2.8X Object-sequence diagrams

Description

Until now, we have been using diagrams to illustrate objects and classes. +++ no classes so far!

The diagrams for objects illustrate snapshots of the state of an object during *program execution* and thus the *dynamic structure* of the program – these diagrams called *object-diagrams*.

The diagrams for illustrating classes is a way to illustrate the *program text* and thus the *static structure* of a program – these diagrams are called *class-diagrams*.

In this section, we introduce diagrams for showing snapshots that further illustrate the dynamic structure including the state of objects, method invocations and method activations during program execution. These diagrams include object diagrams and are called *Object-Sequence Diagrams* (*OSD*). In addition to object diagrams, they illustrate method invocations.

The *state* of a program execution at a given point in time is:

- The *set of objects* (including method objects) in the program execution.
- The *datums* currently hold by objects in the program execution
- As mentioned a *datum* may be a *reference* to an object or a *value*.
- The point of execution of each object.

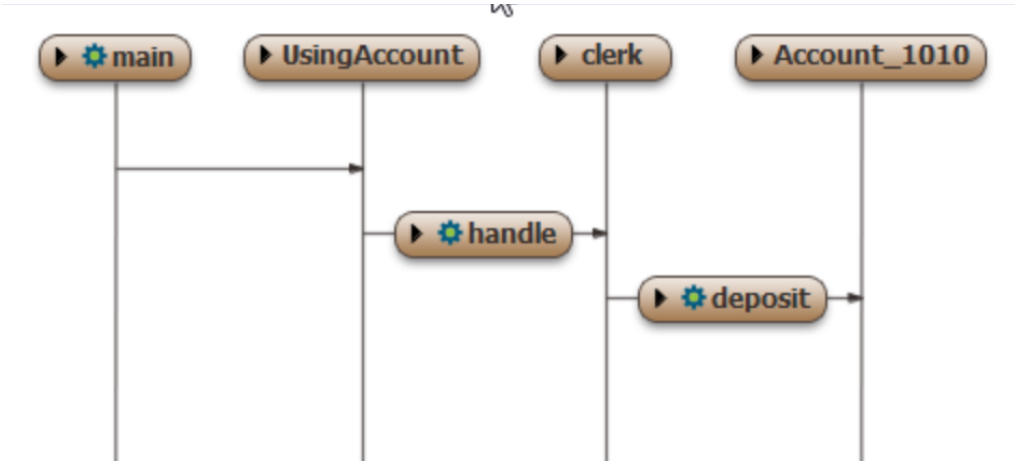
> In general, a *snapshot* of a program execution is part of the state of the program execution at particular points in time. The term snapshot is used as an analogy to that a in photography.

We use the object `usingAccount` from the previous section. The diagram below shows a snapshot at the following point of the execution `usingAccount`:

- The system object `main` has invoked the `usingAccount` object.
- `usingAccount` has invoked the `handle` method of `clerk`.
- The `clerk` has invoked `deposit` on `Account_1010`.

As said, `main` is an object that is generated by the computer-system with the purpose of initiating execution of `usingAccount`.

Skal vi gentage usingAccounts her?



The diagrams four columns representing the four objects, `main`, `usingAccount`, `clerk` and `account_1010`. The vertical

lines from these objects are called *lifelines* and represents time as seen by the object.

A method invocation is shown as an arrow from the lifeline of the caller object to the lifeline of the receiver. The arrows are labeled by the name of the method being invoked.

The arrow from main to `UsingAccount` has no method label. Such an arrow represents the situation where the caller (in this case `main`) is generating the object (in this case `UsingAccount`).

