Relative Values

Defining Comparison Methods for Value Objects in Java

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

December 2005

Abstract

Nowadays we are apt to be conditioned to believe that value rests on quantity rather than quality, and so we take it for granted that large or obvious shapes matter more than small or subtle ones and that the total effect of a thing as a work of art depends on the gross shape of it – the sort of shape you can describe in a full-size drawing. We pay no attention to such mere kickshaws as surface quality and fine detail; yet their aggregate effect is very nearly as important. To achieve anything worthy of to be called quality you will have to do a good deal more than follow a drawing and specification, whoever made them and however carefully.

David Pye, *The Nature & Aesthetics of Design*

This paper presents and connects three patterns for partitioning the logic of comparison operations for value objects in Java: OVERRIDE–OVERLOAD METHOD PAIR, TYPE-SPECIFIC OVERLOAD, and BRIDGE METHOD. This mini language forms a natural extension to an existing pattern language, *Patterns of Value*, which addresses the idioms suitable for defining classes for value objects in Java.

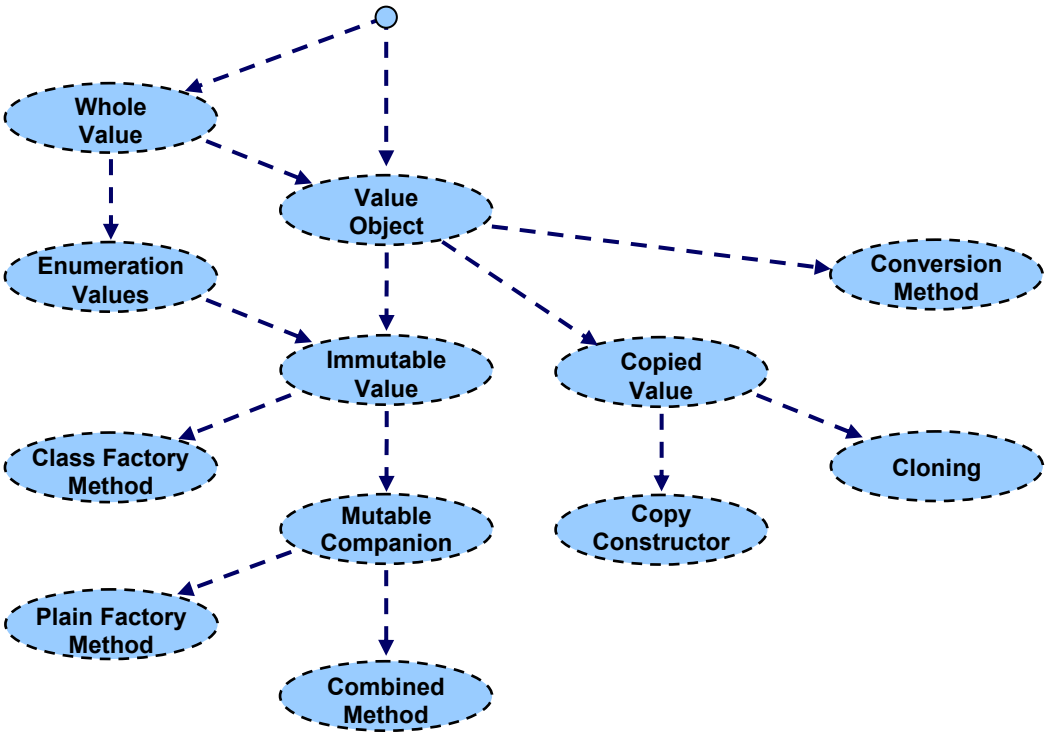
Content-based equality comparison, expressed though the `equals` method, is an intrinsic property of all value objects, e.g. date objects should compare based on the values of their fields not on their respective object identities. Relational comparison, often expressed as the `compareTo` method of the `Comparable` interface, is an additional property of some value objects. Although related in the sense that they are both concerned with comparison, these methods also share some similarities with respect to their internal logic. The common approach to implementation is simply to define all the logic within each method directly. However, this can lead to clumsy control flow in a method or unnecessary type casting at runtime. Partitioning the logic into an OVERRIDE–OVERLOAD METHOD PAIR, realized as a monomorphic TYPE-SPECIFIC OVERLOAD and a polymorphic BRIDGE METHOD, offers a cleaner separation of concerns. This method may seem novel to some readers, but it has prior art in existing application source code, published advice, the implementation of the standard Java library, and in the underlying mechanism that realizes the J2SE 5.0 generics model.

