

5 Programming with SimTalk

The basic behavior of the Plant Simulation objects often in practice is not sufficient to generate realistic system models. For extending of the standard features of the objects, Plant Simulation provides the programming language SimTalk. With it you modify the basic behavior of individual objects. SimTalk can be divided into two parts:

1. Control structures and language constructs (conditions, loops...).
2. Standard methods of the material and information flow objects. They are built-in and they form the basic functionality, which you can use.

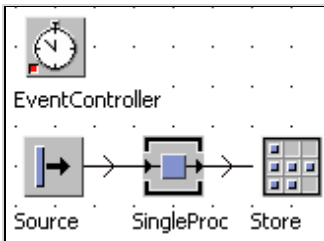
You develop SimTalk programs in an instance of the information flow object Method.

5.1 The Object Method

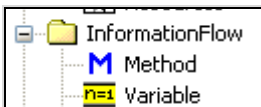
5.1.1 Introductory Example

Example 35: Stock Removal

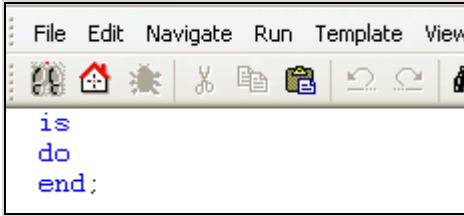
We want to simulate a small production with a store. The capacity of the store is 100 parts. The workplace produces one part every minute (the source delivers ...). Name the Frame “storage”.



You can create controls with Method objects, which are then called and started from the basic objects using their names.




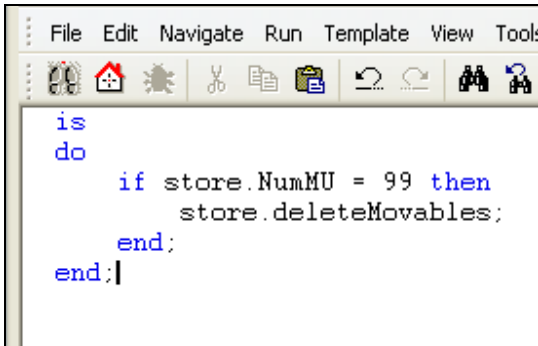
You find the Method in the class library in the folder InformationFlow. Drag a method object to the Frame. A double-click on the icon opens the method.




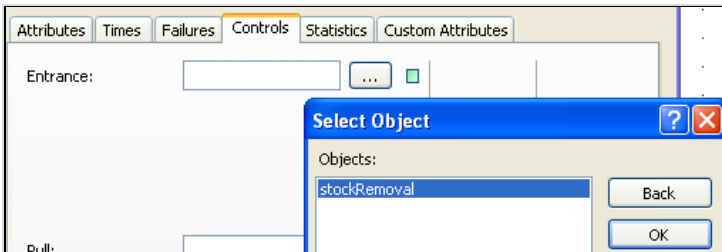
The methods (functions) have always a body:

```
is
do
  -- Statements
end;
```

Declare variables between “is and do”, enter your source code between “do and end”. First, you have to turn off inheritance. Click on the icon  in the editor, you have to formulate the instructions in SimTalk (call your method “stockRemoval”).



Confirm your changes with . You now have to assign the method to an object. For this purpose, each object has one or more sensors. When an MU pass through a sensor, the relevant method is triggered. Double-click on **STORE – CONTROLS – ENTRANCE**; select the correct method, ready we are!



Now start the simulation. If you were successful, then there is no jam. The Store will verify the quantity for each entry. If it is 99, the store will be emptied (a simple solution).

5.2 The Method Editor


Double-clicking a method object opens an editor. You will find a number of functions in the editor, which facilitates your work while programming. If you cannot enter your source code into the method editor, inheritance is still turned on (see above).

5.2.1 Line Numbers, Entering Text




You can display line numbers with the command: **VIEW – DISPLAY LINE NUMBERS**. The following rules apply for entering text:

- Double-clicking selects a word.
- Clicking three times selects a row.
- Ctrl + A selects everything.
- Copy does not work with the right mouse button (until version 9). Use Ctrl + C to copy and Ctrl + V to insert text or use the menu commands Edit - Copy, etc.
- Use Ctrl + Z to undo the last change (or Edit Undo)...
- Move also works by dragging with the mouse.

5.2.2 Bookmarks

For faster navigation, you can set bookmarks in your code. The bookmarks are displayed in red. To insert a bookmark, select any text and click: 

Bookmark functions:

Icon	Description
	Deletes all bookmarks in the method.
	The cursor moves to the previous bookmark.
	The cursor moves to the next bookmark...

5.2.3 Code Completion

The editor supports automatic code completion. If there is only one possibility of completion, Plant Simulation shows the attribute, the method, or variable as a light blue label. You can accept the suggestion with Ctrl + space bar.

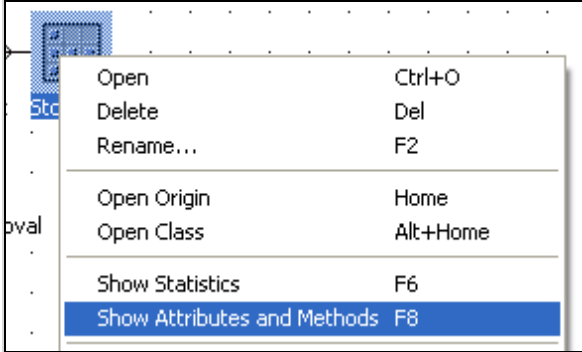
```
if store.NumMU = deleteMovable;
    store.deleteM;
end;
```

Starting from an object you can display all possible completions. Simply press CTRL + SPACE. In the list you can scroll with the direction buttons, an entry will be accepted with Enter.



