Primitive data types

There are eight *primitive data types* in Java, but in AnyLogic models we typically use these four:

Type name	Represents	Examples of constants
int	Integer numbers	12 10000 -15 0
double	Real numbers	877.13 12.0 12. 0.153 .153 -11.7 3.6e-5
boolean	Boolean values	true false
String	Text strings	"AnyLogic" "X = " "Line\nNew line" ""

The word "*double*" means real value with double precision. In AnyLogic engine all real values (such as time, coordinates, length, speed, random numbers) have double precision. The type `String` is actually a class (a non-primitive type, notice that its name starts with a capital letter), but it is a fundamental class, so some operations with strings are built into the core of Java language.

Consider the *numeric constants*. Depending on the way you write a number, Java will treat it either as real or as integer. Any number with the decimal delimiter "." is treated as a real number, even if its fractional part is missing or contains only zeros (this is important for integer division). If integer or fractional part is zero, it can be skipped, so ".153" is the same as "0.153", and "12." is the same as "12.0".

Boolean constants in Java are `true` and `false` and, unlike in languages like C or C++, they are not interchangeable with numbers, so you cannot treat false as 0, or non-zero number as `true`.

String constants are sequences of characters enclosed between the quote marks. The empty string (the string containing no characters) is denoted as `""`. Special characters are included in string constants with the help of escape sequences that start with the backslash. For example the end of line is denoted by `\n`, so the string `"Line one\nLine two"` will appear as:

Line one

Line two

If you wish to include the quote character into a string, you need to write `\"`, for example the string constant `"String with \" in the middle"` will print as:

String with " in the middle.

The backslash itself is included as double backslash: `"This is a backslash: \\"`.

Classes

Structures more complex than primitive types are defined with the help of classes and in object-oriented manner. The mission of explaining the concepts of object-oriented design is impossible in the format of a small section: the subject deserves a separate book, and there are a lot of them already written. All we can do here is give you a feeling of what object-orientedness is about, introduce such fundamental terms as class, method, object, instance, subclass, and inheritance, and show how Java supports object-oriented design. You should not try to learn or fully understand the code fragments in this section. It will be sufficient if, having read this section, you will know that, for example, a statechart in your model is an instance of AnyLogic class `Statechart` and you can find out its current state by calling its method: `statechart.getActiveSimpleState()`.

Class as grouping of data and methods. Objects as instances of class

Consider an example. Suppose you are working with local map and use coordinates of locations and calculate distances. Of course you can remember two double values `x` and `y` for each location and write a function `distance( x1, y1, x2, y2 )` that would calculate distances between two pairs of coordinates. But it is a lot more elegant to introduce a new entity that would group together the coordinates and the method of calculating distance to another location. In object-oriented programming such entity is called a *class*. Java class definition for the location on a local map can look like this:

```java
class Location {

    //constructor: creates a Location object with given coordinates
    Location( double xcoord, double ycoord ) {
        x = xcoord;
        y = ycoord;
    }

    //two fields of type double
    double x; //x coordinate of the location
    double y; //y coordinate of the location

    //method (function): calculates distance from this location to another one
    double distanceTo( Location other ) {
        double dx = other.x - x;
        double dy = other.y - y;
        return sqrt( dx*dx + dy*dy );
    }

}
```

As you can see, a class combines data and methods that work with the data.

Having defined such class, we can write very simple and readable code when working with the map, like this:

```java
Location origin = new Location( 0, 0 ); //create first location

Location destination = new Location( 250, 470 ); //create second location

double distance = origin.distanceTo( destination ); //calculate distance
```

The locations `origin` and `destination` are *objects* and are *instances* of the class `Location`. The expression `new Location( 250, 470 )` is a *constructor call*, it creates and returns a new instance of the class `Location` with the given coordinates. The expression `origin.distanceTo( destination )` is a *method call* – it asks the object `origin` to calculate the distance to another object `destination`.

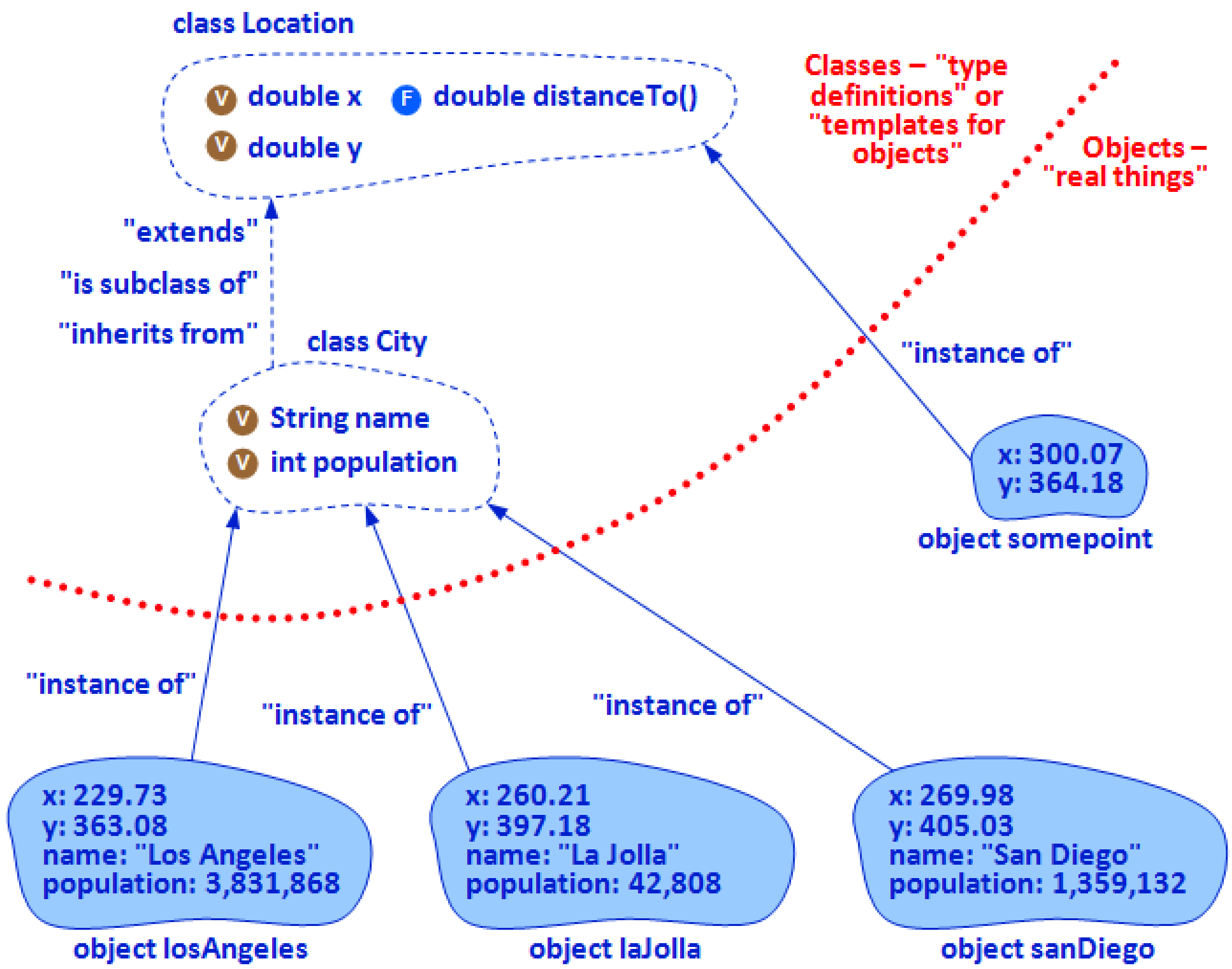
If you declare a variable of a non-primitive type (of a class) and do not initialize it, its value will be set to `null` (`null` is a special Java literal that denotes "*nothing*"). Sometimes you explicitly assign `null` to a variable to "forget" the object it referred to and to indicate that the object is missing or unavailable.

```
     Location target; //a variable is declared without initialization. target equals null
     target = warehouse; //assign the object (pointed to by the variable) warehouse to target
     //now target and warehouse point to the same object
     …
     target = null; //target forgets about the warehouse and equals null again
```

