Some objects are more equal than others. **Kevlin Henney** examines objects that represent values

Objects of value

WHAT IT MEANS TO BE AN INDIVIDUAL is a topic that has vexed philosophers and political theorists through the ages. In the world of systems analysis, however, it is more easily reducible, according to Michael Jackson [1]:

"An individual is something that can be named and reliably distinguished from other individuals. There are three kinds of individual.

- Events. An event is an individual happening, taking place at some particular point in time....
- Entities. An entity is an individual that persists over time and can change its properties and states from one point in time to another. Some entities may initiate events; some may cause spontaneous changes to their own states; some may be passive....
- Values. A value is an intangible individual that exists outside time and space, and is not subject to change...."

In addition to the three types of individual, Jackson identifies three relations of interest in modelling a problem domain: states, truths and roles. Any developer versed in object orientation will find some correspondence between these phenomena and object models.

There is a gentle but pervasive myth that classes should all follow the same coding guidelines. The concern for consistency is a laudable one, but it is misplaced. Many coding guidelines go to great length to spell out how the subtle mechanisms of a language should be used – how to write an assignment operator in C++, how to override the `equals` method in Java – without ever mentioning when or why they should (or should not) be used.

The truth is that, in contrast to the aspirations of many human individuals, objects do not live in a free and egalitarian society. They are brought forth into a class-based system, differentiated by role, to serve an explicit purpose. Their make up is dictated by requirements and each one has a place in the order of your program. Huxley's Brave New World perhaps offers a better model of how to organise most programs than Marx and Engels' Communist Manifesto.

Objects can be characterised with respect to identity, state and behaviour. However, the relative significance of each of these properties varies between objects, as the following stereotypical object categories illustrate:

- Entity: Entities express system information, typically of a persistent nature. Identity is important in distinguishing entity objects from one another.
- Service: Service-based objects represent system activities. Services are distinguished by their behaviour rather than their state content or identity.
- Value: For value-based objects interpreted content is the dominant characteristic, followed by behaviour in terms of this state. In contrast to entities, values are transient and do not have significant enduring identity.
- Task: Like service-based objects, task-based objects represent system activities. However, they have an element of identity and state, e.g. command objects and threads.

We can identify further profiles, but these four are common enough and different enough to appreciate the variation between object types.

Perhaps the most overlooked object category is values. Value types are tactically useful, but do not tend to appear in more strategically focused class diagrams. Nor should they: such attribute types would add nothing to a diagram except clutter. Programmer-defined value types model the fine-grained information in a system, enforcing basic rules and factoring out repetitive common code that would otherwise accumulate in attempting to marshal meaning out of more primitive value types, such as integers and strings.

Now, there is an apparent inconsistency that you may have either missed or dismissed: Jackson defined values as effectively immortal individuals, whereas my description of value objects suggested that identity was not important and that they were transient. Two more opposite definitions it would be hard to find. However, the difference is neither arbitrary nor contradictory. The difference is between the world of the solution – object-oriented programming – and the world of problem – the real world of individuals and other phenomena.

An example of value

To take a classic OO example, a book qualifies as an entity object. It would be reasonable to represent its author and title as strings, but its ISBN would be better off expressed through its own type rather than as a string. There are obvious constraints on its length (10 characters), the range of its characters (decimal digits except for the last character, which can also be 'X'), the relationship between its characters (the last character is a check digit calculated from the rest), and of course its presentation (with spaces, without spaces or with hyphens).

You could place all this logic in the book class, but it would make that class less cohesive by cluttering it up with housekeeping code that distracted from its core role. Where all this logic is required, it makes more sense to create an ISBN class and use that. At this point, you might get a bingo urge to stand up and shout "Reuse!", but resist that temptation. This is little to do with reuse: it is all about use. Reuse is a commonly misused word and a poorly understood concept that has wasted a lot of development on a lot of projects that should have been focusing on their more significant requirements [2, 3].

The identity of a real-world value is one and the same as its content: there is no distinction. In the implementation domain, value objects are used to represent the abstract Platonic ideal of values as we find them in maths or everyday conversation. The representation mechanism in a program is through objects, which have their own identity along with any state and behaviour. It happens that for value objects in most languages the object identity is not an important characteristic.

As an example of content-based versus identity-based usage, consider how you would compare two strings for equality. In Java or C, comparing two strings using the `==` operator is unlikely to be what you intended. You will be comparing object identity rather than object content. In Java the correct approach is to use the `equals` method, which is overridden to compare states between `String` objects, and in C you would use the `strcmp` standard library function. C++ and C# both support operator overloading, so you would use the somewhat more intuitive and pitfall-free `==` operator for the respective string types.

So, if you are defining your own value type, you need to consider equality comparison. Defining equality comparison makes little sense for most other kinds of objects, so you would not override `equals` in Java or overload `operator==` in C++ unless you were defining a value type. Each language has its own idioms for defining value types. For example, in Java when you override `equals` you should also override `hashCode`, and in C++ you would also overload `operator!=` when overloading `operator==`.

Some values have a natural total ordering, e.g. dates, money and postcodes. Meaningful ordering is either by magnitude – £10 is more than £5 – or lexical – BS6 comes before N6. In Java you would implement the `Comparable` interface and supply the `compareTo` method; in C++, C# and many other languages you can work with the more readable `<`, `<=`, `>` and `>=` operators.

Although all value types can be compared for equality, not all have an unambiguous or meaningful total ordering, e.g. co-ordinates, complex numbers and seasons. Whilst it is possible to make up some kind of ordering for otherwise unordered value types, so that they may be used in sorted collections, such fictional orderings are external to the value type rather than intrinsic to it. These orderings should normally be expressed through separate functional objects rather than directly as features of the value, e.g. a `Comparator` object in Java or a binary predicate function object in C++. Embodying fictional orderings as a direct feature can be subtle and confusing because they are not natural to the type and there may be more than one possible ordering. For example, although summer follows spring, that does not necessarily mean that it compares greater than spring: autumn follows summer, winter follows autumn and spring follows winter, which implies, transitively, that summer can also be considered less than spring.

Because identity is not important to the users of a value object, it matters little whether a value object is shared by reference or copied. The overall effect is the same, although the choice of whether indirection or copying is used depends mostly on the programming language.

For languages that support some consistent form of automatic copying, passing objects around by copy is the preferred mechanism. In C# `struct` types fulfil this role and in C++ passing by copy. For languages that support a transparent mechanism for working with objects through a level of indirection, immutability is needed to ensure value object transparency. In C++ passing a value object around by `const` reference offers immutability through a level of indirection that is mostly transparent. This is in contrast to using pointers for that level of indirection, which require a change in usage notation unsuitable for values but appropriate for other object types. In Java, all objects are manipulated consistently via a level of indirection and there is no concept in the core of the language of passing objects by copy. Immutability is effected as a property of the class – only query methods are supported – and not of the usage – such as `const` qualification in C++. C# backs two horses: `struct` types are normally used for values, but some types, such as strings, use the immutable `class` approach.

And what of their parents? In common across most languages, values do not generally live in class hierarchies. They may inherit from a universal `Object` class, and they may implement interface capabilities, but they do not otherwise get involved in the business of family. In C++, the class introduces no virtual functions and is not intended for use as a base class [4]. In Java, value types should be declared `final` [5]. In C#, a `struct` cannot be further derived from and a value `class` can be defined as `sealed`.

To summarise, then, value objects are fully paid-up objects that model values as we perceive them in the world – both real and imagined – around us. The Platonic ideal of equating identity and content is achieved for a few primitive types in a few languages, such as Smalltalk, but for most value objects, their identity plays little or no role in how they are used. The expression of how a value object should be used is strongly dependent on the target language. Although there are common themes to the idioms across languages, the detail is often quite different. ■

References/bibliography

1. Michael Jackson, *Problem Frames*, Addison-Wesley, 2001.
2. Kevlin Henney, "The Imperial Clothing Crisis", *Overload*, February 2002, available from http://www.curbralan.com.
3. Kevlin Henney, "Inside requirements", *Application Development Advisor*, May 2003.
4. Kevlin Henney, "Highly strung", *Application Development Advisor*, September 2002.
5. Kevlin Henney, "Value-based Programming in Java", *JAOO*, September 2002, available from http://www.curbralan.com.

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