5.2.4 Information About Attributes and Methods

You can always get information about the built-in attributes and methods of an object by Show Attributes and Methods in the context menu of an object.



In the table, all methods and attributes are shown (even those you have defined).

Name	Signature	Value	i.
addObserver	(string,method)		
attributeWatchable	(string) : boolean		
Availability	real	100	i
AvailabilityOn	boolean	true	

The column signature allows you to deduce whether it is a method or an attribute and which data you need to pass or what type is returned. If the column only shows the data type, the entry then is an attribute.

Example: Show the attributes and methods of the store.

RecoveryTime	time	0.0000	i
--------------	------	--------	---

Recovery time is an attribute; the data is not in parentheses. To set the recovery time, you have to type:

```
Store.recoveryTime:=120;
```

Set the value of an attribute with „:=“.

PE	(integer, integer) : object
----	-----------------------------

PE is a method. The method PE expects two arguments of data-type integer and returns an object. PE allows you to access a particular place of the store. You will call the method with parentheses:

```
Store.PE(1,1);
```

mirrorX			
mirrorY			

The row mirrorX does not contain a value in the column signature. MirrorX flips the icon on the x-axis. It is a method which has no arguments and returns no value. You will call this method without parentheses.

```
Store.mirrorX;
```

startPause	([time])		
------------	----------	--	--

The parentheses in the column signature indicate that startPause is a method. The data-type is given within square brackets. This means that the argument is optional. This results in two possibilities of the call:

```
Store.startPause; -- no time limit
Store.startPause(120); -- pause for 120 seconds
```

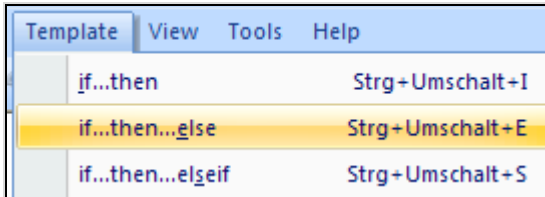
Note:

Some attributes are read-only. You can assign no value to these attributes. Online help describes whether an attribute is read-only.

5.2.5 Templates

For a number of cases, Plant Simulation includes templates, which you can insert into your source code, or which you can use as a starting point for developing controls.

You reach Templates via the menu **TEMPLATE**:

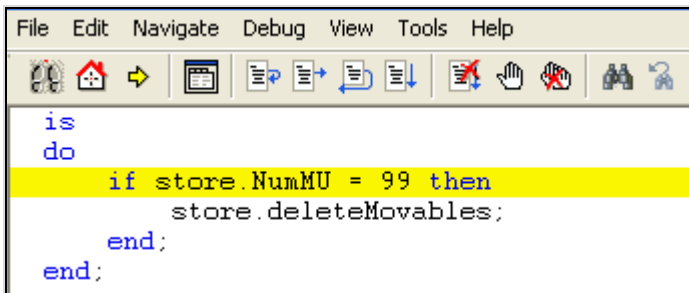


In the method editor click into the row in which the snippet should be inserted. Then click, e.g., **TEMPLATE – IF ... THEN ...** Under **SELECT TEMPLATE** you find more templates. Watch out! Most templates in this selection completely replace your source code. You can use the Tab key for moving through the template (between the areas in angle brackets).

5.2.6 The Debugger

The debugger helps you to correct your methods. Using the F11 key, you can quickly change between editor and debugger (or **RUN – DEBUG**).

Example: Open the method from the example above, place the cursor in the text, and then press F11.



Using F11 you can, for example, move through your method stepwise and see what happens. If the method is completed, the method will open again in the editor. If an error occurs during the simulation, the debugger automatically opens and displays error messages.

Note:

If you have to make changes in the source code in the debugger, then you can save the changes by pressing the F7 key.

5.3 SimTalk

SimTalk does not differentiate between upper- and lowercasing in names and commands. At the end of a statement, you need to type a semicolon. Blocks are bounded by an “end” (no curly brackets as in Java or C++). If you forget the “end”, Plant Simulation always looks for it at the end of the method.

5.3.1 Names

Plant Simulation identifies all objects and local variables using their name or their path. You can freely select new names with the exception of a few key words. The following rules apply:

- The name must start with a letter. Letters, numbers, or the underscore “_” may follow.
- Special characters are not allowed.
- There is no distinction between capital and lowercase letters.
- Names of key words and names of the built-in functions are not allowed.

For methods some names are reserved:

- **Reset**: Will be executed when clicking Reset in the Eventcontroller.
- **Init**: Will be executed when clicking Init in the Eventcontroller.
- **EndSim**: End of simulation (reaching the end time for simulation).

Generally, key words from SimTalk are prohibited names (all terms in SimTalk, which are highlighted in blue).

5.3.2 Anonymous Identifiers

SimTalk uses anonymous identifiers as wild cards. When running the method, these anonymous identifiers will be replaced by real object references.

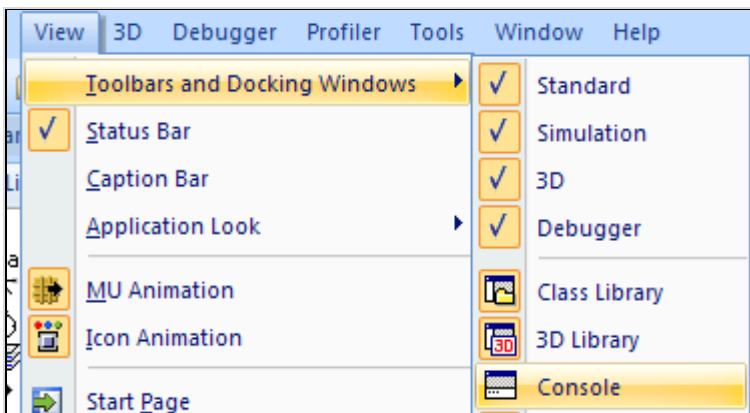
root

The anonymous identifier “root” always addresses the top of the Frame hierarchy. Starting from this Frame, you can access underlying elements.

