2.9 The program execution

The general theme in this book is that programming is modeling and that the model is the dynamic system of objects generated in a computer when executing a program describing the model – referred to as the *program execution*.

It is thus important to understand what goes on in a program execution. In the previous sections, we have shown program text and corresponding diagrams described by a given program, but these are *static* descriptions of the program execution.

We have also shown snapshots of objects as they may appear in a program execution. Such diagrams illustrate snapshots of the state of an object during *program execution*, and they thus illustrate part of the dynamic structure of a program execution.

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
    - A *datum* may be a *reference* to an object or a *value*. For a further description of datum, value and reference see chapter

- The *stage of execution* – the state of the activities currently taking place in the program execution.
    - Below we explain what exactly is meant by stage of execution.

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

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

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

- The computer-system object `VM` has invoked the `mySecondProgram` object. As mentioned before, an invocation of an object implies generation of the object and execution of its statements.
- `mySecondProgram` has invoked the `handle` method of `clerk`.

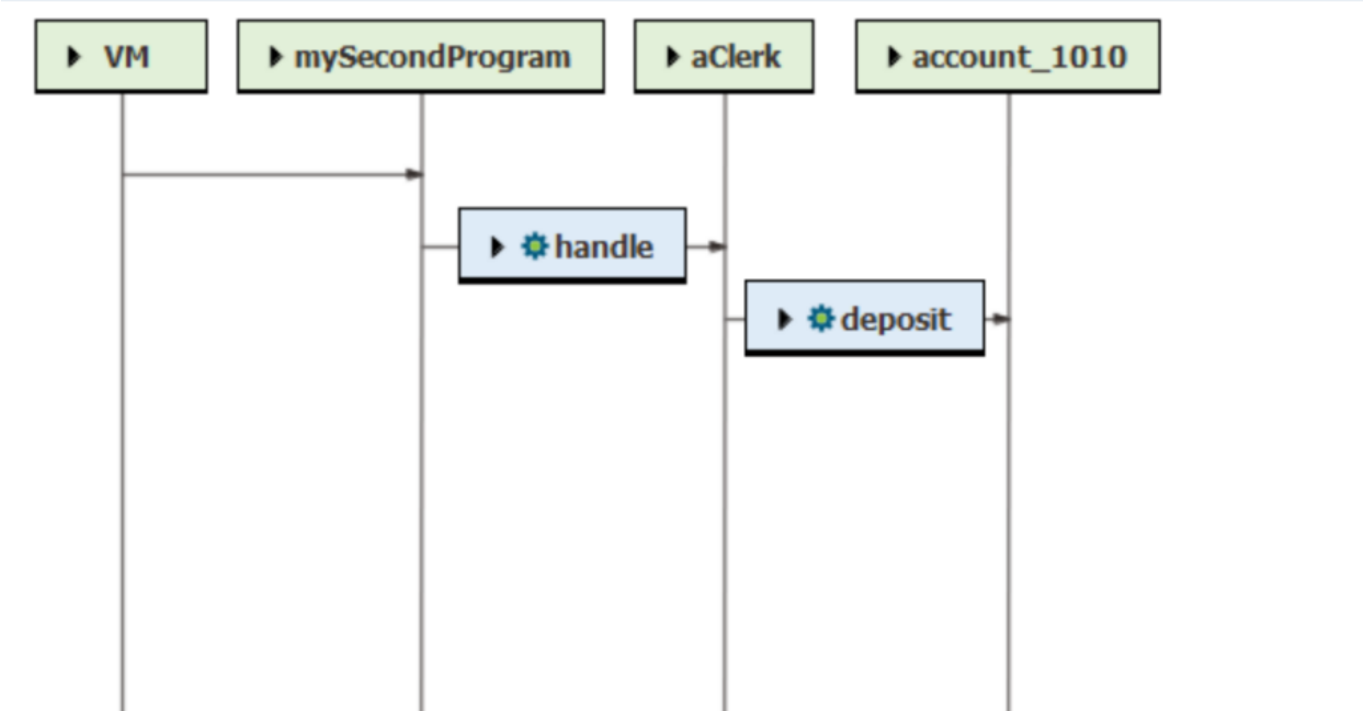
- The `clerk` has invoked `deposit` on `account_1010`.

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

As mentioned in , the abbreviation VM stands for Virtual Machine, which is the component in the computer system that takes care of executing qBeta programs.

As mentioned, the symbols `-"-` stands for code, which has been shown in previous examples – see chapter .

```
mySecondProgram: obj
   aClerk: obj
      handle:
         newBalance :=
                  account_1010.deposit(100)
      -"-
   account_1010: obj
      -"-
      deposit(amount: var float):
         balance := balance + amount
      -"-
   aClerk.handle
```



The four columns of the diagram represent the four objects, `VM`, `mySecondProgram`, `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 invoker object to the lifeline of the

context of the method. The arrows are labeled by the name of the method being invoked.

The arrow from `VM` to `mySecondProgram` has no method label. Such an arrow represents the following situation.

- The invoker (in this case `VM`) has generated the `mySecondProgram` object;
- the generation of the `mySecondProgram` Object implies generation of the `aClerk` object and the `account_1010` object;
- Execution the `mySecondProgram` object, i.e. execution of `aClerk.handle`.

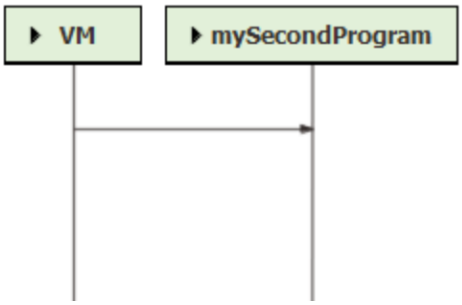
The arrow from `VM` to `mySecondProgram` is above the arrow from `clerk.handle`, which is above the arrow for `account.deposit(100)`, illustrating the order in time of these method invocations.

Generation of objects

In the next snapshots, we show the generation of the objects during program execution.

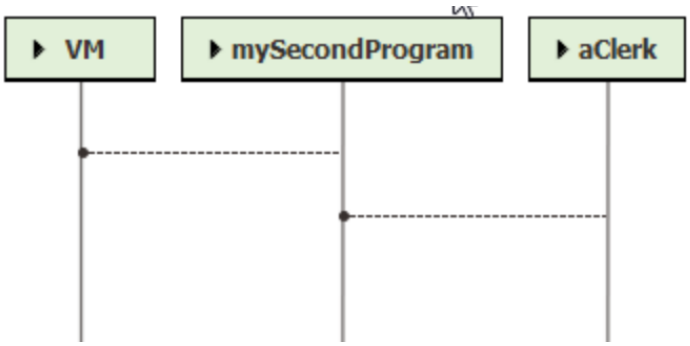
The snapshot below shows the situation after `VM` has generated the object `mySecondProgram`. The local objects `aClerk` and `account_1010` are generated as part of the generation of `mySecondProgram`. The red arrow (▀) before `aClerk`, illustrates that the next object to be generated is `aClerk`:

```
mySecondProgram: obj▀▀▀aClerk: obj
        -"-
    account_1010: obj
        -"-
    -"-
```



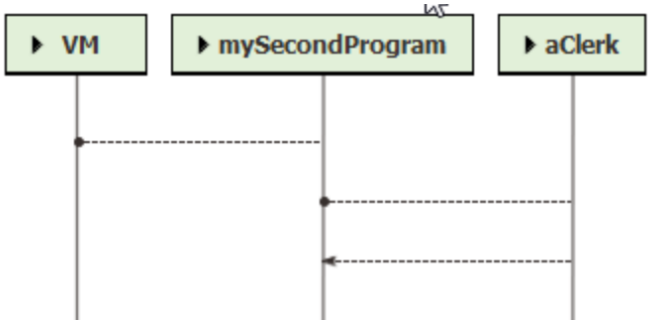
The next snapshot shows the situation after `aClerk` has been generated. The red arrow is positioned at the beginning of the items in `aClerk`, but since `aClerk` has no local objects no more objects are generated here:

```
mySecondProgram: obj▀▀▀aClerk: obj
▀▀      -"-
    account_1010: obj
        -"-
    -"-
```



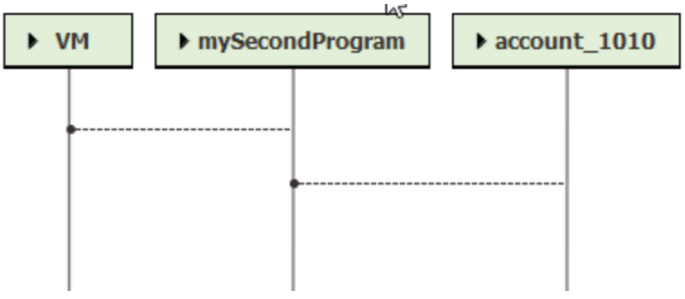
The situation here is after generation of `aClerk` has returned to `mySecondProgram` and the next object to be generated is `account_1010`;

```
mySecondProgram: obj     aClerk: obj
        -"-
    account_1010: obj
        -"-
    -"-
```



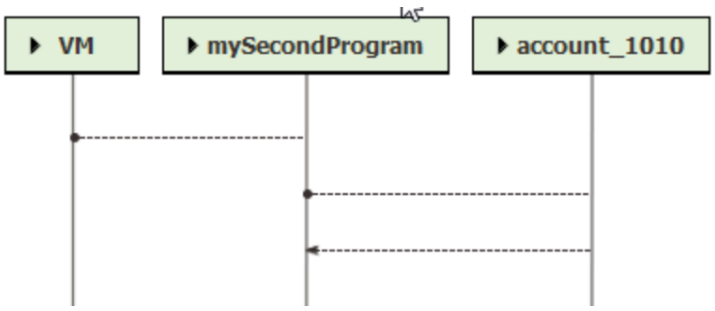
This snapshot shows the situation after `account_1010` has been generated:

```
mySecondProgram: obj     aClerk: obj
        -"-
    account_1010: obj
        -"-
    -"-
```



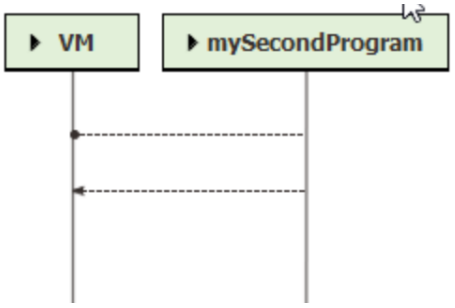
The situation here is after generation of `account_1010` has returned to `mySecondProgram` and generation of `mySecondProgram` has been completed:

```
mySecondProgram: obj     aClerk: obj
        -"-
    account_1010: obj
        -"-
    -"-
```



The situation here is after generation of `mySecondProgram` has returned to `VM` and generation of `mySecondProgram` has been completed:

```
mySecondProgram: obj    aClerk: obj
        -"-
    account_1010: obj
        -"-
    aClerk.handle
```



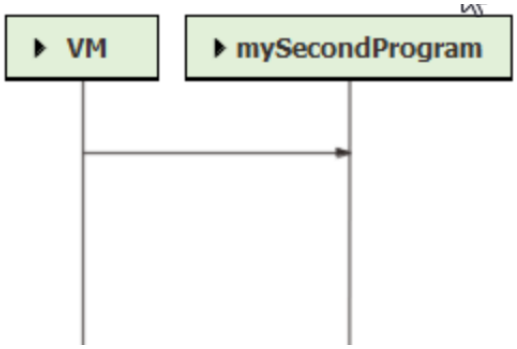
Execution of statements

The next snapshots show how the execution of statements in `mySecondProgram` takes place.