Patterns of Value

The *Patterns of Value* pattern language for value-based programming in Java has evolved over a number of years, from its pre-history as the IMMUTABLE VALUE seed pattern [Henney1997], to its first write up as a pattern language [Henney2000a, Henney2000b], and various treatments of some of its individual patterns since (e.g. CLASS FACTORY METHOD [Henney2003b], IMMUTABLE VALUE and MUTABLE COMPANION [Henney2003c]). In the fashion of any other work of this nature, *Patterns of Value* is a work in progress that grows and reacts in response to the realizations of the author, suggestions from others, and the changes in the Java language and its core libraries.

In an object system, values are fine-grained, stateful objects used to express quantities and other simple information in a system. Object identity is not significant for a value but state is. However, Java is a reference-based language, so that the principal means for passing objects around is via their identity. Classes and subclassing provide the mechanism for extending Java's type system, but not in a way that is consistent with the primitive value types built into the language, such as `int` and `char`. Hence value-based programming must be treated largely as a matter of idiom than of mechanism.

Excluding the patterns presented in this paper, the following diagram outlines the structure of the language as the author considers it to be at the time of writing:



The diagram shows inclusion relationships between the patterns: the use of one pattern to fulfill the solution of a dependent pattern. Whether usage is optional or required, complementary or mutually exclusive, preferred or less desirable is not shown, although such rules of combination must be a part of any pattern language, whether expressed diagrammatically or in prose. For example, a WHOLE VALUE is completed as a VALUE OBJECT, whereas an IMMUTABLE VALUE is not necessarily accompanied by a MUTABLE COMPANION; an IMMUTABLE VALUE is preferred to a COPIED VALUE; a VALUE OBJECT would not normally be both an IMMUTABLE VALUE and a COPIED VALUE.

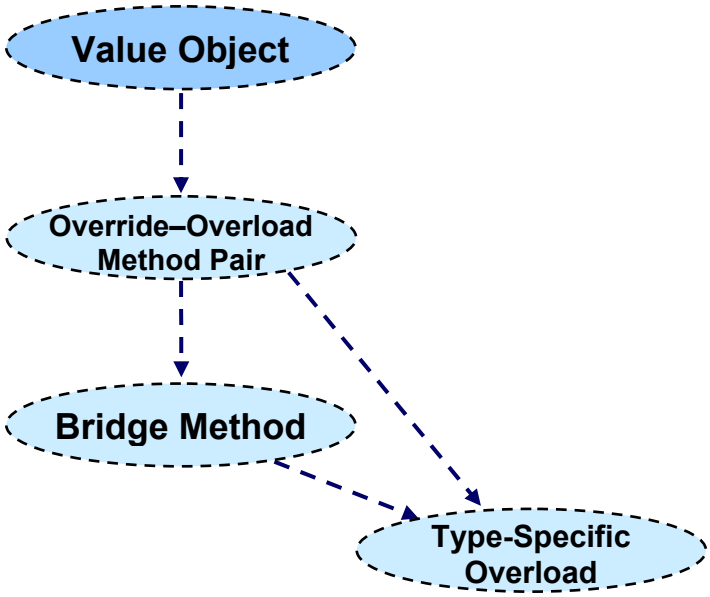
Patterns of Comparison

A VALUE OBJECT's raison d'être is in terms of the state it represents and not the identity it might have as a runtime object. Working with a value inevitably draws comparisons with other values. Such comparisons need to be based on information content rather than identity. However, in the case of equality the default `equals` method inherited from `Object` is based on identity, using the `==` operator on object references, and not on fields. Invariably a value type requires its own overridden `equals` method.

Some value types have a natural total ordering, whether magnitude (e.g. 6th April 2002 is later than 21st February 1998) or lexicographical (e.g. "zebra" sorts after "aardvark" in a dictionary). Ideally these types would provide relational operators for comparison, but deficiencies in the Java language prevent such an obvious solution, so comparisons must be carried out via named methods. The conventional protocol for this is to implement the sole method of the `Comparable` interface, `compareTo`, which offers `strcmp`-based semantics for its `int` result. Non-standard protocols include providing a `lessThan` or equivalent method.

However, it is not only their role in comparison that brings together consideration of `equals`, `compareTo`, and other like-minded methods, but the similarity of their common logic: each method is covariantly dependent on the argument type for its result. The question that the patterns in this paper seek to answer is not so much what criteria are used to determine equality or ordering in a VALUE OBJECT, but how to partition the logic and realization for these methods most effectively. The meaning of comparison is left to the VALUE OBJECT pattern to consider, as is the motherhood advice to override `hashCode` whenever `equals` is overridden.