Example 36: root

First, open the console in Plant Simulation:

VIEW – TOOLBARS AND DOCKING WINDOWS – CONSOLE



Create a new method. Use the method from the introduction example. Complete the method as follows:

```
is
do
  -- writes the root name in the console
  print root.name;
end;
```

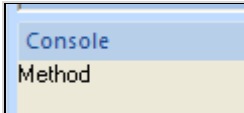
The console will show the name of the Frame in which the method was placed.

self

Self returns a reference to the current object (itself).

Example 37: Anonymous Identifier self

```
is
do
  -- The name of the current method will be
  -- written to the console
  print self.name;
end;
```



current

Current is a reference to the current Frame.

?

? denotes the object that has called the method (e.g., the object in which the method is entered as an exit control). The question mark allows a method to be used without modification by several objects.

@

@ refers to the MU which has triggered the method (so you can access, e.g., on all outgoing MUs).

basis

Basis denotes the class library. You can only use it in comparisons.

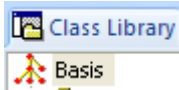
5.3.3 Paths

When objects are not located in the same frame or folder (name space), a path has to be put in front of the name. Only the path allows clearly identifying an object,

so that it can be reached. A path is composed of names and periods (which serve as a separator). Paths are divided into two kinds of paths:

- Absolute paths
- Relative paths

5.3.3.1 Absolute Path



The starting point of the absolute path is the root of the class library. From here on objects are addressed to the “bottom”. An absolute path always starts with a period.

Example:

```
Modelle.Frame.workplace_1
```

5.3.3.2 Relative Path

A relative path starts within the frame, within which the method is located (without the first period).

Example:

```
workingplace_1
```

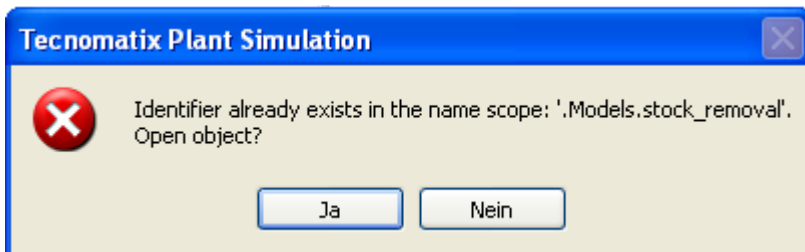
Workingplace_1 is located in the same Frame as the method.

```
controls.Method1
```

This address refers to an object with the name “Method1” in a subframe “controls”.

5.3.3.3 Name Scope

All objects, which are located in the same frame or folder, form a name scope. Within a name scope identical names are not allowed. In other words, names in a name scope may occur only once, all objects must have different names. In different Frames identical names may occur. Their path distinguishes the objects. If you try to assign a name twice, Plant Simulation shows an error message:



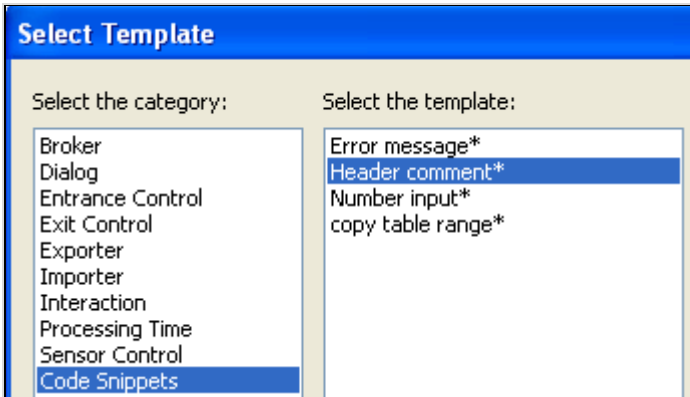
5.3.4 Comments

Comments explain your source code. Plant Simulation distinguishes between two types of comments:

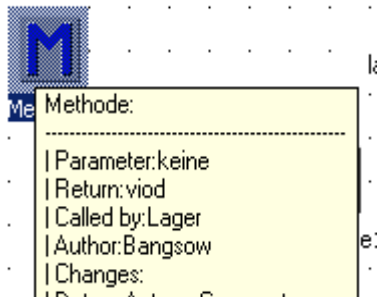
```
-- Comment until the end of line
/* Beginning of a comment, which
   extends over several lines
*/
```

Plant Simulation displays comments in green. We recommend to comment your source code. In this comment, you should enter pertinent information about your method. Plant Simulation provides a template for this purpose: **TEMPLATE – SELECT TEMPLATE ...**

You can find the header comment in **CODE SNIPPETS**.



The header comment (started with “--” in front of the keyword is) is shown as a Tooltip in the Frame, when you place the mouse on the method.



5.4 Variables and Data Types

5.4.1 Variables

A variable is a named location in memory (a place where the program stores information). Variables should have meaningful symbolic names. You first have to declare a variable (introduce its name) before you can use it. Plant Simulation distinguishes between local and global variables. A local variable can only be accessed within the respective method. This variable is unreachable for other methods. Global variables are visible for all methods (in every Frame). You can set values and get values from all methods. This way, you can organize data exchange between the components.

5.4.1.1 Local Variables

Local variables are declared between “is” and “do”. A declaration consists of the name of the local variable, a colon, and a data type. The keyword “do” follows after the last declaration.