The snapshot below shows the situation just before statements in `mySecondProgram` are executed. As mentioned before, the line with arrow with no method from `VM` to `mySecondProgram` illustrates that v has been generated by `VM`. The snapshot shows the situation before invocation of `clerk.handle` in

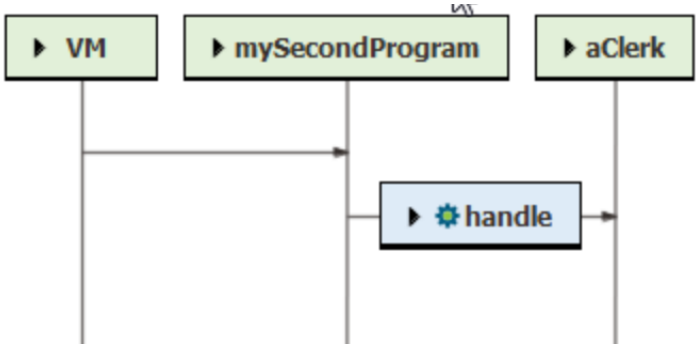
```
mySecondProgram:


mySecondProgram: obj    aClerk: obj
       -"-
    account_1010: obj
       -"-
   aClerk.handle
```



Snapshot when `clerk.handle` has been invoked; before first statement in `handle`:

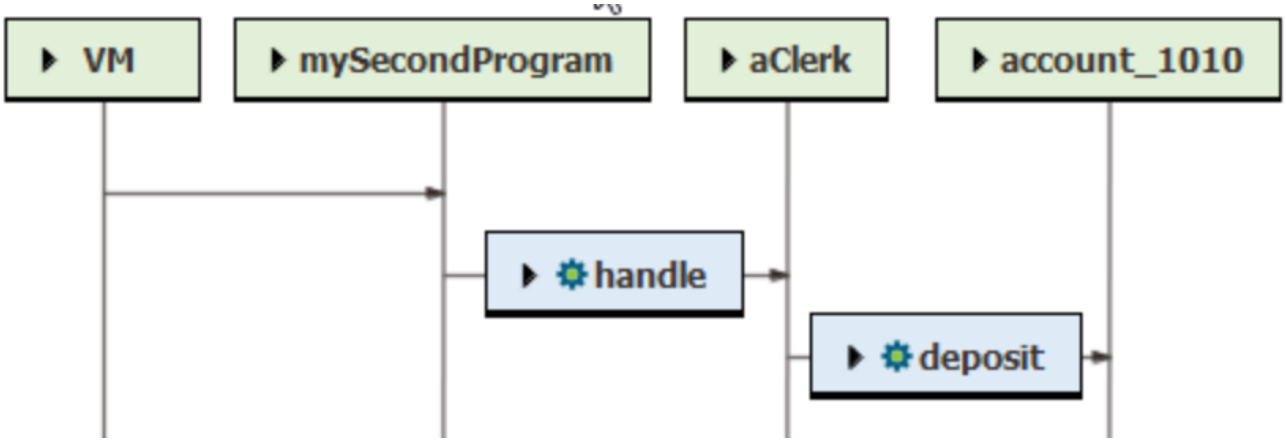
```
mySecondProgram: obj    aClerk: obj
       newBalance: var float
       newBalance :=
                account_1010.deposit(100)
       -"-
    account_1010: obj
       -"-
   aClerk.handle
```



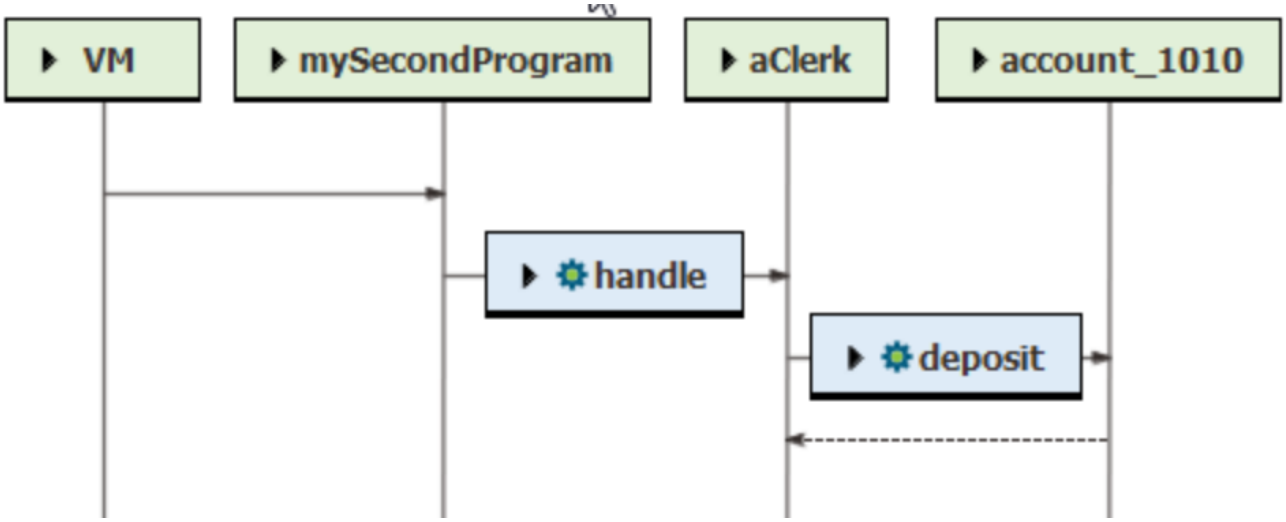
Snapshot when `account_1010.deposit` has been invoked, but before execution of the statement:

```
mySecondProgram: obj    aClerk: obj
    -"-
    account_1010: obj
       deposit(amount: var float):
```

```
        balance := balance + amount
    -"-
aClerk.handle
```