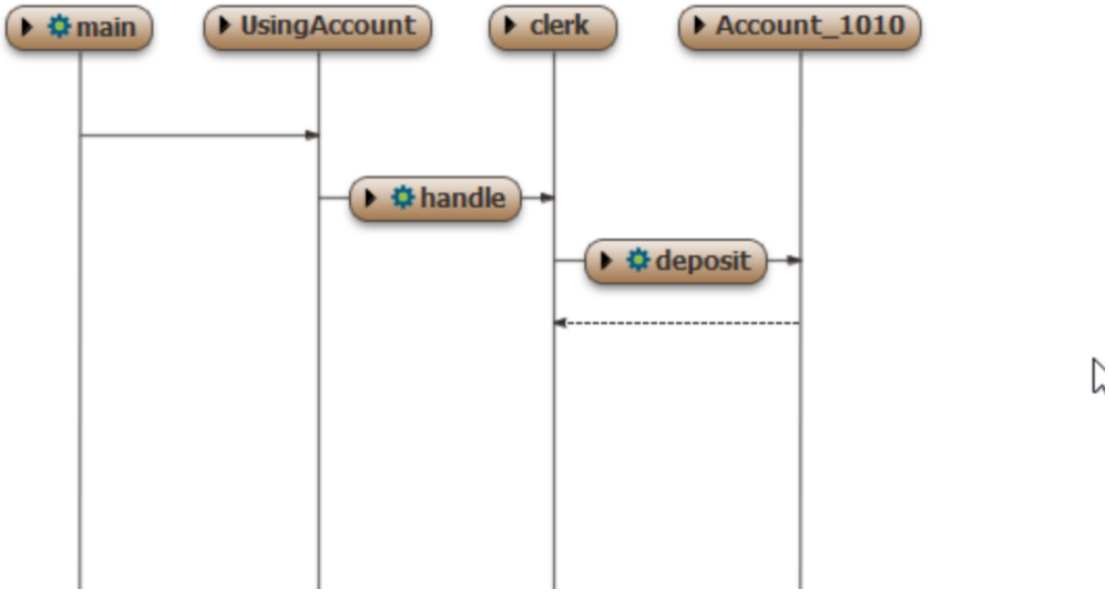
The arrow from `main` to `usingAccount` is above the arrow from `clerk.handle`, which is above the arrow for `account.deposit(100)`, illustrating the order in time of these method invocations.

The horizontal lines showing the active method invocations are often referred to as the i*nvocation stack*, *execution stack* or just *stack*. In the figure above, `deposit` is on top of the stack and `main` is the bottom of the stack.

A stack is a data-structure that may contain a collection of objects where you may insert and remove objects. Insertion is called *push* and removal is called *pop*. Elements are pushed on top of the stack which means that when you pop an element, you get the one that has been pushes latests. We further explain stack sin section X

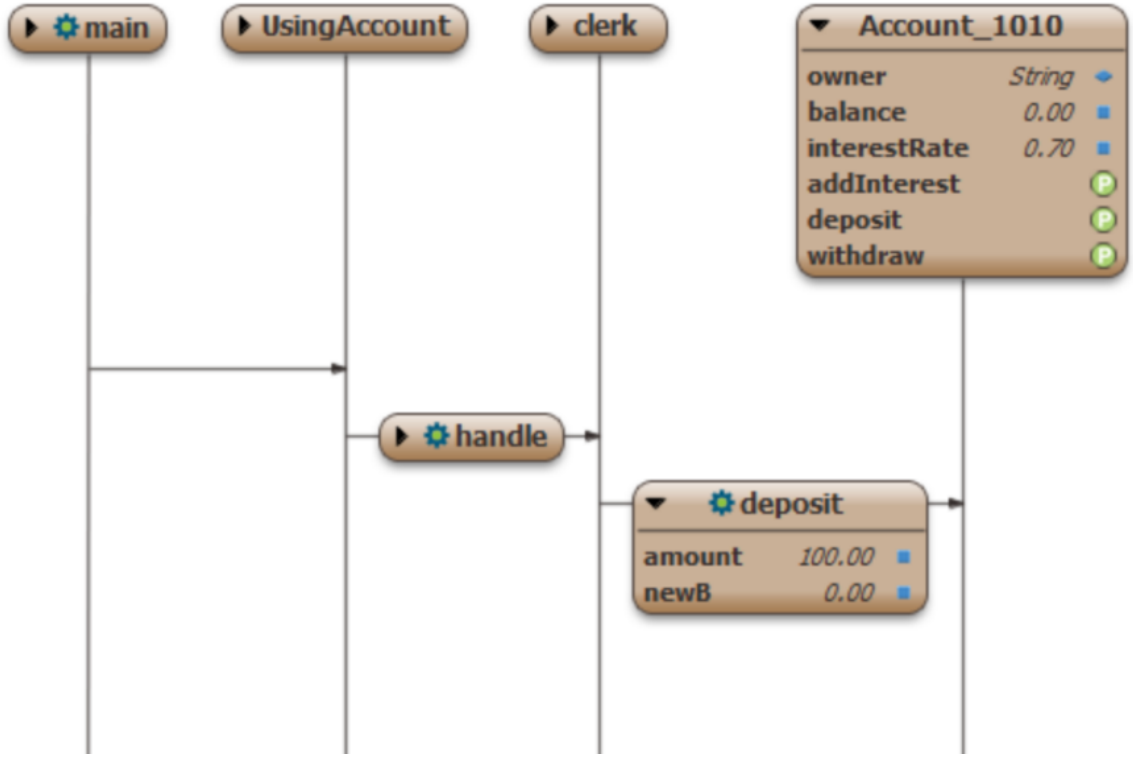
Det viste sig at forklaringen af stack er mere omfattende end jeg lige troede. Vi skal forklare push/pop, etc. Og lige nu er ikke forklaret hvorfor execution stack er en stack. Så måske vi skal forklare stack senere? Og skal vi indføre stack i kapitlet om collecions?