Example:

```
is
  Name : Type;
do
  -- statements
end;
```

For instance:

```
is
  stock_in_store : integer;
do
  ... -- statements
end;
```

The declaration reserves an address in the memory and you define access on it by a name. For this reason, the operating system must know what you want to save. A true/false value requires less memory (1 bit) as a floating-point number with double precision (min. 32 bit). The information about the memory size takes place through the so-called data types. The data type determines the maximum value range of variables and regulates the permissible operations.

A value is assigned to a variable with :=.

Example 38: Declaring Variables

The circumference of a circle is to be calculated from a radius and Pi. Pi is defined in Plant Simulation and can be retrieved via Pi. The result is written on the console with the command print.

```

is
  radius :integer; --integer number
  circum :real; --floating point number
do
  radius:=200;
  circum:= radius*PI;
  print circum;
end;

```

SimTalk provides the following data types: acceleration, any, boolean, date, datetime, integer, length, list, money, object, queue, real, speed, stack, string, table, time, timeSequence, and weight.

Name	Range of values
acceleration	real, m/s ²
any	the data type will be determined only after the assignment of the value (like VB: variant)
boolean	TRUE or FALSE
integer	-2.147.483.648 bis 2.147.483.647
real	floating-point numbers
string	character (each letter, numbers)
object	Reference to an object (except comment)
table	local variable with the behavior of a table
list	see above
stack	see above
queue	see above
money	...
Length	as real, the value is interpreted as meters
weight	see above, kg
speed	real → m/s
time	real → sec.; output: <hh>:<mm>:<ss.sss>
date, datetime	date from 1.1.1970 to 31.12.2038

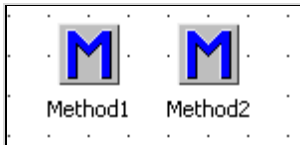
Data types can be converted to a limited extent. Plant Simulation initializes all local variables automatically. The value depends on the data type:

Type	Initialization
boolean	FALSE
integer	0
real	0
string	"" (empty string)
object	Void
table	Void
list	Void
stack	Void
queue	Void
money	0
length	0.0
weight	0.0
speed	0.0
time	0:00:00
date	1.1.1970 0:00:00

If you want to define another start value in the simulation, you can, for example, use the `init` method.

Example 39: Global Variables

You need two methods in a Frame (Method1 and Method2):



In Method2, define a variable of type integer with the name “number”. Assign the value 11 to the variable.

Method 2:

```
is
    number:integer;
do
    number:=11;
end;
```

In Method1, you now try to read the variable “number” and to write the value of “number” to the console.

Method1:

```
is
do
```

```

    print number;
end;

```

If you run Method1 (F5 or Run – Run), you get an error. It opens the debugger and the faulty call is highlighted. The error description is displayed in the status bar of the debugger:

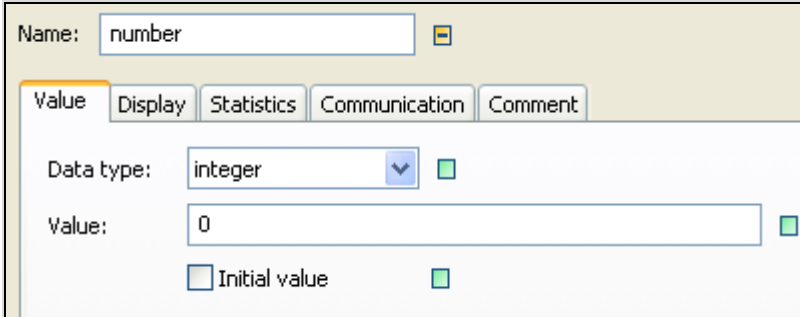


Method1 cannot access a variable “number” of Method2.

If you need the data in several methods, you have to define a global variable (variable object from the folder information flow **n=1** Variable) to exchange data. All methods can set and get the value of these variables. The global variable is defined as an object in the class library and is addressed just like the other objects by name and path. You have to determine the data type of the variable and can specify a start value.

Example 40: Global Variable 2

Insert a global variable into the Frame above. Rename the variable “number”, data type integer (start value remains 0).



Change Method2 like this: Delete the variable declaration of “count”, leave the rest unchanged:

```

is
do
    count := 11;
end;

```

Now start Method2 (the value of the global variable would have to change), and then Method1. The value of the global variable is displayed in the console.

Global variables are reset to the start value when you press the **RESET** button, if you select the option **INITIAL VALUE** and specify a start value. In the previous example, the value will be set to 0 when you enter the following setting:



5.5 Operators

By a combination of constants and variables, you can define complex expressions (e.g., calculations). Operators are used for concatenating expressions. Plant Simulation differentiates between:

- Mathematical operators
- Logical (relational) operators
- Assignment operators

Logical Operators are, for example, needed for comparisons.

5.5.1 Mathematical Operators

SimTalk recognizes the following mathematical operators:

- Algebraic sign, subtraction
- * Multiplication
- / Division
- // Integer division
- \ \ Modulo (remainder of integer division)
- + Addition/concatenation of strings

The integer division, which is defined for the data-type Integer, always delivers a whole number. Any decimal places are suppressed. When calculating data for the data type real, the result is output up to seven valid digits (eighth digit rounded, working with decimal power).

5.5.2 Logical (Relational) Operators

Operator	Function	Result
AND	logical AND	TRUE, if all expressions are TRUE
OR	logical OR	TRUE, if at least one expression is TRUE
NOT	Not	Invert the boolean-value
<	Less than	
<=	Less than or equal	

>	Greater than	
>=	Greater than or equal	
=	equal	
/=	unequal	

A logical expression is always interpreted from left to right. The evaluation is completed once the value of an expression is established.