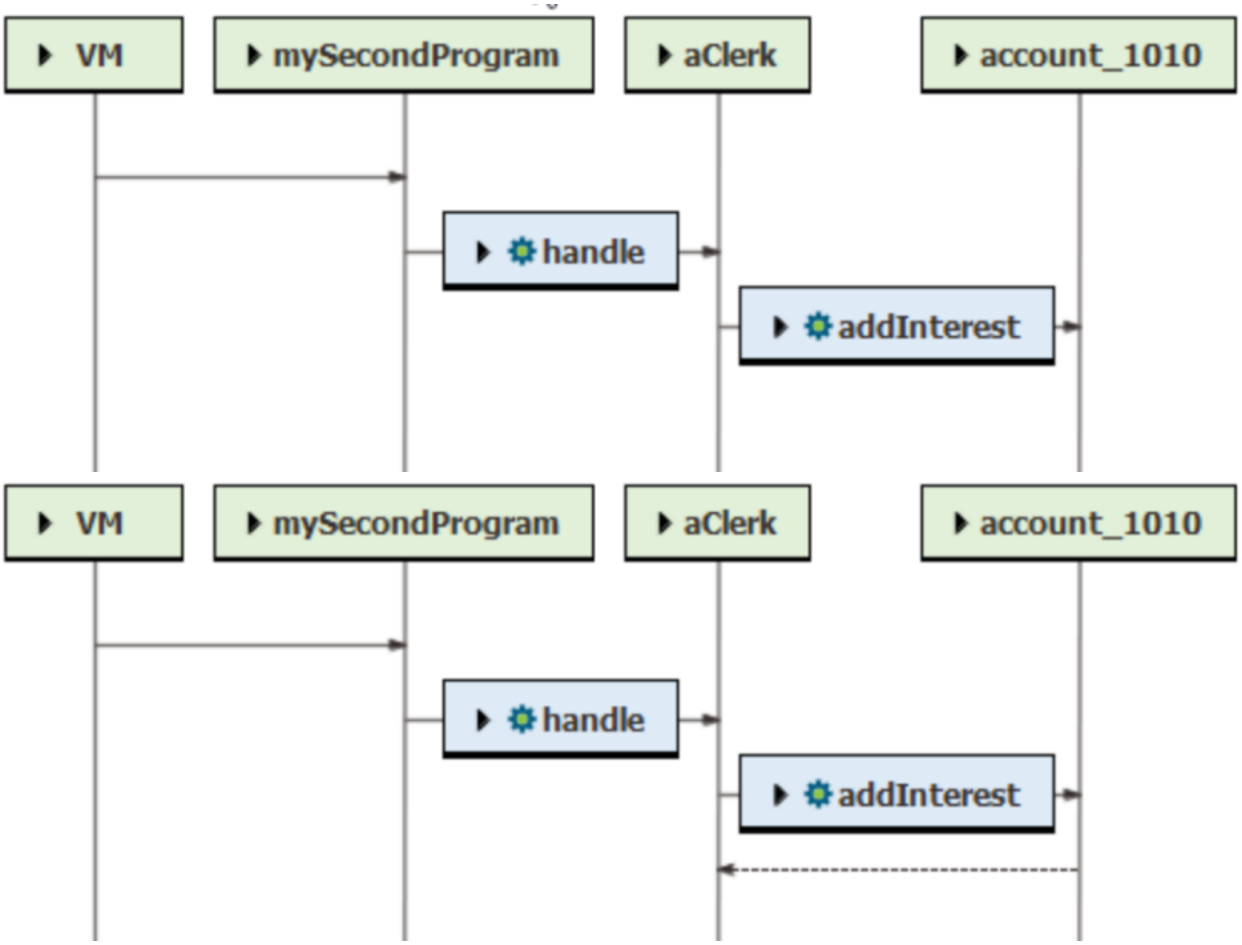


Return from `deposit`:

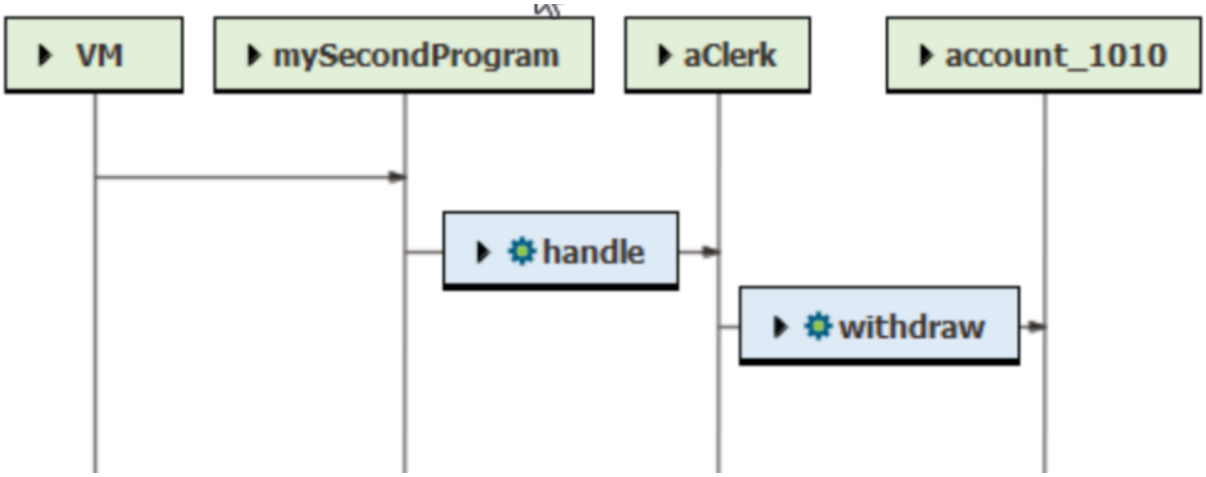


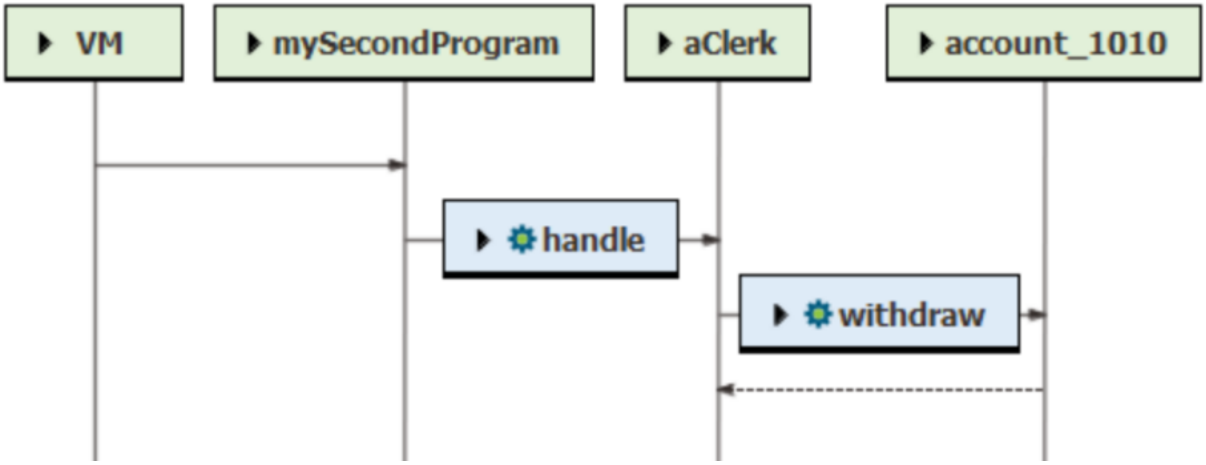
Next `account_1010.deposit(225)` is executed, but no snapshots are shown since they are similar to the above two.

These two snapshots show invocation and return of `addInterest`.

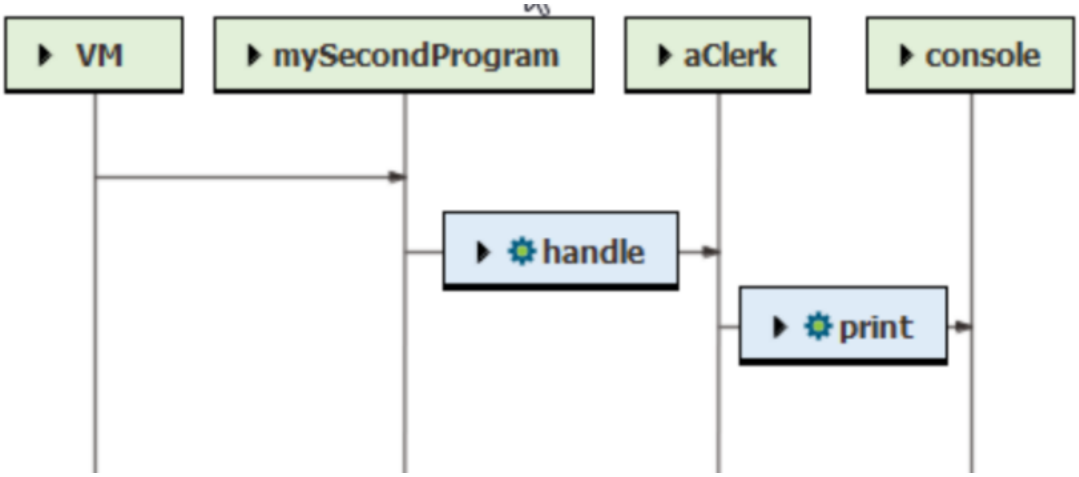


These two snapshots show invocation and return of `withdraw`.

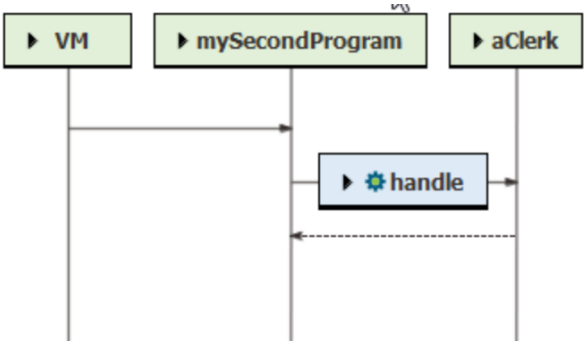




This snapshot shows return from `console.print`.

