Luke's Language로 파싱된 이펙티브 자바

Object 재정의

제작. 홍성혁

3장

모든 객체는 Object를 상속한다

- 재정의를 염두한 메서드들
  - equals
  - hashCode
  - toString
  - clone
  - finalize

Item10. equals 빙산의 윗부분

가능하면 재정의 하지 않는다

Why?

Item10. equals 빙산의 윗부분

가능하면 재정의 하지 않는다

Why?

그 이유를 찾기에 앞서 “equal”이 무슨 뜻이지?

Equality (mathematics)

From Wikipedia, the free encyclopedia

In mathematics, **equality** is a relationship between two quantities or, more generally, two mathematical expressions, asserting that the quantities have the same value, or that the expressions represent the same mathematical object. Equality between A and B is written $A = B$, and pronounced "A equals B". In this equality, A and B are the *members* of the equality and are distinguished by calling them *left-hand side* or *left member*, and *right-hand side* or *right member*. Two objects that are not equal are said to be **distinct**.

A formula such as $x = y$, where x and y are any expressions, means that x and y denote or represent the same object.[1] For example,

$$1.5 = 3/2,$$

are two notations for the same number. Similarly, using set builder notation,

$$\{x \mid x \in \mathbb{Z} \text{ and } 0 < x \leq 3\} = \{1, 2, 3\},$$

since the two sets have the same elements. (This equality results from the axiom of extensionality that is often expressed as "two sets that have the same elements are equal".[2])

The truth of an equality depends on an interpretation of its members. In the above examples, the equalities are true if the members are interpreted as numbers or sets, but are false if the members are interpreted as expressions or sequences of symbols.

An identity, such as $(x + 1)^2 = x^2 + 2x + 1$, means that if x is replaced with any number, then the two expressions take the same value. This may also be interpreted as saying that the two sides of the equals sign represent the same function (equality of functions), or that the two expressions denote the same polynomial (equality of polynomials).[3][4]

Equality (mathematics)

41 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

수학 에서 **등식은** 두 양 또는 보다 일반적으로 두 수학적 표현식 사이의 관계 로, 양이 같은 값을 갖거나 표현식이 같은 수학적 객체를 나타낸다고 주장합니다. A 와 B 사이의 등식은 $A = B$ 로 작성되고 " A 는 B 와 같다 "로 발음됩니다. 이 등식에서 A 와 B 는 등식의 멤버이고 *왼쪽* 또는 *왼쪽 멤버* , *오른쪽* 또는 *오른쪽 멤버 라고* 부르면서 구별됩니다 . 같지 않은 두 객체는 **서로 다르다고** 합니다 .

다음과 같은 공식$x = y$,여기서 x 와 y 는 임의의 표현식이며, 이는 x 와 y 가 동일한 객체를 나타내거나 표현함을 의미합니다. [1] 예를 들어,

$$1.5 = 3/2,$$

같은 숫자에 대한 두 가지 표기법입니다. 마찬가지로, 세트 빌더 표기법을 사용하면 ,

$$\{x \mid x \in \mathbb{Z} \text{ and } 0 < x \leq 3\} = \{1, 2, 3\},$$

두 집합이 같은 원소를 가지고 있기 때문입니다. (이러한 동등성은 종종 "같은 원소를 가지는 두 집합은 동일하다"로 표현되는 확장성 공리 에서 비롯됩니다. [2])

등식의 진실은 멤버의 해석에 따라 달라집니다. 위의 예에서 등식은 멤버가 숫자나 집합으로 해석되면 참이지만, 멤버가 표현식이나 기호 시퀀스로 해석되면 거짓입니다.

다음 과 같은 신원$(x + 1)^2 = x^2 + 2x + 1$,x 를 어떤 숫자로 대체하면 두 표현식이 같은 값을 취한다는 것을 의미합니다 . 이것은 등호의 두 변이 같은 함수 (함수의 동등성)를 나타내거나 두 표현식이 같은 다항식 (다항식의 동등성)을 나타낸다는 것을 의미하는 것으로 해석될 수도 있습니다. [3][4]

Equality (mathematics)

From Wikipedia, the free encyclopedia

수학 에서 **등식은** 두 양 또는 보다 일반적으로 두 수학적 표현식 사이의 관계 로, 양이 같은 값을 갖거나 표현식이 같은 수학적 객체를 나타낸다고 주장합니다. A 와 B 사이의 등식은 $A = B$ 로 작성되고 " A 는 B 와 같다 "로 발음됩니다. 이 등식에서 A 와 B 는 등식의 멤버이고 *왼쪽* 또는 *왼쪽 멤버* , *오른쪽* 또는 *오른쪽 멤버 라고* 부르면서 구별됩니다 . 같지 않은 두 객체는 **서로 다르다고** 합니다 .