The connected patterns in this paper can be considered either as a corner of the *Patterns of Value* pattern language or as a mini language in their own right:



As a mini language, *Relative Values* could be scoped more generally, dealing with other overrides that have a strong covariant element to their logic. However, in spite of such potential for generality, the scope presented here is specific and focused on extending the *Patterns of Value* language, using the standard `equals` and `compareTo` methods as examples. It is left to the reader to see the recurrence of these three patterns in other designs.

OVERRIDE–OVERLOAD METHOD PAIR

VALUE OBJECTs are fine-grained, stateful objects used to express quantities and other simple information in a system. Object identity is not necessarily significant for comparing one value with another, but state is. The correct semantics for comparison can be provided by defining the appropriate methods. However, comparison protocols expressed in Java are typically polymorphic and covariant: the result of the comparison depends on the comparison method called and the actual type of the argument passed, which both depend on the type of the target object. Any implementation of comparison has to mix both type-specific comparison logic and the covariant checking of the argument. Even when both the left- and right-hand sides of the comparison have the same declared type, so that polymorphism is not required, the types are still laundered.

The most commonly used comparison operation is `equals`, which is introduced into the Java class hierarchy with a default, identity-based implementation in `Object`. Introducing it at the root of the hierarchy allows arbitrary equality comparison between any object and any other object (or `null`). This is important because genericity in Java has traditionally been achieved via this ability to ignore the specific type of any object and generalize it to `Object`, allowing heterogeneous collections and class-independent frameworks, as well as forming the basis for the generics mechanism in J2SE 5.0. Hence, any override of the `equals` method must account for the dynamic type of its argument, whose static type is necessarily declared as `Object`. On the other hand, except in a few special cases, a VALUE OBJECT's class should be declared `final`, so the extent of polymorphism is bounded.

Among other problems a failure to restrict inheritance can lead to contract failures in the implementation of `equals`, namely the requirements that it should be symmetric and transitive [Bloch2001, Langer+2002]. For example, if a date class were extended to include time of day, an instantiated date–time object would correctly determine that it could not be equal to any plain date object because of declared type incompatibility. However, if the comparison were reversed, so that a plain date object compared itself to a date–time object, it might miss the fact that not only was the other object a date, it was little bit more as well, and therefore of a different type and in possession of more state.

Type recovery is also significant for the relational comparison interface, `Comparable`, whose `compareTo` method is bound by a contract more strictly covariant than `equals`: any mismatch between the type of the target and the type of the argument can be met with a `ClassCastException`. It is not considered meaningful to relate objects of arbitrary types, e.g. sorted collections of objects should be type homogeneous rather than heterogeneous. In common with most object-oriented languages, Java supports single-dispatch runtime polymorphism rather than multiple-dispatch runtime polymorphism. Multi-methods would help to ease this particular tension between dynamic dispatch and a static type system. As with `equals`, this aspect of the type system must be handled explicitly by the programmer with casts, the `instanceof` operator, or the `getClass` method, as appropriate.

The following `Date` class, expressed as an IMMUTABLE VALUE using J2SE 1.4, illustrates the standard method signatures for equality and relational comparison:

```
public final class Date implements Comparable ...
{
    ...
    public boolean equals(Object other) ...
    public int compareTo(Object other) ...
    ...
}
```

Use of a `Date` objects falls into two categories: either it will be type specific, in terms of references declared to be of type `Date`, or it will be in terms of more general types, with references declared of type `Object` or `Comparable`. In either case, the methods above will be called. Although not normally an issue, the additional dynamic type checking in the case where the declared types match one another and the class of the object seems redundant. The exact types are known at the point of call, so in principle there is no need for any additional dispatch. The polymorphic comparison method is unaware of this, but is the only option, combining both the type filtering and the detailed comparison logic.

Therefore, divide the responsibility for comparison between VALUE OBJECTs across two public methods: one method, the BRIDGE METHOD, for when polymorphic comparison is needed; the other method, the TYPE-SPECIFIC OVERLOAD, for when the class and the declared type of both the references all match. The methods are overloaded and their separation is transparent to the calling code.

The two roles of a polymorphic comparison method are divided across two overloaded public methods: one overload is a TYPE-SPECIFIC OVERLOAD, holding the logic relevant when there is an exact type match, and the other is a BRIDGE METHOD, overriding the superclass method to handle the dispatch logic relevant when polymorphic comparison is genuinely required. Thus, where there is some covariant argument dependency an OVERRIDE–OVERLOAD METHOD PAIR splits the monomorphic and polymorphic aspects of comparison into two overloaded variants.