Inheritance. Subclass and super class

Now suppose some locations in your 2D space correspond to cities. A city has a name and population size. To efficiently manipulate cities we can extend the class `Location` by adding two more fields: `name` and `population`. We will call the new agent `City`. In Java this will look like:

```
class City extends Location { //declaration of the class CIty that extends the class Location

    //constructor of class City
    City( String n, double x, double y ) {
        super( x, y ); //call of constructor of the super class with parameters x and y
        name = n;
        //the population field is not initialized in the constructor – to be set later
    }

    //fields of class City
    String name;
    int population;

}
```



Classes and inheritance

`City` is called a *subclass* of `Location`, and `Location` is correspondingly a *super class* (or *base class*) of `City`. `City` *inherits* all properties of `Location` and adds new ones. See how elegant is the code that finds the biggest city in the range of 100 kilometers from a given `point` of class `Location` (we assume that there is a collection `cities` where all cities are included):

```
int pop = 0; //here we will remember the largest population found so far
City biggestcity = null; //here we will store the best city found so far

for( City city : cities ) { //for each city in the collection cities
    if( point.distanceTo( city ) < 100 && city.population > pop ) { //if best so far
        biggestcity = city; //remember it
        pop = city.population; //and remember its population size
    }
}
traceln( "The biggest city within 100km is " + city.name ); //print the search result
```

Notice that although `city` is an object of class `City`, which is "bigger" than `Location`, it still can be treated as Location when needed, in particular when calling the method `distanceTo()` of class `Location`.

This is a general rule: an object of a subclass can always be considered as an object of its base class.

How about vice versa? You can declare a variable of class `Location` and assign it an object of class `City` (a subclass of `Location`):

```
Location place = laJolla;
```

This will work. However, if you will then try to access the population of place, Java will signal an error:

```
int p = place.population; //error: "population cannot be resolved or is not a field"
```

This happens because Java does not know that `place` is in fact an object of class `City`. To handle such situations you can:

- test whether an object of a certain class is actually an object of its particular subclass by using the operator `instanceof`: *<object>* `instanceof` *<class>*
- "*cast*" the object from a super class to a subclass by writing the name of the subclass in parenthesis before the object: (*<class>*)*<object>*

A typical code pattern is:

```
if( place instanceof City ) {
    City city = (City)place;
    int p = city.population;
    …
}
```

Classes and objects in AnyLogic models

Virtually all objects that you use to create your models are instances of AnyLogic Java classes. Please refer to AnyLogic class reference to find the Java class names for most model elements you work with. Whenever you will need to find out what can you do with a particular object using Java, you should find the corresponding Java class in *AnyLogic API Reference* and look through its methods and fields.

Variables (local variables and class fields)

In this section we are considering only plain Java variables.

Depending on where a variable is declared it can be either a:

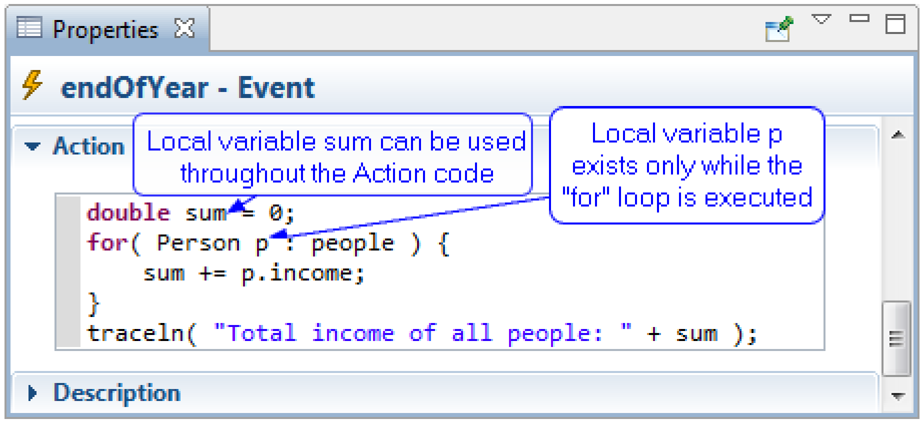
- *Local variable* – an auxiliary temporary variable that exists only while a particular function or a block of statements is executed, or
- *Class variable* (or *class field* – more correct term in Java) – a variable that is present in any object of a class, and whose lifetime is the same as the object lifetime.

Local (temporary) variables

Local variables are declared in sections of Java code such as a block, a loop statement, or a function body. They are created and initialized when the code section execution starts and disappear when it finishes. The declaration consists of the variable type, name, and optional initialization. The declaration is a statement, so it should end with a semicolon. For example:

```
double sum = 0; //double variable sum initially 0

int k; //integer variable k, not initialized

String msg = ok ? "OK" : "Not OK"; //string variable msg initialized with expression
```

Local variables can be declared and used in AnyLogic fields where you enter actions (sequences of statements), such as **On startup** code of agent type, **Action** field of events or transitions, **Entry action** and **Exit action** of state, **On enter** and **On exit** fields of flowchart objects. The variables `sum` and `p` are declared in the action code of the event `endOfYear` in the Figure and exist only while this portion of code is being executed.


Properties
endOfYear - Event
Action
Local variable sum can be used throughout the Action code
Local variable p exists only while the "for" loop is executed
Description

```
double sum = 0;
for( Person p : people ) {
    sum += p.income;
}
traceln( "Total income of all people: " + sum );
```

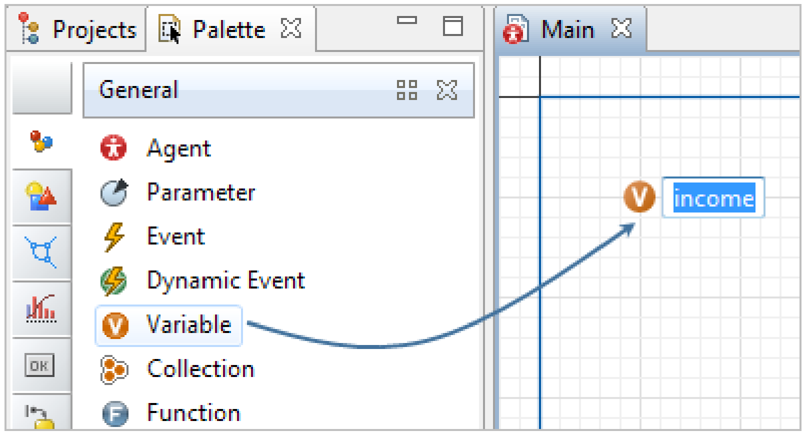
Local variables declared in the Action code of an event

Class variables (fields)

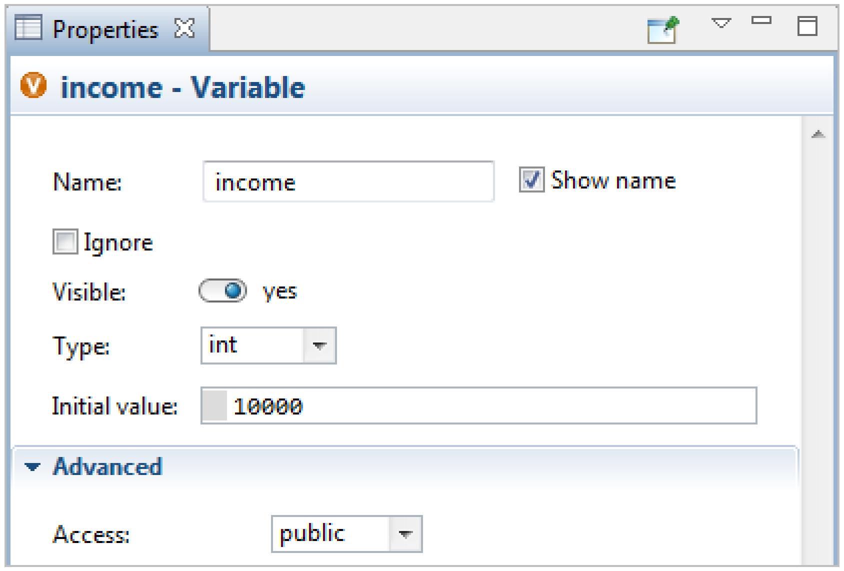
Java variables (fields) of agent class are part of "memory" or "state" of agents. They can be declared graphically or in code.

