

정확한 답이 필요하다면
float 과 double 은 피하라

소수점에 관한 유명한 사고

패트리엇 미사일 격추 실패 사건

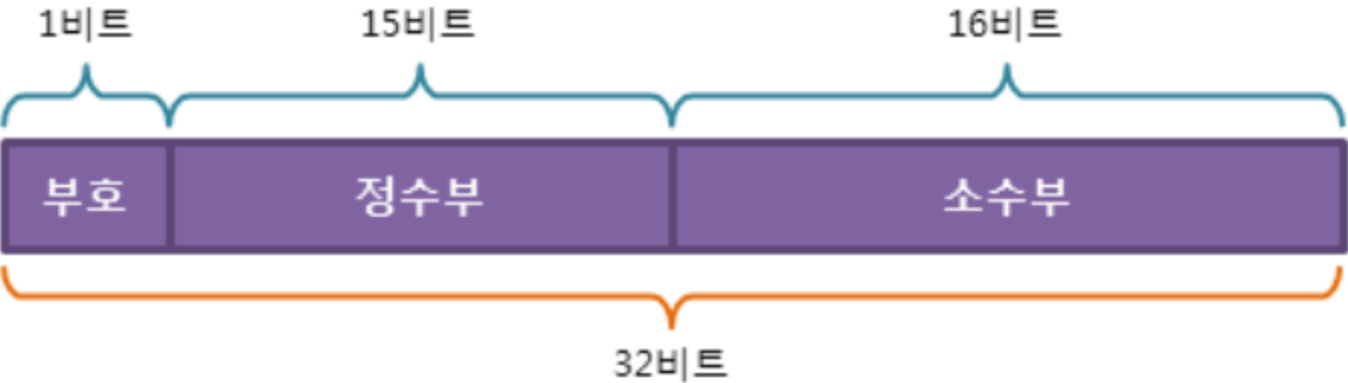


아리안5 로켓 폭발 사고



실수 표현 방식

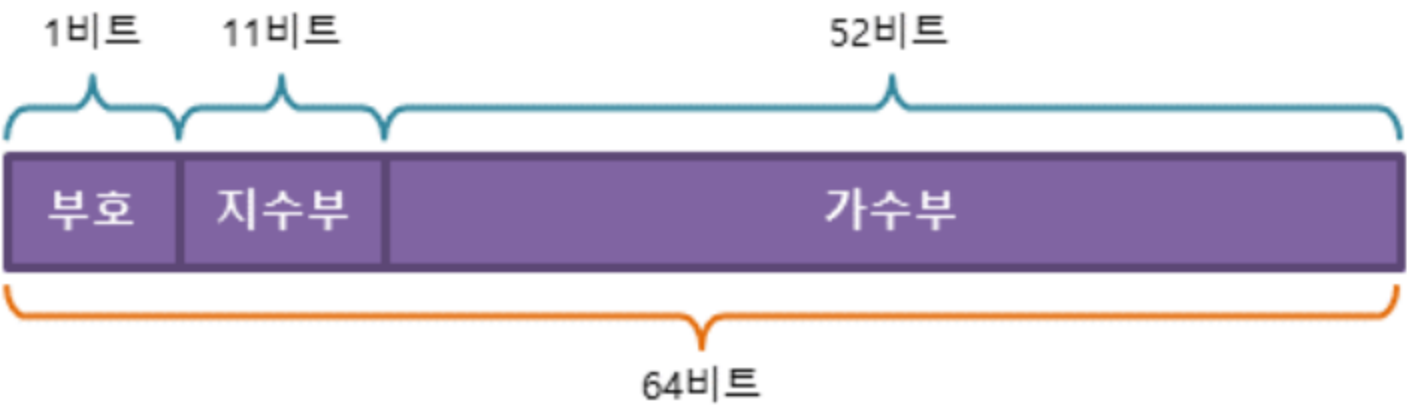
고정 소수점 방식



IEEE float형 부동 소수점 방식



IEEE double형 부동 소수점 방식



float과 double이 사용하는 이진 부동소수점의 취약점

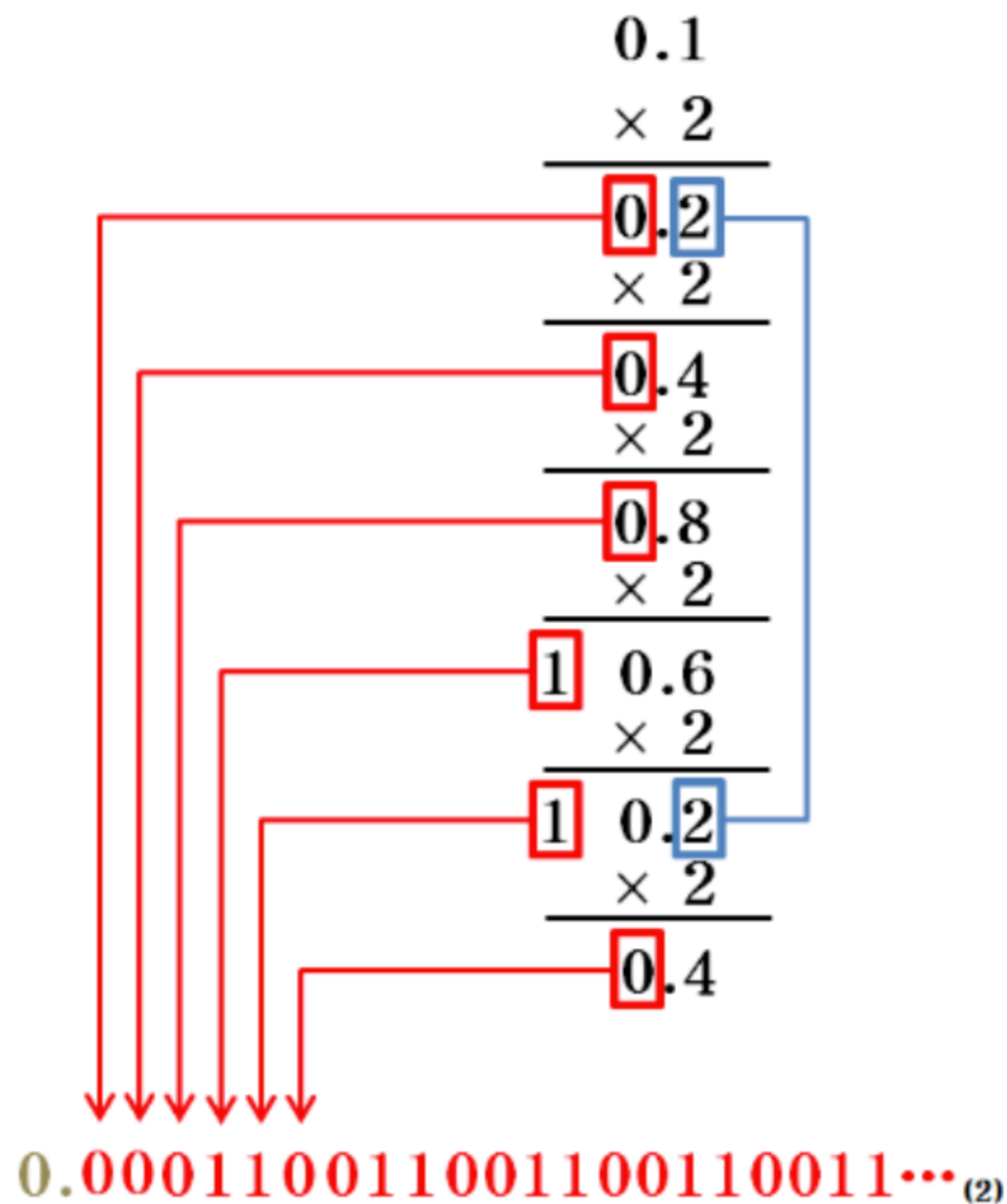
```
public static void main(String[] args) {  
    System.out.println(1.03 - 0.42);  
    System.out.println(1.00 - 9 * 0.10);  
    System.out.println(0.3 - 0.2);  
}
```

0.610000000000000001

0.099999999999999998

0.099999999999999998

float과 double이 사용하는 이진 부동소수점의 취약점



순환소수로 비트 수의 한계까지 표현하다보면 위와 같은 결과가 나온다.

float과 double이 사용하는 이진 부동소수점의 취약점

```
public static void main(String[] args) {  
    System.out.println(1.03 - 0.42);  
    System.out.println(1.00 - 9 * 0.10);  
    System.out.println(0.3 - 0.2);  
}
```

→ 0.610000000000000001

→ 0.099999999999999998

→ 0.099999999999999998

float 과 **double** 은 이진 부동소수점 연산에 쓰이며 근사치로 계산하도록 설계 되었다.

