부 배열

No. 외부에서 값을 변경할 수 있음

```
public static final Thing[] VALEUS = {...};
```

대안 1

```
private static final Thing[] VALEUS = {...};
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES))
```

대안 2

```
private static final Thing[] VALEUS = {...};
public static final Thing[] values() {
    return PRIVATE_VALUES.clone();
 }
```

JPMS (Java Platform Module System)

자바 9에서 공식적으로 도입된 모듈 시스템.

사용이유

1. 컴포넌트간의 관계를 더 잘 표현하기 위해.
2. 모듈들간의 관계 안정성
3. 캡슐화
4. 확장성

모듈 이전에는..

JAR에서 다른 JAR에 있는 클래스를 사용하기 위해서는 classpath를 작성하는게 유일한 방법

근데 이 방법은 컴파일 시점에 에러를 찾지 못하므로, 런타임 에러가 발생할 수 있음

JPMS (Java Platform Module System)

(관례상) module-info.java

```
module com.test {
    requires com.good;
}

module com.good {
    exports com.good;
}
```

하지만 실제로는 잘 사용하지 않음..

1. 모듈 시스템 사용이 강제가 아님
2. 개발자들이 캡슐화를 좋아하지 않음. (더 깊게 내려가서 조작하기를 원한다.)

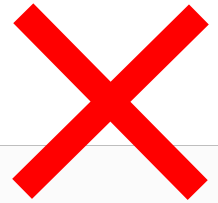
JDK 외에도 모듈 개념을 널리 받아들여질지 예측하기는 아직 이른 감이 있다.
그러니 꼭 필요한 경우가 아니라면 당분간은 사용하지 않는 게 좋을 것 같다.

effective java 3rd. 101 p

Item 16

public 클래스에서 public 필드는 피하라

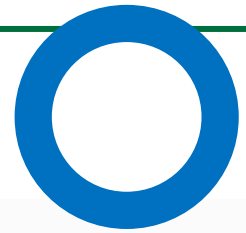
public 클래스



```
public class Point {
    public double x;
    public double y;
}
```

- 내부 표현을 자유롭게 수정 불가능
- 불변식 보장 불가능
- 외부에서 필드 접근 시 부수작업 수행 불가능

→ 필드가 불변이라도 여전히 단점은 존재



```
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }
}
```

default 클래스, private 중첩 클래스

코드면에서 public 필드 사용이 오히려 더 깔끔하다.

default 클래스

```
class Outer {
    public int outerField = 1;
}
```

같은 패키지에서 접근 가능

private 중첩 클래스

```
public class Outer {

    private static class Inner {
        public int innerField = 2;
    }
}
```

Outer에서만 접근 가능

CF) 내부 클래스는 최대한 static이어야 한다.

https://inpa.tistory.com/entry/JAVA-%E2%98%95-%EC%9E%90%EB%B0%94%EC%9D%98-%EB%82%B4%EB%B6%80-%ED%81%B4%EB%9E%98%EC%8A%A4%EB%8A%94-static-%EC%9C%BC%EB%A1%9C-%EC%84%A0%EC%96%B8%ED%95%98%EC%9E%90

Item 17

변경 가능성을 최소화하라

불변 클래스

인스턴스 내부 값이 고정되어, 객체가 파괴되는 순간까지 절대 달라지지 않는 클래스

대표적인 예시: String, Wrapper 클래스, BigInteger, BigDecimal

String.class

```
public static String toUpperCase(String str, byte[] value, Locale locale) {
    //.. 생략 ..
    byte[] result = new byte[len];
    System.arraycopy(value, 0, result, 0, first);

    for (int i = first; i < len; i++) {
        int cp = value[i] & 0xff;
        cp = CharacterDataLatin1.instance.toUpperCaseEx(cp);
        if (!canEncode(cp)) {
            return toUpperCaseEx(str, value, first, locale, false);
        }
        result[i] = (byte)cp;
    }
    return new String(result, LATIN1);
}
```

불변 클래스 5가지 규칙

1. 객체의 상태를 변경하는 메서드를 제공하지 않는다.
2. 클래스를 상속할 수 없도록 한다.
3. 모든 필드를 final로 선언한다.
4. 모든 필드를 private으로 선언한다.
5. 자신 외에는 내부에 가변 컴포넌트에 접근할 수 없도록 한다.

