5.4 Composite values

Description

In the previous chapters, we have introduced objects and classes for representing physical entities like accounts and customers ; we have introduced references to objects, primitive values like float, integer, char, Boolean, and String for representing properties of physical entities, like owner, balance, and interestRate – cf. section .

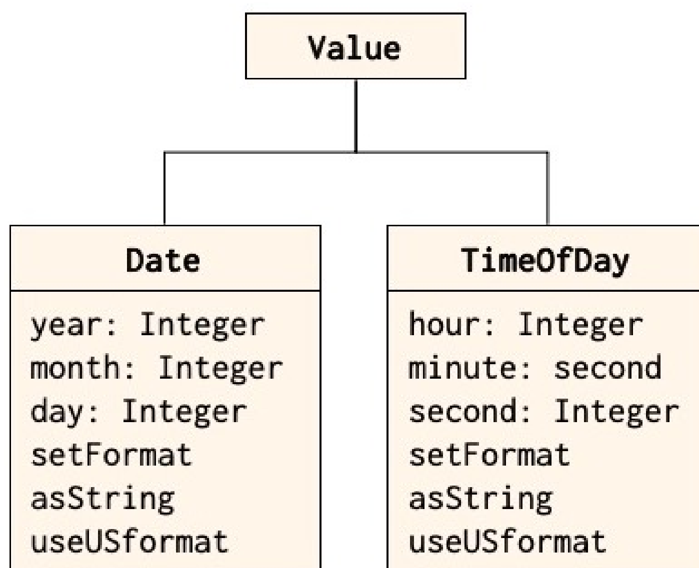
In some cases primitive value types are not adequate for the types of data items. In a domain where e.g. date (in terms of year, month and day) and time of day (in terms of hour, minute, second) are required to represent values of properties, we need composite values in order to represent these properties.

It is important here to notice that objects and values are different entities – objects have state that may vary over time whereas values are timeless abstractions – we discuss this further in section .

We do, however, need objects to *implement* values, but such objects differ from objects *representing* physical entities. Objects representing values are called *value objects*, and a class defining value objets is called a *value type* or just a *type*.

In the United States the notation for date and time differs from most of the rest of the world. In the US a date is written as month-day-year whereas the rest of the world use day-month-year. A time of day in the USA is written using a 12-hour clock and the suffixes a.m. and p.m. In addition, many other formats are used like year-mont-day. In the examples below, we just represent dates and time of days that may be in either US-notation og non-US notation.

In qBeta a value type is defined by means of a class with the `Value` as superclass. `Value` is actually the name of a class and we explain how this works in section . For now, just think of `Value` as a keyword like **`class`**.



Date and TimeOfDay as special Value types

```
class Date(year, month, day: var Integer): Value
   setFormat(asUS: var Boolean):
      useUSformat := US
   asString -> S: var String:
      ...
   useUSformat: var Boolean

class TimeOfDay(hour, minute, second: var Integer): Value
```


© Copyright by Ole Lehrmann Madsen and Birger Møller-Pedersen 2023.

```
   setFormat(asUS: var Boolean):
      useUSformat := US
   asString -> S: var String:
      ...
   useUSformat: var Boolean
```

`Date` has data items `year`, `month`, `day`. November 11, 1949 will therefore be represented by a `Date` value with `year = 1949`, `month = 11` and `day = 11`, i.e `Date(1949, 11, 11)`.

In addition, class `Date` has:

- a Boolean `useUSFormat`, representing whether or not to use US notation when displaying the `Date` – `true`, US-notation is used – and since a `Boolean` variable by default is `false`, non-US notation is the default;
- a method `setFormat` that may be used to set the format;
- a method `asString` that returns a `String` representing the `Date` in either US-notation or non-US-notation.

Class `TimeOfDay` has a similar structure and `TimeOfDay(1,2,3)` represents the time of day being 1 hour, 2 minutes and 3 seconds.

