今さら聞けない!
equals()/hashCode()

--- Javaの同値判定メソッド ---

2011/6/30
Shisei Hanai

ruimo@ruimo.com
twitter: @ruimo
Facebook: www.facebook.com/unokichi
mixi: るいも
LinkedIn: Shisei Hanai

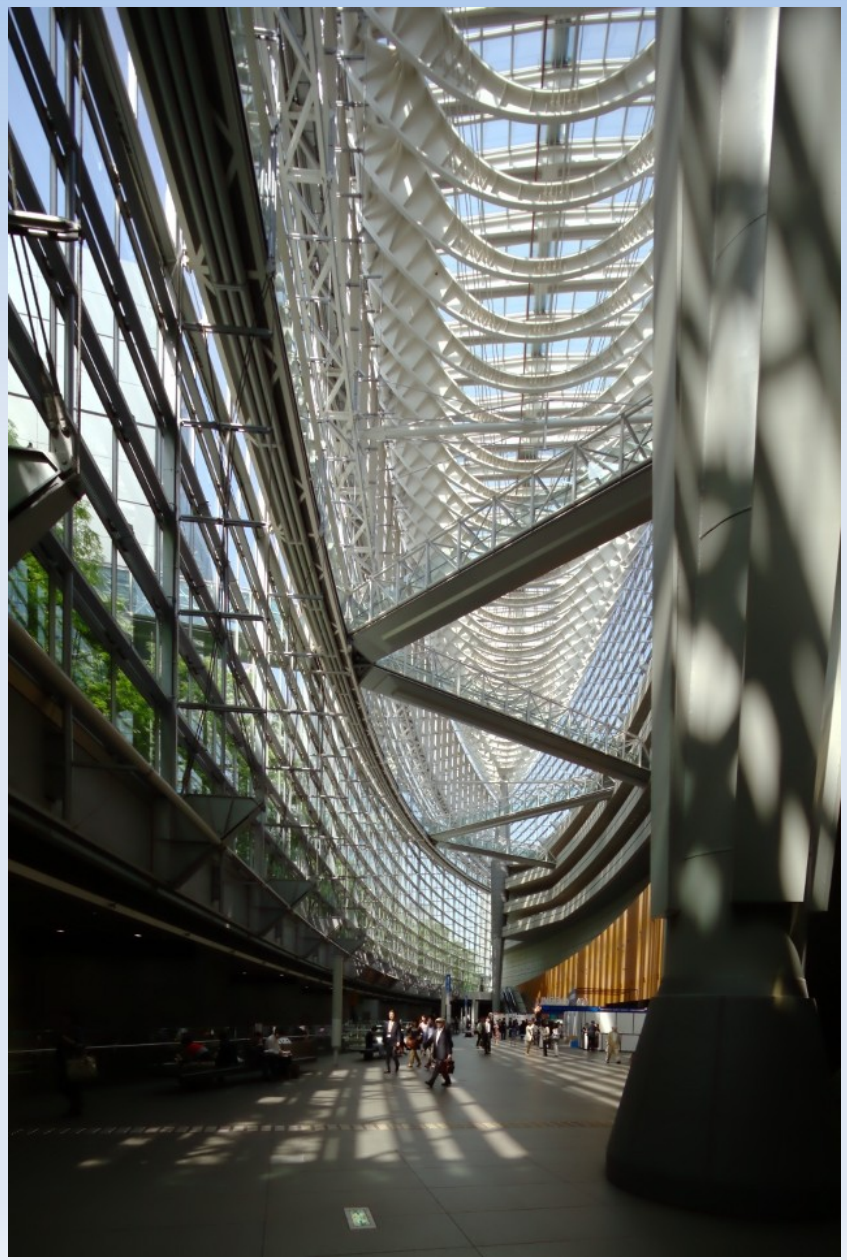
equals()とは

- Objectクラスが提供する、オブジェクトの同値性を判定するメソッド。
- 別のクラスのオブジェクト同士も比較できるように設計されている。
- デフォルト（Object.equals()）では、全てのオブジェクトは、等しくないと判定される。

hashCode()とは

- HashMap、HashSetといったハッシュを用いたコンテナのキーとして用いるオブジェクトで、ハッシュ値を提供するためにObjectクラスが提供するメソッド。
- デフォルト（Object.hashCode())では、オブジェクトごとにバラバラな値が返されるように実装される（ただし値が一意である保証はない）。

equals()の実装



equals()の実装

- Object.equals()のAPI仕様書を参照。
- nullに対してfalseを返すこと
- 反射的（reflexive）であること
- 対称的（symmetric）であること
- 推移的（transitive）であること
- 整合性を持っていること

実は結構難しい。

反射的である

- nullでない参照xに対して、x.equals(x)は常にtrueを返すこと。

悪い例

```
public class Foo {
    static int seed;

    public int getSeed() {return ++seed;}

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o.getClass() != getClass()) return false;
        Foo opponent = (Foo)o;
        return getSeed() == opponent.getSeed();
    }
}
```

対称的である

- nullでない参照、xとyに対し、x.equals(y)がtrueになるのは、y.equals(x)がtrueになる時のみである。

悪い例

```
public class Age {
    final int age;

    public Age(int age) {this.age = age;}

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o.getClass() == Integer.class)
            return ((Integer)o).intValue() == age;
        else if (o.getClass() != getClass())
            return false;

        Age opponent = (Age)o;
        return opponent.age == age;
    }
}
```

整数とも比較できるようにした。

```
Age age = new Age(23);
System.err.println(age.equals(23));    // true
System.err.println(Integer.valueOf(23).equals(age));  // false
```

推移的である

- nullでない参照、x, y, zがある時、x.equals(y)かつ、y.equals(z)なら、x.equals(z)である。
- 一見当然そうで難しい。

悪い例

```
public class Version {
    public final int version;

    public Version(int version) {
        this.version = version;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        return (o instanceof Version) &&
            ((Version)o).version == version;
    }
}
```

推移的である

- リリース番号も持たせた。

```
public class Version2 extends Version {
    public final int release;

    public Version2(int version, int release) {
        super (version);
        this.release = release;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        if (o instanceof Version2) {
            Version2 opponent = (Version2)o;
            return opponent.version == version &&
                opponent.release == release;
        }
        else if (o instanceof Version) {
            Version opponent = (Version)o;
            return opponent.equals(this);
        }
        return false;
    }
}
```

悪い例

古いVersionとも比較可能にした。

推移的である

```
Version x = new Version2(1, 2); // Ver 1.2
Version y = new Version(1); // Ver 1
Version z = new Version2(1, 3); // Ver 1.3

System.err.println(x.equals(y));
System.err.println(y.equals(z));
System.err.println(x.equals(z));
```