CF) 함수형 프로그래밍과 절차형(또는 명령형) 프로그래밍
함수형 프로그래밍 : 메서드 반환 시, 인스턴스 자신은 수정하지 않고 새로운 인스턴스를 만들어 반환
절차형 프로그래밍 : 메서드에서 피연산자인 자신을 수정해 자신의 상태가 변함

불변 클래스의 이점

1. 단순하다.
2. 쓰레드 안전하다.
3. 자유롭게 공유할 수 있고, 불변 객체끼리도 내부 데이터를 공유할 수 있다.
4. 실패 원자성을 제공한다.

```
public class BigInteger {
  final int[] mag;
  final int signum;

  BigInteger(int[] magnitude, int signum) {
    this.signum = (magnitude.length == 0 ? 0 : signum);
    this.mag = magnitude;
  }

  public BigInteger negate() {
    return new BigInteger(this.mag, -this.signum);
  }
}
```

CF) clone 메서드나 복사생성자는 제공하지 않는 게 좋다.

불변 클래스 단점 : 성능이슈

1. 자주 쓰이는 값은 public static final로 제공

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

2. 가변 동반 클래스(companion class) 사용

StringBuilder, StringBuffer

3. 캐시 사용

```
private in hashCode;

@Override
public int hasCode( ) {
    int result = hasCode;
    if (result ==0) { .. 계산 ..}
    return result;
}
```

상속할 수 없도록 하는 방법

1. final 클래스로 선언
2. 모든 생성자를 private(또는 default)로 만들고 public 정적 팩터리 제공

```
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }
}
```

유연성 증가

정리

1. 클래스는 꼭 필요한 경우가 아니라면 불변이어야 한다.

   setter를 만들지 말자

   성능 이슈가 있다면 가변 동반 클래스를 만들자

2. 불변으로 만들 수 없더라도 변경 가능한 부분을 최소한으로 줄이자

   다른 합당한 이유가 없다면 모든 필드는 private final이어야 한다.

3. 생성자는 초기화가 완벽히 끝난 객체를 반환해야 한다.

record

자바 14 preview, 자바 16에서 정식 도입

특징 : 불변 클래스, 메서드 자동 생성(getter, equals, hashcode, toString), 생성자 자동 생성

```
public record Book(String title, String author, String isbn) {
    public Book {
        Objects.requireNonNull(title);
        Objects.requireNonNull(author);
        Objects.requireNonNull(isbn);
    }
    public Book(String title, String isbn) {
        this(title, "Unknown", isbn);
    }
}
```

Extra

BigDecimal

https://dev.gmarket.com/75

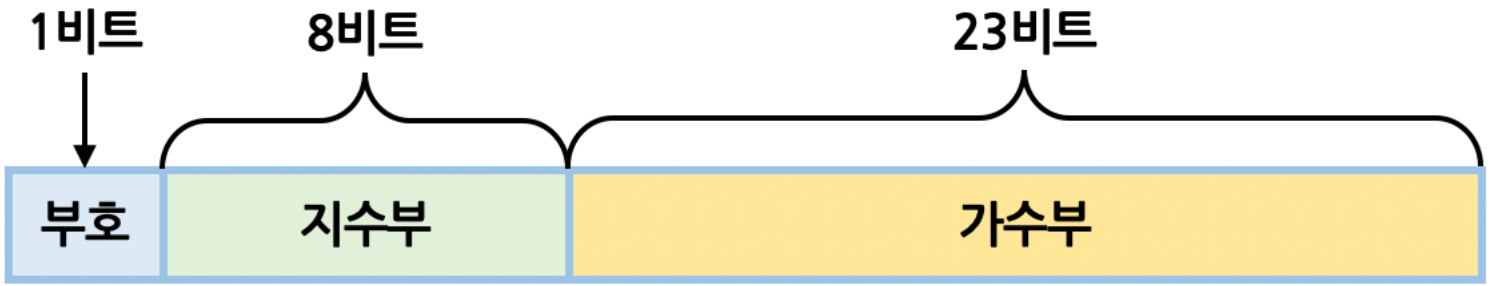
float, double의 문제점

```
double a = 0.1;
double b = 0.2;

// expect true
if (a + b == 0.3) {
    doSomething(); // but not invoked..
}

System.out.println(a + b); // 0.30000000000000004
```

0.3의 2진수 표현 : 0.0100110011...



IEEE 부동 소수점 방식

12.3456
1. 정규화
   0.123456 * 10^2
2. 지수부 : 2
   가수부 : 0.123456

BigDecimal 이란?