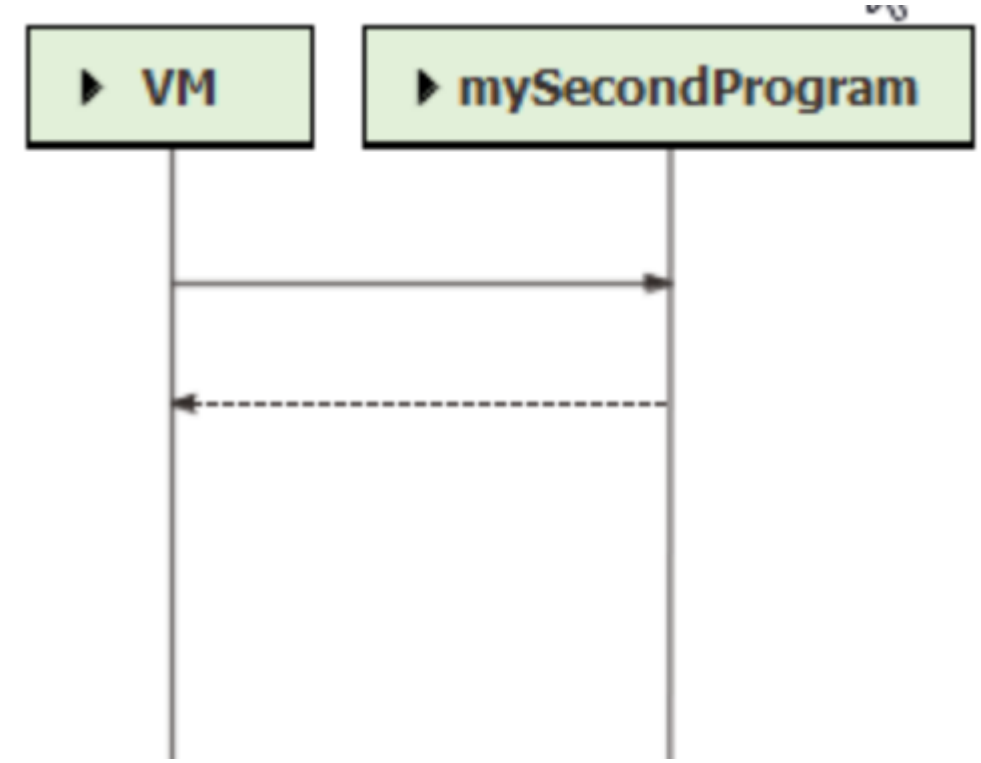


Finally, `aClerk` has no more statements to execute, and this snapshot shows the return from `handle`.



Termination

After return from `handle`, `mySecondProgram` there is no more statement; `mySecondProgram` returns to `VM` and execution is terminated.



Return from `mySecondProgram`.

Execution of `mySecondProgram` has terminated.

As said above, the lifelines and arrows showing method invocations represents how actions are

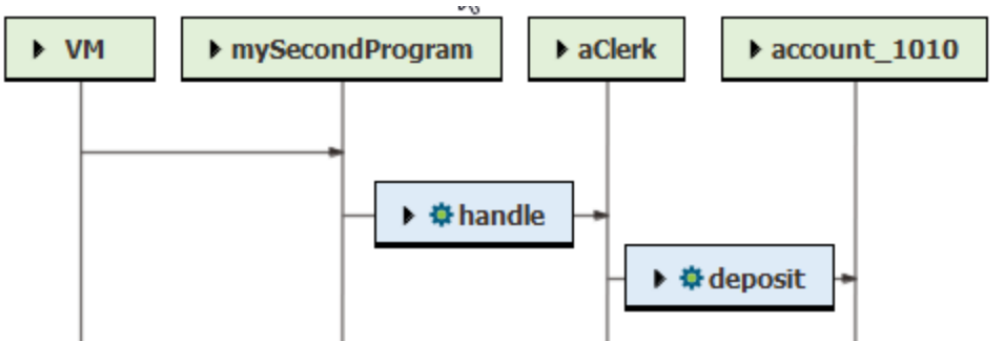
ordered in time. The ordering of the objects in the columns does not matter, but usually a diagram may be more readable if the ordering in time flows from left to right to the extent that this is possible.

The stage of execution

The stage of execution includes the objects and method objects currently executing actions as described below.

The horizontal lines in the diagrams show the active method objects and/or objects of the program execution. For the snapshot here, these are:

```
VM; mySecondProgram; handle; deposit
```



As said `VM` is executing `mySecondProgram`; `mySecondProgram` is executing `handle`; and `handle` is executing `deposit`.

As can be seen, the active objects may include objects like `mySecondProgram` and method objects like `handle` and `deposit`. *Below the term object also covers method object.*

The active objects are often referred to as the *invocation stack*, *execution stack* or just *stack*. The *top* element of a stack is the object that has been invoked last. The *bottom* element is the first object being invoked. In the figure above, `deposit` is on top of the stack and `VM` is the bottom of the stack.

The term stack comes from the fact that a new invocation is always made from the top element, and this new object must return to the previous top object before the previous top object can return. That is, the last object on the stack is the first to return.

In general, 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 pushed latest. We further explain stacks in section .

The active objects on the stack have a *local sequence control* (LSC), which points to the next statement to be executed:

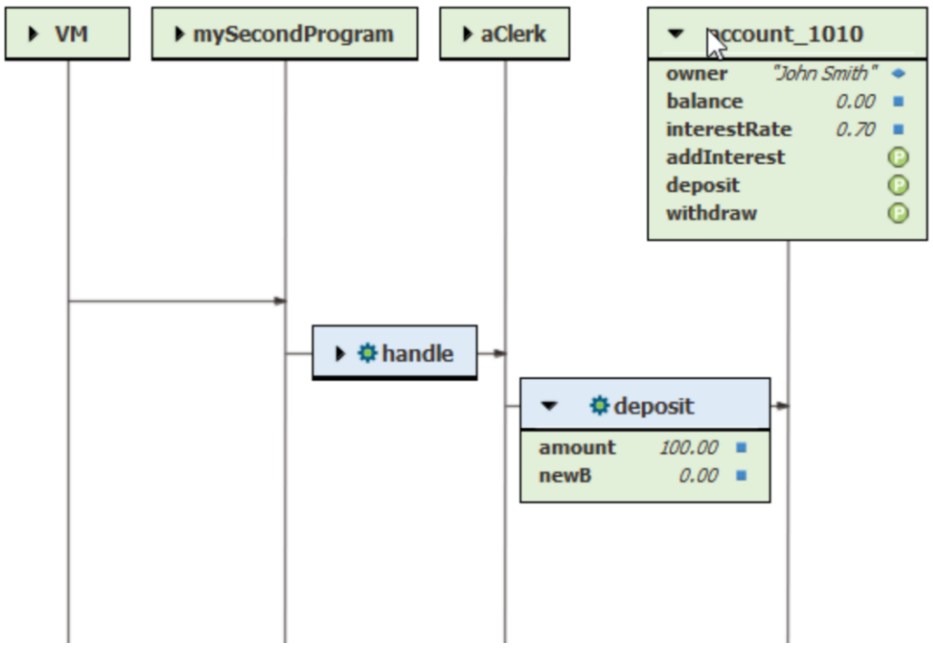
- For the object top of the stack, its LSC points to the next statement to be executed.

- For an object, that has invoked a method, its LSC points to the statement/action to be executed when the method returns.
- Each object has an implicit statement/action at the very end, which when executed returns to its invoker.

The *stage of execution* at a given point in time is thus the current stack including the LSC's of the active objects on the stack.

Inspecting objects

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

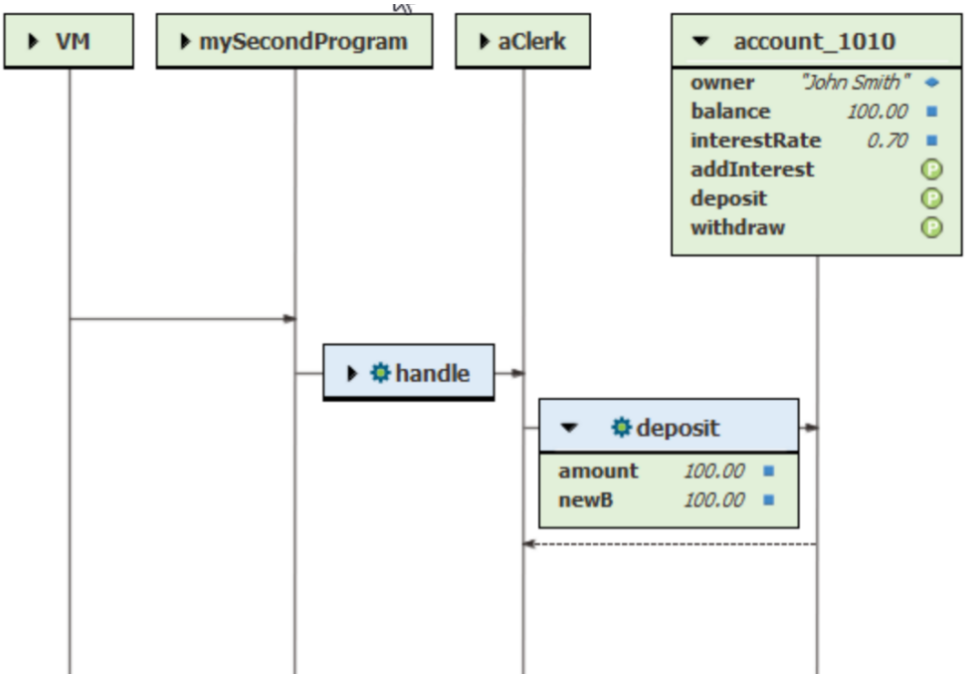


In this snapshot, the `Account_1010` object and the `deposit` invocation have been expanded, and we may thus see the datums of these objects at this point of execution.

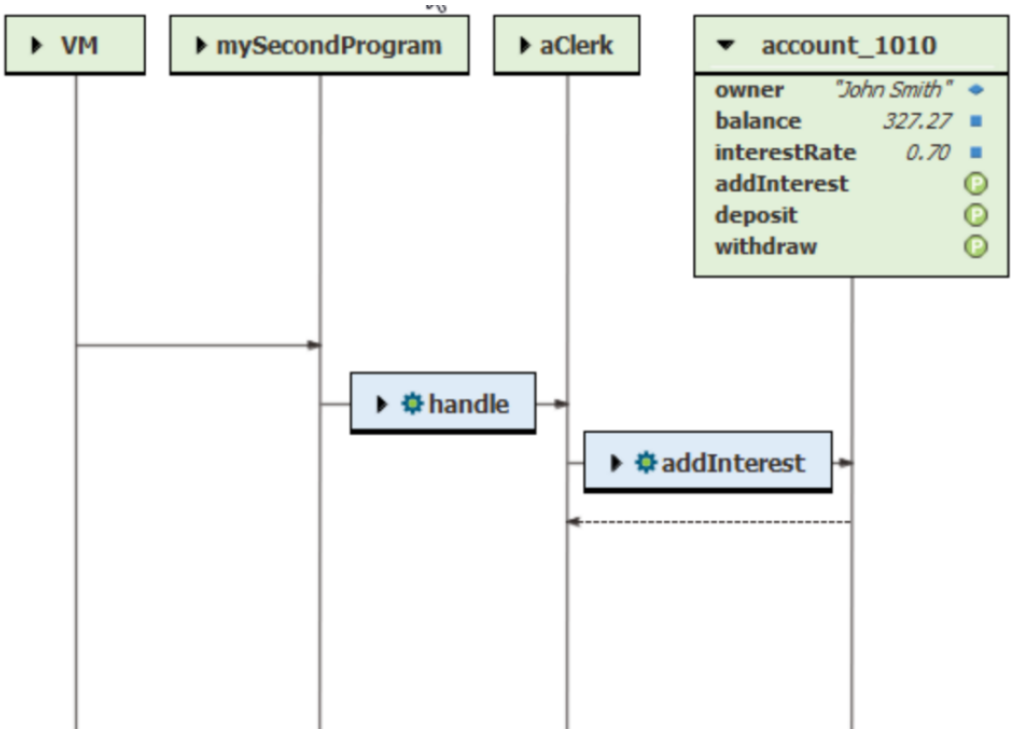
The `amount` parameter of `deposit` has the value 100.00; the return value `newB` has the value 0.00.

