

Equality, Identity, and a Modified Contract

Beate Ritterbach, Axel Schmolitzky
ritterbach@informatik.uni-hamburg.de
schmolitzky@informatik.uni-hamburg.de

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg

Abstract

This paper describes a software-engineering problem, proposes a solution and shows how that solution influences language design. In many object-oriented programming languages, when implementing equality the programmer has to make sure that it obeys a set of rules, called equality contract. Not only is it difficult to adhere to these seemingly simple rules, but the equality contract itself is a source of potential errors. Even if equality complies with the contract, it can lead to faulty, unintended, indeterministic program behavior. This paper proposes a modified contract that avoids these problems. Additionally, the modified contract describes equality unambiguously, and it implies that equality for values and identity for objects can be regarded as the very same concept. Based on the modified contract the language design can be enhanced in a way that supports value equality and object identity more clearly and more safely.

1 Introduction

Equality seems to be a simple and basic concept. However, dealing with equality is not as easy as it looks. As a guideline for the programmer, many languages stipulate a set of rules – called “*equality contract*” – that the programmer should adhere to when programming equality. For example, in Java the API documentation of `java.lang.Object` specifies that the `equals` method must be an equivalence relation (i. e. *reflexive*, *symmetric* and *transitive*) for non-null references, it must be *consistent* (i. e. it must yield the same results unless one or both of the objects involved are modified) and comparing something with `null` must always yield `false`. The equality contract states formal rules for the behavior most people intuitively expect from equality. For example, if `a` and `b` are equal, we also expect `b` and `a` to be equal. The equality contract is one of the basics of Java programming; many books (e. g. [Blo08, Item 8, p. 33-44]) explain in detail how to adhere to it.

Section 2 argues that the contract itself is a source of problems. We list a number of errors that the contract can cause. Section 3 proposes a modified contract. We show that complying with the modified contract eliminates the errors stated before and that additionally the modified contract serves as a unique specification of value equality and object identity. Section 4 points out that, depending on the programming language, some conditions of the modified contract can be kept with adequate programming discipline, yet other conditions are hard to satisfy. Section 5 describes a language support for equality and identity that adheres to the modified contract, its prerequisites and its benefits.

Copyright © by the paper’s authors. Copying permitted for private and academic purposes.

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at <http://ceur-ws.org>

2 Problems with the Equality Contract

Adhering to the contract is amazingly difficult. Even in professional code and in textbooks you find `equals` methods that violate it; [LK02a] and [VTFD07] cite some examples. Many papers propose guidelines and techniques for implementing `equals` in a contract-conforming way, mostly for Java, e. g. [BV02], [Coh02], [LK02a], [LK02b], [SP03], [RH08].

This paper, however, addresses a different issue. It claims that *the contract itself is a source of problems*: it is not sufficiently restrictive to prevent erroneous code and subtle bugs. Even though equality adheres to the contract, several programming errors can occur:

1. According to the contract, equality may *depend on mutable state*, yet it leads to indeterministic, contradictory behavior, e. g. when adding an element to a collection and trying to retrieve it after it has been modified (see [OSV09]). For this reason, Vaziri et al. propose a “revised contract” that demands equality not to depend on mutable state [VTFD07].
2. If `a` and `b` are equal, then the `hashCode`s of `a` and `b` must be equal, too. This “hashcode rule” is not a part of the equality contract but a programming guideline on its own. Inadvertently or unknowingly *violating the hashcode rule* also leads to unexpected, unintended behavior in collections (see [Blo08, Item 9, p. 46]).
3. What pertains to `hashCode` actually holds true for every read-only method `q`: If `a` and `b` are equal, then `a.q()` and `b.q()` must be equal as well. As stated by Liskov and Guttag, “it should be impossible to distinguish between two equal objects” [LG86, p. 93]. As for the hashcode rule, it can be shown that *violating indistinguishability* can result in unexpected, indeterministic behavior.
4. In Java, the `equals` method is inherently *asymmetric for null references*. Because `equals` is an instance method, `null.equals(a)` throws a `NullPointerException`, whereas, according to the contract, `a.equals(null)` must yield `false`. This asymmetry results in unpredictable behavior (see, e. g. [Hor03]). Moreover, it leads to intricate, verbose code, because it forces the clients to check for `null`.
5. Since equality is usually defined in a class at the top of the hierarchy (like `Object` in Java), it can compare expressions of *incomparable types* (like `String` and `Button`). However, in such cases the result can never be `true`, and mostly the comparison is a programming error (see [VTFD07]).
6. In many object-oriented languages, there is *more than one comparison*, for example `==` and `equals` in Java. Inadvertently using the “wrong” comparison leads to subtle and hard-to-find errors. For example, comparing two `String` expressions with `==` instead of `equals` is a well-known, yet widespread source of errors in Java.