NAVER

kakao

LINE

coupang

배달의민족

 당근마켓

 toss

해결 방법

BigDecimal, int 혹은 **long**을 사용해야 한다.

int 혹은 long

```
System.out.println(1.03 - 0.42);  
System.out.println(1.00 - 9 * 0.10);  
System.out.println(0.3 - 0.2);
```

문제의 연산



```
public static void main(String[] args) {  
    System.out.println((103 - 42) / 100d);  
    System.out.println((100 - 9 * 10) / 100d);  
    System.out.println((30 - 20) / 100d);  
}
```

0.61

0.1

0.1

다룰 수 있는 값의 크기가 제한되고 소수점을 직접 관리해야 한다.

int 혹은 long

자료형	데이터	메모리 크기	표현 가능 범위
boolean	참/거짓	1 byte	true, false
char	문자	2 byte	모든 유니코드 문자
byte	정수	1 byte	-128~127
short		2 byte	-32768~32767
int		4 byte	-2147483648~2147483647
long	실수	8 byte	-9223372036854775808~9223372036854775807
float		4 byte	1.4E-45~3.4028235E38
double		8 byte	4.9E-324~1.7976931348623157E308

int 혹은 long

- 숫자를 9자리 십진수로 표현할 수 있으면 **int**
- 18자리 십진수로 표현할 수 있다면 **long**
- 18자리가 넘어간다면 **BigDecimal**

BigDecimal

```
System.out.println(1.03 - 0.42);  
System.out.println(1.00 - 9 * 0.10);  
System.out.println(0.3 - 0.2);
```

문제의 연산



```
public static void main(String[] args) {  
    System.out.println(new BigDecimal("1.03").subtract(new BigDecimal("0.42")));  
                                                                    0.61  
    System.out.println(new BigDecimal("1.00").subtract(  
        new BigDecimal("9").multiply(new BigDecimal("0.10")))); 0.1  
    System.out.println(new BigDecimal("0.3").subtract(new BigDecimal("0.2"))); 0.1  
}
```

BigDecimal 의 생성자 중 반드시 문자열로 숫자를 받는 생성자를 이용해야 한다.

BigDecimal

연산 메서드

```
BigDecimal add(BigDecimal val)
BigDecimal subtract(BigDecimal val)
BigDecimal multiply(BigDecimal val)
BigDecimal divide(BigDecimal val)
BigDecimal remainder(BigDecimal val)
```

소수점 처리하는 방법

```
RoundingMode.CEILING - 올림 (음수일 경우 내림, 숫자가 더 큰 쪽으로)
RoundingMode.UP - 올림 (음수 양수 상관 없이 무조건 올림)
RoundingMode.FLOOR - 내림 (음수일 경우 올림, 숫자가 더 작은 쪽으로)
RoundingMode.DOWN - 내림 (음수 양수 상관 없이 무조건 내림)
RoundingMode.HALF_DOWN - 반내림
RoundingMode.HALF_EVEN - 짝수를 만드는 쪽으로 올리거나 내림
RoundingMode.HALF_UP - 반올림
RoundingMode.UNNECESSARY - 사용안함
```

소수점 몇번째 자리까지 RoundingMode를 적용할지는
setScale() 메서드를 사용하면 된다.

BigDecimal

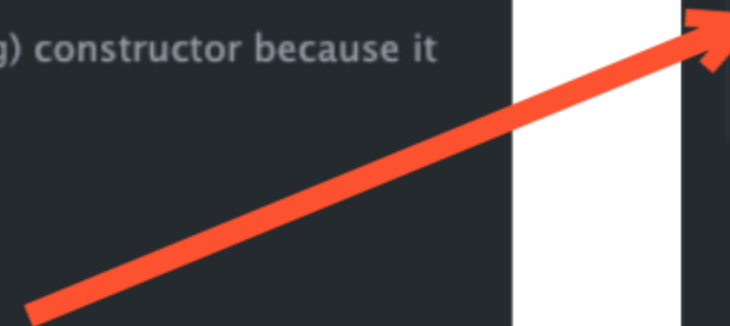
Translates a long value into a BigDecimal with a scale of zero.

