What does it mean for two objects to compare equal? Kevlin Henney pores over some contracts to find out

FIRST AMONG EQUALS

WHETHER FORMAL OR INFORMAL, A CONTRACT defines an (in principle) enforceable agreement between two or more parties with respect to a specific undertaking. The same is also true in code. The contract metaphor is an effective way of approaching API design and use: [1] "A contract is effectively an agreement on the requirements fulfilled by a component between the user of the component and its supplier".

One way of capturing a functional contract is through the use of pre- and postconditions that state what must be true before an operation is to be called, for it to be called correctly, and what must be true after an operation has returned, for it to be considered correct [2]. To consider contracts only with respect to pre- and postconditions, however, offers a slightly limiting and incomplete – albeit common – view of the contract metaphor, although it clearly offers a great deal of coverage, utility and insight [3].

Comparing two objects for some kind of equality offers fertile ground for exploring different conventions and contracts [4,5].

Beyond pre- and postconditions

The contract for the `equals` method in Java can be phrased most easily and clearly in terms of named constraints, each one stated as a simple truth:

reflexive:
`a.equals(a)`

symmetric:
`a.equals(b)` if and only if `b.equals(a)`

transitive:
`a.equals(b) && b.equals(c)` implies `a.equals(c)`

consistent:
`a.equals(b)` returns the same as long as `a` and `b` are unmodified

`null` *inequality:*
`!a.equals(null)`

Two objects can be considered equal if they compare neither greater nor less than one another

`hashCode` *equality:*
`a.equals(b)` implies `a.hashCode() == b.hashCode()`

This contract is binding on any override of the `Object.equals` method. If you try to state this in terms of preconditions – what must be true before a successful call to `equals` – and postconditions – what must be true after a successful call to `equals` – from a strictly object-centric viewpoint you will find a loss of clarity as well as a loss of part of the contract. Assuming that the argument to `equals` is named `other`:

postcondition where `other == null`: The result is `false`.

postcondition where `other == other`: The result is `true`.

postcondition otherwise: The result is the same as the result of `other.equals(this)`, and where `true` then `hashCode() == other.hashCode()`.

As you can see, there is no useful precondition and the postcondition is in part partitioned by the named constraints we had before, but not as clearly. More reading between the lines is needed to get the full sense of the contract. The consistency constraint is a difficult one to express at the best of times, but its temporal nature means that it cannot be properly expressed as the postcondition of an operation, which describes only what is true at the point of completion of the method call, and nothing beyond.

There is also a subtle loop here: if the result of calling `equals` is required to be the same as the result of making the same call but with the argument and receiving object switched round, what is the requirement on that second call? It would be the first call. Any literal implementation of this would find itself caught in infinite recursion.

Although the contract for overriding

Facts at a glance

- Contracts can be described in many ways, not just functional pre- and postconditions.
- Working out a contract that describes equality directly takes more care than you might first think.
- Working out a contract describing total ordering takes even more.
- Contracts need to strike a balance between strict necessity and unnecessary strictness.

`Object.equals` in Java cannot be as well stated in terms of pre- and postconditions, the truths it uses are assertible from a testing perspective. For a given `a`, `b` and `c` that are supposed to compare equal we can assert the following, using the Java 1.4 assertion mechanism:

```
assert a.equals(a)                  : "reflexive";
assert !a.equals(null)              : "null inequality";
assert a.hashCode() == b.hashCode() : "hashCode equality";
assert a.equals(b) && b.equals(a)   : "symmetric";
assert b.equals(c) && a.equals(c)   : "transitive";
assert a.equals(b)                  : "consistent";
```

Of course, the consistency constraint check cannot be a thorough one, but it simply reinforces that in the simple test suite shown yet another equality comparison between `a` and `b` will yield `true`.

It is worth noting that contrary to the way that it is often presented – including in Sun's own documentation – the relationship between `equals` and `hashCode` is more properly a part of the equality contract, which deals with relations between objects, than it is of the hashing contract, which deals with individual objects.

Interface is not implementation

The contract for C#'s `Object.Equals` is similar to the corresponding Java contract. It clarifies more clearly that exceptions should not be thrown as a result of equality comparison and makes some qualifications concerning floating-point number comparison.

However, the most obvious shortcoming is not so much in the content of the contract as in its context: it is considered under the heading "Notes to Implementers". Whilst it is certainly true that the contract binds and guides an implementation, that is only one half of the contractual relationship. A contract is not just the view from the supplier: it also describes what the client can rely on. A contract is therefore more than just an implementation guideline: it is part of the interface.

Contracts form the basis of substitutability: an implementation can be said to satisfy the contract described by an interface, so any implementation satisfying that contract can be substituted wherever that interface is expected. It is why class hierarchies should follow a notion of substitutability – a strong form of the "is a" or "is a kind of" view of inheritance – before any commonality of implementation through inheritance is considered. A subclass follows and specialises the contract of its parents. This notion of substitutability, known as the Liskov Substitution Principle [6], can be derived from the contract model.