In general, there may be more than one operation that adheres to the contract. E. g., identity is one of them, see item no. 6. At first glance, this is not a problem. After all, the contract does not claim to be a unique description, but its ambiguity gives rise to questions, such as: Can a type have more than one equality? Is equality a universal concept, or does it mean something different for each type?

3 A Modified Equality Contract

To prevent problems 1 to 6, we propose a contract with more, and stronger conditions than the ones described in section 2. We call it the *modified equality contract*:

- A. Equality must *always* be an *equivalence relation* (i. e. reflexive, symmetric and transitive), this also applies to `null` references.
- B. Equality must be *independent of mutable state*. In this respect, we follow the revised contract of Vaziri et al. [VTFD07] (see section 2, problem no. 1).
- C. Equality must be *indistinguishable*: if `a` and `b` are equal, there must not exist a read-only operation `q` that yields different results for `a` and `b`. (Condition C refers solely to read-only operations, because for mutating operations in general it cannot be satisfied. If operation `m` modifies the instance it is called for, even two calls to `a.m()` can yield different results.)
- D. Equality must compare only expressions of *comparable types*.

By definition, equality that adheres to the modified contract suffers none of the problems described in the previous section. Problem 1 is prevented by condition B. Problems 2 and 3 are prevented by condition C (actually, problem 2 is a special case of problem 3), problem 4 by condition A, and problem 5 by condition D. It can be proved that there exists at most one operation that adheres to the modified contract.¹ This uniqueness solves problem 6: If there is only one equality, then there is no danger of accidental confusion.

Because of its uniqueness, the modified contract serves as an implementation-independent specification. When we define equality as the concept described by the modified contract, we receive the following answers to the conceptual questions posed in section 2: There is one equality only. Equality means the same thing for every type. Equality is the most fine-grained distinction possible (“the finest distinction” [Bak93, p. 3]). It determines what we are referring to when we talk about “one” instance of a type.

Uniqueness has yet another implication: For *objects*, the concept described by the modified contract means *identity*. For *values*, it means *value equality*.

We define objects as stateful abstractions like persons, cars, or bank accounts. Objects can basically be created, destroyed and changed (even though they don’t have to). To put it more abstractly, their operations can be referentially opaque and cause side effects. Values, by contrast, are stateless; they comprise abstractions such as numbers and characters, strings, points and monetary amounts. Values cannot be changed, created or destroyed, they exist per se. Operations on values are always referentially transparent and never cause side effects. Distinguishing objects and values separates two fundamentally different programming paradigms, with objects representing the imperative side and values representing the functional side. Table 1 lists the defining characteristics of objects and values.

