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 attributes 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.
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++, passing objects around by copy is the preferred mechanism, but for most value objects, their identity plays little role in how they are used. The expression of how a value object is used is mostly straightforward.

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 theequals method, which is overridden to compare states betweenString objects, and in C you would use thestrcmp 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 comparisons makes little sense for most other kinds of objects, so you would not overrideequals in Java or overloadoperator== 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 overrideequals you should also overridehashCode, and in C++ you would also overloadoperator! when overloadingoperator==.

Some values have a natural total ordering, e.g. dates, money and postcodes. Meaningful ordering is either by magnitude or lexical, -5 is more than 5, or the most familiar is by sorting lexicographic. In Java you would implement theComparable interface and supply thecompareTo 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. For example, a Comparator object in Java or a binary predicate function object in C++.

Reference/bibliography

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