結果

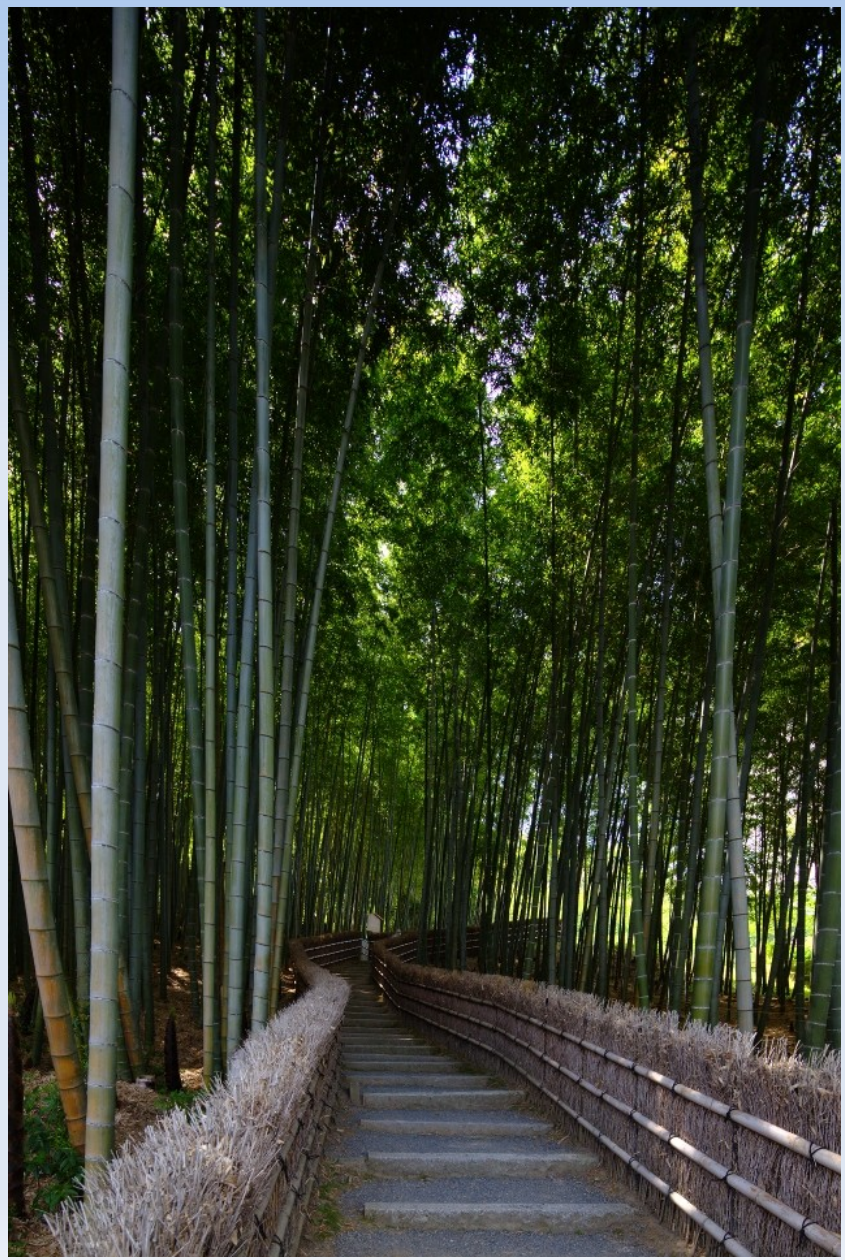
```
true
true
false
```

あれ？

整合性を持っていること

- 非nullの参照、x, yに対し、同値性に関連する状態が変わらない限り、x.equals(y)の結果は変化しないこと。

hashCode()の実装



hashCode()の実装

- equals()の結果に影響を及ぼす状態に変化が無い限り、あるJavaアプリケーション実行中には、hashCode()の値が変わらないこと。
- equals()で同値と判定される2つのオブジェクトのhashCode()は等しいこと。
- 2つのオブジェクトがequals()で同値とならなかったからといって、hashCode()が異なる必要はないが、なるべく異なるように実装することで、ハッシュ値を用いる処理が効率化される。

hashCode()の実装

```
public class Employee {
    public final String name;
    public final int age;

    public Employee(String name, int age) {
        if (name == null) throw new NullPointerException();
        this.name = name;
        this.age = age;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        if (o.getClass() != getClass()) return false;
        Employee opponent = (Employee)o;
        return opponent.name.equals(name) &&
            opponent.age == age;
    }

    @Override public int hashCode() {
        return name.hashCode() ^ age;
    }
}
```

最も簡単な実装は、equals()に関わるフィールドに対し、プリミティブ型は、値そのものを、オブジェクトはそのhashCode()をXOR演算する。

equals()とhashCode()実装の ベストプラクティス

ベストプラクティス

- equals()とhasCode()はセットで実装する。片方のみ実装するのは厳禁。
- 値オブジェクト（イミュータブル）でのみで実装した方が良い。値オブジェクトの例：String、Integer、BigDecimal
- ビジネスオブジェクトでは使用しない（同値の定義が多様）。
- 継承クラス間では同値性を定義しない方が良い。
- ORマッパのエンティティクラスはDB上でのレコードに対応させる。
- equals()/hashCode()の実装ミスの検出にはFindBugsを利用。

equals()とhashCode()はセットでオーバーライドする。

```
public class Version {
    public final int version;

    public Version(int version) {
        this.version = version;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        return (o instanceof Version) &&
            ((Version)o).version == version;
    }
}
```

hashCode()をオーバーライドし忘れ。

Setなので、同値オブジェクトは1つにまとめられるはずが…

```
        Set<Version> tbl = new HashSet<Version>
            (Arrays.asList(new Version(1), new Version(1)));
        System.err.println(tbl.size());
```

結果は2になる。

値オブジェクトでのみ利用する

- 値オブジェクトとは、
  - ある値を表現するオブジェクト
  - イミュータブル（変更不能）
  - 一般にはサイズの小さなオブジェクト
  - Nullオブジェクトパターン、Special caseパターンと併用する場合が多い。
  - 例：String、Integer、BigDecimal、...
  - 値を変更できるようにビルダークラスとセットで用意する。
    例：String <=> StringBuilder（StringBuilderは、equals/hashCodeをオーバーライドしない）。

ミュータブルオブジェクトと**equals**

- 設計を失敗している例：java.awt.Point、java.util.Date
- 日付、時刻についてはJDK 8で修正予定（JSR 310)。ようやくイミュータブルに。JDK 8までは、Joda-Timeが利用可能。

```
Point p = new Point(0, 0);
Set<Point> set = new HashSet<Point>();
set.add(p);
p.translate(10, 10);
System.err.println(set.contains(p));

for (Point po: set) {
    if (p == po) System.err.println("found");
}
```