다음과 같은 공식$x = y$,여기서 x 와 y 는 임의의 표현식이며, 이는 x 와 y 가 동일한 객체를 나타내거나 표현함을 의미합니다. [1] 예를 들어,

$$1.5 = 3/2,$$

같은 숫자에 대한 두 가지 표기법입니다. 마찬가지로, 세트 빌더 표기법을 사용하면 ,

$$\{x \mid x \in \mathbb{Z} \text{ and } 0 < x \leq 3\} = \{1, 2, 3\},$$

두 집합이 같은 원소를 가지고 있기 때문입니다. (이러한 동등성은 종종 "같은 원소를 가지는 두 집합은 동일하다"로 표현되는 확장성 공리 에서 비롯됩니다. [2])

등식의 진실은 멤버의 해석에 따라 달라집니다. 위의 예에서 등식은 멤버가 숫자나 집합으로 해석되면 참이지만, 멤버가 표현식이나 기호 시퀀스로 해석되면 거짓입니다.

다음 과 같은 신원$(x + 1)^2 = x^2 + 2x + 1$,x 를 어떤 숫자로 대체하면 두 표현식이 같은 값을 취한다는 것을 의미합니다 . 이것은 등호의 두 변이 같은 함수 (함수의 동등성)를 나타내거나 두 표현식이 같은 다항식 (다항식의 동등성)을 나타낸다는 것을 의미하는 것으로 해석될 수도 있습니다. [3][4]

Equality (mathematics)

From Wikipedia, the free encyclopedia

수학 에서 **등식은** 두 양 또는 보다 일반적으로 두 수학적 표현식 사이의 관계 로, 양이 같은 값을 갖거나 표현식이 같은 수학적 객체를 나타낸다고 주장합니다. A 와 B 사이의 등식은 $A = B$ 로 작성되고 " A 는 B 와 같다 "로 발음됩니다. 이 등식에서 A 와 B 는 등식의 멤버이고 *왼쪽* 또는 *왼쪽 멤버* , *오른쪽* 또는 *오른쪽 멤버 라고* 부르면서 구별됩니다 . 같지 않은 두 객체는 **서로 다르다고** 합니다 .

다음과 같은 공식$x = y$,여기서 x 와 y 는 임의의 표현식이며, 이는 x 와 y 가 동일한 객체를 나타내거나 표현함을 의미합니다. [1] 예를 들어,

$$1.5 = 3/2,$$

같은 숫자에 대한 두 가지 표기법입니다. 마찬가지로, 세트 빌더 표기법을 사용하면 ,

$$\{x \mid x \in \mathbb{Z} \text{ and } 0 < x \leq 3\} = \{1, 2, 3\},$$

두 집합이 같은 원소를 가지고 있기 때문입니다. (이러한 동등성은 종종 "같은 원소를 가지는 두 집합은 동일하다"로 표현되는 확장성 공리 에서 비롯됩니다. [2])

등식의 진실은 멤버의 해석에 따라 달라집니다. 위의 예에서 등식은 멤버가 숫자나 집합으로 해석되면 참이지만, 멤버가 표현식이나 기호 시퀀스로 해석되면 거짓입니다.

다음 과 같은 신원$(x + 1)^2 = x^2 + 2x + 1$,x 를 어떤 숫자로 대체하면 두 표현식이 같은 값을 취한다는 것을 의미합니다 . 이것은 등호의 두 변이 같은 함수 (함수의 동등성)를 나타내거나 두 표현식이 같은 다항식 (다항식의 동등성)을 나타낸다는 것을 의미하는 것으로 해석될 수도 있습니다. [3][4]

Item10. equals 빙산의 윗부분

Equality 동등성이란?

다음 네 가지를 만족한다.

- Reflexivity
- Symmetry
- Transitivity
- Consistency

\+ 그리고 Not Null

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분

Equality 동등성이란?

다음 네 가지를 만족한다.

- Reflexivity
  - 난, 나야

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분

Equality 동등성이란?

다음 네 가지를 만족한다.

- Symmetry
- 너도? 나도!

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분

Equality 동등성이란?

다음 네 가지를 만족한다.

- Transitivity
- 삼단논법
  사람은 똥을 싼다
  당신은 사람이다
  당신은 똥을 싼다.

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분

Equality 동등성이란?

다음 네 가지를 만족한다.

- Consistency
- 사람이 변하면 죽는다.
  동등성도 같나보다.