To declare a variable of agent (or experiment) class

1. Open the **Agent** palette and drag the **Variable** element from the palette to the canvas.
2. Type the variable name in the in-place editor or in the variable properties.



3. Choose the variable **Access** type in the variable properties. In most cases you can leave **default**, which means the variable will be visible within the current model. **Public** opens the access to the variable from other models, and **private** limits the access to this agent only.
4. Choose the variable type. If the type is not one of the primitive types, you should choose **Other** and enter the type name in the field nearby.
5. Optionally you can enter the variable **Initial value**.
6. If you do not specify the initial value, it will be `false` for `boolean` type, 0 for numeric variables, and `null` ("nothing") for all other classes including `String`.


Properties

income - Variable

Name: income ☑ Show name

☐ Ignore

Visible: yes

Type: int

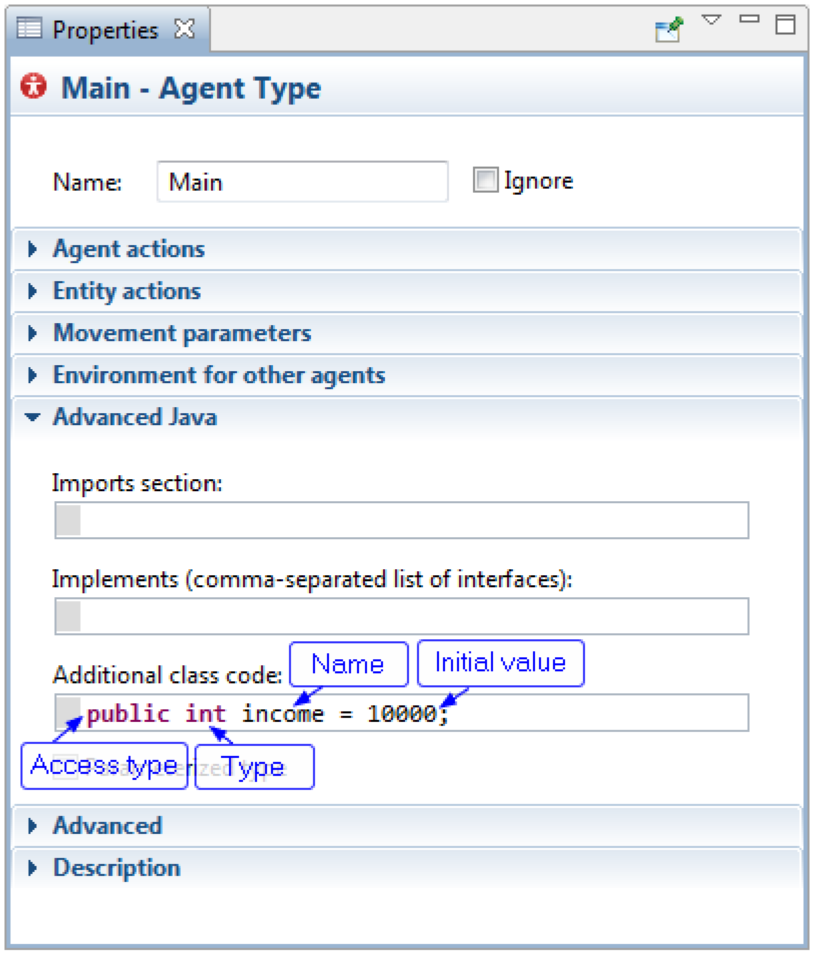
Initial value: 10000

Advanced

Access: public

A variable of an agent declared in the graphical editor

In the Figure a variable `income` of type `int` is declared in an agent (or experiment) type. Its access type is public, therefore it will be accessible from anywhere. The initial value of the variable is 10000. The graphical declaration above is equivalent to a line of code of the class, which you can write in **Additional class code** field in the **Advanced Java** property section of the agent type, see the Figure:



```
public int income = 10000;
```

The same variable declared in the Additional class code field

Graphical declaration of a variable allows you to visually group it together with related functions or objects, and also to view or change the variable value at runtime with one click.

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

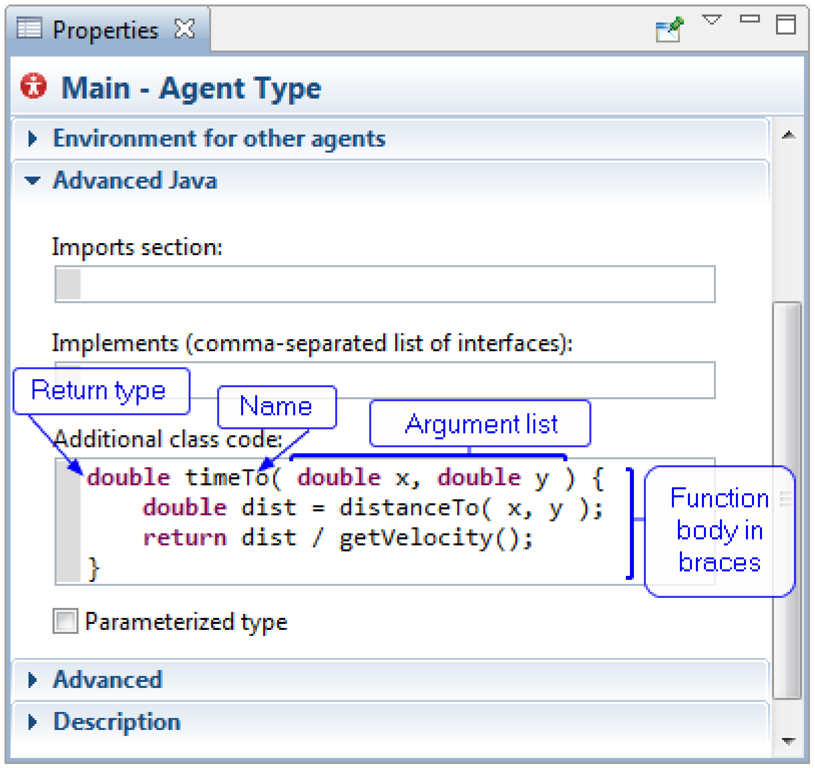
Note that the parameter rate appears as **Arrival rate** in the source object properties. To find out the Java names of the parameters you should hover the mouse pointer on the properties.

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.


Properties

Main - Agent Type

Environment for other agents

Advanced Java

Imports section:

Implements (comma-separated list of interfaces):

Additional class code:

```
double timeTo( double x, double y ) {
    double dist = distanceTo( x, y );
    return dist / getVelocity();
}
```

Parameterized type

Advanced

Description

Function defined in the Additional class code of the agent

Arithmetic expressions

Arithmetic expressions in Java are composed with the usual operators +, –, *, / and the remainder operator %. Multiplication and division operations have higher priority than addition and subtraction. Operations with equal priority are performed from left to right. Parenthesis are used to control the order of operation execution.

```
a + b / c ≡ a + ( b / c )

a * b – c ≡ ( a * b ) – c

a / b / c ≡ ( a / b ) / c
```

It is recommended to always use parenthesis to explicitly define the order of operations, so you do not have to remember which operation has higher priority.

A thing worth mentioning is *integer division*.

The result of division in Java depends on the types of the operands. If both operands are integer, the result will be integer as well. The unintended use of integer division may therefore lead to significant precision loss. To let Java perform real division (and get a real number as a result) at least one of the operands must be of real type.

For example:

```
3 / 2 ≡ 1

2 / 3 ≡ 0
```

because this is integer division. However,

```
3 / 2. ≡ 1.5

2.0 / 3 ≡ 0.66666666…
```

because 2. and 2.0 are real numbers. If `k` and `n` are variables of type `int`, `k/n` is integer division. To perform a real division over two integer variables or expressions you should force Java to treat at least one of them as real. This is done by *type casting*: you need to write the name of the type you are converting to before the variable in parentheses, for example `(double)k/n` will be real division with result of type `double`.

Integer division is frequently used together with the *remainder operation* to obtain the row and column of the item from its sequential index. Suppose you have a collection of 600 items, say, seats in a theater, and want to arrange them in 20 rows, each row containing 30 seats. The expressions for the seat number in a row and the row would be:

Seat number: `index % 30` (remainder of division of index by 30: 0 - 29)

Row number: `index / 30` (integer division of index by 30: 0 - 19)

where index is between 0 and 599. For example, the seat with index 247 will be in the row 8 with seat number 7.

The power operation in Java does not have an operand (if you write `a^b` this will mean bitwise OR and not power). To perform the power operation you need to call the `pow()` function:

`pow( a, b ) ≡` a^b

Java supports several useful shortcuts for frequent arithmetic operations over numeric variables. They are:

`i++ ≡  i = i+1` (increment i by 1)

`i-- ≡  i = i-1` (decrement i by 1)

`a += 100.0  ≡ a = a + 100.0` (increase a by 100.0)

`b -= 14 ≡ b = b - 14` (decrease b by 14)

Note that, although these shortcuts can be used in expressions, their evaluation has effect, it changes the value of the operands.

Conditional operator ? :

Conditional operator is helpful when you need to use one of the two different values in an expression depending on a condition. It is a ternary operator, i.e. it has three operands:

```
<condition> ? <value if true> : <value if false>
```

It can be applied to values of any type: numeric, boolean, strings, any classes. The following expression evaluates to 0 if the backlog contains no orders and the amount of the first order in the backlog queue otherwise:

```
backlog.isEmpty() ? 0 : backlog.getFirst().amount
```

Conditional operators can be nested. For example, the following code line prints the level of income of a person (High, Medium, or Low) depending on the value of the variable `income`:

```
traceln( "Income: " + ( income > 10000 ? "High" : ( income < 1500 ? "Low" : "Medium" ) ) );
```

This single code line is equivalent to the following combination of "if" statements:

```
trace( "Income: " );
if( income > 10000 ) {
    traceln( "High" );
} else if( income < 1500 ) {
    traceln( "Low" );
} else {
    traceln( "Medium" );
}
```

Logical expressions

There are three logical operators in Java that can be applied to `boolean` expressions:

`&&`	logical AND
`||`	logical OR
`!`	logical NOT (unary operator)

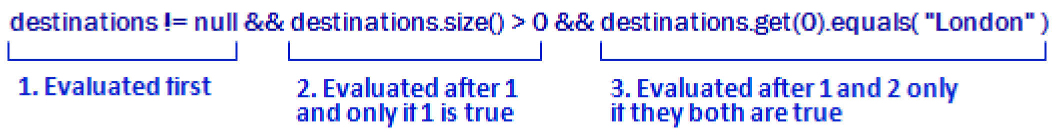
AND has higher priority than OR, so that

```
a || b && c ≡ a || ( b && c )
```

but again, it is always better to put parentheses to explicitly define the order of operations.

The logical operations in Java exhibit so-called *short-circuiting behavior*, which means that the second operand is evaluated only if needed.

This feature is very useful when a part of an expression cannot be evaluated (will cause error) if another part is not true. For example, let `destinations` be a collection of places an agent in the model must visit. To test if the first place to visit is London, you can write:


```
destinations != null && destinations.size() > 0 && destinations.get(0).equals( "London" )
```
1. Evaluated first
2. Evaluated after 1 and only if 1 is true
3. Evaluated after 1 and 2 only if they both are true

Short-circuiting behavior of logical operations

Here we first test if the list of destinations exists at all (does not equal `null`), then, if it exists, we test if it has at least one element (its size is greater than 0), and if yes, we compare that element with the string `"London"`.

Relations and equality

Relations between two numeric expressions are determined using the following operators:

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

You can test if two operands (primitive or objects) are equal using the two operators:

`==`	equal to
`!=`	not equal to

For non-primitive objects (i.e. for those that are not numeric or `boolean`) the operators "`==`" and "`!=`" test if the two operands are the same object rather than *two objects with equal contents*. To compare the contents of two objects, e.g. two strings, you should use the function `equals()`.

For example, to test if the string message msg equals "Wake up!" you should write:

```
msg.equals( "Wake up!" )
```

Do not confuse the equality operator "==" with the assignment operator "="!

`a = 5` means assign value of 5 to the variable `a`, whereas

`a == 5` is `true` if a equals 5 and `false` otherwise

The result of all comparison operations is of `boolean` type (`true` or `false`).

String expressions

Strings in Java can be concatenated by using the `+` operator. For example:

`"Any" + "Logic"` results in `"AnyLogic"`

Moreover, this way you can compose strings from objects of different types. The non-string objects will be converted to strings and then all strings will be concatenated into one. This is widely used in AnyLogic models to display the textual information on variable values. For example, in the dynamic property field **Text** of the Text object you can write:

```
"x  = " + x
```

And then at runtime the text object will display the current value of x in the textual form, e.g.:

x = 14.387

You can use an empty string in such expressions as well: `"" + x` will simply convert x to string. Another example: the following string expression:

```
"Number of destinations: " + destinations.size() + "; the first one is " + destinations.get(0)
```

results in a string like this:

Number of destinations: 18; the first one is London

Please use the function `equals()` and not the `==` operator to compare strings!

Statements

Actions associated with events, transitions, process flowchart objects, agents, controls, etc. in AnyLogic are written in Java code. Java code is composed of statements. *Statement* is a unit of code, an instruction provided to a computer. Statements are executed sequentially one after another and, generally, in top-down order. The following code of three statements, for example, declares a variable `x`, assigns it a random number drawn from a uniform distribution, and prints the value to the model log.

```
double x;

x = uniform();

traceln( "x = " + x );
```

You may have noticed that there is a semicolon at the end of each statement. In Java this is a rule:

Each statement in Java must end with semicolon except for the block {…}.

We will consider these types of Java statements:

- Variable declaration, e.g.: `String s;` or `double x = getX();`
- Function call: `traceln( "Time:" + time() );`
- Assignment: `shipTo = client.address;`
- Decisions: `if( … ) then { … } else { … }`, `switch( … ) { case … : case …; … }`
- Loops: `for() { … }`, `while( … ) { … }`, `do { … } while( … )` and also `break;` and `continue;`
- Block: `{ … }`

It is important that, regardless of its computational complexity and the required CPU time, any piece of Java code written by the user is treated by AnyLogic simulation engine as indivisible and is executed logically instantly: the model clock is not advanced.

Variable declaration

Variable declarations are considered in the section Variables. There are two syntax forms:

```
<type> <variable name> ;

<type> <variable name>  = <initial value>;
```

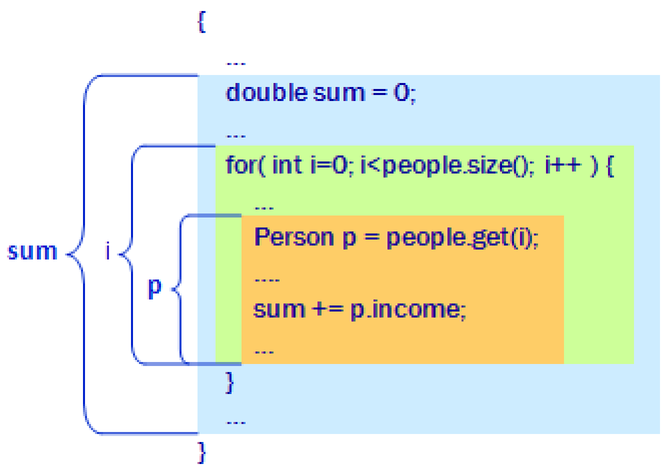
Examples:

```
double x;

Person customer = null;

ArrayList<Person> colleagues = new ArrayList<Person>();
```

Please keep in mind that:

- A local variable must be declared before it is first used.
- A local variable declared within a block `{…}` or a function body exists only while this block is being executed.
- If the name of a variable declared in a block or function body is the same as the name of the variable declared in an enclosing (higher level) block or of the current class variable, the lower level local variable is meant by default when the name is used. We however strongly recommend to avoid duplicate names.