Params: val - value of the BigDecimal.

Returns: a BigDecimal whose value is val.

API Note: This static factory method is provided in preference to a (long) constructor because it allows for reuse of frequently used BigDecimal values.

```
public static BigDecimal valueOf(long val) {  
    if (val ≥ 0 && val < ZERO_THROUGH_TEN.length)  
        return ZERO_THROUGH_TEN[(int) val];  
    else if (val ≠ INFLATED)  
        return new BigDecimal( intVal: null, val, scale: 0, prec: 0);  
    return new BigDecimal(INFLATED_BIGINT, val, scale: 0, prec: 0);  
}
```



// Cache of common small BigDecimal values.

```
private static final BigDecimal ZERO_THROUGH_TEN[] = {  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.ONE, val: 1, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.TWO, val: 2, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(3), val: 3, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(4), val: 4, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(5), val: 5, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(6), val: 6, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(7), val: 7, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(8), val: 8, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.valueOf(9), val: 9, scale: 0, prec: 1),  
    new BigDecimal(BigInteger.TEN, val: 10, scale: 0, prec: 2),  
};
```

// Cache of zero scaled by 0 - 15

```
private static final BigDecimal[] ZERO_SCALED_BY = {  
    ZERO_THROUGH_TEN[0],  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 1, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 2, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 3, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 4, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 5, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 6, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 7, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 8, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 9, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 10, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 11, prec: 1),  
    new BigDecimal(BigInteger.ZERO, val: 0, scale: 12, prec: 1),  
};
```


BigDecimal

```
public static BigDecimal valueOf(double val) {  
    // Reminder: a zero double returns '0.0', so we cannot fastpath  
    // to use the constant ZERO. This might be important enough to  
    // justify a factory approach, a cache, or a few private  
    // constants, later.  
    return new BigDecimal(Double.toString(val));  
}
```

```
public BigDecimal(String val) {  
    this(val.toCharArray(), offset: 0, val.length());  
}
```



Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the `BigDecimal(String)` constructor, while allowing a sub-array to be specified and with rounding according to the context settings.

Params: `in` - char array that is the source of characters.
`offset` - first character in the array to inspect.
`len` - number of characters to consider.
`mc` - the context to use.

Throws: `NumberFormatException` - if `in` is not a valid representation of a `BigDecimal` or the defined subarray is not wholly within `in`.

Implementation Note: If the sequence of characters is already available within a character array, using this constructor is faster than converting the char array to string and using the `BigDecimal(String)` constructor.

Since: 1.5

```
public BigDecimal(char[] in, int offset, int len, MathContext mc) {  
    // protect against huge length, negative values, and integer overflow  
    try {  
        Objects.checkFromIndexSize(offset, len, in.length);  
    } catch (IndexOutOfBoundsException e) {  
        throw new NumberFormatException  
            ("Bad offset or len arguments for char[] input.");  
    }  
  
    // This is the primary string to BigDecimal constructor; all  
    // incoming strings end up here; it uses explicit (inline)  
    // parsing for speed and generates at most one intermediate  
    // (temporary) object (a char[] array) for non-compact case.  
  
    // Use locals for all fields values until completion  
    int prec = 0;           // record precision value  
    int scl = 0;           // record scale value  
    long rs = 0;           // the compact value in long  
    BigInteger rb = null;   // the inflated value in BigInteger
```

정리

- 소수점 연산이 필요한 경우 정확한 답이 필요할 때는 float 과 double 을 사용하지 말고 성능저하가 있더라도 BigDecimal 을 사용하는 것이 정확하다.
- BigDecimal이 제공하는 여덟 가지 반올림 모드를 이용하여 반올림을 완벽히 제어할 수 있다.
- 성능이 중요하고 소수점을 직접 추적할 수 있고 숫자가 너무 크지 않다면 int나 long을 사용하자.