5.1.1 Reference assignment and comparison

Description

Der er nok for mange gentagelser her i forhold til forrige afsnit?
Skal det være en selvstændig side eller integreres med siden foran, eller på niveau 2 som 5.2?
Value assignment er 5.5 – altså niveau 2.

In the previous section, we have seen examples of variables that holds references to objects. These include `owner`, `aCustomer`, and `JohnSmithProfile`, which all may refer to `Customer` objects. `JohnSmithProfile` is constant in the sense that it always refer to the `Customer` object representing John Smiths profile. `Owner` and `aCustomer` are variable references and may refer to different `Customer` objects during a program execution and they may hold a `none` reference, which means that they do not refer to any object.

We have also seen examples of assignment of references that has the effect that two or more references may refer to the same object. In the figure +++ in the previous section, `aCustomer` and the `owner` attribute of two `Account` objects all refer to the same `Customer` object.

Here will summarise assignment of references in general and also describe comparison of references. We use class `Customer` for the example.

```
JohnSmithProfile: obj Customer("John Smith")
LizaJonesProfile: obj Customer("Liza Jones")
customerA, customerB: ref Customer
customerA := JohnSmithProfile
customerB := LizaJonesProfile
```

Figure 3.4.1 Customer references

In the above example, we have two `Customer` objects `JohnSmithProfile` and `LizaJonesProfile`, and reference variables `customerA` and `customerB`.

We have assigned `JohnSmithProfile` to `customerA`. This is actually a reference to the `Customer` object representing John Smiths profile. We then have that `JonhSmithProfile` as well as `customerA` both refers to the same `Customer` object.

This is similar for `LizaJonesProfile` – after the assignment `customerB := LizaJonesProfile` both `LizaJonesProile` and `customerB` refers to the `Customer` object representing the profile of Liza Jones.

As mentioned this is illustrated by the figure in the previous section.

Next we may execute an assignment:

```
customerA := customerB
```

The effect of this is that also `customerA` refers to the object representing Liza Jones profile.

See also section .

Comparison

We may also compare references using = (equality) and <> (inequality).

Efter execution of figure 3.4.1, we have:

Og her bruges kommentarer næste gang

```
JohnSmithProfile = LizaSmithProfile    -- false, they refer to different objects
JohnSmithProfile <> LizaSmithProfile   -- true
JohnSmithProfile = customerA           -- true, they refer to the same object
JohnSmithProfile <> customerB          -- false, they refer to different objects
```

After the assignment `customerA := customerB`, we have:

```
JohnSmithProfile = customerA   -- false
customerA = customerB          -- true
LizaJonesProfile = customerA   -- true
```

Assignment between data items being references is called r*eference assignment* and comparison of references is called *reference comparison*.

Reference assignment and reference comparison is fundamentally different from assignment between data items representing values.

The `withdraw` method has a statement:

```
newB := balance
```

Here the value hold by `balance` is copied to `newB`, which then holds the same value as `balance`. The data items `newB` and `balance` are not references to some objects. As we shall se in section X, they are a special kind of objects called *value objects* that may represent values – in section , we describe *value assignment* and *value comparison*.

Type rule

As shown above, we may assign a reference to a `Customer` object to a reference variable that has the type `Customer`. It is not possible to assign a reference to an `Account` object to the reference variable `aCustomer`.

In general the type of an expression in an assignment must be the same as the type of the reference variable being assigned to. This is also the case for passing an expression as an argument to a parameter of a method being a reference.

The above rule does also apply to comparisons using = (equality) and <> (inequality) where both arguments must be of the same type

Consider the following example:

```
aCustomerA, aCustomerB: ref Customer
anAccountA, anAccountB: ref Account
B: var Boolean
aCustomerA := Customer("John Smith")   -- legal
anAccountA := Account(aCustomerA)      -- legal
aCustomerA := anAccountA               -- illegal
anAccountB := Account(anAccountA)      -- illegal
B := aCustomerA = aCustomerB           -- legal
B := aCustomerA <> anAccountA          -- illegal
```

The assignment `anAccountB := Account(anAccountA)` is illegal since the `owner` parameter of `Account` is of type `Customer` whereas the argument `anAccountA` is of type `Account`.

The purpose of the type rule is two fold: from a programming and modeling point of view it does not make sense to allow assignments like `aCustomerA := anAccountA`.

Secondly the type rule is necessary to prevent errors at run-time. Assume that we allow the assignment then we may write code as

```
aCustomerA := anAccountA
aCustomerA.addAccount(JohnSmithProfile)
```

This does not makes sense since a `Customer` object does not have an `addAccount` method.

In chapter , we extend the type rule for assignment, parameter transfer and comparison.

Parameter passing

Passing a parameter as part of a method invocation or class invocation is similar to assignment in the sense that the actual parameter is assigned to the formal parameter of the method or class respectively. This also means that the type rules for parameter passing is the same as the type rule for assignment.