The Figure below illustrates the code areas where local variables exist and can be used.



```
{
  ...
  double sum = 0;
  ...
  for( int i=0; i<people.size(); i++ ) {
    ...
    Person p = people.get(i);
    ....
    sum += p.income;
    ...
  }
  ...
}
```

Code areas where local variables exist and can be used

Assignment

To assign a value to a variable in Java you write:

```
<variable name> = <expression>;
```

Remember that "=" means assignment action, whereas "==" means equality relation.

Examples:

```
distance = sqrt( dx*dx + dy*dy ); //calculate distance based on dx and dy

k = uniform_discr( 0, 10 ); //set k to a random integer between 0 and 10 inclusive

customer = null; //"forget" customer: reset customer variable to null ("nothing")
```

The shortcuts for frequently used assignments can be executed as statements as well (see Arithmetic expressions), for example:

```
k++; //increment k by 1

b *= 2; //b = b*2
```

if-then-else

As in any language that supports imperative programming, Java has *control flow statements* that "break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code" [The Java Tutorials, Oracle].

The "if-then-else" statement is the most basic control flow statement that executes one section of code if a condition evaluates to true, and another – if it evaluates to false. The statement has two forms, short:

```
if( <condition> )
    <statement if true>
```

and full:

```
if( <condition> )
    <statement if true>
else
    <statement if false>
```

For example, this code assigns a client to a salesman only if the salesmen is currently following up less than 10 clients.

```
if( salesman.clients.size() < 10 )
    salesman.assign( client );
```

And this code tests if there are tasks in a certain queue and, if yes, assigns the first one to a truck, otherwise sends the truck to the parking position:

```
if( tasks.isEmpty() )

    truck.setDestination( truck.parking );

else

    truck.assignTask( tasks.removeFirst() );
```

In case "then" or "else" code section contains more than one statement, they should be enclosed in braces `{ … }` to become a block, which is treated as a single statement, see the code below. We however recommend to always use braces for "then" and "else" sections to avoid ambiguous-looking code. Braces are specifically important when there are nested "if" statements or when lines of code in the "if" neighborhood are added or deleted during editing or debugging.

```
if( friends == null ) {

    friends = new ArrayList< Person >();

    friends.add( john );

} else {

    if( ! friends.contains( john ) )

        friends.add( john );
}
```

switch

The `switch` statement allows you to choose between arbitrary number of code sections to be executed depending on the value of an integer expression. The general syntax is:

```
switch( <integer expression> ) {
case <integer constant 1>:
    <statements to be executed in case 1>
    break;
case <integer constant 2>:
    <statements to be executed in case 2>
    break;
…
default:
    <statements to be executed if no cases apply>
    break;
}
```

The `break` statements at the end of each `case` section tell Java that the `switch` statement execution is finished. If you do not put `break`, Java will continue executing the next section, no matter that it is marked as a different case. The `default` section is executed when the integer expression does not match any of the cases. It is optional.

The integer values that correspond to different cases of the switch are usually pre-defined as integer constants. Imagine you are developing a model of an overhead bridge crane where the crane is an agent controlled by a set of commands. The response of the crane to the commands can be programmed in the form of a `switch` statement:

```
switch( command ) {
case MOVE_RIGHT:
    speed = 10;
    break;
case MOVE_LEFT:
    speed = -10;
    break;
case STOP:
    speed = 0;
    break;
case RAISE:
    …
    break;
case LOWER:
    …
    break;
default:
    error( "Invalid command: " + command );
}
```

For loop

There are two forms of `for` loop and two forms of `while` loop in Java. We will begin with the easiest to use form of the `for` loop, the so-called "enhanced" `for` loop:

```
for( <element type> <name> : <collection> ) {
    <statements> //the loop body executed for each element
}
```

This form is used to iterate through arrays and collections. Whenever you need to do something with each element of a collection, we recommend to use this loop because it is more compact and easy to read, and is supported by all collection types (unlike index-based iteration).

Example: in an agent-based model, a firm's product portfolio is modeled as a replicated `products` object (remember that a replicated object is a collection). The following code goes through all products in the portfolio and kills those, whose estimated ROI is less than some allowed minimum:

```
for( Product p : products ) {
    if( p.getEstimatedROI() < minROI )
        p.kill();
}
```

Another example: the loop counts the number of sold seats in a movie theater. The seats are modeled as the `seats` Java array with the elements of the `boolean` type (`true` means sold):

```
boolean[] seats = new boolean[600]; //array declaration
…
int nsold = 0;
for( boolean sold : seats )
    if( sold )
        nsold++;
```

Note that if the body of the loop contains only one statement, the braces `{…}` can be dropped. In the code above, braces are dropped both in the `for` and the `if` statements.

Another more general form of the `for` loop is typically used for index-based iterations. In the header of the loop you can put the initialization code, for example, the declaration of the index variable, the condition that is tested before each iteration to determine whether the loop should continue, and the code to be executed after each iteration that can be used, say, to increment the index variable:

```
for( <initialization>; <continue condition>; <increment> ) {
    <statements>
}
```

The following loop finds all circles in a group of shapes and sets their fill color to red:

```
for( int i=0; i<group.size(); i++ ) { //index-based loop
    Object obj = group.get( i ); //get the i-th element of the group
    if( obj instanceof ShapeOval ) { //test if it is a ShapeOval - AnyLogic class for ovals
        ShapeOval ov = (ShapeOval)obj; //if it is oval, "cast" it to ShapeOval
```

```
            ov.setFillColor( red ); //set the fill color to red
        }
    }
```

As long as there is no other way to iterate through the `ShapeGroup` contents than accessing the shapes by index, only this form of loop is applicable here.

Many Process Modeling Library objects also offer index-based iterations. For example, this code goes through all agents in the queue from the end to the beginning and removes the first one that does not possess any resource units:

```
    for( int i=queue.size()-1; i>=0; i-- ) { //the index goes from queue.size()-1 down to 0
        Agent a = queue.get(i); //obtain the i-th agent
        if( a.resourceUnits().isEmpty() ) { //test
            queue.remove( a ); //remove the agent from the queue
            break; //exit the loop
        }
    }
```

Note that in this loop the index is decremented after each iteration, and correspondingly the continue condition tests if it has reached `0`. Once we have found the agent that satisfies our condition, we remove it and do not need to continue. The `break` statement is used to exit the loop immediately. If the agent is not found, the loop will finish in its natural way when the index after a certain iteration becomes `-1`.

While loop

"While" loops are used to repeatedly execute some code while a certain condition evaluates to `true`. The most commonly used form of this loop is:

```
while( <continue condition> ) {
    <statements>
}
```

The code fragment below tests if a `shape` is contained in a given `group` either directly or in any of its subgroups. The function `getGroup()` of `Shape` class returns the group, which is the immediate container of the shape. For a top-level shape (a shape that is not contained in any group) it returns `null`. In this loop we start with the immediate container of the `shape` and move one level up in each iteration until we either find the `group` or reach the top level:

```
ShapeGroup container = shape.getGroup(); //start with the immediate container of shape
while( container != null ) { //if container exists
    if( container == group ) //test if it equals the group we are looking for
        return true; //if yes, the shape is contained in the group
    container = container.getGroup(); //otherwise move one level up
}
return false; //if we are here, the shape is definitely not contained in the group
```

The condition in the first form of "while" loop is tested before each iteration; if it initially is false, nothing will be executed. There is also a second form of while loop – `do…while`:

```
do {
    <statements>
} while( <continue condition> );
```

The difference between `do…while` loop and `while` loop is that `do…while` evaluates its condition *after* the iteration, therefore the loop body is executed at least once. This makes sense, for example, if the condition depends on the variables prepared during the iteration.