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분

Equality 동등성이란?

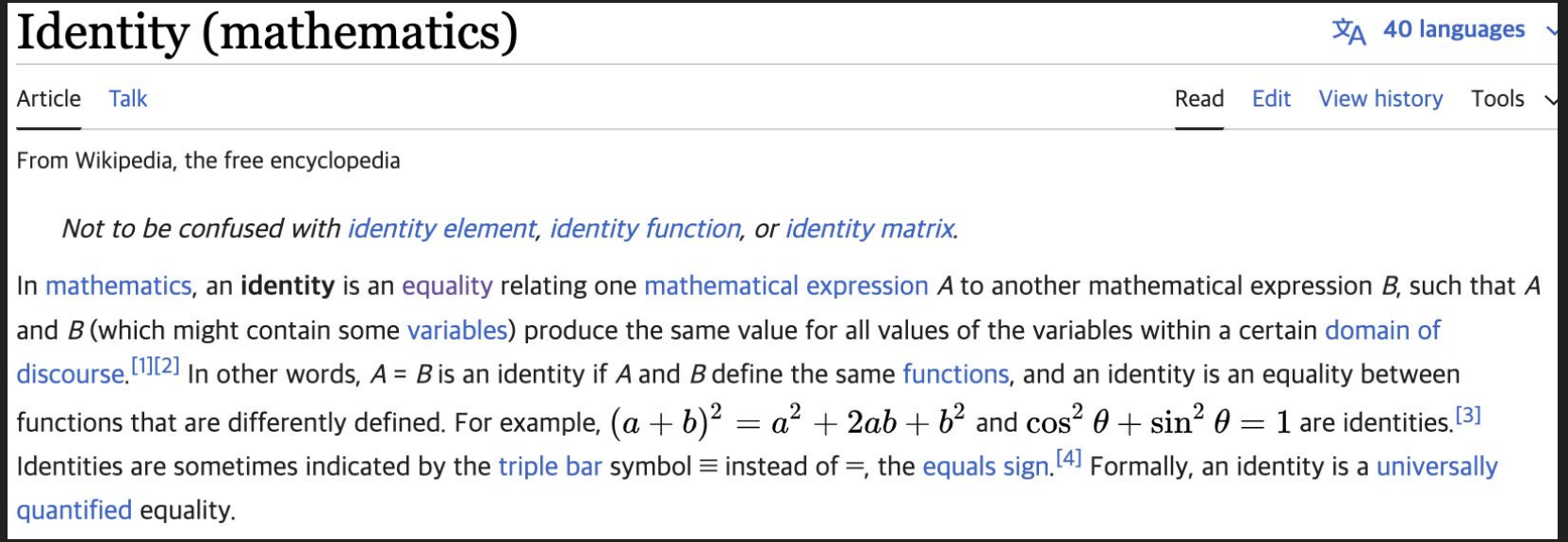
다음 네 가지를 만족한다.

+ 그리고 Not Null
  - 0으로 나누면,
    그때부턴 수학이 아니다.

equals 메서드는 동치관계(equivalence relation)를 구현하며, 다음을 만족한다.

- **반사성(reflexivity)**: null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다.
- **대칭성(symmetry)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다.
- **추이성(transitivity)**: null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)도 true면 x.equals(z)도 true다.
- **일관성(consistency)**: null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다.

Item10. equals 빙산의 윗부분



Identity (mathematics)

40 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with identity element, identity function, or identity matrix.

In mathematics, an **identity** is an equality relating one mathematical expression *A* to another mathematical expression *B*, such that *A* and *B* (which might contain some variables) produce the same value for all values of the variables within a certain domain of discourse.[1][2] In other words, *A* = *B* is an identity if *A* and *B* define the same functions, and an identity is an equality between functions that are differently defined. For example, $(a+b)^2 = a^2 + 2ab + b^2$ and $\cos^2\theta + \sin^2\theta = 1$ are identities.[3] Identities are sometimes indicated by the triple bar symbol ≡ instead of =, the equals sign.[4] Formally, an identity is a universally quantified equality.

cf) 동일성

Item10. equals 빙산의 윗부분

Identity (mathematics)

40 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

항등원소, *항등함수*, *항등행렬* *과 혼동하지 마십시오*.