As an aside, it can be considered ironic that Eiffel [2], the language that has done the most to promote and embody the concept of contracts, using pre- and postconditions, actually fails the Liskov criteria for its own class-based type system.

Good relations

There is another way to test equality between objects: relational comparison. Two objects can be considered equal if they compare neither greater nor less than one another.

In Java the `Comparable` interface provides the standard protocol for querying the relation between two objects

defined to have a natural ordering. There is only one method, `compareTo`, and this uses `strcmp` semantics, so called because of the standard string comparison function in C that uses the same result model: a negative value if the left-hand side of the comparison compares less than the right-hand side; zero if they are considered equal; a positive value if the left-hand side compares greater than the right-hand side.

Assuming that `sgn` returns the value `-1`, `0` or `+1` when its operand is less than, equal to or greater than zero, respectively, the `Comparable` contract can be stated as follows:

antisymmetric:
`sgn(a.compareTo(b)) == -sgn(b.compareTo(a))`
transitive:
`sgn(a.compareTo(b)) == s && sgn(b.compareTo(c)) == s` implies `sgn(a.compareTo(c)) == s`
consistent equivalence:
`a.compareTo(b) == 0` implies `sgn(a.compareTo(c)) == sgn(b.compareTo(c))` for all `c`
consistent:
`a.compareTo(b)` returns the same as long as `a` and `b` are unmodified
`null` *incomparability:*
`a.compareTo(null)` throws a `NullPointerException`
incompatible type incomparability:
`a.compareTo(b)` throws a `ClassCastException` if the types of `a` and `b` cannot be meaningfully compared
incompatible type symmetry:
`a.compareTo(b)` throws a `ClassCastException` if and only if `b.compareTo(a)` throws a `ClassCastException`

No mention is made of reflexivity, i.e. `a.compareTo(a) == 0`, and indeed there is no strict requirement that `a.compareTo(b) == 0` has the same result as `a.equals(b)`, although it is strongly recommended.

The contract for `IComparable.CompareTo` in C# is similar but has a couple of notable differences: there is a requirement to be reflexive and any object compares greater than `null` instead of throwing an exception.

In common with `strcmp` and other similarly specified operations, note that the contracts are not defined in terms of `-1`, `0` and `+1` return values, which sometimes programmers mistakenly assume to be the case. It is OK to implement ordering functions to return `-1`, `0` and `+1`, because these certainly satisfy the contractual requirement for returning less than, equal to and greater than zero, but it is not OK for a caller to rely on these specific values.

For example, imagine a `Date` class whose representation is scalar, counting the number of days since a given epoch [7]:

```
public final class Date implements Comparable
{
    ...
    public int compareTo(Object other)
    {
        return day - ((Date) other).day;
    }
    private int day;
}
```

This is about the simplest, contract-satisfying implementation you can imagine. A contract over-specified to require the result to `-1`, `0` or `+1` would not be a great benefit to the caller but it would be a burden to the implementer:

```
public final class Date implements Comparable
{
    ...
    public int compareTo(Object other)
    {
        return sgn(day - ((Date) other).day);
    }
    private static int sgn(int value)
    {
        return value < 0 ? -1 :
               value > 0 ? +1 : 0;
    }
    private int day;
}
```

Alternatively, the implementer could write out the logic longhand and inline within the `compareTo` method.

The same justification of the less strict contract also applies to field-based representations [7]. Instead of comparing field by field, treat the date as an ordered but discontinuous number range and subtract the right-hand side from the left-hand side. For example, the dates 6th April 2002 and 21st February 1998 can be treated as decimal numbers, 20020406 and 19980221, which are easily obtained by multiplying up and adding the respective year, month and day fields. To discover the relative ordering of 6th April 2002 and 21st February 1998 one simply has to subtract the second figure from the first, which yields a positive number and, therefore, the conclusion that 6th April 2002 naturally orders after 21st February 1998.

Most programmers attempting this task get caught up in a logic and control flow thicket that, as a by-product, happens to return precisely `-1`, `0` or `+1`. Neither the code expressing the logic nor the precision of the result is necessary. Part of the art of decision making is knowing which decisions need not be taken: letting arithmetic take the place of control flow sometimes offers such a route; having a contract that is no stricter than is necessary makes that path an easier one to follow. ■

References

1. Kevlin Henney, "Sorted", *Application Development Advisor*, July 2003.
2. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
3. Kevlin Henney, "No Memory for Contracts", *Application Development Advisor*, September 2004.
4. Kevlin Henney, "Objects of Value", *Application Development Advisor*, November 2003.
5. Kevlin Henney, "Conventional and Reasonable", *Application Development Advisor*, May 2004.
6. Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the Proceedings*, October 1987.
7. Kevlin Henney, "The Taxation of Representation", artima.com, July 2003, www.artima.com/weblogs/viewpost.jsp?thread=8791

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