Both methods in the OVERRIDE–OVERLOAD METHOD PAIR are bound by the same contract, so the consistency between them is more than signature deep. For `equals` the contract, when stated fully, requires reflexivity ("I am myself"), symmetry ("If you're the same as me, I'm the same as you"), transitivity ("If I'm the same as you, and you're the same as them, then I'm the same as them too"), consistency ("If there's no change, everything's the same as it ever was"), null inequality ("I am not nothing"), `hashCode` equality ("If we're the same, we also share the same magic numbers"), and no exceptions to be thrown ("If you call, I won't hang up"). For `compareTo` the contract requires asymmetry, transitivity, consistency, and admits a `ClassCastException`.

The two method variants should be textually located next to one another to ensure that the reader does not mistake the intent or believe that the intent is a mistake:

```
public final class Date implements Comparable
{
    ...
    public boolean equals(Object other) ... // Bridge Method
    public boolean equals(Date other) ...   // Type-Specific Overload
    public int compareTo(Object other) ...  // Bridge Method
    public int compareTo(Date other) ...    // Type-Specific Overload
    ...
}
```

Although not the most common approach to expressing these methods, an OVERRIDE–OVERLOAD METHOD PAIR expressed as TYPE-SPECIFIC OVERLOAD and a BRIDGE METHOD can be found in the standard Java libraries, in the implementation of generics in J2SE 5.0, in application code, and also as published advice [Henney2002]. The common approach to expressing such comparisons, where the type handling and field comparison are handled in the same method, can be considered to be less effective in many ways, not least because it often makes control flow a messy and jumpy affair. Good practice must be considered more a matter of sound rationale than a matter of democracy — popularity is not necessarily an indication of goodness or reason.

TYPE-SPECIFIC OVERLOAD

The responsibility for comparison between VALUE OBJECTs has been divided across two methods: one method for when polymorphic comparison is needed; the other method for when the class and the declared type of both the references all match.

When two objects of two references are compared, the need for polymorphism (and therefore covariant argument checking) is apparent because either reference type can be declared of a more general object type. However, when such variation is neither present — because the declared types are all specific — nor possible — because the class permits no further extension — the question of polymorphism and type checking disappears completely.

Therefore, the monomorphic comparison can focus specifically on the logic of comparison. It does not need to handle type recovery for the argument type. Static checking and binding ensure that this variant is called when the declared reference types involved match one another and that of the VALUE OBJECT class.

With a TYPE-SPECIFIC OVERLOAD there is no type filtering when there is no need for it. This eliminates runtime type checking and filtering control flow, reducing the need for additional temporary variables or complex expressions to express the comparison:

```
public final class Date implements Comparable
{
    ...
    public boolean equals(Date other)
    {
        return other != null &&
               year  == other.year &&
               month == other.month &&
               day   == other.day;
    }
    ...
    public int compareTo(Date other)
    {
        return (year  – other.year)  * 10000 +
               (month – other.month) * 100 +
               (day   – other.day);
    }
    ...
    private final int year, month, day;
}
```

In the event of a change in comparison logic, such as caused by a change in representation or an alternative algorithm of choice, the TYPE-SPECIFIC OVERLOAD need to be modified accordingly.

Note that a field-based representation (YYYY-MM-DD) is used for `Date` to emphasize the practical coding issues programmers may encounter when defining these methods in similar or more complex classes. A simpler, scalar representation (e.g. the number of days since 1st January 1900) offers different benefits and should obviously be considered, depending on the application [Henney2003a].

BRIDGE METHOD

The responsibility for comparison between VALUE OBJECTs has been divided across two methods: one method for when polymorphic comparison is needed; the other method for when the class and the declared type of both the references all match. The polymorphic method needs to handle type recovery for the argument type as well as the logic of comparison.

