Functions (methods)

In Java to call a *function* (or *method*, which is more correct in object-oriented languages like Java, as any function is a method of a class) you write the function name followed by the argument values in parentheses. For example, this is a call of a triangular probability distribution function with three numeric arguments:

```
triangular( 2, 5, 14 )
```

And the next function call prints the coordinates of an agent to the model log with a timestamp:

```
traceln( time() + ": X = " + getX() + " Y = " + getY() );
```

The argument of this function call is a string expression with five components; three of them are also function calls: `time()`, `getX()`, and `getY()`.

Even if a function has no arguments, you must put parentheses after the function name, like this: `time()`

A function may or may not return a value. For example, the call of `time()` returns the current model time of type `double`, and the call of `traceln()` does not return a value. If a function returns a value it can be used in an expression (like `time()` was used in the argument expression of `traceln()`). If a function does not return a value it can only be called as a statement (the semicolon after the call of `traceln()` indicates that this is a statement).

Standard and system functions

Most of the code you write in AnyLogic is the code of a subclass of `Agent` (a fundamental class of AnyLogic). For your convenience, AnyLogic system functions and most frequently used Java standard functions are available there directly (you do not need to think which package or class they belong to, and can call those functions without any prefixes). Below are some examples (these are only a few functions out of several hundreds, see AnyLogic API Reference for the full list).

The type name written before the function name indicates the type of the returned value. If the function does not return a value, we write void instead of type, but we are dropping it here.

Mathematical functions (imported from `java.lang.Math`, about 45 functions in total):

- `double min( a, b )` – returns the minimum of `a` and `b`
- `double log( a )` – returns the natural logarithm of `a`
- `double pow( a, b )` – returns the value of `a` raised to the power of `b`
- `double sqrt( a )` – returns the square root of `a`

Functions related to the model time, date, or date elements (about 20 functions):

- `double time()` – returns the current model time (in model time units)
- `Date date()` – returns the current model date (`Date` is standard Java class)
- `int getMinute()` – returns the minute within the hour of the current model date
- `double minute()` – returns one minute time interval value in the model time units

Probability distributions (over 30 distributions are supported):

- `double uniform( min, max )` – returns a uniformly distributed random number

- `double exponential( rate )` – returns an exponentially distributed random number

Output to the model log and formatting:

- `traceln( Object o )` – prints a string representation of an object with a line delimiter at the end to the model log
- `String format( value )` – formats a value into a well-formed string

Model execution control:

- `boolean finishSimulation()` – causes the simulation engine to terminate the model execution after completing the current event.
- `boolean pauseSimulation()` – puts the simulation engine into a "paused" state.
- `error( String msg )` – signals an error. Terminates the model execution with a given message.

Navigation in the model structure and the execution environment:

- `Agent getOwner()` – returns the upper level agent where this agent lives, if any
- `int getIndex()` – returns the index of this agent in the list if it is a replicated agent population
- `Experiment<?> getExperiment()` – returns the experiment controlling the model execution
- `Presentation getPresentation()` – returns the model GUI
- `Engine getEngine()` – returns the simulation engine

Agents have more functions available specific to the particular type of agent, for example:

- `inState( Truck.Moving )` – tests if the agent (of type `Truck`) is currently in the `Moving` state of its statechart.

Network and communication-related functions:

- `connectTo( agent )` – establishes a connection with another agent
- `send( msg, agent )` – sends a message to given agent

Space and movement-related functions:

- `double getX()` – returns the X-coordinate of the agent in continuous space
- `moveTo( x, y, z )` – starts movement of the agent to the point (x,y,z) in 3D space

The functions available in the current context, for example, in a property field where you are entering some code, are always listed in the code completion window that opens if you press Ctrl+Space (Mac OS: Alt+Space), please refer here for more details.

Functions of the model elements

All elements in AnyLogic model (events, statecharts, table functions, plots, graphics shapes, controls, library objects, and so on) are mapped to Java objects and expose Java API (Application Programming Interface) to the user. You can retrieve information about the objects and control them using their API.