수학 에서 항등 **성은** 한 수학적 표현 *A*를 다른 수학적 표현 *B* 에 연결하는 등식 이며, *A* 와 *B* (몇몇 변수를 포함할 수 있음)는 특정 담론 도메인 내의 모든 변수 값에 대해 동일한 값을 생성합니다. [1][2] 즉, *A* = *B* 는 *A* 와 *B*가 동일한 함수를 정의할 때 항등성이고, 항등성은 다르게 정의된 함수 간의 등식입니다. 예를 들어, $(a+b)^2 = a^2 + 2ab + b^2$ 그리고 $\cos^2\theta + \sin^2\theta = 1$ 항등성입니다. [3] 항등성은 때때로 = 대신 등호 (≡) 로 표시됩니다. [4] 형식적으로 항등성은 보편적으로 양화된 동등성 입니다.

quantified equality.

cf) 동일성

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

가능하면 재정의 하지 않는다

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

가능하면 재정의 하지 않는다

Why?

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

가능하면 재정의 하지 않는다

Why? 설계가 그래,

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

가능하면 재정의 하지 않는다

Why? 설계가 그래,

대수학적 동등성의 논리를 바탕으로 설계된 언어이기 때문 !

Item10. equals 빙산의 윗부분

컴퓨터 언어란, 논리적(수학적) 사고를 바탕으로 설계된 언어

가능하면 재정의 하지 않는다

Why? 설계가 그래,

대수학적 동등성의 논리를 바탕으로 설계된 언어이기 때문 !

설계대로 안쓰면, 그 객체를 사용하는 객체가 과연 정상적으로 동작할까?

- ex) Collection

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

1. 각 인스턴스가 본질적으로 고유하다.

값이 아닌, 동작 객체

ex) Thread

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

2. 인스턴스의 논리적 동등성(logically equality)를 검사할 일이 없다.

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

3. 상위 클래스에서 재정의한 equals가 하위클래스의 경우에도 동일하다.

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

4. 클래스가 private이거나 package-private이고, 어쨌든 equal 호출될 일 없다.

```
@Override public boolean equals(Object o) {
    throw new AssertionError(); // 호출 금지!
}
```

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

1. 각 인스턴스가 본질적으로 고유하다.
2. 인스턴스의 논리적 동등성(logically equality)를 검사할 일이 없다.
3. 상위 클래스에서 재정의한 equals가 하위클래스의 경우에도 동일하다.
4. 클래스가 private이거나 package-private이고, 어쨌든 equal 호출될 일 없다.

Item10. equals 빙산의 윗부분

그래서, 다음 조건에 해당하면, 그냥 쓰자

1. 각 인스턴스가 본질적으로 고유하다.
2. 인스턴스의 논리적 동등성(logically equality)를 검사할 일이 없다.
3. 상위 클래스에서 재정의한 equals가 하위클래스의 경우에도 동일하다.
4. 클래스가 private이거나 package-private이고, 어쨌든 equal 호출될 일 없다.

=> 코드는 부채다!
그냥 쓸 수 있다면, 그냥 쓰자

Item10. equals 빙산의 윗부분

그래도 써야한다면?

1. ==
2. instance of 와 형변환
3. 핵심 필드의 동등성 확인

```
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "지역코드");
        this.prefix = rangeCheck(prefix, 999, "프리픽스");
        this.lineNum = rangeCheck(lineNum, 9999, "가입자 번호");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // 나머지 코드는 생략
}
```

Item10. equals 빙산의 윗부분

그래도 써야한다면?

1. ==
2. instance of 와 형변환
3. 핵심 필드의 동등성 확인

```
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "지역코드");
        this.prefix = rangeCheck(prefix, 999, "프리픽스");
        this.lineNum = rangeCheck(lineNum, 9999, "가입자 번호");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // 나머지 코드는 생략
}
```

Item10. equals 빙산의 윗부분

그래도 써야한다면?

1. ==
2. instance of 와 형변환
3. 핵심 필드의 동등성 확인

```
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "지역코드");
        this.prefix = rangeCheck(prefix, 999, "프리픽스");
        this.lineNum = rangeCheck(lineNum, 9999, "가입자 번호");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // 나머지 코드는 생략
}
```

Item10. equals 빙산의 윗부분

그래도 써야한다면?

1. ==
2. instance of 와 형변환
3. 핵심 필드의 동등성 확인

```
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "지역코드");
        this.prefix = rangeCheck(prefix, 999, "프리픽스");
        this.lineNum = rangeCheck(lineNum, 9999, "가입자 번호");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // 나머지 코드는 생략
}
```

Item11. hashCode도 재정의 하시오

Object 명세에서 발췌한 규약이다.

> - equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
> - equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
> - equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

equals랑
hashCode는
세트네?

Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

equals랑
hashCode는
세트네?

Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

equals랑
hashCode는
세트네?

Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

그래서,
Hash 가 뭔데?