false

found

非常に分かりにくいバグを生むので、ミュータブルオブジェクトでは、equals()の再定義を避けるのが無難。

ビジネスオブジェクトでは使用しない

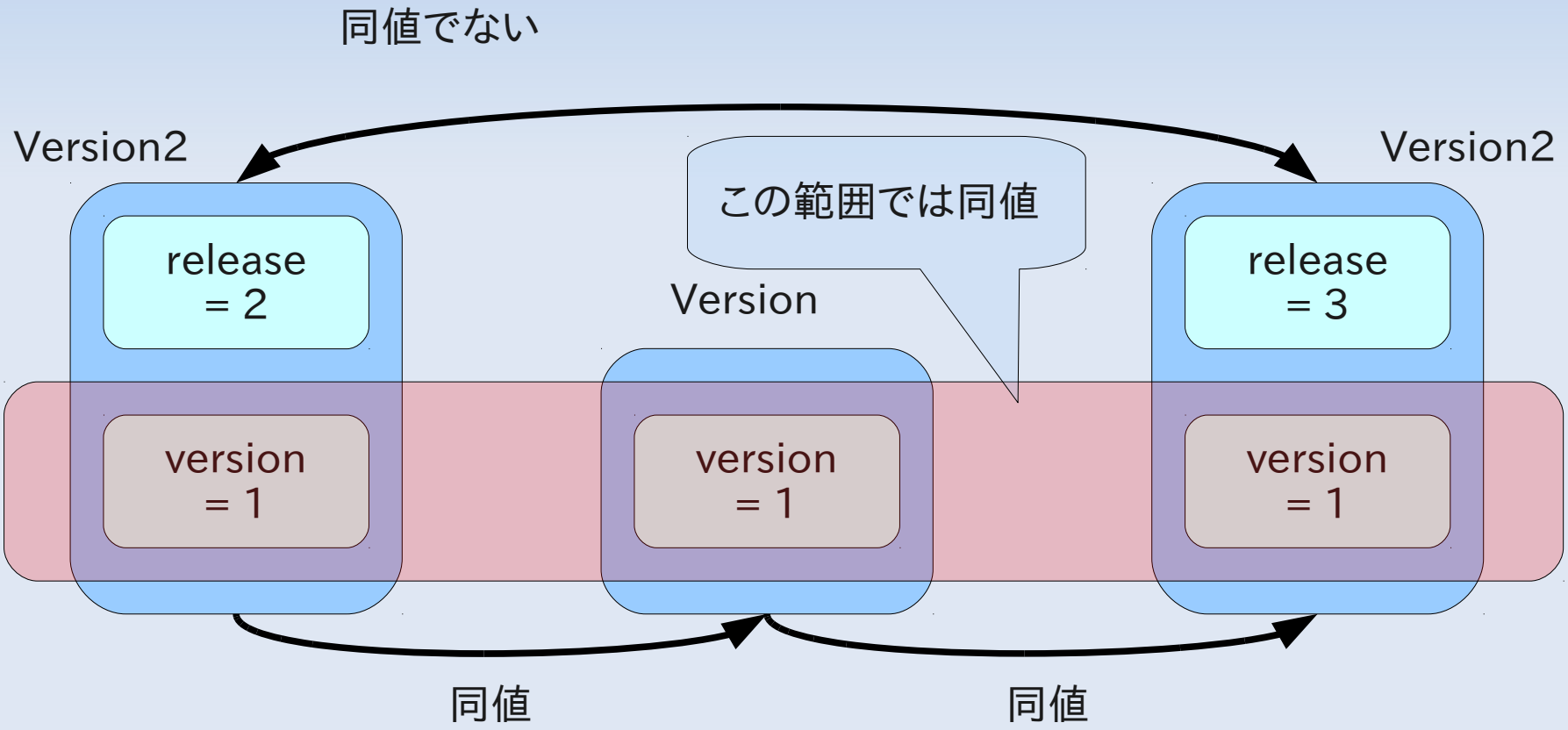
```
public class Transaction {
    final long id; // 取引番号
    final BigDecimal totalAmount; // 合計額
    final BigDecimal taxAmount; // 税額
    final BigDecimal tenderAmount; // 預り額
    final BigDecimal changeAmount; // 釣り銭

    public Transaction
        (long id,
         BigDecimal totalAmount,
         BigDecimal taxAmount,
         BigDecimal tenderAmount,
         BigDecimal changeAmount)
    {
        this.id = id;
        this.totalAmount = totalAmount;
        this.taxAmount = taxAmount;
        this.tenderAmount = tenderAmount;
        this.changeAmount = changeAmount;
    }

    // equals() ???
}
```

何を持って同値とするかは、状況による。
それでも定義するとしたら、取引番号が良いが、例えば番号が4桁で再使用されるようなケースでは不適切。

継承クラス間での同値性定義

継承クラス間で同値性を定義しない

前掲のVersion/Version2の問題



全く新規にequals()の同値性に関係するフィールドが増えていることが根本原因

継承クラス間で同値性を定義しない

うまくいくこともある。

```
abstract public class Rectangle {
    abstract public int getWidth();
    abstract public int getHeight();

    public static Rectangle getInstance(int width, int height) {
        if (width == height) return new SquareImpl(width);
        else return new RectangleImpl(width, height);
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        if (! (o instanceof Rectangle)) return false;
        Rectangle opponent =(Rectangle)o;
        return opponent.getWidth() == getWidth() &&
            opponent.getHeight() == getHeight();
    }
}
```

