Here `read_character` is a command, consuming a character from the input; `last_character` is a query, returning the last character read (both features are from the basic I/O library). A contiguous sequence of calls to `last_character` would be guaranteed to return the same result repeatedly. For both theoretical and practical reasons detailed elsewhere (Meyer 1997), the command-query separation principle is a methodological rule, not a language feature, but all serious software developed in Eiffel observes it scrupulously, to the benefit of referential transparency. Although other schools of object-oriented programming do not apply it (continuing instead the C style of calling functions rather than procedures to achieve changes), it is in our view a key element of the object-oriented approach. It seems like a viable way to obtain the referential transparency goal of functional programming—since expressions, which only involve queries, will not change the state, and hence can be understood as in traditional mathematics or a functional language—while acknowledging, through the notion of command, the fundamental role of the concept of state in modeling systems and computations.

An Object-Oriented View

We now consider how to devise an object-oriented architecture for the designs discussed in the presentation and article.

Combinators Are Good, Types Are Better

So far we have dealt with operations and combinators. Operations will remain; the key step is to discard combinators and replace them with types (or classes—the distinction only arises with genericity as discussed below). This brings a considerable elevation of the level of abstraction:

- A combinator describes a specific way of building a new mechanism from existing ones. The combination is defined in a rigid way: a `take` combination (as in `take` 3 apples) associates one quantity element and one food element. As noted earlier, this is the mathematical equivalent of defining a structure by its implementation.
- A class defines a type of objects by listing the applicable features (operations). It provides abstraction in the sense of abstract data types: the rest of the world knows the corresponding objects solely through the applicable operations, not from how they were constructed. We may capture these principles of data abstraction and object-oriented design by noting that the approach means knowing objects not from what they *are* but through what they *have* (their public features and the associated contracts). This also opens the way to taxonomies of types, or *inheritance*, to keep the complexity of the model under control and take advantage of commonalities.

By moving from the first approach to the second one, we do not lose anything, since classes trivially include combinators as a special case. It suffices to provide features giving the constituents, and an associated creation procedure (constructor) to build the corresponding objects. In the `take` example: