In type systems there is more to making equality work consistently in practice than just hand waving and good intentions

UNEQUAL EQUIVALENCE

LAST TIME [1] THE CONTRACT FOR OBJECT EQUALITY was put under the microscope, in most detail for `Object.equals` in Java but also with a brief look at `Object.Equals` in C#. The idea that equality can also be assessed by relational comparison between two objects was also examined, looking again in most detail at the contract for `Comparable.compareTo` in Java and then briefly at some differences in `IComparable.CompareTo` in C#. Certain obvious similarities between the languages, in this respect, make their direct comparison easy.

However, one point that was mentioned briefly, but otherwise glossed over, deserves more attention. The idea of determining equality from a total ordering is essentially based on the idea that if two objects are neither greater than nor less than one another they can be considered equal, i.e. when `compareTo` in Java returns a value of zero rather than a positive or negative value. What may come as a surprise is that this does not necessarily — and is not required to — give the same measure of equality as the direct notion of equality comparison embodied in the `equals` method.

When are equivalent objects not equal?

The leeway on allowing equality according to `equals` to be inconsistent with equality determined by `compareTo` may seem strange at first, but there are cases when a strict ordering and a notion of exact equality do not necessarily reach the same conclusion. Of course, the wording of Sun's JDK documentation strongly encourages the two concepts to be consistent, but that is not a hard and fast contractual binding.

Moreover, you don't have to look very far for an everyday example. In Britain, an alphabetically sorted listing of names, such as a telephone directory, should order names beginning "Mc" and "Mac" together and as the latter. However, although they are equivalent with respect to ordering there is no mistaking "McDonald" as equal to "MacDonald".

A similar case can be made for case-insensitive ordering of strings: `"Mongoose"` should order after `"aardvark"` but before `"ZEBRA"`. However, `"mongoose"`, `"MONGOOSE"` and `"Mongoose"` should be considered equivalent with respect to ordering even though they are not strictly equal. A practical application of this concept can be seen in the syntax of identifiers in CORBA's Interface Definition Language. Because of its intended role as an interoperability standard, IDL cannot reasonably favour either case-sensitive naming, as found in the C family of languages, or case-free naming, as found in Pascal, Fortran and other languages. The compromise is to ensure that spelling must be unique — so `"Mongoose"` has the same spelling as `"mongoose"`, and in a sorted table would map to the same place — but case is preserved — so `"Mongoose"` is not considered equal to `"mongoose"`, so you cannot redeclare one as the other or use one as the other. This is a good compromise that works with both case-sensitive and case-free languages, and a middle path that should be considered by more language designers.

A further example can be found in the `java.math` package. In contrast to floating-point numbers, a `BigDecimal` holds a scaled, arbitrary precision integer, where the scale indicates the number of digits to the right of the decimal point. A strict interpretation of the notion of equality between two objects suggests that a representation of `2.5` (`25` with a scale of `1`) and one of `2.50` (`250` with a scale of `2`) are not truly equal, and therefore `equals` returns `false`. However, no matter what the representation, the values represented by `BigDecimal` are reasonably subject to a total and natural strict ordering. In that case, `2.5` is neither greater nor less than `2.50`, so they are considered equivalent and `compareTo` returns `0`.

A similar model exists in C++. Equality is expressed through the `==` operator and bound by the *EqualityComparable* requirements, which require conforming implementations of `==` to be reflexive, symmetric and transitive. Reasonably enough, inequality is normally considered to be defined with respect to equality, so `a != b` is equivalent to `!(a == b)`. This is not just a good logical relationship: it's good implementation advice as well. Rather than duplicating the concept of equality comparison in both `operator==` and `operator!=` functions, it exists in only a single place and therefore, should it need to be changed to correct a defect or to modify representation, the change is needed in only one place. This leads to a more stable design with fewer hidden dependencies.

The C++ standard also defines *LessThanComparable* requirements to govern the `<` operator. To satisfy the *LessThanComparable* contract an implementation of the operator must define an ordering on its arguments and it must be irreflexive, i.e. `!(a < a)` must be `true`. Comparison in the C++ standard library is based solely on this operator. However, one would also expect — in the name of consistency, convention and reason [2,3] — the other relational operators to

At a glance

- A type that supports a natural, total ordering need not necessarily consider equivalently ordered items as equal.
- Java, C# and C++ each have their own models for ordering that accommodate this.
- The special NaN ("not a number") value presents an interesting challenge when comparing values of floating-point types.