Objects:	Values:
- can be created / destroyed	- are never created / destroyed
- can be mutable	- are always immutable
Operations on Objects:	Operations on Values:
- can cause side effects	- never cause side effects
- can be referentially opaque	- always referentially transparent

Table 1: Defining properties of objects and values

Distinguishing objects and values is a wide-spread modeling approach (see [Mac82], [BRS⁺98], [Fow09, p. 486-495], [Eva04, p. 97-103]). Fowler even defines values as “abstractions whose equality is not based on identity” in line with the modified contract. Liskov and Gutttag hold the view that “in the case of mutable objects, all distinct objects are distinguishable (i. e. equals has the same meaning as ==)” and “if two immutable objects have the same state, they should be considered equal because there will not be any way to distinguish among them by callig their methods.” [LG01, p. 94] Liskov and Gutttag distinguish “mutable objects” and “immutable objects” – which comes close to our distinction between objects and values, though it is not exactly the same.

The distinction of objects and values is not fully supported by current programming languages. Note that so-called “value types” in C#, e. g. `structs`, are not necessarily stateless and thus are *not* the same concept as the values mentioned above. `struct` does not support values, but “value semantics”, an implementation technique also used for primitive types. This also applies to similar language mechanisms like expanded types in Eiffel, `structs` in X10, “value objects” in Fortress etc.

Because the modified contract describes both object identity and value equality, calling this comparison “equality” is not quite appropriate. However, neither is the term “identity” appropriate, because “identity” does not include value equality. For want of a better name, we shall call that comparison “equality/identity”.

The modified contract excludes some comparisons that, in colloquial language, are also termed “equality”. For example, two cars are often regarded as “equal” if they are the same model from the same manufacturer, yet they are two distinct physical entities. In this case, the objects are distinguishable, at least by identity, and mostly by other properties as well (owner, serial number etc). Thus this “object equality” violates the modified

¹Proof: Assume that operations `eq1` and `eq2` both adhere to the modified contract. We show that `eq1(a,b)` yields `true` if and only if `eq2(a,b)` yields `true`: If `eq1(a,b)` yields `true`, then `eq2(a,b)` is the same as `eq2(a,a)` because `eq1` is indistinguishable, and `eq2(a,a)` is `true` because `eq2` is reflexive. It can be proved analogously that “`eq2(a,b)` yields `true`” implies “`eq1(a,b)` yields `true`”.

contract. Object equality and value equality are different concepts. Value equality does adhere to the modified contract, and it results, e. g., in recognizing $3/4$ and $6/8$ as the very same value.

Figure 1 depicts object identity, value equality, object equality and their relationships. It illustrates that, despite a different name, object identity and value equality can be regarded as the very same concept, whereas, despite the same name, value equality and “object equality” denote two fundamentally different concepts.

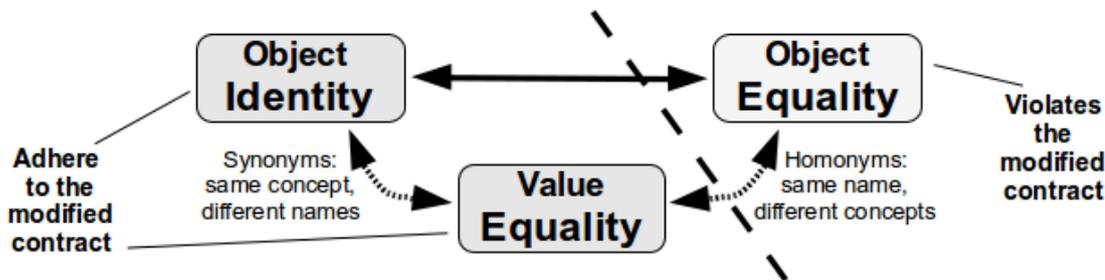


Figure 1: Object identity/ value equality versus object equality

There is no such thing as a “right” or a “wrong” equality contract. Choosing the characteristics you expect from equality is a mere matter of definition. However, rejecting the modified contract and starting out from a less restrictive contract (e. g. the one for Java) means implicitly accepting the inconsistencies and potential errors described in section 2.

In the next sections, we focus on (value) equality/ (object) identity more extensively; we do not consider “object equality” any further.