The next diagram shows a snapshots where invocation of `deposit` returns to the caller (`clerk`). The dotted line indicates a return of a method.



As said above, the lifelines and arrows showing method invocations represents how actions are ordered in time. The ordering of the objects in the columns does not matter, but usually a diagram may be more readable if the ordering in time flows from left to right to the extend that this is possible.

It is possible to inspect the state of the objects and method activations as illustrated by the next diagram:

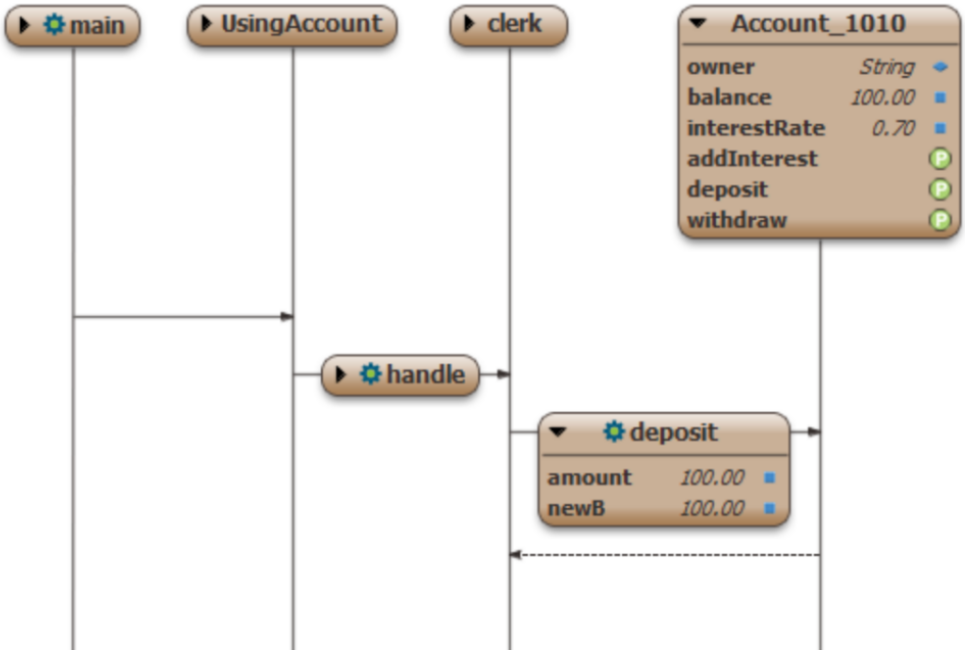


In this snapshot, the `Account_1010` object and the `deposit` invocation have been expanded, and we may thus see the datums of these object at this point of execution.