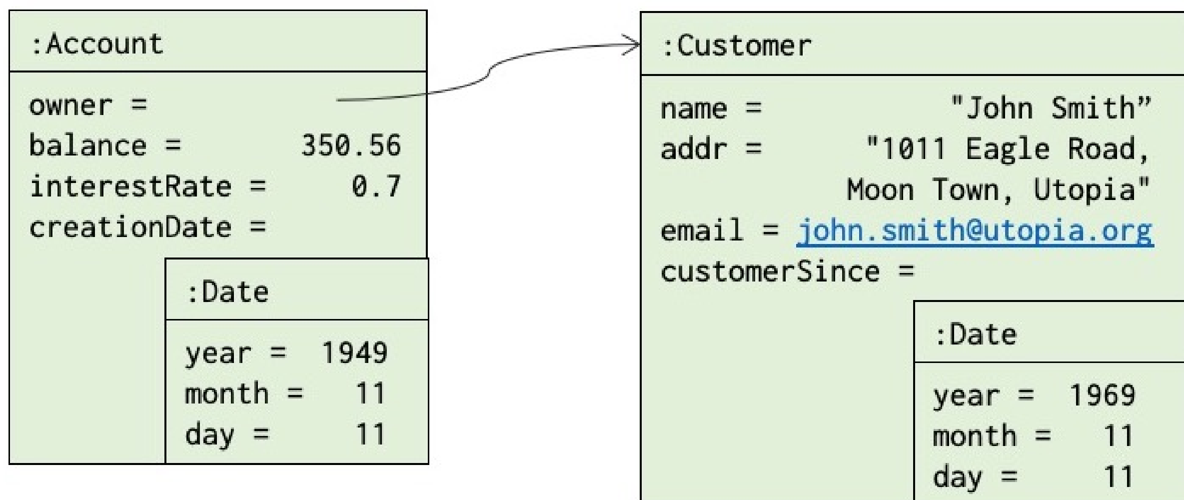
The following example illustrates the difference between references (to objects) and values. We have added a `Date` property `customerSince` to `Customer`:

```
class Customer(name: var String):
   addr: var String
   email: var String
   customerSince: var Date
```

In a similar way, we have also added a data-item, `creationDate` of type `Date` to `Account`:

```
class Account(owner: ref Customer):
   balance: var float
   interestRate: var float
   creationDate: var Date
   -"-
```

The diagram below illustrates the difference between a reference data-item (`owner`) and value data-items (`creationDate`, `customerSince`).



creationDateObject

As can be seen, `owner` refers to a separate `Customer` object.

© Copyright by Ole Lehrmann Madsen and Birger Møller-Pedersen 2023.

The primitive value data-items `balance`, and `interestRate` are shown embedded in the `Account` object. The same is the case for the `String` values in `Customer`.

The composite value object `creationDate` is also embedded in the `Account` object since it is a sole property of the `Account` object. It is not possible for other objects to have a reference to such a value object. This is an important distinction between objects and values.

Another example using composite values

In the following we have another example of using composite values. In the bank domain, each account keeps track of transactions with information about date and time of the transaction, and what kind of transaction (like `deposit` and `withdraw`). This information is important for the bank, and for the customer in order to check what kind of transaction was done when. A transaction is represented by instances of class `Transaction`:

```
class Transaction(
   what: var String,
   whichDate: var Date,
   whichTime: var TimeOfDay,
   whichAmount: var float):
   ...
```

The three dots `'...'` indicates possible additional attributes of class `Transaction`, but in the examples in this book, we only make use of the attributes defined as parameters.

We add a `Set`-object, `transactions`, to keep track of the transactions on an `Account`. The methods `addInterest`, `deposit`, and `withdraw` add transaction objects to this set:

```
class Account(owner: ref Customer):
   balance: var float
   interestRate: var float
   transactions: obj Set(Transaction)
   addInterest:
      interest: var (balance * interestRate) / 100
      balance := balance + interest
      transactions.insert(
         Transaction("interest", clock.today, clock.now, interest))
   deposit(amount: var float):
      balance := balance + amount
      transactions.insert(
         Transaction("deposit", clock.today, clock.now, amount))
   withdraw(amount: var float) -> newB: var
 float:
      balance:= balance - amount
      transactions.insert(
         Transaction("withdraw", clock.today, clock.now, amount))
      newB := balance
```


© Copyright by Ole Lehrmann Madsen and Birger Møller-Pedersen 2023.