5.7 Objects and values

Description

From a modeling point-of-view, it is quite important to be able to distinguish between values and objects, and since programming is modeling, it is important for programming in general.

Values and objects are different entities. An object exists in time and space, has identity, and may have state that changes over time, and often represents a physical entity. A value on the other hand is a timeless concept that does not change over time. The number seven is an example of a value, and there is a clear distinction between the concept of the number seven and the representations of this number by means of symbols like: 7, VII, seven, and syv.

The characterization of value is to a large extent borrowed from [Hoare, MacLennan and Dahl], but the authors are solely responsible for the formulations used here.

There is of course a relationship between objects and values – objects may have a state represented by data-items that hold datums being values and/or references to other objects. This state may be changed by means of methods. In addition, properties of objects may be computed by means of functions returning datums.

An ***object*** exists in time and space, has identity, and may have state that changes over time, and often represents a physical phenomena from the application domain. Objects can be created, destroyed and shared. Objects have substance. E.g., the substance of a living physical phenomena consists of its cells, while the substance of a program execution object consists of the memory cells allocated for it.

Objects are typically used to represent phenomena that may change state during its lifetime. Vehicles in a vehicle registration system, medical records in an electronic health record and books in a library system are examples of phenomena that may be represented by objects in a program execution.

Objects may be referred to by means of references, and an object may be referred to by several other objects. This implies that a data-item holding a reference may have aliases in the form of other reference variables that refer to the same object.

Most OO languages have data-items that may hold variable or constant references to objects. A reference variable may be assigned a new reference using reference assignment just as references may be compared for equality. This is where identity comes in, two references are equal with respect to reference equality if and only if they refer to the same object.

A ***value*** on the other hand is a timeless concept that does not change over time. Mathematical entities such as integer, float, and complex numbers, Booleans, and points are examples of values.

A value is an abstraction (a concept) – the number 4 is an abstraction that subsumes all collections of four elements. An abstraction/concept is determined by its extension: the phenomena subsumed by the abstraction, its intension: the properties characterizing phenomena in the extension, and the name of the abstraction – see +++BetaBook chp. 18 for a further discussion of abstractions / concepts.

A value is timeless (or atemporal) and it does not change over time. Put another way, a value is immutable as is the case for abstractions.

A value cannot be counted in the sense that you cannot take a copy of a value just as you cannot copy an abstraction. Abstractions including values are uniquely determined by its intension, extension and name.

Here is a quote from MacLennan:

> Although values can be operated on, in the sense of relating values to other values, they cannot be altered. That is, 2 + 1 = 3 states a relation among values; it does not alter them. When in a programming language we assign x the value 2, x := 2, and later add one to x , x := x + 1, haven’t we changed a number, which is a

> value? No, we haven't; the number 2 has remained the same. What we have changed is the number that the name 'x' denotes. We can give names to values, and we can change the names that we give to values, but this doesn't change the values.
>
> *MacLennan: …*

A value has no substance. It cannot be instantiated or destroyed as is the case for objects – the number 4 exist independent any expression like 4, 3 + 1, etc. The number 4 exist independently of any program or program execution. As a consequence, a value does not have an identity in the same way as an object.

There is a clear distinction between the concept of a given value and the *representation* of this value. The concept of the number seven may be represented by means of symbols like: 7, 111, VII, *seven,* and *syv* – see (Hoare 1972).

In program executions values must be represented by objects or other data-structures. Most programming languages have variable data-items that may represent a given value at a given point in time. And they may have constant (immutable) data-items that represent the same value during the life-time of a given program execution.

The difference between the concept of a value and its representation and the notion of variables in programming languages may perhaps have contributed to blur the understanding of values. A value may have many representations just as several variables may represent the same value. Neither of these imply that there are several instances of a given value.

Another source of confusion about values in object-oriented languages may be Smalltalk and Self. Here a value is considered an object that may be copied and destroyed. A value may also be referred to. Thus there is no clear distinction between objects and values.

Many object-oriented languages support immutable objects – i.e. objects that cannot change state. Such objects are important for parallel programming since concurrent processes may share references to the same immutable object without risking race conditions. Immutable objects may of course be used to represent values, but there is more to supporting values than immutable objects.

Values play a central role in programming since all programming languages have mechanisms for representing values. Imperative languages and object-oriented languages have variables that may hold values and be assigned new values.

Functional languages are applicative in the sense that the result of a computation is formed by expressions and without value assignment and other imperative mechanisms. This form of programming has many advantages since such a program has no side-effects, and it is easier to use mathematical reasoning to prove the correctness of such a program.

In order to prevent side-effects, it is important to avoid aliases for a given variable representing values. In traditional object-oriented programming with references to objects, it is hard to avoid aliases. This makes it hard to use mathematical techniques to reason about programs.

As mentioned, we think that one needs objects as well as values for object-oriented modeling and programming and there is no contradiction in supporting objects and values in the same language. We may thus not agree with advocators of pure object-oriented languages.

Also, for some programmers the distinction between value and object appears quite subtle and of no real practical value. This is however, one of the reasons to consider programming as modeling and teach programmers that a program should be a description of a selected part of the application domain. And this includes that programmers must be aware of when representing objects or when representing values.

To repeat, the creation of a model of a system in a given domain consist of identifying relevant phenomena and relevant properties of these phenomena and identifying relevant concepts subsuming these phenomena. Phenomena in a given domain may be physical entities that may change state over time. Such phenomena should be represented by objects – vehicles, medical records and books as mentioned above are examples of this.

There are, however, other kinds of phenomena than those having substance. One important class of such phenomena are quantities characterizing physical entities. The color, weight, speed, etc. are examples of quantities characterizing a vehicle. Quantities may be represented by values and often including a unit, the color of a vehicle is a value, the speed and weight of a car are all values. And such quantities are fundamentally different from phenomena being physical entities.

Activities may change the measurable properties of a physical entity – the color, weight and speed may vary over time. Activities are another class of phenomena that is important for modelling and may be represented by the activities generated as part of the program execution. A method of an object describing an activity of the state of the object may in many cases be expressed as a function from the input parameters of the method and the state of the object leading to a change of state and/or measurement of part of the state of the object.

To sum up: objects and values are fundamentally different entities and modelling and programming languages should support both equally well.