1

Fundamentals on the Semantics of Self and Super

Main Author(s): Ducasse, Wuyts

This lesson wants you to give a better understanding of self and super.

1.1 self

When the following message is evaluated:

```
aWorkstation originate: aPacket
```

The system starts to look up the method originate: starts in the class of the message receiver: Workstation. Since this class defines a method originate:, the method lookup stops and this method is executed.

Following is the code for this method:

```
Workstation>>originate: aPacket

  aPacket originator: self.
  self send: aPacket
```

1. It first sends the message originator: to an instance of class Packet with as argument self which is a pseudo-variable that represents the receiver of originate: method. The same process occurs. The method originator: is looked up into the class Packet. As Packet defines a method named originator:, the method lookup stops and the method is executed. As shown below the body of this method is to assign the value of the first argument (aNode) to the instance variable originator. Assignment is one of the few constructs of Smalltalk. It is not realized by a message sent but handle by the compiler. So no more message sends are performed for this part of originator:.

```
    Packet>>originator: aNode

        originator := aNode
```

2. In the second line of the method originate:, the message send: thePacket is sent to self. self represents the instance that receives the originate: message. **The semantics of self specifies that the method lookup should start in the class of the message receiver.** Here Workstation. Since there is no method send: defined on the class Workstation, the method lookup continues in the superclass of Workstation: Node. Node implements send:, so the method lookup stops and send: is invoked

```
Node>>send: thePacket

    self nextNode accept: thePacket
```

The same process occurs for the expressions contained into the body of the method send:.

1.2 super

Now we present the difference between the use of self and super. self and super are both pseudo-variables that are managed by the system (compiler). They both represents the receiver of the message being executed. However, there is no use to pass super as method argument, self is enough for this.

The main difference between self and super is their semantics regarding method lookup.

- The semantics of self is to start the method lookup **into the class of the message receiver and to continue in its superclasses.**

- The semantics of super is to start the method look into **the superclass of class in which the method being executed was defined and to continue in its superclasses.**. Take care the semantics is **NOT** to start the method lookup into the superclass of the receiver class, the system would loop with such a definition (see exercise 1 to be convinced). Using super to invoke a method allows one to invoke overridden method.

Let us illustrate with the following expression: the message accept: is sent to an instance of Workstation.

```
aWorkstation accept: (Packet new addressee: #Mac)
```

As explained before the method is looked up into the class of the receiver, here Workstation. The method being defined into this class, the method lookup stops and the method is executed.

```
Workstation>>accept: aPacket

    (aPacket addressee = self name)
        ifTrue: [ Transcript show: 'Packet accepted', self name asString ]
        ifFalse: [ super accept: aPacket ]
```

Imagine that the test evaluates to false. The following expression is then evaluated.

```
super accept: aPacket
```

The method accept: is looked up in the superclass of the class in which the containing method accept: is defined. Here the containing method is defined into Workstation so the lookup starts in the superclass of Workstation: Node. The following code is executed following the rule explained before.

```
Node>>accept: aPacket

    self hasNextNode
        ifTrue: [ self send: aPacket ]
```

Remark. The previous example does not show well the vicious point in the super semantics: the method look into **the superclass of class in whichor the method being executed was defined and not in the superclass of the receiver class.**

You have to do the following exercise to prove yourself that you understand well the nuance.

Exercise: 1. Imagine now that we define a subclass of Workstation called AnotherWorkstation and that this class does NOT defined a method accept:. Evaluate the following expression with both semantics:

```
anAnotherWorkstation accept: (Packet new addressee: #Mac)
```

You should be convinced that the semantics of super change the lookup of the method so that the lookup (for the method via super) does NOT start in the superclass of the receiver class but in the superclass of the class in which the method containing the super. With the wrong semantics the system should loop.