To call a function of a particular model element that is inside the agent you should put the element name followed by dot "." before the function call: *<object>.<method call>*

These are some examples of calling functions of the elements of the current agent (the full list of functions for a particular element is available in AnyLogic Help):

Scheduling and resetting events:

- `event.restart( 15*minute() )` – schedules the `event` to occur in 15 minutes
- `event.reset()` – resets a (possibly scheduled) `event`

Sending messages to statecharts and obtaining their current states:

- `statechart.receiveMessage( "Go!" )` – delivers the message "Go!" to the `statechart`

Adding a sample data point to a histogram:

- `histData.add( x )` – adds the value of `x` to the histogram data object `histData`

Display a view area:

- `viewArea.navigateTo()` – displays the part of the canvas marked by the viewArea

Changing the color of a shape:

- `rectangle.setFillColor( red )` – sets the fill color of the `rectangle` shape to red

Retrieving the current value of a checkbox:

- `boolean checkbox.isSelected()` – returns the current state of the `checkbox`

This statement hides or shows the shape depending on the state of the checkbox:

- `rectangle.setVisible( checkbox.isSelected() );`

Changing parameters and states of embedded agents:

- `source.set_rate( 100 )` – sets the `rate` parameter of `source` object to 100
- `hold.setBlocked( true )` – puts the `block` object to the blocked state

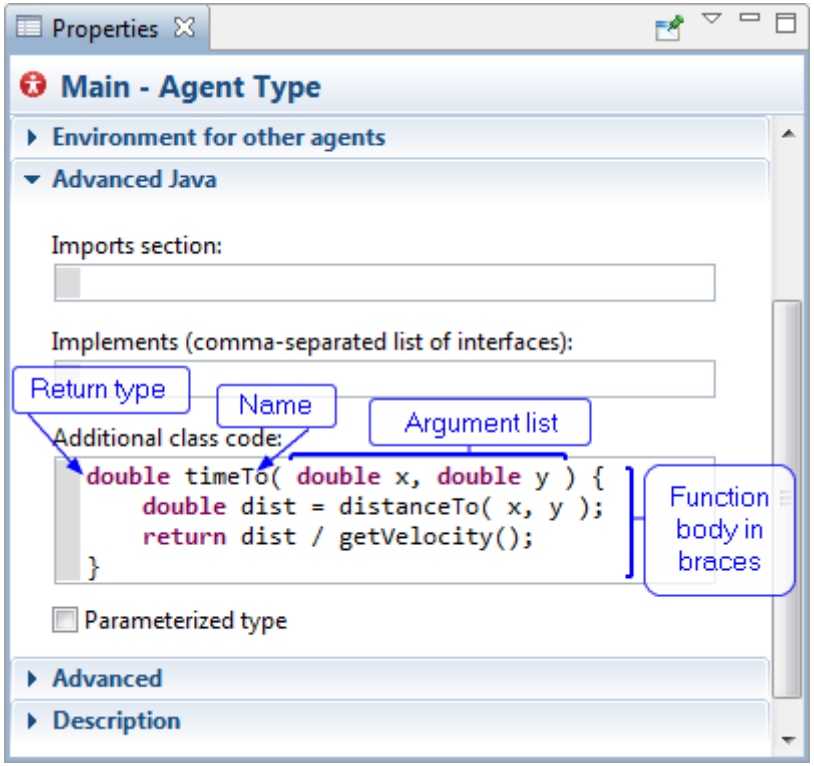
Note that the parameter rate appears as **Arrival rate** in the source object properties. To find out the Java names of the parameters you should hover the mouse pointer on the properties.

You can easily find out which functions are offered by a particular object is by using code completion. In the code you are writing you should type the object name, then dot ".", and then press Ctrl+Space (Alt+Space on Mac). The pop-up window will display the list of functions you can call.

Defining your own function

You can define your own functions of agents, experiments, custom Java classes. For agents and experiments functions can be defined as objects in the graphical editor.

Another way of defining a function is writing its full Java declaration and implementation in **Additional class code** field in the **Advanced** property section of the agent type or experiment, see the figure below. The two definitions of the functions are absolutely equivalent, but having the function icon on the canvas is preferred because it is more visual and provides quicker access to the function code in design time.



Function defined in the Additional class code of the agent