Example 41: Logical Operators

Simple method (+ start the console)

```

is
  local
    num1:integer;
    num 2:integer;
    num 3:integer;
    val1:boolean;
    val2:boolean;
    val3:boolean;
  do
    num1:=10;
    num2:=23;
    num3:=num1*num2;
    val1:=num1<num2;
    val2:=num1<num2;
    val3:=val1 AND val2;
    print val3;
  end;

```

Try out the operators. Let the console show different variables (Save + F5).

5.5.3 Assignments

The operator “:=” assigns a new value to a variable. First, the expression to the right of the operator is calculated. If the value and the variable have the same data type, then the value is assigned to the variable.

Example 42: Variable – Value Assignment

```

is
  num:integer;
do
  num:=1;

```



```

    num:=num+1;
-- first num+1, then assignment to num
    print num;  -- print the new value of num
end;

```

If the data types are different, the values have to be converted. Plant Simulation automatically converts real into integer and vice versa. For other types of data, you need to use type conversion functions.

Example 43: Type Conversion 1

```

is
    num:integer;
    text:string;
    res:integer;
do
    num:=10;
    text:="20";
    res:=num*text;
    print res;
end;


```

Run time error:

```

    text:="20";
    res:=num*text;
    print res;

```



Type mismatch.

stockRemoval

The most important type conversion functions are:

Syntax	Return value data type
bool_to_num(<boolean>)	real
num_to_bool(<integer>)	boolean
str_to_bool(<string>)	boolean
str_to_date(<string>)	time
str_to_datetime(<string>)	datetime
str_to_length(<string>)	length
str_to_num(<string>)	real
str_to_obj(<string>)	object

<code>str_to_speed(<string>)</code>	<code>speed</code>
<code>str_to_time(<string>)</code>	<code>time</code>
<code>str_to_weight(<string>)</code>	<code>weight</code>
<code>To_str(<any>, ...)</code>	<code>string</code>

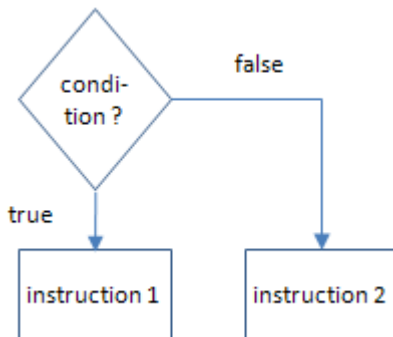
In the example above, a conversion from string to integer is required. You can realize a type conversion of text with the help of the function `str_to_num (...)`. The method has the following syntax:

```
str_to_num (text)
```

Expand the example:

```
is
  num:integer;
  text:string;
  res:integer;
do
  nun:=10;
  text:="20";
  res:=num*str_to_num(text);
  print res;
end;
```

5.6 Branching



After testing a condition, the branch decides which of the following instructions should be executed. If the condition is met, the if-branch (TRUE) will be executed. If the condition is not met, the else branch (False) will be executed.

The general syntax is as follows:

```
if condition then
  instruction1;
else
  instruction2;
end;
```

Example 44: Branch 1

You only need one method for the example. We want to query if value1 is less than 10. If yes, then a message "If branch is executing" should be displayed in the console, otherwise "Else branch is executing"

```

is
  local
    value1:integer;
do
  value1:=12;
  if value1< 10 then
    print " If branch is executing";
  else
    print " Else branch is executing";
  end;
  print " Here we continue normally";
end;

```

Try different queries!

After passing through the branch, the execution of the code continues. If more than one condition is to be checked, the conditions can be nested. The nesting depth is not limited. In this case, a new condition begins after the if-branch with "elseif".

Example 45: Branch 2

Extension above:

```

is
  local
    value1:integer;
do
  value1:=7;
  if value1= 10 then
    print " value1 is 10.";
  elseif value1=9 then
    print " value1 is 9";
  elseif value1=8 then
    print " value1 is 8";
  elseif value1=7 then
    print " value1 is 7";
  -- and so on ...
  end;
  print "Here we continue normally. ";
end;

```

```

Console
value1 is 7
Here we continue normally.

```

If you have to check many conditions, this construction gets complicated quickly. You can then use so-called case differentiations.

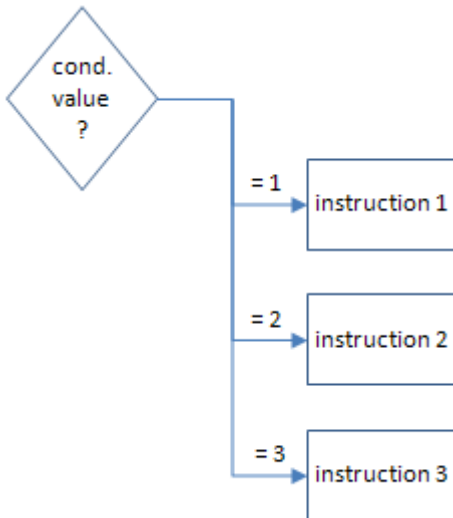
5.7 Case Differentiation

Case differentiation in SimTalk has the following syntax:

```

inspect <expression>
  WHEN <constant_1> THEN <instruction 1>
  WHEN <constant_2> THEN <instruction 2>
  -- ...
end;

```



Example 46: Case Differentiation

```

is
  local
    num:integer;
do
  num:=2;
  inspect num
    when 1 then print "Num is 1.";
    when 2 then print "Num is 2.";
    when 3 then print "Num is 3.";

