19. Generation of objects

Description

As we have seen, invocation of methods and instantiation of objects includes generation of method objects and objects. *Generation* of an object includes (1) allocation of possible declarations followed by execution of possible statements.

Allocation of data-items of a primitive type and reference variables consist of initializing these to their default values.

Vi skal tage stilling til om obj betyder ***part object*** eller ***static object*** – vi bruger begge.
Måske static object er bedst da part object signalerer composition og vi bruger jo også obj til aspects unden vi vel siger det er part objects?
Men vi bruger jo også constant object reference – men det går på referencen og ikke objektet!?

Mener vi bruger part object ved aspects

Allocation of a static object specified by **`obj`** or a value object specified by **`var`** and **`val`** implies generation of the objects specified in the declaration. Note that generation of such objects happens recursively starting with allocation of possible declarations followed by execution of possible statements, etc.

Allocation of a class or a method declaration does not imply any actions – and do not confuse this with *generation* of objects from a class or method.

As said, after allocation of possible declarations, possible statements in the object-descriptor are executed. For a coroutine and a parallel object only statements until the first suspend are executed as part of the generation.

If an object has a superclass the declarations in the superclass are also allocated and possible statements in the superclass are executed. The allocation of declarations in a possible superclass takes place before allocation of declarations in the subclass. Execution of statements starts with executing the statements in the superclass. The statements in the subclass are only executed if an `inner` is executed in the superclass.

The description below for generation of objects is in principle the same for method objects – we return to that below. The details of generation is described below.

Generation of objects with no superclass

We start by the simple case with generation of objects with no superclass and which are not coroutines or parallel objects.

The generation of an object takes place in 3 phases: storing of possible arguments in the object including the origin-reference, allocation of declarations, and execution of statements:

1. AllocDeclarations:
    - Space for data-items is allocated as part of invocation of the object being initialized.
    - Data-items holding values of primitive types or references are initialized to default values:
        - The default value of a variable reference declared as `R: ref T` is `none`.
        - The default value of a primitive number is `0` (zero).
        - The default value of a boolean is `false`.
        - The default value of a char is the `null`-character.
    - StoreArguments
        - Possible arguments of the object are assigned to the corresponding data-items in the object.
        - This includes assigning the origin-reference to the data-item holding the origin.
    - Data-items declared using **`obj`**, **`var`** and **`val`** are generated in the order they appear in the object-descriptor:
        - An object declared using `R: obj T` is allocated by generating an instance of `T` and assigning the reference to new `T`-object to `R`.

        - A composite value object declared using `v: val P` or `v: var P` is allocated by generating an instance of `P` and assigning its reference to `v`.
    - Declarations of local classes and methods does not imply any actions.
2. ExecuteStatements:
    - The statements in the object-descriptor are executed in the specified order of appearance.

The purpose of these actions is to initialise the state of the object to ensure that all data-items of the object are well-defined with respect to representing a corresponding phenomena from the application domain, but also so that the possible data-item for implementing the object are well-defined.

Generation of objects with a superclass

Generation of an object with a superclass takes place as follows::

- AllocDeclarations:
    - Space for data-items is allocated as part of invocation of the object being initialized.
    - Data-items holding values of primitive types or references are initialized to default values:
        - The default value of a variable reference declared as `R: ref T` is `none`.
        - The default value of a primitive number is `0` (zero).
        - The default value of a boolean is `false`.
        - The default value of a char is the `null`-character.
    - StoreArguments
        - Possible arguments of the object are stored; this includes storing the origin-reference
        - super.StoreArguments
    - Super.AllocDeclarations
    - Data-items declared using **`obj`**, **`var`** and **`val`** are allocated in the order they appear in the object-descriptor:
        - An object declared using `R: obj T` is allocated by generating an instance of `T` and assigning the reference to new `T`-object to `R`.
        - A composite value object declared using `v: val P` or `v: var P` is allocated by generating an instance of `P` and assigning its reference to `v`.
    - Declarations of local classes and methods does not imply any actions.
- ExecuteMainPart
    - Super.ExecuteMainPart
    - The statements in the object-descriptor are executed in the order of appearance, but only if an inner is executed in super. Execution of an `inner` in super implies execution of the statements in the main part.

Generation of method object

Generation of a method object is the same as generation of an object as described above. However, when the statements have been executed (ExecuteMainPart), the method invocation has finished and the method object is no longer accessible.

Generation of coroutines

Generation of a coroutine – cooperative or preemptive (parallel object) takes place as for an ordinary object. However, by convention, the statements being executed as part of the generation are the statements before the first suspend of the coroutine.

Using initializer methods

Sometimes you may want to define a class that includes different methods for initializing the state of its objects.

Consider the `Customer` class in section . This class has a parameter representing the name of the customer. When we create a `Customer` object, the name attribute is initialized whereas attributes like the address (`addr`) and `email` are not defined. We may add an initializer method to set up these attributes as shown below:

```
class Customer(name: var String):
```

```
   addr: var String
   email: var String
   contacts(a: var String, e: var String) -> Customer:
      addr := a
      email := e
      return this(Customer)
   ...
```

We may then create instances of class `Customer` in the following way:

```
aCustomer: ref Customer
aCustomer :=
   Customer("John Smith).contacts("...  Utopia","john.smith@...")
```

What happens here is that the expression `Customer("John Smith)` creates an instance of `Customer` as described in previous sections. This expression returns a reference to this new `Customer` object. The method `contacts("... Utopia","john.smith@...")` is then invoked on this reference implying that `addr` and `email` are assigned initial values.

Technically there is no difference between a method like `contacts` and other methods. Any method may be invoked as part of the generation of an object. Methods to be used as initialisers must return a reference to the object being generated since this reference in most cases is assigned to a reference variable.

Although technically there is no difference between an initializer method and other methods it is good practice to be explicit about which methods play the role of being initializers.

It is important that the default initialization of an object species by its statements ensures that the state of the object is well-defined since it is optional to invoke additional initializer methods during generation.