4 Programming Equality According to the Modified Contract

A contract can be regarded as a set of criteria that guide the programmer how (and how not) to implement equality. In general, the criteria are not assured or checked by the language. Programming equality that adheres to the modified contract is easier to achieve in some respects and more difficult in others.

Some conditions of the modified contract can be met with adequate programming discipline: writing equality as an equivalence relation for non-null references, avoiding dependence on mutable state, and implementing all read-only operations (especially hashCode) so that equal instances are indistinguishable.

With the understanding that for object types equality *is* identity, these tasks get even easier: For object-like, i. e. stateful types like `Person` or `BankAccount`, implement equality as a call to identity (or, if this is the standard behavior anyway, as in Java, simply refrain from overriding `equals`).

For value-like, i. e. stateless classes like `MonetaryAmount`, in many cases *canonical equality* (see [Bak93, p. 18]) – compare all matching data fields and link results by logical `And` – is appropriate and straightforward. For example, comparing two `MonetaryAmounts` `m1` and `m2` can be implemented as

```
m1.amount == m2.amount && m1.currency == m2.currency
```

Canonical equality is already the most fine-grained value comparison. Thus making equality indistinguishable does not require any further programming effort.

For some value-like classes, a more elaborate equality may be required. For example, comparing rational numbers `r1` and `r2` may need an implementation like

```
r1.numerator * r2.denominator == r2.numerator * r1.denominator
```

Without canonical equality, special care must be taken to make equality indistinguishable. E. g., in the example above an operation like `getNumerator` is not permitted because it would destroy indistinguishability: Equality recognizes $3/4$ and $6/8$ as equal, but `getNumerator` would enable clients to distinguish between them.

At first glance, it may seem strange for a class `RationalNumber` *not* to provide a method like `getNumerator`. However, it consequently conveys the notion of rational numbers as abstract entities, characterized by their operations (addition, multiplication etc). If we say that the numerator of “ $3/4$ ” is 3, then we refer to a specific representation. But the rational number denoted by “ $3/4$ ” can have more than one representation, therefore it does not make sense to ascribe it a numerator. It is exactly the purpose of a class to present an abstract view

(“specification view”) of the type towards clients and to shield them from the representation and other details that are needed for implementing the class’ behavior.

In the case of rational numbers, it is conceivable to provide an operation that computes the numerator of the reduced representation. For reasons of clarity, it should be termed something like “`getNumeratorOfReducedRepresentation`”. Programming it this way, the operation would satisfy indistinguishability, as demanded by the modified contract. For “3/4” and for “6/8”, this operation would yield the same result, 3. Possibly, this operation may be required under certain conditions. However, the result of arithmetical operations on rational numbers does not depend on their representation.

Many conditions of the modified contract are hard to adhere to. If equality is implemented as an instance method (like `equals` in Java), it is difficult to make it symmetric for `null` references. One could do so by throwing a `NullPointerException` if the parameter is `null`, violating the Java equality contract which states that `a.equals(null)` must yield `false`. For this problem, Scala has a ready-made solution: The operator “`==`” handles `null` references; it does so in a symmetrical way, and in the case of non-`null` references, “`==`” implicitly calls `equals`. From the client’s perspective, in Scala “`==`” can be regarded as a “null-safe” version of `equals`. As another consequence of this design, in Scala both object identity and value equality can be called via the operator “`==`”, provided `equals` is implemented in a way matching the object nature or value nature of the class. With respect to equality/identity, the language model of Scala is closer to the modified contract than the language model of Java.

Yet, there are still open issues both in Java and in Scala. For value-like classes, equality is *not* indistinguishable. “Equal” instances can be distinguished by “`==`” in Java, and by `eq` in Scala. These comparisons are not appropriate for value-like classes, and they violate the modified contract, but nonetheless they are always available.