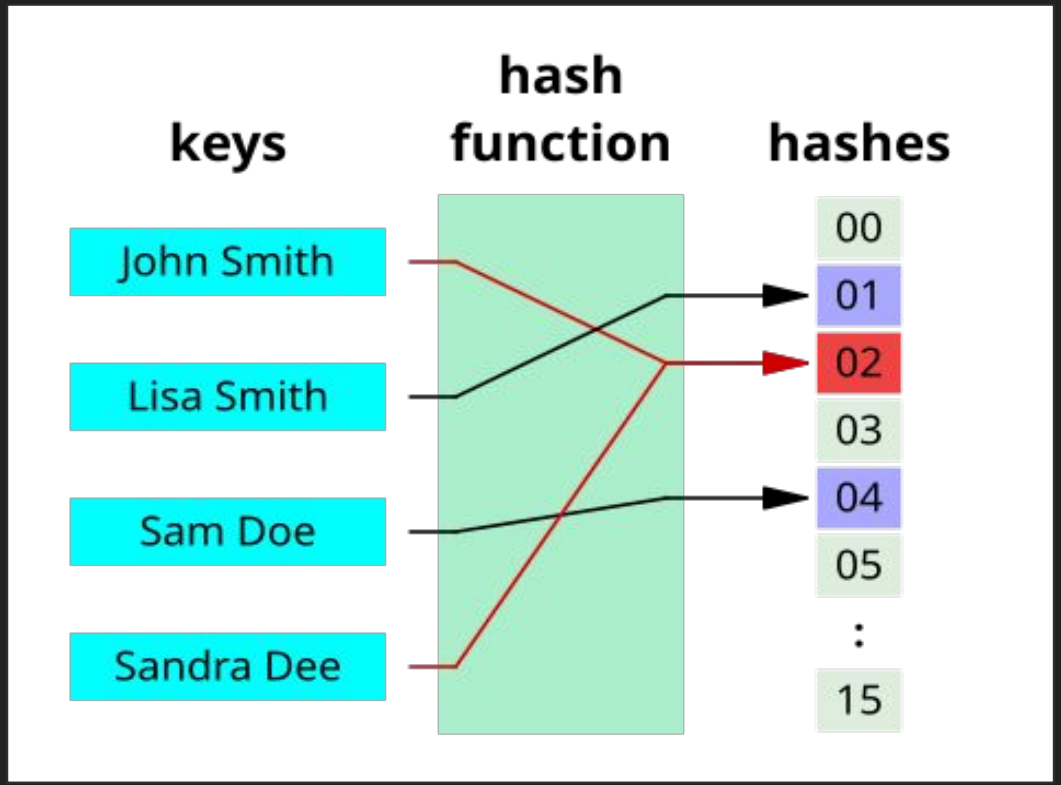
Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

그래서,
Hash 가 뭔데?

Hash Function을 알아보자



Hash function

Hash function - Wikipedia

53 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values, though there are some hash functions that support variable-length output.[1] The values returned by a hash function are called *hash values*, *hash codes*, *hash digests*, *digests*, or simply *hashes*.[2] The values are usually used to index a fixed-size table called a *hash table*. Use of a hash function to index a hash table is called *hashing* or *scatter-storage addressing*.

Hash functions and their associated hash tables are used in data storage and retrieval applications to access data in a small and nearly constant time per retrieval. They require an amount of storage space only fractionally greater than the total space required for the data or records themselves. Hashing is a computationally- and storage-space-efficient form of data access that avoids the non-constant access time of ordered and unordered lists and structured trees, and the often-exponential storage requirements of direct access of state spaces of large or variable-length keys.

Use of hash functions relies on statistical properties of key and function interaction: worst-case behavior is intolerably bad but rare, and average-case behavior can be nearly optimal (minimal collision).[3]:527

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, lossy compression, randomization functions, error-correcting codes, and ciphers. Although the concepts overlap to some extent, each one has its own uses and requirements and is designed and optimized differently. The hash function differs from these concepts mainly in terms of data integrity. Hash tables may use non-cryptographic hash functions, while cryptographic hash functions are used in cybersecurity to secure sensitive data such as passwords.

Hash function

Hash function - Wikipedia

53 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

해시 **함수는 임의 크기의** 데이터를 고정 크기 값으로 매핑하는 데 사용할 수 있는 모든 함수 이지만 가변 길이 출력을 지원하는 해시 함수도 있습니다. [1] 해시 함수에서 반환되는 값을 *해시 값* , *해시 코드* , *해시 다이제스트* , *다이제스트* 또는 간단히 *해시* 라고 합니다 . [2] 값은 일반적으로 *해시 테이블* 이라고 하는 고정 크기 테이블을 인덱싱하는 데 사용됩니다 . 해시 함수를 사용하여 해시 테이블을 인덱싱하는 것을 *해싱* 또는 *분산 저장 주소 지정* 이라고 합니다 .