For `Account_1010`, the value of `owner` is the `String` `"John Smith"`; the value of `balance` is 0.00 and the value of `interestRate` is 0.70.

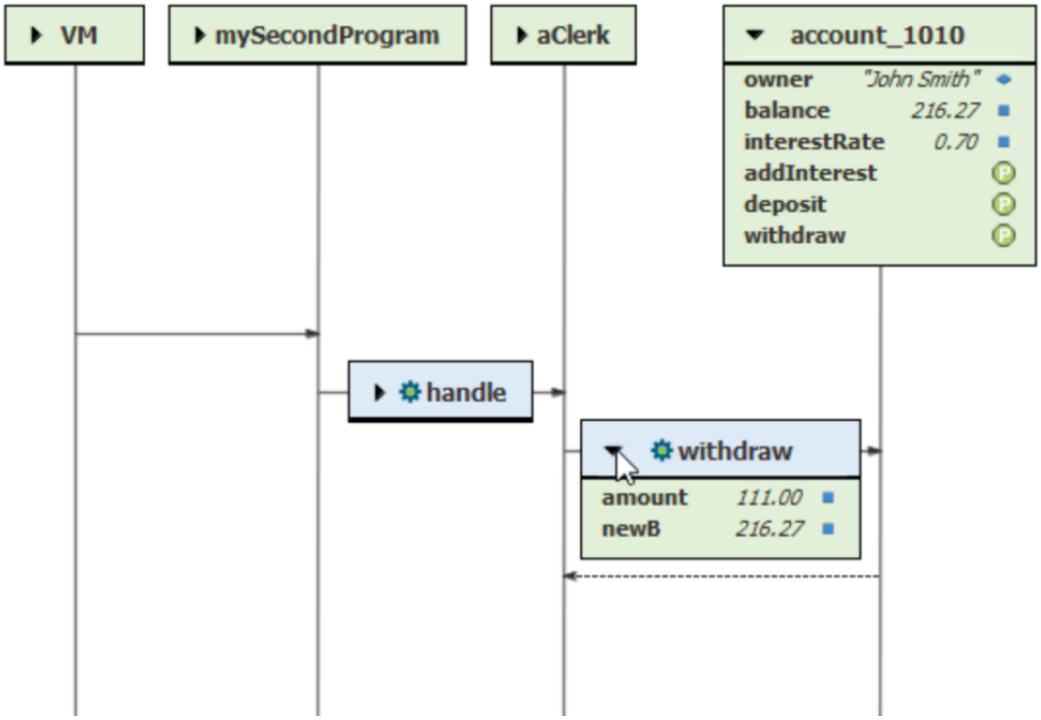
The next snapshot shows the state at the return of `deposit`. Here we can see that the `balance` of `Account_1010` has been updated to 100.00 and that `newB` of `deposit` also has the value 100.00.



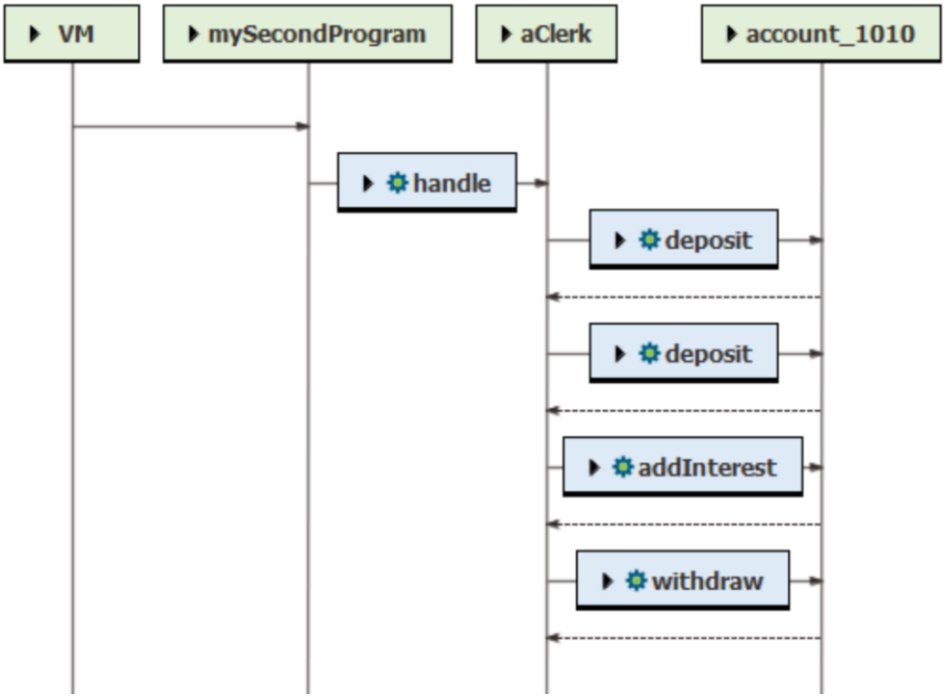
The next snapshot shows the state at the return of `addInterest` and the `balance` is now 327.27.



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`.



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