継承クラス間で同値性を定義しない

```
class RectangleImpl extends Rectangle {
    final int width;
    final int height;

    RectangleImpl(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override public int getWidth() {return width;}
    @Override public int getHeight() {return height;}
}
```

```
class SquareImpl extends Rectangle {
    final int length;

    SquareImpl(int length) {
        this.length = length;
    }

    @Override public int getWidth() {return length;}
    @Override public int getHeight() {return length;}
}
```

継承クラスと同値性

- 基本は継承クラス間で同値性を定義しない方が無難（instanceofを使わずに、getClass()で型を判定する）

```
@Override public boolean equals(Object o) {
    if (o == null) return false;
    if (o == this) return true;
    return (o instanceof Version) &&
        ((Version)o).version == version;
}
```

これだと継承クラス間に同値関係が持ち込まれる。

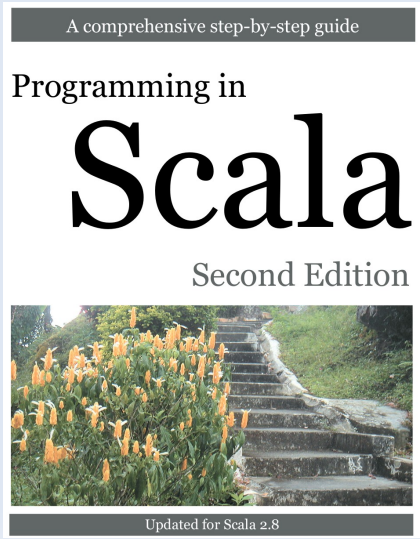
```
@Override public boolean equals(Object o) {
    if (o == null) return false;
    if (o == this) return true;
    return (o.getClass() == getClass()) &&
        ((Version)o).version == version;
}
```

これなら同一クラス内のみで同値関係が定義される。

継承クラスと同値性

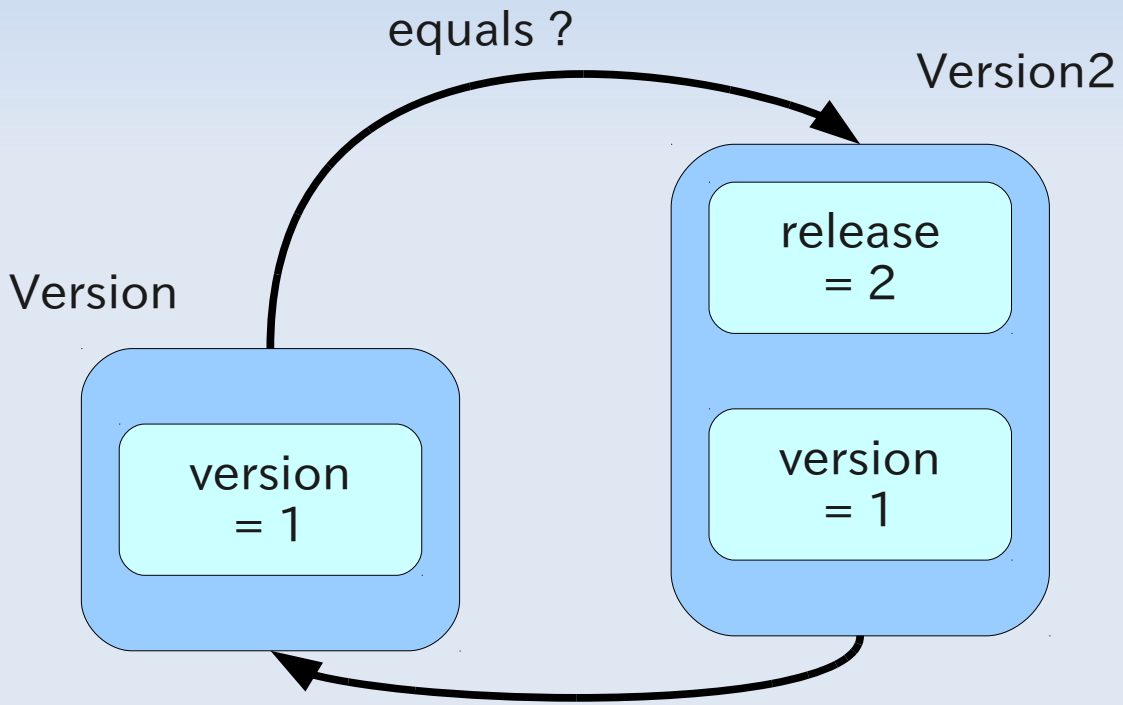
- 継承クラス間での同値性判定を一切認めないのは若干制限がキツすぎる。
- Scalaでは、同値性の制御のためのメソッド(canEqual)を1つ追加して解決しており、これはJavaでも流用できる。
- 書籍で紹介されているが、
  該当箇所は以下のサイトでも読める。