해시 함수와 연관된 해시 테이블은 데이터 저장 및 검색 애플리케이션에서 검색당 짧고 거의 일정한 시간 내에 데이터에 액세스하는 데 사용됩니다. 이는 데이터 또는 레코드 자체에 필요한 총 공간보다 약간 더 큰 양의 저장 공간을 필요로 합니다. 해싱은 정렬된 목록과 정렬되지 않은 목록 및 구조화된 트리의 비일정한 액세스 시간과 크거나 가변 길이 키의 상태 공간에 직접 액세스하는 데 필요한 지수적 저장 요구 사항을 피하는 계산 및 저장 공간 효율적인 데이터 액세스 형태입니다.

해시 함수 사용은 키와 함수 상호 작용의 통계적 속성에 의존합니다. 최악의 경우 동작은 참을 수 없을 정도로 나쁘지만 드물고 평균적인 경우 동작은 거의 최적(최소 충돌)일 수 있습니다. [3] : 527

해시 함수는 체크섬 , 체크 디지트 , 지문 , 손실 압축 , 난수화 함수 , 오류 정정 코드 및 암호 와 관련이 있으며 종종 혼동됩니다 . 개념이 어느 정도 겹치지만 각각 고유한 용도와 요구 사항이 있으며 다르게 설계 및 최적화됩니다. 해시 함수는 주로 데이터 무결성 측면에서 이러한 개념과 다릅니다 . 해시 테이블은 비암호화 해시 함수를 사용할 수 있는 반면, 암호화 해시 함수는 사이버 보안에서 비밀번호와 같은 민감한 데이터를 보호하는 데 사용됩니다.

Hash function

Hash function - Wikipedia

53 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

해시 **함수는 임의 크기의** 데이터를 고정 크기 값으로 매핑하는 데 사용할 수 있는 모든 함수 이지만 가변 길이 출력을 지원하는 해시 함수도 있습니다. [1] 해시 함수에서 반환되는 값을 *해시 값* , *해시 코드* , *해시 다이제스트* , *다이제스트* 또는 간단히 *해시* 라고 합니다 . [2] 값은 일반적으로 *해시 테이블* 이라고 하는 고정 크기 테이블을 인덱싱하는 데 사용됩니다 . 해시 함수를 사용하여 해시 테이블을 인덱싱하는 것을 *해싱* 또는 *분산 저장 주소 지정* 이라고 합니다 .

해시 함수와 연관된 해시 테이블은 데이터 저장 및 검색 애플리케이션에서 검색당 짧고 거의 일정한 시간 내에 데이터에 액세스하는 데 사용됩니다. 이는 데이터 또는 레코드 자체에 필요한 총 공간보다 약간 더 큰 양의 저장 공간을 필요로 합니다. 해싱은 정렬된 목록과 정렬되지 않은 목록 및 구조화된 트리의 비일정한 액세스 시간과 크거나 가변 길이 키의 상태 공간에 직접 액세스하는 데 필요한 지수적 저장 요구 사항을 피하는 계산 및 저장 공간 효율적인 데이터 액세스 형태입니다.

해시 함수 사용은 키와 함수 상호 작용의 통계적 속성에 의존합니다. 최악의 경우 동작은 참을 수 없을 정도로 나쁘지만 드물고 평균적인 경우 동작은 거의 최적(최소 충돌)일 수 있습니다. [3] : 527

해시 함수는 체크섬 , 체크 디지트 , 지문 , 손실 압축 , 난수화 함수 , 오류 정정 코드 및 암호 와 관련이 있으며 종종 혼동됩니다 . 개념이 어느 정도 겹치지만 각각 고유한 용도와 요구 사항이 있으며 다르게 설계 및 최적화됩니다. 해시 함수는 주로 데이터 무결성 측면에서 이러한 개념과 다릅니다 . 해시 테이블은 비암호화 해시 함수를 사용할 수 있는 반면, 암호화 해시 함수는 사이버 보안에서 비밀번호와 같은 민감한 데이터를 보호하는 데 사용됩니다.