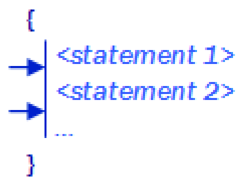
Consider the following example. The local area map is a square 1000 by 1000 pixels. The city bounds on the map are marked with a closed polyline `citybounds`. We need to find a random point within the city bounds. As long as the form of the polyline can be arbitrary we will use Monte Carlo method: we will be generating random points in the entire area until a point happens to be inside the city. The `do…while` loop can be naturally used here:

```
//declare two variables
double x;
double y;
do {
    //pick a random point within the entire area
    x = uniform( 0, 1000 );
    y = uniform( 0, 1000 );
} while( ! citybounds.contains( x, y ) ); //if the point is not inside the bounds, try again
```

Block {…} and indentation

A *block* is a sequence of statements enclosed in braces `{…}`. There can be one, several, or even no statements inside a block. Block tells Java that the sequence of statements should be treated as one single statement, and therefore blocks are used with `if`, `for`, `while`, etc.

Java code conventions recommend to place the braces on separate lines from the enclosed statements and use *indentation* to visualize the nesting level of the block contents:



If the block is a part of a decision or loop statement, the braces can be placed on the same line with `if`, `else`, `for`, or `while`:

```
if( … ) {
    <statements>
} else {
    <statements>
}

while( … ) {
    <statements>
}
```

In `switch` statement it is recommended not to indent the lines with `case`:

```
switch( …  ) {
case …:
    <statements>
    break;
case …:
    <statements>
    break;
    …
}
```

Return statement

The `return` statement is used in a function body and tells Java to exit the function. Depending on whether the function returns a value or not, the syntax of the `return` statement will be one of the following:

```
return <value>; //in a function that does return a value
return; //in a function that does not return a value
```

Consider the following function. It returns the first order from a given client found in the collection `orders`:

```
Order getOrderFrom( Person client ) { //the function return type is Order
    if( orders == null )
        return null; //if the collection orders does not exist, return null
    for( Order o : orders ) { //for each order in the collection
        if( o.client == client ) //if the field client of the order matches the given client
            return o; //then return that order and exit the function
    }
    return null; //we are here if the order was not found - return null
}
```

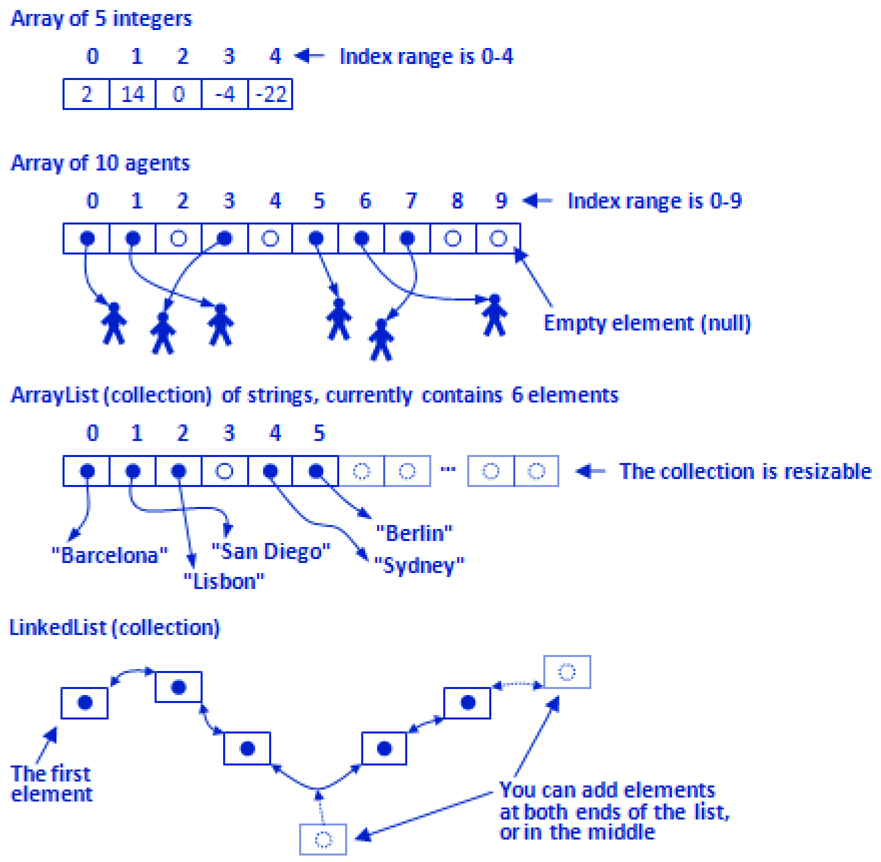
If the `return` statement is located inside one or several nested loops or if statements, it immediately terminates execution of all of them and exits the function. If a function does not return a value, you can skip the `return` at the very end of its body: the function will be exited in its natural way:

```
void addFriend( Person p ) { //void means no value is returned
    if( friends.contains( p ) )
        return; //explicit return from the middle of the function
    friends.add( p );
    //otherwise the function is exited after the last statement - no return is needed
}
```

Java arrays and collections

Java offers two types of constructs where you can store multiple values or objects of the same type: arrays and collections (for System Dynamics models AnyLogic also offers `HyperArray`, also known as "subscripts", – a special type of collection for dynamic variables).

Array or collection? Arrays are simple constructs with linear storage of fixed size and therefore they can only store a given number of elements. Arrays are built into the core of Java language and the array-related Java syntax is very easy and straightforward, for example the nth element of the `array` can be obtained as `array[n-1]`. Collections are more sophisticated and flexible. First of all, they are resizable: you can add any number of elements to a collection. A collection will automatically handle deletion of an element from any position. There are several types of collections with different internal storage structure (linear, list, hash set, tree, etc.) and you can choose a collection type best matching your problem so that your most frequent operations will be convenient and efficient. Collections are Java classes and syntax for obtaining, e.g., the nth element of a `collection` of type `ArrayList` is `collection.get(n)`.



Java arrays and collections

Please note that indexes in Java arrays and collections start with 0, not with 1! In an array or collection of size 10 the index of the first element is 0, and the last element has index 9.

Arrays

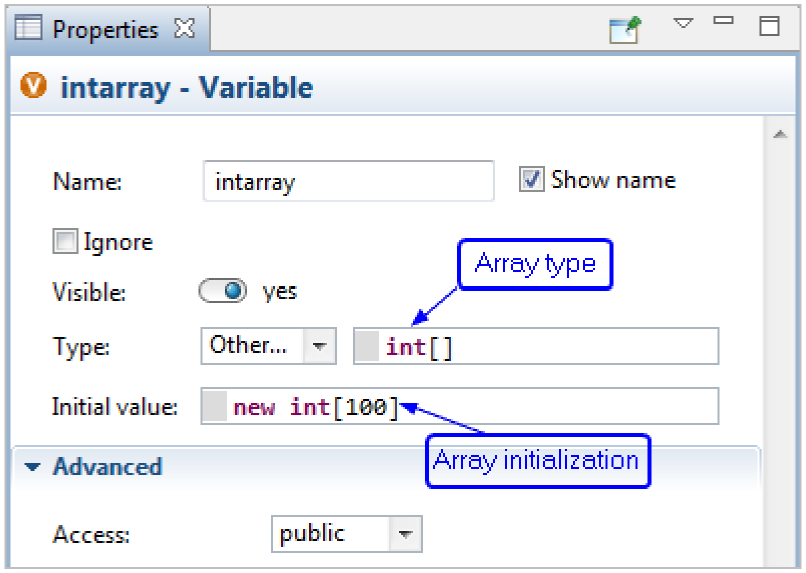
Java arrays are containers with linear storage of fixed size. To create an array you should declare a variable of array type and initialize it with a new array, like this:

```
int[] intarray = new int[100]; //array of 100 integer numbers
```

Array type is composed of the element type followed by square brackets, e.g. `int[]`, `double[]`, `String[]`, `Agent[]`. The size of the array is not a part of its type. Allocation of the actual storage space (memory) for the array elements is done by the expression `new int[100]`, and this is where the size is defined. Note that unless you initialize the array with the allocated storage, it will be equal to `null`, and you will not be able to access its elements.