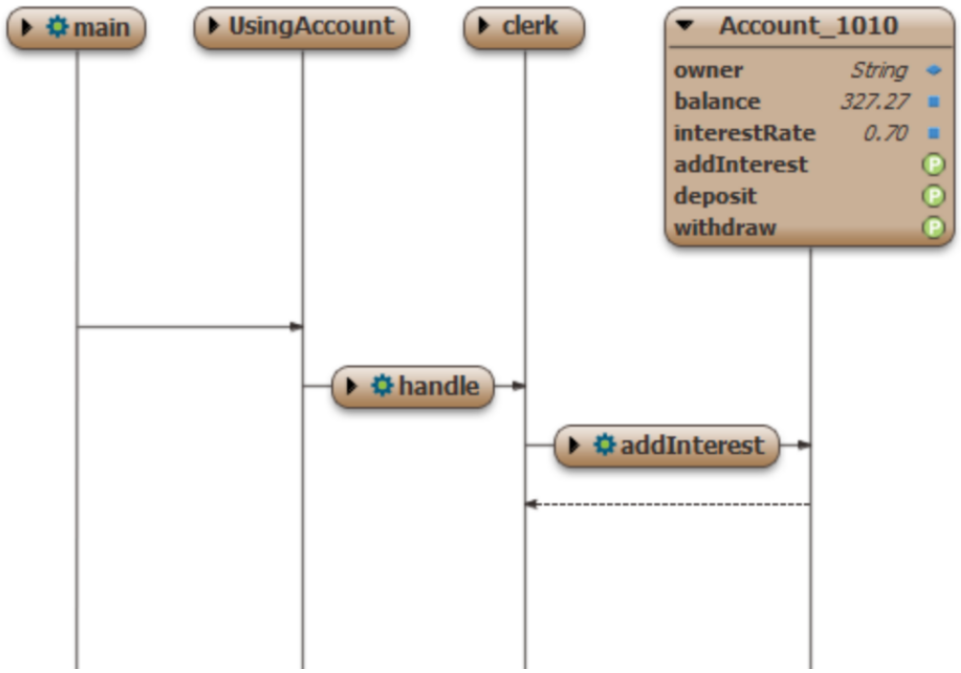
The `amount` parameter of `deposit` has the value 100.00; the return value `newB` has the value 0.00.

For `Account_1010`, the datum of `owner` is a `String`; the values of `balance` is 0.00 and the `interestRate` is 0.70.

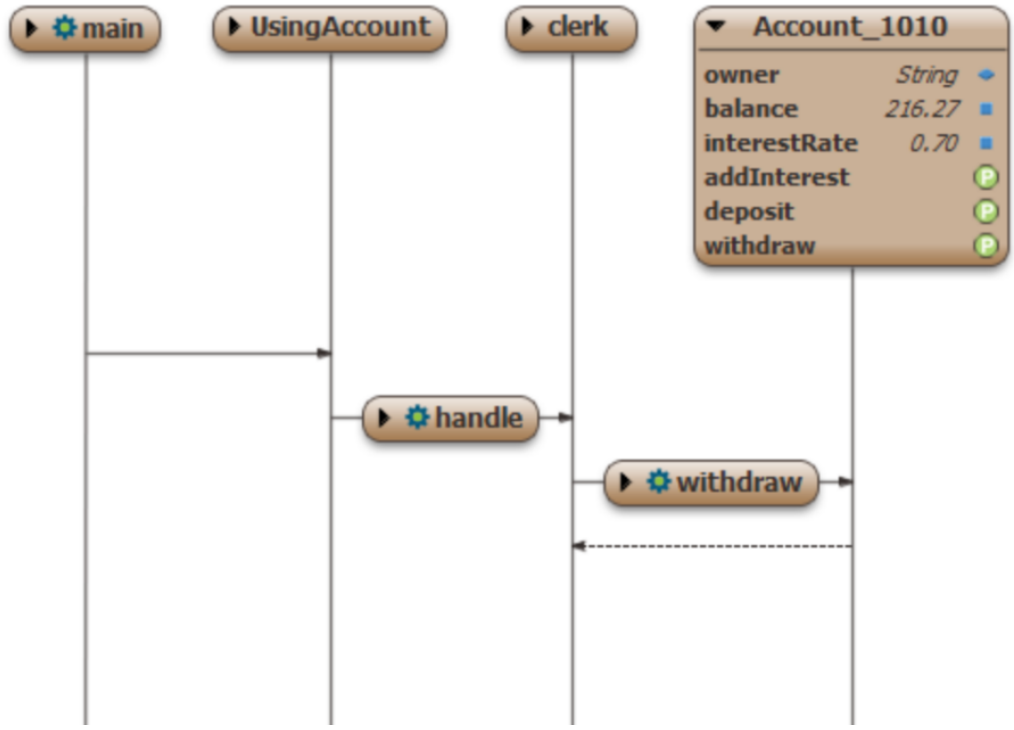
The next snapshot shows the state at the return of `deposit`. Here we can see that the `balance` of `Account_1010` has been updated to 100.00 and that `newB` of `deposit` also has the value 100.00.