```

```

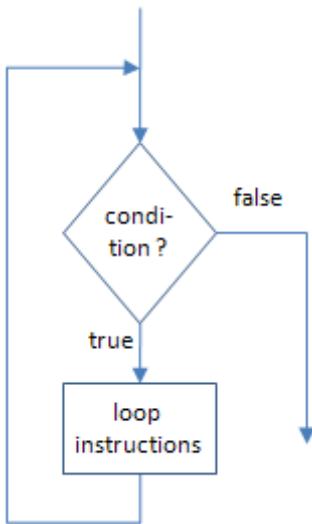
-- and so on
else
    print "Not 1, not 2, not 3 !";
end;
end;

```

5.8 Loops

5.8.1 Conditional Loops

5.8.1.1 Header-Controlled Loops



Before passing through the loop instructions, Plant Simulation checks whether a condition is met or not. The loop is repeated if the validation of the condition returns true. If the condition before the first loop is not met, the loop instructions will not be executed.

Make sure that the loop condition is false some of the time (e.g., increase in the value of a variable, until their value exceeds a certain limit).

Endless loops are terminated with the key combination CTR + ALT + SHIFT.

Syntax:

```

while <condition> loop
<instructions>
end;

```

Example 47: while-Loop

Loop 1, Loop 2 to Loop 10 should be written to the console.

```

is
i:integer;
do
i:=1;
while i<10 loop
    print "Loop run number:" + to_str(i);

```

```

    i:=i+1;
end;
end;

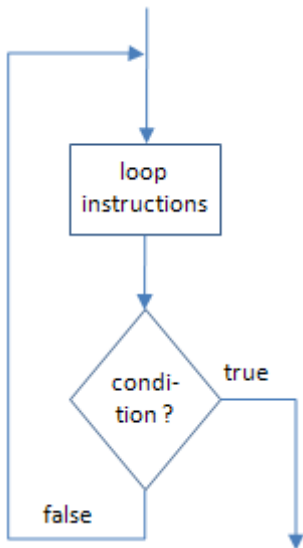
```

```

Console
Loop run number:1
Loop run number:2
Loop run number:3

```

5.8.1.2 Footer-Controlled Loops



The condition is checked after the execution of the loop statement. If the condition for a termination of the loop is not met, the loop statement will be executed one more time. The loop statement is executed at least once.

Syntax:

```

repeat
  -- instructions
until <break condition is met>;

```

Example 48: repeat-Loop

```

is
  i:integer;
do
  i:=1;
  repeat
    print "Loop number:" + to_str(i);
    i:=i+1;
  until i>5;
end;

```

5.8.2 For-Loop

If you know exactly how often the loop is to be iterated, you can use the for-loop or the from-loop. It needs a running variable to control the number of runs of the loop. The variable is increased or decreased during each run starting from an initial value by a certain value. When a predetermined end value is reached, the loop will be terminated.

Syntax1:

```
from < Initialization > until <condition> loop
  <instructions>
end;
```

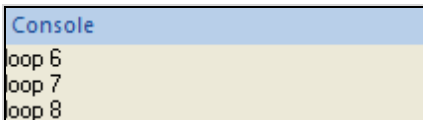
Syntax 2:

```
for < Initialization > to
  <end value> loop
  -- loop instructions
next;
```

Example 49: from-Loop

Outputs will be shown in the console (loop 1, loop 2, and so on)

```
is
  local
  i:integer;
do
  from i:=1; until i=10 loop
    print "loop " + to_str(i);
    i:=i+1;
  end;
end;
```



```
Console
loop 6
loop 7
loop 8
```

Example 50: for-Loop

A loop should be executed 5 times:

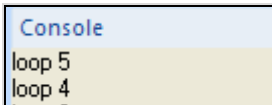
```
is
  i:integer;
do
  for i:=1 to 5 loop
    print "loop " + to_str(i);
  next;
end;
```

You can count in the loop backwards with “downto” instead of “to”.

Example 51: for-Loop with downto

Similar to the previous example:

```
is
  i:integer;
do
  for i:=5 downto 1 loop
    print "loop " + to_str(i);
  next;
end;
```



5.9 Methods and Functions

A function is the definition of a sequence of statements, which are processed when the function is called. There are functions in different variants:

Arguments are passed to some functions, but not to others. (Arguments are values, which must be passed to the function, so the function can meet its purpose.) Some functions give back a value, others do not.

5.9.1 Passing Arguments

Arguments serve the purpose for passing data on during the function call (not just a stock removal function call, for example, but at the same time the number of the average stock removal per day is handed over by function call). The data type for the given value must match the data type of the argument declared in the function. The arguments have to be declared in the function. The declaration will be made at the beginning of the method before “is” in parenthesis in the following format:

```
(name : type)
```

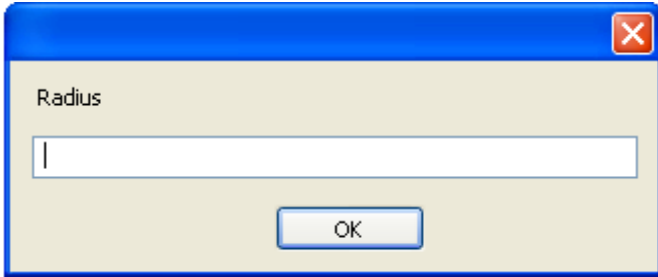
For example:

```
(Stock_removal : integer)
```

Within the method, the arguments can be used like local variables, with the caller determining the initial values.

Example 52: Passing Arguments 1

The user is to enter the radius of a circle, and the size of the circumference is to be displayed in the console. To enter the argument, you need a text box. This is called by the function “prompt”. Using the method “prompt”, you can ask the user for input. If you pass a string to the method “prompt”, then this string will be shown as a command prompt:



Names: Frame: Programming, Method: Test, the type of the data to be read is string, a type conversion is therefore necessary (`str_to_num` (identifier)).