A graphical declaration of the same array of 100 integers will look like the Figure. You should use a Variable or Parameter object, choose **Other** for **Type** and enter the array type in the field on the right.

Do not be confused by the checkbox **Array** in the properties of Parameter: that checkbox sets the type of parameter to System Dynamics `HyperArray` and not to Java array.



intarray - Variable

Name: intarray ☑ Show name

☐ Ignore

Visible: yes

Type: Other... int[]

Initial value: new int[100]

Access: public

Array variable declared graphically

All elements of the array initialized that way will be set to 0 (for numeric element type), `false` for `boolean` type and `null` for all classes including `String`. Another option is to explicitly provide initial values for all array elements. The syntax is the following:

```
int[] intarray = new int[] { 13, x-3, -15, 0, max{ a, 100 ) };
```

The size of the array is then defined by the number of expressions in braces. To obtain the size of an existing array you should write *<array name>*`.length`, for example:

```
intarray.length
```

The i^th element of an array (remember that array indexes are 0-based) can be accessed as:

```
intarray[i-1]
```

Iteration over array elements is done by index. The following loop increments each element of the array:

```
for( int i=0; i<intarray.length; i++ ) {
    intarray[i]++;
}
```

Java also supports a simpler form of the "for" loop for arrays. The following code calculates the sum of the array elements:

```
int sum = 0;
for( int element : intarray ) {
   sum +=  element;
}
```

Arrays can be *multidimensional*. This piece of code creates a two-dimensional array of double values and initializes it in a loop:

```
double[][] doubleArray = new double[10][20];
for( int i=0; i<doubleArray.length; i++ ) {
    for( int j=0; j<doubleArray[i].length; j++ ) {
        doubleArray[i][j] = i * j;
    }
}
```

You can think of a multidimensional array as of "array of arrays". The array initialized as `new double[10][20]` is array of 10 arrays of 20 double values each. Notice that `doubleArray.length` returns 10 and `doubleArray[i].length` returns 20.

Collections

Collections are Java classes developed to efficiently store multiple elements of a certain type. Unlike Java arrays collections can store any number of elements. The simplest collection is `ArrayList`, which you can treat as a resizable array. The following line of code creates an (initially empty) `ArrayList` of objects of class `Person`:

```
ArrayList<Person> friends = new ArrayList<Person>();
```

Note that the type of the collection includes the element type in angle brackets. This "tunes" the collection to work with the specific element type, so that, for example, the function `get()` of `friends` will return object of type `Person`. The `ArrayList<Person>` offers the following API (for the complete API see Java Class Reference):

- `int size()` – returns the number of elements in the list
- `boolean isEmpty()` – tests if this list has no elements
- `Person get( int index )` – returns the element at the specified position in this list
- `boolean add( Person p )` – appends the specified element to the end of this list
- `Person remove( int index )` – removes the element at the specified position in this list
- `boolean contains( Person p )` – returns true if this list contains a given element
- `void clear()` – removes all of the elements from this list

The following code fragment tests if the `friends` list contains the person `victor` and, if, victor is not in there, adds him to the list:

```
if( ! friends.contains( victor ) )
    friends.add( victor );
```

All collection types support iteration over elements. The simplest construct for iteration is a "for" loop. Suppose the class `Person` has a field `income`. The loop below prints all friends with income greater than 100000 to the model log:

```
for( Person p : friends ) {
    if( p.income > 100000 )
        traceln( p );
}
```

Another popular collection type is `LinkedList`.

Linked lists are used to model stack or queue structures, i.e. sequential storages where elements are primarily added and removed from one or both ends.

Consider a model of a distributor that maintains a backlog of orders from retailers. Suppose there is an `Order` class with the `amount` field. The backlog (essentially a FIFO queue) can be modeled as:

```
LinkedList<Order> backlog = new LinkedList<Order>();
```

`LinkedList` supports functions common to all collections (like `size()` or `isEmpty()`) and also offers a specific API:

- `Order getFirst()` – returns the first element in this list

- `Order getLast()` – returns the last element in this list
- `addFirst( Order o )` – inserts the given element at the beginning of this list
- `addLast( Order o )` – appends the given element to the end of this list
- `Order removeFirst()` – removes and returns the first element from this list
- `Order removeLast()` – removes and returns the last element from this list

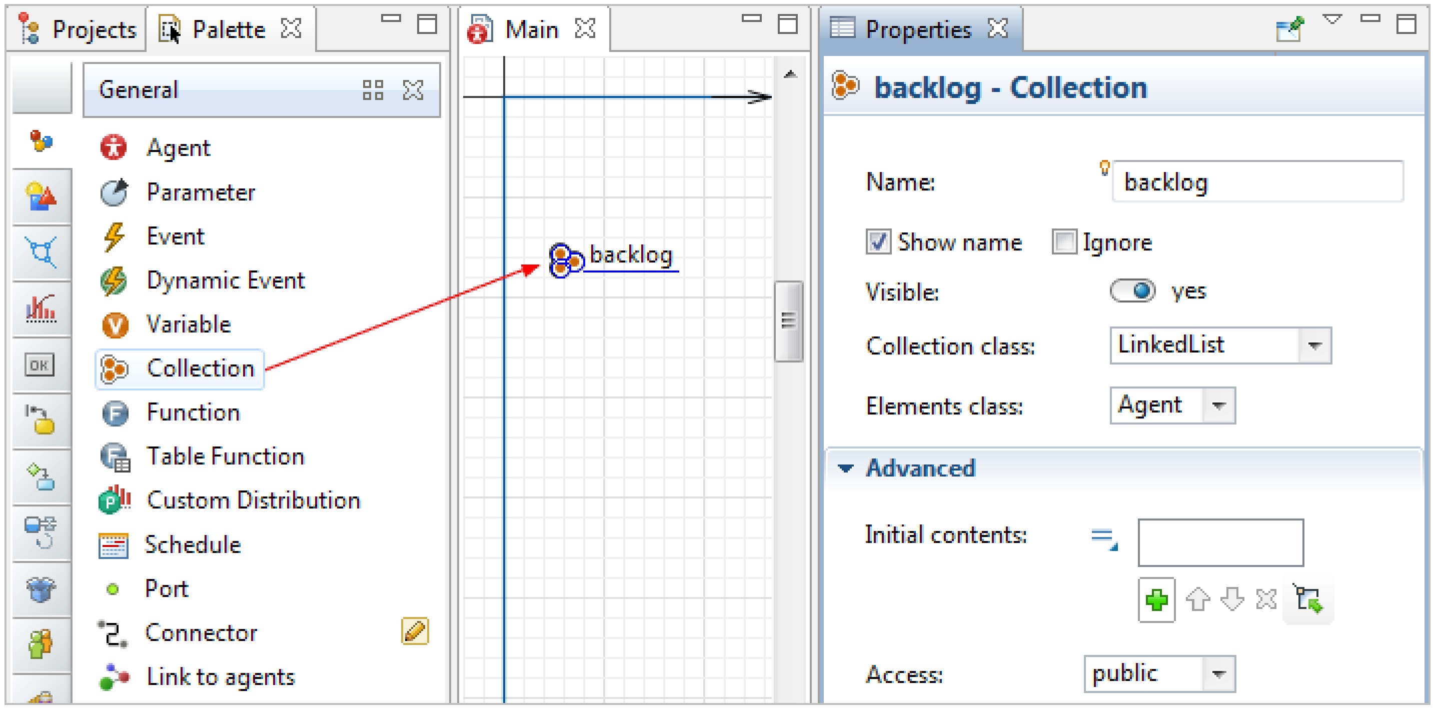
When a new order is received by the distributor it is placed at the end of the backlog:

```
backlog.addLast( order );
```