  http://www.artima.com/lejava/articles/equality.html



継承クラスと同値性

基本的な考え



同値性を判定する際、逆向きに
比較可能かを判定する。

継承クラスと同値性

```
public class Version {
    public final int version;

    public Version(int version) {
        this.version = version;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        if (o instanceof Version) {
            Version opponent = (Version)o;
            return
                opponent.canEqual(this) &&
                opponent.version == version;
        }
        else return false;
    }

    public boolean canEqual(Object o) {
        return o instanceof Version;
    }
}
```

引数に与えられたオブジェクト
は、Versionと同値比較可能ですか？

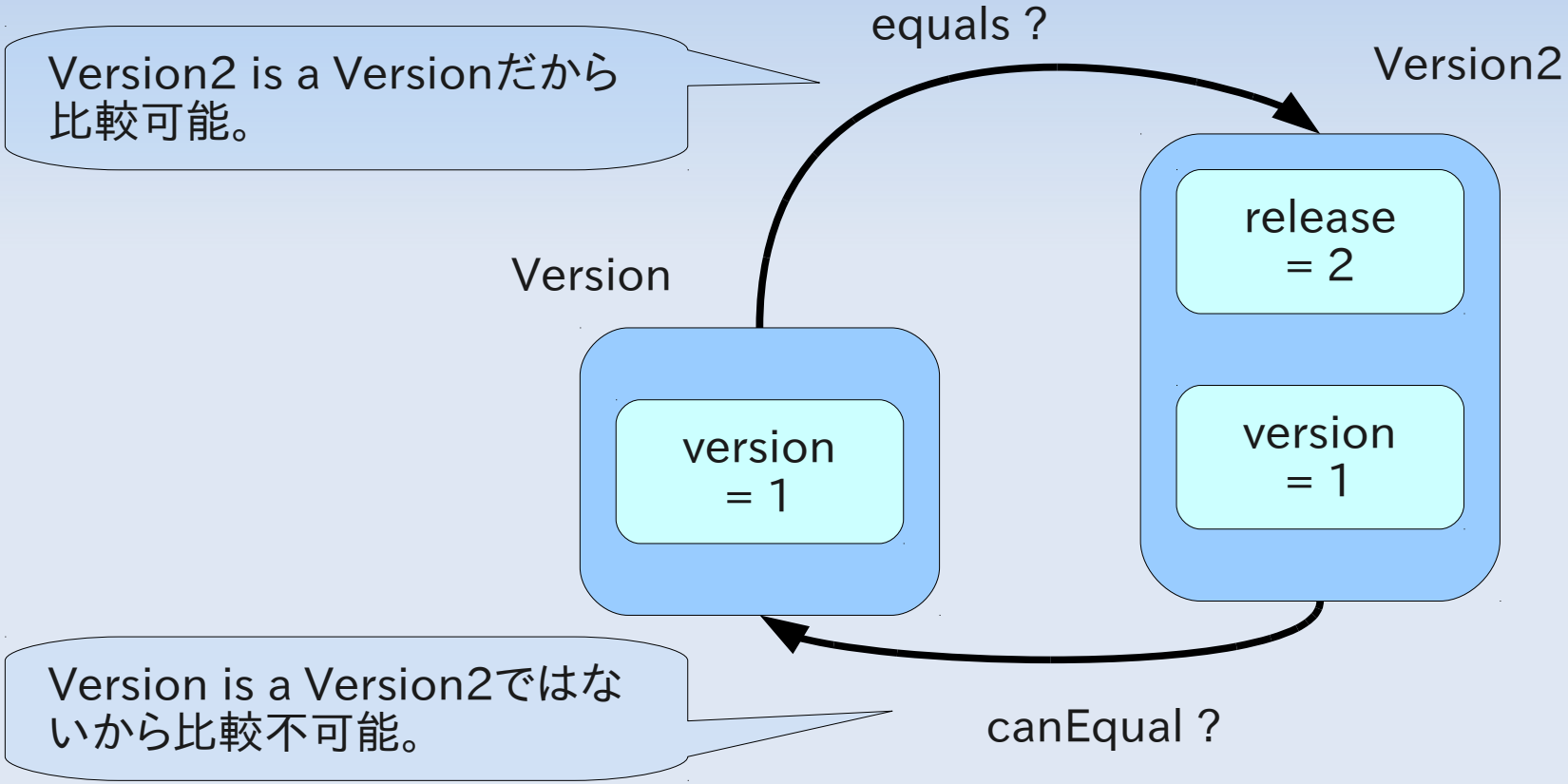
継承クラスと同値性

```
public class Version2 extends Version {
    public final int release;

    public Version2(int version, int release) {
        super (version);
        this.release = release;
    }

    @Override public boolean equals(Object o) {
        if (o == null) return false;
        if (o == this) return true;
        if (o instanceof Version2) {
            Version2 opponent = (Version2)o;
            return
                opponent.canEqual(this) &&
                opponent.version == version &&
                opponent.release == release;
        }

        return false;
    }

    @Override public boolean canEqual(Object o) {
        return o instanceof Version2;
    }
}
```