Method Test:

```
is
  radius :string;
  circumference : real;
  val:real;
do
  -- prompt
  radius:=prompt("Radius");
  -- type conversion to real
  val:=str_to_num(radius);
  -- calculate circumference and display in
  --the console
  circumference:=val*2*PI;
  print  circumference;
end;
```

Open the console to see the result.

5.9.2 Passing Several Arguments at the Same Time

Within the definition, a semicolon separates several arguments. When you call the function, you have to pass the same number of arguments that you have defined in the function.

Example 53: Passing Arguments 2

For two given numbers, the larger number is to be returned after the call of the function. Name the function getMax (number1, number2).

```
(number1:integer;number2:integer)
-- passing arguments
:integer --type of the return value
is
do
  if number1 >= number2 then
    result:=number1; -- return value
  else
    result:=number2;
  end;
end;
```

Call the function (from another method):

```
is
do
  print getMax(85,23);
end;
```

5.9.3 Result of a Function

Methods can return back results (usually a method that returns a value is called a function). For this purpose, you have to enter a colon and the type of the return value before “is”. The result of the function has to be assigned within the function to the automatically declared local variable “result”. Another possibility is to use “return”. Return passes program control back to the caller. You can also pass a value on this occasion. After processing, the function will return the content of the variable “return” to the caller (the return value will replace the function call).

Example 54: Results of a Function

A function “circumference” is to be written. The radius will be passed to the function and the function returns the circumference.

Function circumference:

```
(radius:real) -- argument radius (2)
:real -- data type of the return value
is
do
  result:=radius*2*PI; -- (3)
end;
```

Method Test (in the same Frame):

```

is
  res:real;
do
  res:= circumference (125); -- (1)
  print res;
end;

```

Explanation:

- (1) A value is given when calling the function.
- (2) The function declares the arguments.
- (3) The function inserted the value passed at the designated position.

If you want to return more than one result from one function, result will not work. One solution is to define arguments as a reference. Usually the program makes copies of the data, and the function continues to work with the copies (except if you are passing objects; objects will always be passed as reference to the object). The original values of simple data types remain unchanged in the calling method. If arguments are defined as reference, you can change the values in the calling method by the called function.

The definition is accomplished with

```
(byref name:data_type)
```

5.9.4 Predefined SimTalk Functions

SimTalk has a range of ready functions.

5.9.4.1 Functions for Manipulating Strings

Function	Description
copy(<string>,<integer1>,<integer2>);	The function "copy" copies a number of characters (integer2) from a string starting from the position integer1. The first character has the position 0.
incl(<string1>,<string2>,<integer>);	The function "incl" inserts a string2 into the string1 before the position integer. The new string is returned.
omit (<string>,<integer1>,<integer2>);	"omit" copies the string and deletes the substring within it from starting position integer1 with number (integer2) characters. The new string will be returned.

<code>pos(<string1>,<string2>)</code>	"pos" shows the position within string2, in which the string1 occurs in for the first time. If the string1 is not contained in string2, 0 is returned, otherwise the position as integer.
<code>strlen(<string>)</code>	"strlen" returns the length of the string passed.

Example 51: Functions for Manipulating Strings

The file extension of a filename is to be identified and re-turned. The dot will be searched for first. Then, starting from a position after the dot all characters until the end of the string will be copied. The result will be displayed in the console.

```

is
  filename, extension:string;
  length, posPoint:integer;
do
  filename:="samplefile.spp";
  -- search point
  posPoint:=pos(".",filename);
  -- find out the length of the text
  length:=strlen(filename);
  -- copy the substring
  extension:=copy(filename,posPoint+1,length-
posPoint);
  -- display in the console
  print extension;
end;

```

5.9.4.2 Mathematical Functions

Function	Description
<code>sqrt(x)</code>	Square root. The argument x has to be greater than or equal to 0.
<code>abs(x)</code>	Absolute amount of x.
<code>round(x)</code>	Rounds x to the nearest whole number (on or off) .
<code>round(x,y)</code>	Rounds x on y digits.
<code>floor(x)</code>	Nearest whole number less than or equal to x
<code>ceil(x)</code>	Nearest whole number greater than or equal to x

<code>min(x,y)</code>	Minimum
<code>max(x,y)</code>	Maximum
<code>pow(x,y)</code>	x^y
...	

5.9.5 Method Call

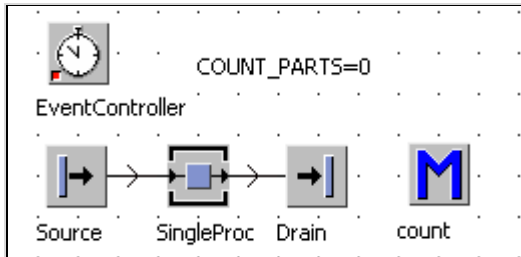
Methods are called by their names. During the simulation, methods have to be called often in connection with certain events.

5.9.5.1 Sensors

Most objects have sensors that are triggered when an MU enters the object and exits again. Length-oriented objects, such as the Line, have separate sensors for moving forward or in reverse.

Example 56: Methodcalls by Sensors

Parts manufactured by a machine are to be counted using a global variable. Create the following simple Frame:



“COUNT_PARTS” is a global variable of type integer. The method “count” should look like this:

```

-----
--| increases the global variable "COUNT_PARTS" with
--| each call by one
-----
-
is
do
    COUNT_PARTS:= COUNT_PARTS +1;
end;

```

The method should now be called if a part is exiting the machine. Open the dialog of the SingleProc and select the tab CONTROLS.