In general, `equals` permits the comparison of incomparable types. The programmer of the `equals` method could implement additional type checks, which would result in runtime errors. Programming a type-safe comparison, i. e. one that is checked at compile-time, requires a language feature like *typeclasses* (as in Haskell, where instances of typeclass `Eq` are compared in a type-safe manner). In Scala, typeclasses can be emulated, using an intricate combination of traits, implicit parameters and implicit conversions; and indeed the library `scalaz` does provide a type-safe comparison (`===`). However, it adds yet one more comparison to the Scala ecosystem. This contradicts the modified contract which implies that there can be one comparison only.

A programming language that provides two or more comparisons, like Java or Scala, undermines equality/identity as a unique concept and leaves loopholes for problem no. 6 (see section 2). Sather, for example, is different in this respect; it provides one language-supported comparison only [SO96, p. 64] (the symbol “`=`” is syntactic sugar for a call to `is_eq`).

The bottom line is: depending on the programming language and its implicit assumptions, adhering to the modified contract can be hard, even virtually impossible. The examples from Java, Scala and Sather described in this section give an idea of many subtle differences in the ways programming languages handle equality/identity.

5 Language Support for the Modified Contract

As section 4 has shown, it is the programming language that prevents implementing equality so that it adheres to all conditions of the modified contract. Therefore a dedicated language support for equality can solve the problem. Section 4 has also shown that, to a certain extent, language support for equality already exists. Equality is deeply interwoven with programming language design. It is a basic concept; as Odersky puts it, “equality is at the basis of many things” [OSV09]. This section will show that language support for equality/identity, as specified by modified contract, is possible and beneficial.

Supporting equality/identity by the language requires separating objects and values on the language level. The language must provide two kinds of classes, object classes and value classes. Otherwise, the language cannot “know” if a class is meant to model object-like or value-like abstractions and therefore, if object identity or value equality is the appropriate comparison. For example, a class with two data fields might as well model an object type (like `Person` with `name` and `salary`) or a value type (like `MonetaryAmount` with `amount` and `currency`). Nevertheless, generating an appropriate comparison is not the primary motivation and by far not the only reason for the separation. Objects and values denote two fundamentally different concepts. Distinguishing between them enhances modeling power, clarity and safety of a language in many respects (see [Mac82], [BRS⁺98], [Fow09, p. 486-487], [Eva04, p. 81]). For example, it can ensure the conceptual properties of objects and values respectively.

In the following points, we shall describe a language support for equality/identity:

- For each class the language provides exactly one comparison.
- The language ensures that the comparison matches the type, i. e. object classes are compared by object identity, value classes by value equality.
- Whenever possible, the language generates an implementation for the comparison. For object classes, the language provides object identity as a language primitive, e. g. based on comparing storage addresses – just as many current object-oriented languages already do. That way, object identity cannot (and need not) be implemented by the programmer. For value classes, the language provides canonical equality (comparing all matching data fields, see section 4) as the standard implementation. As section 4 has argued, canonical equality is just what many value classes need. As has been shown previously, there may be value classes where canonical equality is not suitable. For these cases, the language provides a mechanism that enables the programmer to implement value equality differently.
- The unified comparison is denoted by a single symbol (or a single keyword). Hence clients always call equality/identity with the same symbol, no matter whether it refers to objects or values, and no matter whether it was generated by the language or implemented by the programmer. Since equality/identity is frequently used, a short notation is advisable. (The symbol “=” is an obvious candidate, because it has been used in mathematics for a long time.)
- The language handles `null` cases. If the right side or the left side of a comparison yields `null`, then the result of the comparison will be `false`.
- The language takes care of incomparable types. Comparing expressions with incomparable types results in a compile time error.