불변의 성질을 띠며, 임의 정밀도와 부호를 지니는 10진수

* 임의 정밀도 : 아무리 큰 숫자라도 표현할 수 있는 것(무한에 가까움)

불변 : 객체간의 연산마다 새로운 객체 생성. float, double보다 훨씬 느림

기본적인 구현 방법 : 큰 숫자를 배열에 나눠 담는 방식

```
큰 숫자 = [int] + [int] + [int] + [int] + ...
```

BigDecimal 구성

표현 : unscaled value와 scale(소수점 자릿수)

ex) 3.14의 경우 : unscaled value=314, scale=2

precision : 총 자릿수

intCompact : 값의 크기가 작아서 유효숫자가 long타입으로 표현될 수 있다면 intVal 대신 사용(메모리 최적화)

```
package java.math;

public class BigDecimal extends Number implements Comparable<BigDecimal> {

    private final BigInteger intVal; // = unscaled value

    private final int scale;

    private transient int precision;

    private final transient long intCompact;

    ...
```

BigDecimal 생성

```
BigDecimal fromBigInteger = new BigDecimal(new BigInteger("1000"));
BigDecimal fromCharArray = new BigDecimal(new char[]{'1', '5', '4', '3'});
BigDecimal fromInt = new BigDecimal(1000);
BigDecimal fromLong = new BigDecimal(10000000L);
BigDecimal fromDouble = new BigDecimal(1.12); // 이렇게 사용하면 안 됨!
```

```
BigDecimal fromDouble = new BigDecimal(1.12);
System.out.println(fromDouble); // 1.1200000000000001065814103640150278806686401367187 5
```

String 생성자나 valueOf() 정적 팩터리 메서드를 사용해서 생성하기.
*valueOf() : double을 String으로 변환 후 생성

```
new BigDecimal("1.12"); // 1.12
BigDecimal.valueOf(1.12); // 1.12
```

BigDecimal 연산

나눗셈 주의 : 정확한 몫을 반환하기 때문에 무한소수이면 ArithmeticException 발생

따라서 나누기 연산을 수행할 땐 반드시 소수점 처리 전략을 지정해줘야 함

```
BigDecimal a = new BigDecimal("7");
BigDecimal b = new BigDecimal("3");

// 더하기
// 10
a.add(b);

// 빼기
// 4
a.subtract(b);

// 곱하기
// 21
a.multiply(b);

// 나누기 - 기본적으로 정확한 몫을 반환함
// 2.33333...
// java.lang.ArithmeticException: Non-terminating
a.divide(b);

// 나누기 - 소수점 아래 첫째 자리까지 반올림
// 2.3
a.divide(b, 1, RoundingMode.HALF_UP);

// 나누기 - 총 자릿수를 34개로 제한하고 반올림(HALF_EVEN)
// 2.3333333333333333333333333333333333
a.divide(b, MathContext.DECIMAL128);
```

```
// 나누기 - 총 자릿수를 제한하지 않고 반올림
// 2.33333...
// java.lang.ArithmeticException: Non-te
a.divide(b, MathContext.UNLIMITED);

// 나머지(%)
// 1
a.remainder(b);

// 나머지(%) - 총 자릿수를 34개로 제한
// 1
a.remainder(b, MathContext.DECIMAL128);

// 절대값
// 3
new BigDecimal("-3").abs();

// 최대값
// 7
a.max(b);

// 최소값
// 3
a.min(b);

// 부호 변환
// -7
a.negate();
```

소수점 처리
RoundingMode Enum 참고

BigDecimal 비교연산

3.14 : unscaled value=314, scale=2

3.140 : unscaled value=3140, scale=3

equals : unscaled value와 scale을 모두 비교

compareTo() : 소수점 가장 뒤의 0은 비교하지 않음

```
// 주솟값을 비교한다
// false
a == b;

// unscaled value와 scale을 비교한다 (값과 소수점 자리까지 함께 비교)
// false
a.equals(b);

// unscaled value만 비교한다 (값만 비교)
// true
a.compareTo(b) == 0;
```

BigDecimal 속도

```
int loop = 1000000000;  10억 loop

long start = System.nanoTime();
long result = 0;
for (int i = 0; i < loop; i++) {
    result += i;
}
long end = System.nanoTime();
System.out.println("[" + (end - start) + "ns] " + " ======> " + result);


start = System.nanoTime();
BigDecimal result2 = new BigDecimal("0");
for (int i = 0; i < loop; i++) {
    result2 = result2.add(new BigDecimal(i));
}
end = System.nanoTime();
System.out.println("[" + (end - start) + "ns] " + " ======> " + result2);
```

약 3배

```
[659759750ns]  ======> 4999999995000000000
[1991709083ns]  ======> 4999999995000000000
```