The main sensors are entrance and exit control. The setup control is triggered when a setup process starts and ends. You can activate the sensor with the front or the rear. The choice depends on the practical case. The rear exit control is triggered when the part has already left the object. For counting, this is the right choice. If you would trigger the control with the front, the MU will still be on the object. If the subsequent object is faulty or is still occupied, the front sensor is triggered twice (once when trying to transfer, the second time when transfer is done). If you select “front” in the exit control, then you need to trigger the transfer by SimTalk, even if a connector leads to the next object (e.g., @.move or @.move(successor)). If you press the F2 key in a field in which a method is registered, Plant Simulation will open the method editor and will load the relevant method.

5.9.5.2 Other Events for Calling Methods

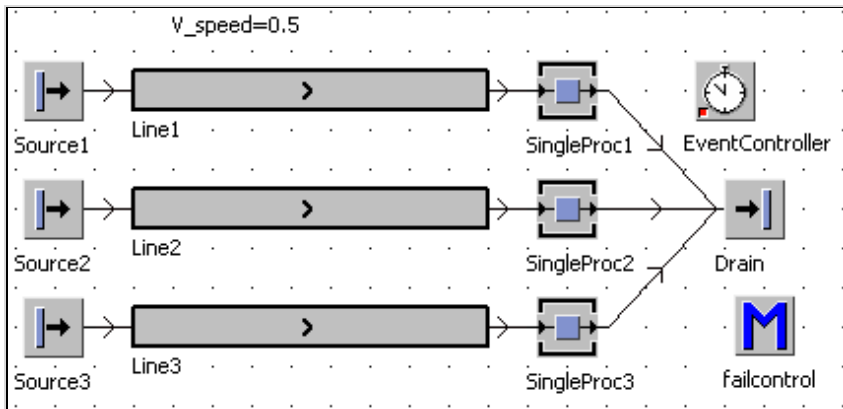
For calling methods, you can use other events. These events can be found in the object dialogs under the command **TOOLS – SELECT CONTROLS ...**



The failure control, for example, is called if a failure of the object begins and ends (when the value of the property failure changes).

Example 57: Fail Control

You are to simulate the following part of a production. On a multilane, non-accumulating line parts are transferred and transported by 3 meters. Processing on each lane by a separate machine follows. If a machine fails, the entire line must be stopped (all lanes have to be stopped). Create the following Frame:



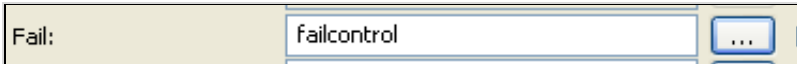
Settings: Source1 to Source3: interval 1 second, not blocking, Conveyors: 12 m, 0.5 m/s, SingleProc: each 1 second processing time, Failures: 90% availability, 10 minutes MTTR, Drain: 0 seconds processing time.

One way to stop all lanes by one machine failure could be: The method “failcontrol” checks whether a machine is failed. If at least one machine is failed, the speed of all three lines is set to 0. If no machine is failed, the speed is set to the value that is stored in the global variable “v_speed”.

Method failcontrol:

```
is
do
  if SingleProc1.failed or
     SingleProc2.failed or
     SingleProc3.failed then
    Line1.speed:=0;
    Line2.speed:=0;
    Line3.speed:=0;
  else
    Line1.speed:=speed;
    Line2.speed:=speed;
    Line3.speed:=speed;
  end;
end;
```

*Now the method “failcontrol” must be called whenever a failure occurs, or when the failure ends. Click in the dialog of the SingleProc objects **TOOLS – SELECT CONTROLS**: Enter the method “failcontrol” as the failure control.*



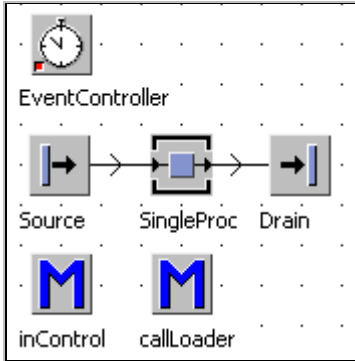
*Repeat this for all three machines. The result can very well be proven statistically. The individual machines can in theory work 90% of the time. If the failures of the other machines lead to no more parts being transported by the conveyor, then the machines remain 20% of their processing time without parts in addition to their own failures. That means the machines can on average operate only 70% of the time. Run the simulation for a while (for at least 2 days). Then click the tab **STATISTICS** in SingleProc1, 2, or 3.*

5.9.5.3 Method Call After a Certain Timeout

You can call SimTalk methods after a certain timeout. This can be useful if you need to trigger calls after an interval referred to an event.

Example 58: Ref-Call

In this simulation, the machine should send a status message (ready) 10 seconds before completion of a part (e.g., to inform a loading device). The message should be shown first as output in the console. Create the following simple frame:



Settings: Source interval 2 minutes, SingleProc 2 minutes processing time, inControl as entrance control SingleProc. The method callLoader writes a short message into the console. Method callLoader:

```
is
do
    print time_to_str(eventController.simTime) +
        " called loader";
end;
```

The call of the method callLoader should now take place 10 seconds before completion of the part on the SingleProc. If you use the entrance control to start the timer, the method should be started after the processing time of SingleProc minus 10 seconds. For calling a method after a certain time period, you may use the method `<ref>.MethCall(<time>,[<parameter>])` in Plant Simulation. You cannot address the method with a path or directly with name, because the method is then called directly. Instead, use the method `ref (<path>)`. This returns a reference to the method by which you get access to the method. In the example above, the method InControl should look like this:

```
is
do
    ref(callLoader).methcall(
        SingleProc.procTime-10);
end;
```