Equality/identity as just described differs largely from the way current object-oriented languages treat equality and identity. Language support for equality/identity, as sketched above, has numerous advantages:

- The comparison is completely – or at least largely – under the control of the language. In the cases of object identity and canonical value equality – both generated by the language – *all* conditions of the modified contract are taken care of by the language. In case of a manually implemented value equality, the programmer has to make sure that equality is an equivalence relation and that it is indistinguishable. Even then, the other conditions of the modified contract can be taken care of by the language. This applies especially to conditions that, for a programmer, are hard or even impossible to ensure: The language can handle `null` references, and it can preclude comparing incomparable types. Because value classes do not possess mutable state, in all cases value equality is necessarily independent of mutable state. In summary, many problems that are frequently associated with programming and using equality (see section 2) are no longer possible. This enhances language safety.
- The language support precludes two kinds of potential errors:
There is no way of confusing equality/identity with “object equality” (see section 3). However, this does not mean that object equality cannot be implemented with such a language. With the approach described above, object equality is just a user-defined operation like any other: it gets no special language support. Just like any other similar operation, the programmer has to implement object equality manually and give it a suitable name.
There is also no way of using an identity-like comparison for values, e. g. based on a technical address. For value classes, such a comparison would have no meaning, it would allow implementation details to “leak through” to the programmer (see [Bak93, p. 6]), and it would violate indistinguishability and thus break the modified contract.
- The largest benefit of language support for the modified contract is a gain in conceptual clarity. When the language provides two kinds of classes (value and object) and a single comparison – instead of one type of class (object) and two or more comparisons – and when it uses the same comparison symbol for every class, then it expresses clearly that it regards object identity and value equality as the very same concept, leaving less room for ambiguity and erroneous behavior.

The language model described in this paper might preferably be taken into consideration when designing a *new* programming language. Current object-oriented programming languages like Java, C# or Scala are implicitly based on language models that differ substantially from the one presented here. The attempt to incorporate equality/identity – and its prerequisite, the separation of object classes and value classes – into an existing language would destroy upwards compatibility and is therefore less promising.

References

- [Bak93] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *SIGPLAN OOPS Mess.*, 4(4):2–27, 1993.
- [Blo08] Joshua Bloch. *Effective Java*. The Java series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition, 2008.
- [BRS⁺98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, and Heinz Züllighoven. Values in Object Systems. Technical report, UBS AG, Zurich, Switzerland, 1998.
- [BV02] Joshua Bloch and Bill Venners. Josh Bloch on Design, A Conversation with Effective Java Author. <http://www.artima.com/intv/blochP.html>, 1 2002.
- [Coh02] Tal Cohen. How Do I Correctly Implement the equals() Method? *Dr. Dobb's Journal*, 5 2002.
- [Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, MA, 2004.
- [Fow09] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, MA, 2009.
- [Hor03] Cay Horstmann. Some Objects are More Equal Than Others. <http://www.artima.com/weblogs/viewpost.jsp?thread=4744>, 5 2003.
- [LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.
- [LG01] Barbara Liskov and John Guttag. *Program Development in Java : Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
- [LK02a] Angelika Langer and Klaus Kreft. Secrets of equals() - Part 1, Not all implementations of equals() are equal. *JavaSolutions*, 4 2002.
- [LK02b] Angelika Langer and Klaus Kreft. Secrets of equals() - Part 2, How to implement a correct slice comparison in Java. *Java Solutions*, 6 2002.
- [Mac82] B. J. MacLennan. Values and objects in programming languages. *SIGPLAN Not.*, 17(12):70–79, 1982.
- [OSV09] Martin Odersky, Lex Spoon, and Bill Venners. How to Write an Equality Method in Java. <http://www.artima.com/lejava/articles/equality.html>, 6 2009.
- [RH08] Chandan R. Rupakheti and Daqing Hou. An empirical study of the design and implementation of object equality in Java. In *CASCON '08*, pages 111–125, NY, 2008. ACM.
- [SO96] David Stoutamire and Stephen Omohundro. The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, 8 1996.
- [SP03] Daniel E. Stevenson and Andrew T. Phillips. Implementing object equivalence in Java using the template method design pattern. *SIGCSE Bull.*, 35(1):278–282, 2003.
- [VTFD07] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative Object Identity Using Relation Types. In *Proc. ECOOP 2007*, pages 54–78. Springer, 2007.