In C++ ordering is defined, by default, in terms of the < operator rather than via a separate operation granted by an inherited class

be defined for a given type, and defined according to logical relationships in terms of `operator<` and not `operator==`, e.g. `a <= b` is defined as `!(b < a)`. Therefore, again, only a single piece of code defines the concept of ordering rather than having it subtly duplicated over four functions.

The *LessThanComparable* requirements are also used to define an equivalence relation along the lines of two values being considered equivalent if neither is less than the other, i.e. `!(a < b) && !(b < a)`. And, as we have seen with Java and C#, this notion of equivalence with respect to ordering does not imply consistency with equivalence defined in terms of *EqualityComparable*. Such consistency may be recommended but, as we have seen, making it a rule may sometimes be too strong an imposition.

When is a number not a number?

In the world of floating-point numbers there is commonly a notion of a special value that represents something that is *not a number*, NaN for short. A NaN value has some interesting properties: it does not order against any other value, and does not even compare equal to itself. Alas, one problem with having "interesting properties" is that, left to its own devices, a NaN rather messes up the idea of a total ordering that can be used for sorting.

Because Java's type system is not fully object oriented there is a divorce between what can be done with primitive types, such as `double` and `float`, and its class-based type system, rooted in `Object`. Floating-point numbers do not implement the `Comparable` interface used for ordering objects and it is not possible to hold floating-point numbers directly within a collection. To do this the programmer works instead in terms of the wrapper types, `Float` and `Double`, which are proper object types that implement `Comparable` and wrap their corresponding primitive type. The implementation of `compareTo` ensures that a NaN will compare equal to another NaN and greater than any other value. As an aside, J2SE 5.0 offers a number of improvements that make working with wrappers less tedious for the programmer, but the underlying model remains unchanged.

C#, by contrast, has a more regular type system, one that incorporates primitive types as first class types in its object system. Wrapper types are not needed, so `double` and `float` are synonyms rather than metonyms of the `System.Double` and `System.Single` types. They implement the `IComparable` interface to define a total ordering for floating-point values, including NaN. However, unlike Java they choose the slightly more intuitive ordering of considering all proper values to be greater than a NaN. This policy is also consistent with the idea of treating any object as comparing greater than `null`[1] (which is, in effect, "not an object").

In C++ ordering is defined, by default, in terms of the < operator rather than via a separate operation granted by an inherited class. This effectively offers the same behaviour as the corresponding operator in C# and Java, which means that because of NaN it doesn't offer a total ordering on all floating-point values.

To understand what this means in practice, consider holding a set of `double` values. A `std::set` orders its elements using the defaulted `std::less` function object type, which corresponds to the < operator in this case, and ensures uniqueness of keys by applying the equivalence outlined earlier — when `!(a < b) && !(b < a)` evaluates to `true` then a and b are considered equivalent. If you insert a NaN as the first element in a set then you will never be able to insert anything else, because all values will appear equivalent to NaN. The presence of a NaN pollutes the data set and therefore the container. And, for the same reason, if you insert something else first you will then never be able to insert a NaN!

However, all is not lost. You can impose your own preferred ordering criterion through a function object type, i.e. a class whose instances look like functions because they can be invoked with the function-call operator. The following type imposes an ordering so that a NaN is considered lower than any proper value:

```
struct total_less
{
    template<typename numeric_type>
    bool operator()(numeric_type lhs, numeric_type rhs) const
    {
        return is_nan(lhs) ? !is_nan(rhs) : lhs < rhs;
    }
};
```

To use this with a `std::set` you replace the defaulted `std::less` comparator with `total_less`, i.e. declare the type as `std::set<double, total_less>` rather than as `std::set<double>`.

The `total_less` function object type is reasonably straightforward in that it is written in terms of a simple helper, `is_nan`, and the ordering option is made explicit. Because of what can only be considered an oversight (OK, a cock up) in the design of the numeric features of C++ standard library, you can use `std::numeric_limits` to determine whether or not a numeric type has a NaN value, and what it is, but you cannot actually test a value to see whether or not it is a NaN — remember that NaN does not compare equal to itself, so you can't use == to check! However, assuming that infinity compares reflexively, we can take advantage of that very property to define our own predicate:

```
template<typename numeric_type>
bool is_nan(numeric_type value)
{
    return !(value == value);
}
```

This is generic and works for any numeric type, so consequently `total_less` will also work for any numeric type. ■

References/bibliography

1. Kevlin Henney, "First Among Equals", *Application Development Advisor*, November 2004.
2. Kevlin Henney, "Form Follows Function", *Application Development Advisor*, March 2004.
3. Kevlin Henney, "Conventional and Reasonable", *Application Development Advisor*, May 2004.

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com