引数に与えられたオブジェクト
は、Version2と同値比較可能ですか？

継承クラスと同値性

基本的な考え



つまり同値関係を築きたく無い場合には、継承クラス側で、canEqual()を再定義すれば、同値関係は継承クラス側まで持ち込まれなくなる。

継承クラスと同値性

```
Version version = new Version(1) {
    @Override public String toString() {
        return String.valueOf(version);
    }
};

Version v = new Version(1);
System.err.println(v.equals(version));
```

同値性に無関係の拡張が行なわれた際には、canEqualを再定義しないでおく。

true

ORマッパと同値性

ORマッパのエンティティクラス

- 現在のORマッパ（JPA等）のエンティティクラスでは、テーブル間関連をSetやMapで表現できる。
- このため、エンティティクラスで適切な同値性定義が必要となる。
- エンティティオブジェクトは、ミュータブル(変更可能)なため、細心の注意が必要。

エンティティクラスのequals()

- ORマッパでは(エンベデッドバリューパターンなどを除くと)、基本的には1レコード、1オブジェクト。
- ORマッパは、アイデンティティマップパターンを用いて、レコードとオブジェクトの一意性を保証しているので、デタッチ状態(DBと切り離された状態)のオブジェクトをSetやMapのキーとして使用しないなら、equals()を再定義しなくて良い(参照一致したオブジェクトは、DB上も同一レコード)。
- あるいは拡張永続コンテキストを用いれば、デタッチ状態を気にしなくて良い(もちろんコンテキストのスコープを超えてオブジェクトを比較しなければならないなら同値性を気にしなければならない)。

全フィールドを**equals()**で比較しないこと

- 異なるレコードかどうかを識別するにはオーバースペック(レコードの一意性を決定しているフィールドだけを比較すべき)。
- エンティティオブジェクトはミュータブルなので、特定のフィールドを更新すると、同じ参照なのに異なる値のオブジェクトと認識されてしまう。
- 特にコレクションで保持しているフィールドをequals()で検査しないこと(equals()呼び出しで、lazy loadが走って著しくパフォーマンスが下がる)。

サロゲートキーを**equals()**で比較しないこと

- レコードを一意に識別する材料として、サロゲートキーは適切ではあるが、サロゲートキーは、レコードを永続化してはじめてアサインされる。
- このため、永続化前のオブジェクトを、SetやMapに格納してから、永続化するとハッシュ値が変わってしまい思わぬ挙動を招く。

ビジネスキーを**equals()**で比較すること

- これまでの問題を解決するには、ビジネスキーをequals()で比較するようにする。
- ビジネスキーとは、そのレコードを一意に識別できるキーのこと。
- 永続化する前にも値が設定されているキー。
- サロゲートキーを用いずにナチュラルキーを使用しているなら、ナチュラルキーが第一候補。
- ORマッパは知らないところでエンティティクラスのサブクラス(動的proxy)を生成している場合があるので、equals()の中ではgetClass()を用いずに、instanceofを使用すること。またフィールドには直接アクセスせずに、getterを用いること。

FindBugsでequals()/hashCode()の問題を発見する。

FindBugのバグパターン

- FindBugsには、equals()/hashCode()の実装ミスを検出するためのバグパターンが数多く用意されている。

  http://findbugs.sourceforge.net/bugDescriptions.html

Eq: Equals checks for noncompatible operand (EQ_CHECK_FOR_OPERAND_NOT_COMPATIBLE_WITH_THIS)

This equals method is checking to see if the argument is some incompatible type (i.e., a class that is neither a supertype nor subtype of the class that defines the equals method). For example, the Foo class might have an equals method that looks like:

```
public boolean equals(Object o) {
  if (o instanceof Foo)
    return name.equals(((Foo)o).name);
  else if (o instanceof String)
    return name.equals(o);
  else return false;
```

This is considered bad practice, as it makes it very hard to implement an equals method that is symmetric and transitive. Without those properties, very unexpected behavoirs are possible.