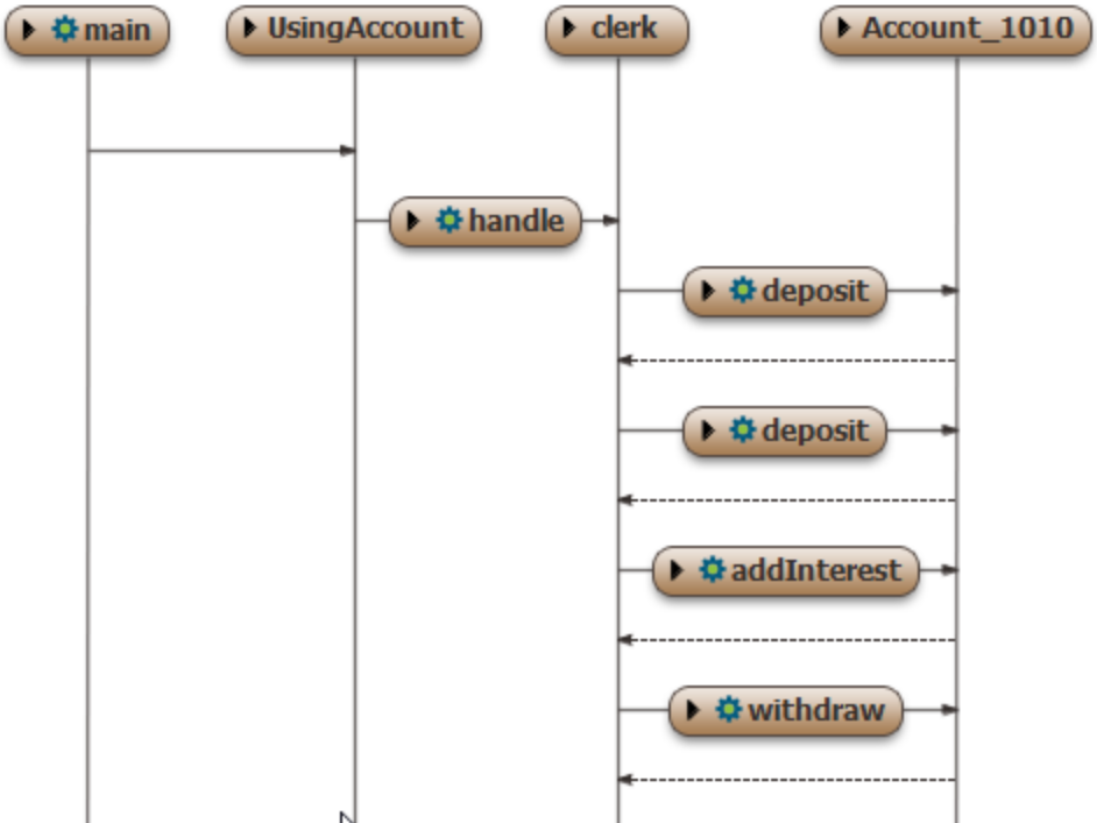
The next snapshot shows the state at the return of `addInterest` and the `balance` is now 327.27.



The final snapshot shows the state at the return of the `withdraw` made by the `clerk` before invoking `console.print`. As can be seen, the `balance` is 216.27.



The above snapshots just show the active method invocations. Sometimes one may want to show (a subset of) previous method invocations. The next diagram shows the history of all method invocations performed by the `clerk` on `account_1010`.



In later examples, we should also show that you can expand references to other objects.

The OSD's being used in this book are generated by a tool called qenv that may execute qBeta programs.