Hash function

Hash function - Wikipedia

53 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

해시 **함수는 임의 크기의** 데이터를 고정 크기 값으로 매핑하는 데 사용할 수 있는 모든 함수 이지만 가변 길이 출력을 지원하는 해시 함수도 있습니다. [1] 해시 함수에서 반환되는 값을 *해시 값* , *해시 코드* , *해시 다이제스트* , *다이제스트* 또는 간단히 *해시* 라고 합니다 . [2] 값은 일반적으로 *해시 테이블* 이라고 하는 고정 크기 테이블을 인덱싱하는 데 사용됩니다 . 해시 함수를 사용하여 해시 테이블을 인덱싱하는 것을 *해싱* 또는 *분산 저장 주소 지정* 이라고 합니다 .

=> 아, 뭔가 매핑 함수구나!

해시 함수와 연관된 해시 테이블은 데이터 저장 및 검색 애플리케이션에서 검색당 짧고 거의 일정한 시간 내에 데이터에 액세스하는 데 사용됩니다. 이는 데이터 또는 레코드 자체에 필요한 총 공간보다 약간 더 큰 양의 저장 공간을 필요로 합니다. 해싱은 정렬된 목록과 정렬되지 않은 목록 및 구조화된 트리의 비일정한 액세스 시간과 크거나 가변 길이 키의 상태 공간에 직접 액세스하는 데 필요한 지수적 저장 요구 사항을 피하는 계산 및 저장 공간 효율적인 데이터 액세스 형태입니다.

해시 함수 사용은 키와 함수 상호 작용의 통계적 속성에 의존합니다. 최악의 경우 동작은 참을 수 없을 정도로 나쁘지만 드물고 평균적인 경우 동작은 거의 최적(최소 충돌)일 수 있습니다. [3] : 527

해시 함수는 체크섬 , 체크 디지트 , 지문 , 손실 압축 , 난수화 함수 , 오류 정정 코드 및 암호 와 관련이 있으며 종종 혼동됩니다 . 개념이 어느 정도 겹치지만 각각 고유한 용도와 요구 사항이 있으며 다르게 설계 및 최적화됩니다. 해시 함수는 주로 데이터 무결성 측면에서 이러한 개념과 다릅니다 . 해시 테이블은 비암호화 해시 함수를 사용할 수 있는 반면, 암호화 해시 함수는 사이버 보안에서 비밀번호와 같은 민감한 데이터를 보호하는 데 사용됩니다.

Item11. hashCode도 재정의 하시오

어?

동일성에서 동등성으로 equals를 재정의 했는데?

Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

어?

동일성에서 동등성으로
equals를 재정의 했는데?

hashcode도
똑같은 값이 나오게
작업이 필요하겠군!

Object 명세에서 발췌한 규약이다.

- equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashCode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. 단, 애플리케이션을 다시 실행한다면 이 값이 달라져도 상관없다.
- equals(Object)가 두 객체를 같다고 판단했다면, 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 요령

1단계. 첫 필드로 해시코드 초기화

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

1. int 변수 result를 선언한 후 값 c로 초기화한다. 이때 c는 해당 객체의 첫 번째 핵심 필드를 단계 2.a 방식으로 계산한 해시코드다(여기서 핵심 필드란 equals 비교에 사용되는 필드를 말한다. 아이템 10 참조).

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

잠깐, 왜 31이죠?

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

잠깐, 왜 31이죠?
=> 홀수 이면서, 소수(prime) 이기 때문

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

짝수가 아닌 이유는?
=> 짝수라면, 2의 배수라, 오버플로우가 일어나면 데이터를 잃는다. (시프트 연산 효과)

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

소수(prime)인 이유는?
=> 관습적으로. 정확한 이유는 모른다.
(있어빌리티구만..?)

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의

hashCode 만들기 - 요령
2단계. 핵심 필드들을 해시코드에 반영

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

잠깐, 왜 31이죠?
=> 추가로, 31은 32와 가까워서, VM이 몰래 시프트 연산으로 최적화해준다.
=> (i << 5) - i

2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
   a. 해당 필드의 해시코드 c를 계산한다.
      i. 기본 타입 필드라면, *Type*.hashCode(f)를 수행한다. 여기서 *Type*은 해당 기본 타입의 박싱 클래스다.
      ii. 참조 타입 필드면서 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(canonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다(다른 상수도 괜찮지만 전통적으로 0을 사용한다).
      iii. 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다. 배열에 핵심 원소가 하나도 없다면 단순히 상수(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.
   b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 요령

3단계. 반환

코드 11-2 전형적인 **hashCode** 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

3. result를 반환한다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 요령

3단계. 반환 그리고 ***단위 테스트 !***

코드 11-2 전형적인 **hashCode** 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

3. result를 반환한다.