equals()/hashCode()の規約が守られているかテストで検証する。

テスト

- 手でequals()/hashCode()の規約チェックを、いちいち書くのは大変。
- 自動で検証するツールが幾つかある。比較的新しいもの：equalsverifier
  http://code.google.com/p/equalsverifier/

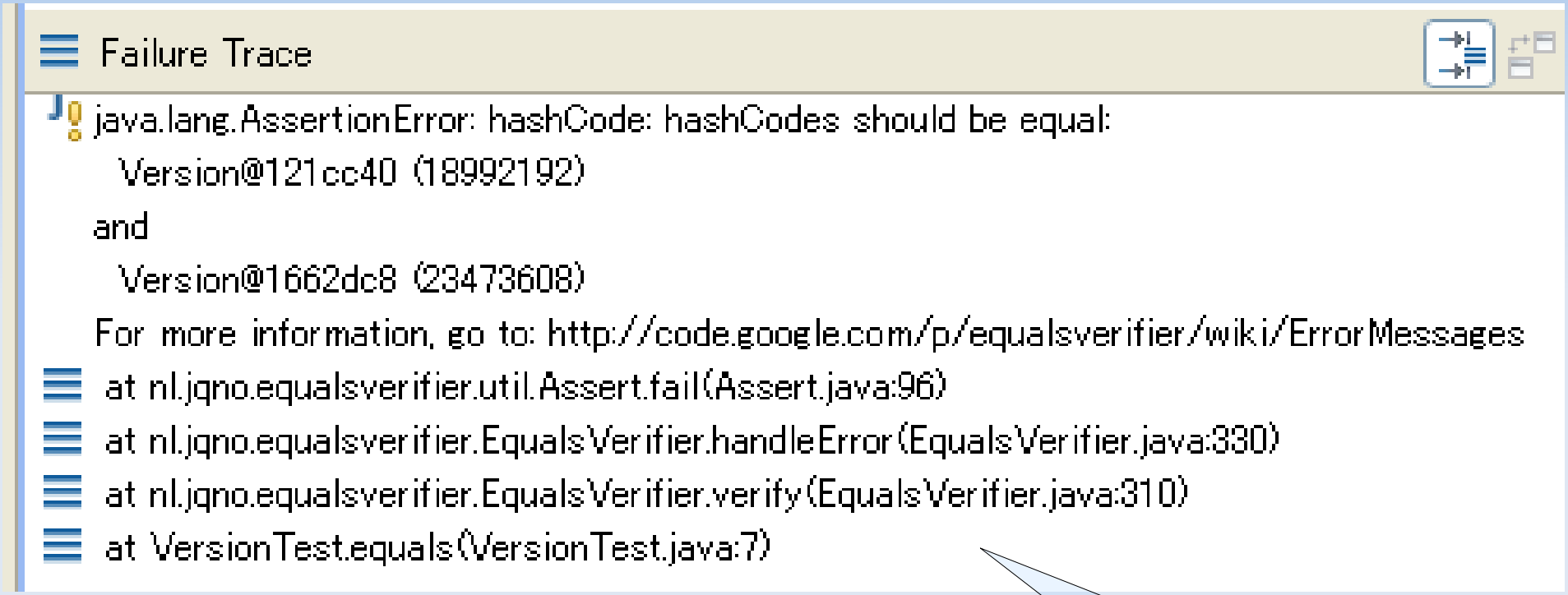
```
@Test
public void equalsContract() {
    EqualsVerifier.forClass(My.class).verify();
}
```

これだけ！

```
@Test
public void equalsContract() {
    EqualsVerifier.forClass(My.class)
            .usingGetClass()
            .verify();
}
```

instanceofではなくgetClass()を使っている時はこちら。

実行例



hashCode()を忘れた例

実行例

- ただし使いこなすには、きちんとequals()/hashCode()について理解しておく必要がある。

```
@Test public void equals() throws Exception {
    EqualsVerifier.forClass(canequal.Version.class)
    .withRedefinedSubclass(canequal.Version2.class)
    .verify();
}

@Test public void equals2() throws Exception {
    EqualsVerifier.forClass(canequal.Version2.class)
    .withRedefinedSuperclass()
    .verify();
}
```

canEqual()の説明で使った、Version/Version2のテストコード。Versionにはサブクラスがあること、Version2にはスーパークラスがあることを教える必要がある。
また、Version2のequals()/hashCode()はfinalにしておかないとエラーになる。

参考文献

- Effective Java 第2版 (The Java Series), ISBN: 489471499X
- Programming in Scala: A Comprehensive Step-by-step Guide, ISBN: 9780981531601
- Java Persistence With Hibernate, ISBN: 1932394885
- 開発のプロが教える標準FindBugs完全解説-Javaバグパターンの詳細と対策 (デベロッパー・ツール・シリーズ), ISBN: 4756146554