Intertwining the logic of type recovery and of representation-related comparison is likely to be messy as these are two different concerns expressed at different levels. The representation-related part is also likely to be essentially the same as that implemented in the TYPE-SPECIFIC OVERLOAD, leading to code duplication as well as complexity:

```
public final class Date implements Comparable
{
    ...
    public boolean equals(Object other)
    {
        Date otherDate = other instanceof Date ? (Date) other : null;
        return otherDate != null &&
               year  == otherDate.year &&
               month == otherDate.month &&
               day   == otherDate.day;
    }
    ...
    public int compareTo(Object other)
    {
        Date otherDate = (Date) other; // throws ClassCastException on mismatch
        return (year  – otherDate.year)  * 10000 +
               (month – otherDate.month) * 100 +
               (day   – otherDate.day);
    }
    ...
    private final int year, month, day;
}
```

There are many different ways to express the logic and the control flow, and the version shown here probably has the fewest control flow jumps of any approach. However, the aim is not to explore them here: none of them offer an entirely satisfactory or clean expression of the logic. A clearer separation of the different roles in the flow is what is needed. There is a type-enquiry part, which on mismatch leads unconditionally to a `false` result in the case of `equals` and an exception in the case of `compareTo`, and a representation-related part, which works with the field representation of the left-hand and right-hand sides directly and in terms of the correct static type. This representation-related part already exists in the form of the TYPE-SPECIFIC OVERLOAD.

Therefore, express the polymorphic comparison method in terms of the TYPE-SPECIFIC OVERLOAD. Call the already implemented comparison logic once a type filter has been satisfied, ensuring that the forwarding call casts the comparison argument to the static type expected by the TYPE-SPECIFIC OVERLOAD.

The anatomy of the BRIDGE METHOD is essentially filter-and-forward. Specifically, for `equals` the filter is a logical predicate and for `compareTo` it is an exception-generating guard:

```
public final class Date implements Comparable
{
    ...
```

```
    public boolean equals(Object other)
    {
        return other instanceof Date && equals((Date) other);
    }
    ...
    public int compareTo(Object other)
    {
        return compareTo((Date) other); // throws ClassCastException on mismatch
    }
    ...
}
```

The BRIDGE METHOD acts purely as a bridge between the monomorphic TYPE-SPECIFIC OVERLOAD, where all the representation-specific comparison logic is held, and the runtime polymorphic demands of callers working in terms of general types. A separation of roles has eliminated duplication and resulted in simplified logic across more focused methods. Should the private representation ever change, only the TYPE-SPECIFIC OVERLOAD will need to be modified to cater for changes in comparison logic: the BRIDGE METHOD is a locus of stability that need not change.

Note that when working with J2SE 5.0 (or later), a programmer providing relational comparison based on the `Comparable` interface has only to provide the TYPE-SPECIFIC OVERLOAD:

```
public final class Date implements Comparable<Date>
{
    ...
    public boolean equals(Object other) ...
    public boolean equals(Date other) ...
    public int compareTo(Date other) ...
    ...
}
```

The compiler provides the BRIDGE METHOD. J2SE 5.0 programmers are not required to follow this generic approach, and may continue to follow the traditional approach if they wish. However, it seems reasonable to take the compiler up on its offer.

Acknowledgments

I would like to thank Alan O'Callaghan for his shepherding of this paper for VikingPLoP 2004, and Jon Jagger and Angelika Langer for their comments on drafts of this paper. From the VikingPLoP 2004 workshop I would like to thank Neil Harrison, Osama Mabrouk Khaled, Juha Parssinen, Vladimir Pavlov, Vladimir Rancic, and Kristian Elof Sørensen for their time and feedback.

Specific References for Relative Values

[Bloch2001] Joshua Bloch, *Effective Java*, 2001, Addison-Wesley.

[Henney2002] Kevlin Henney, "Value-Based Programming in Java", the *JAOO* conference, September 2002, available from `http://www.curbralan.com`.

[Henney2003a] Kevlin Henney, "The Taxation of Representation", *The Road to Code* blog at *Artima*, 30th July 2003, `http://www.artima.com/weblogs/viewpost.jsp?thread=8791`.

[Langer+2002] Angelika Langer and Klaus Kreft, "Secrets of `equals`", *C/C++ Users Journal Java Supplement*, April 2002, `http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html`.

Background References for Patterns of Value

The *Patterns of Value* pattern language has benefited directly and indirectly from the work and feedback of others, some of whom can be found listed in the references or acknowledgments of papers listed here.

[Henney1997] Kevlin Henney, "Java Patterns and Implementations", *BCS OOPS Pattern Day*, October 1997, available from `http://www.curbralan.com`.

[Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from `http://www.curbralan.com`.

[Henney2003b] Kevlin Henney, "Factory and Disposal Methods", *VikingPLoP 2003*, available from `http://www.curbralan.com`.

[Henney2003c] Kevlin Henney, "The Good, the Bad, and the Koyaanisqatsi", *VikingPLoP 2003*, available from `http://www.curbralan.com`.