hashCode를 다 구현했다면 이 메서드가 동치인 인스턴스에 대해 똑같은 해시코드를 반환할지 자문해보자. 그리고 여러분의 직관을 검증할 단위 테스트를 작성하자(equals와 hashCode 메서드를 AutoValue로 생성했다면 건너뛰어도 좋다). 동치인 인스턴스가 서로 다른 해시코드를 반환한다면 원인을 찾아 해결하자.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 요령

3단계. 반환 그리고 ***단위 테스트 !***

코드 11-2 전형적인 hashCode 메서드

```
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

3. result를 반환한다.

hashCode를 다 구현했다면 이 메서드가 동치인 인스턴스에 대해 똑같은 해시코드를 반환할지 자문해보자. 그리고 여러분의 직관을 검증할 단위 테스트를 작성하자(equals와 hashCode 메서드를 AutoValue로 생성했다면 건너뛰어도 좋다). 동치인 인스턴스가 서로 다른 해시코드를 반환한다면 원인을 찾아 해결하자.

Why?

1. 직관을 검증하고,
2. 회귀 테스트로 사용하기 위함 (뒤에서 추가 설명)

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
1. 성능을 높이겠다고, 핵심 필드 빼먹으면 안된다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
1. 성능을 높이겠다고, 핵심 필드 빼먹으면 안된다.

Why?
hash 값을 연산하는 속도는 빨라질 수 있다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
1. 성능을 높이겠다고, 핵심 필드 빼먹으면 안된다.

Why?
hash 값을 연산하는 속도는 빨라질 수 있다.
하지만 hash 품질을 떨어뜨리기 때문에,
해시 테이블의 성능을 심각하게 낮출 수 있다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
2. hashCode가 반환하는 값의 생성규칙(hash function의 구현)을 API 사용자에게 알리지 마라.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
2. hashCode가 반환하는 값의 생성규칙(hash function의 구현)을 API 사용자에게 알리지 마라.

Why?
사용자는 짱구라, 정확한 규칙을 알려주면, 문제가 있어도 그걸 써먹는다.

Item11. hashCode도 재정의 하시오

hashCode 만들기 - 주의사항
2. hashCode가 반환하는 값의 생성규칙(hash function의 구현)을 API 사용자에게 알리지 마라.

Why?
사용자는 짱구라, 정확한 규칙을 알려주면, 문제가 있어도 그걸 써먹는다.

=> 자세한 규칙을 이야기하지 않았다면,
결함이나 더 나은 해시 알고리즘을 발견했을 때, ***잠수함 패치가 가능하다!***

Item12. toString 써먹기

toString을 잘 구현한 클래스는 사용하기에 훨씬 ***즐겁고***,
그 클래스를 사용한 시스템은 ***디버깅하기 쉽다.***

Item12. toString 써먹기

toString을 잘 구현한 클래스는 사용하기에 훨씬 ***즐겁고***,
그 클래스를 사용한 시스템은 ***디버깅하기 쉽다.*** (퇴근 시간을 줄여줄 수 있다.)

Item12. toString 써먹기

toString을 잘 구현한 클래스는 사용하기에 훨씬 ***즐겁고***,
그 클래스를 사용한 시스템은 ***디버깅하기 쉽다.*** (퇴근 시간을 줄여줄 수 있다.)

=> 보기에 좋다

Item12. toString 써먹기

toString을 잘 구현한 클래스는 사용하기에 훨씬 ***즐겁고***,
그 클래스를 사용한 시스템은 ***디버깅하기 쉽다.*** (퇴근 시간을 줄여줄 수 있다.)

=> 보기에 좋다

toString 규약
간결하면서 사람이 읽기 쉬운 형태의 유익한 정보를 반환한다.

Item12. toString 써먹기

toString을 잘 구현한 클래스는 사용하기에 훨씬 ***즐겁고***,
그 클래스를 사용한 시스템은 ***디버깅하기 쉽다.*** (퇴근 시간을 줄여줄 수 있다.)

=> 보기에 좋다

toString 규약
간결하면서 사람이 읽기 쉬운 형태의 유익한 정보를 반환한다.

=> 보기 좋게 하라

Item12. toString 써먹기

Object의 toString()

- Class이름 + “@” + 16진수해시코드
- “PhoneNumber@ab7c”

vs

@Override toString()

- telecom + “-” + prefix + “-” + lineNum
- “010-1234-5678”

Item12. toString 써먹기

toString은 그 객체가 가진 주요 정보 모두를 반환하는 게 좋다.

포맷을 명시하든 아니든 **여러분의 의도는 명확해야한다.**

toString이 반환한 값에 포함된 정보를 얻어올 수 있는 API를 제공 하자.

E.O.D.