Each time the inventory gets replenished, the distributor tries to ship the orders starting from the head of the backlog. If the amount in an order is bigger than the remaining inventory, the order processing stops. The order processing can look like this:

```
while( ! backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
    Order order = backlog.getFirst(); //pick the first order in the backlog
    if( order.amount <= inventory ) { //if enough inventory to satisfy this order
        ship( order ); //ship
        inventory -= order.amount; //decrease available inventory
        backlog.removeFirst(); //remove the order from the backlog
    } else { //not enough inventory to ship
        break; //stop order backlog processing
    }
}
```

We recommend to declare collections in agents and experiments graphically. The **Collection** object is located in the **Agent** palette. All you need to do is to drop it on the canvas and choose the collection and the element types. At runtime you will be able to view the collection contents by clicking its icon.



backlog - Collection

Name: backlog

[✓] Show name [ ] Ignore

Visible: yes

Collection class: LinkedList

Elements class: Agent

Initial contents:

Access: public

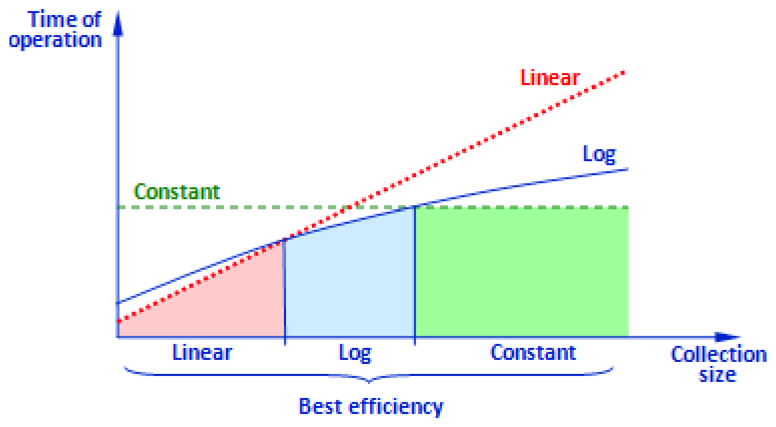
Declaring Java collection graphically

Different types of collections have different *time complexity* of operations. For example, checking if a collection of 10,000,000 objects contains a given object may take 80 milliseconds for `ArrayList`, 100 milliseconds for `LinkedList`, and less than 1 millisecond for `HashSet` and `TreeSet`. To ensure maximum efficiency of the model execution you should analyze, which operations are most frequent and choose the collection type correspondingly. The Table below contains the time complexities of the most common operations for four collection types.

Operation	ArrayList	LinkedList	HashSet	TreeSet
Obtain size	Constant	Constant	Constant	Constant
Add element	Constant	Constant	Constant	Log
Remove given element	Linear	Linear	Constant	Log
Remove by index	Linear	Linear	–	–
Get element by index	Constant	Linear	–	–
Find out if contains	Linear	Linear	Constant	Log

The terms *constant*, *linear*, and *logarithmic complexity* mean the following. Linear complexity means that the worst time required to complete the operation grows linearly as the size of the collection grows. Constant means that it does not depend on the size at all, and Log means the time grows as the logarithm of the size. You should not treat the constant complexity as the unconditionally best choice. Depending on the size, different types of collection may behave better than others.

Consider the Figure. It may well happen that with relatively small number of elements the collection with linear complexity will behave better than the one with constant complexity.



Depending on the size, different types of collections may behave better that others

And a couple of things you should keep in mind when choosing the collection type:

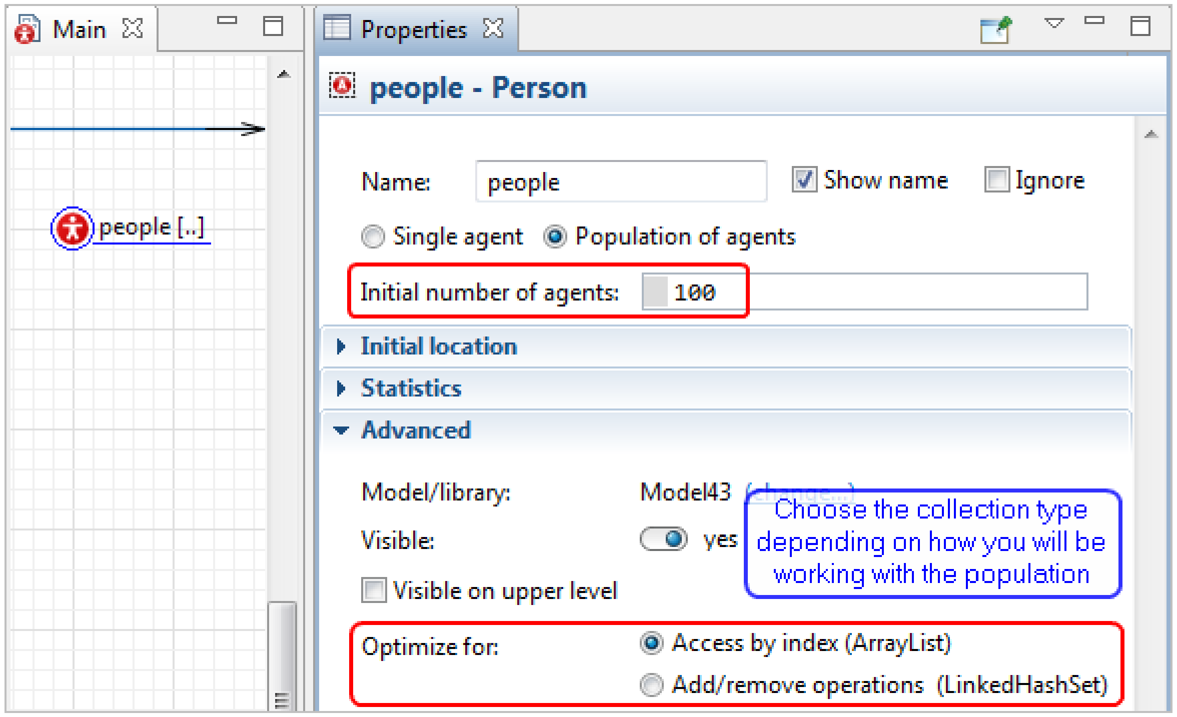
- `HashSet` and `TreeSet` do not support element indexes, so it is not possible to e.g. get an element at position 32.
- `TreeSet` is a naturally sorted collection: elements are stored in a certain order defined by a natural or a user-provided comparator.

Agent populations are collections too

When you declare an embedded agent as replicated, AnyLogic creates a special type of collection to store the individual agents. You have two options:

- `AgentArrayList` – choose this collection type if the set of agents is more or less constant or if you need to frequently access individual objects by index.
- `AgentLinkedHashSet` – choose this option if you plan to intensively add new agents and remove existing ones. For example, if you are modeling a population of a city for a relatively long period so that people are born, die, move out of the city, and new people arrive.

The options for the type of collection appear in the **Advanced** section of the embedded object properties, see the Figure.


Main
Properties
people [..]
Initial location
Statistics
Advanced
Choose the collection type depending on how you will be working with the population

people - Person

Name: people [✓ Show name] Ignore

Single agent [✓ Population of agents]

Initial number of agents: 100

Model/library: Model43 (change...)

Visible: yes

Visible on upper level

Optimize for: [✓ Access by index (ArrayList)] Add/remove operations (LinkedHashSet)

Options for collection type of a replicated object

Both collections support functions like `size()`, `isEmpty()`, `get( int index )`, `contains( Object ao )`, and iteration. If index is not specifically needed during iteration, it is always better to use the enhanced form of "for" loop:

```
for( Person p : people ) {
    …
}
```

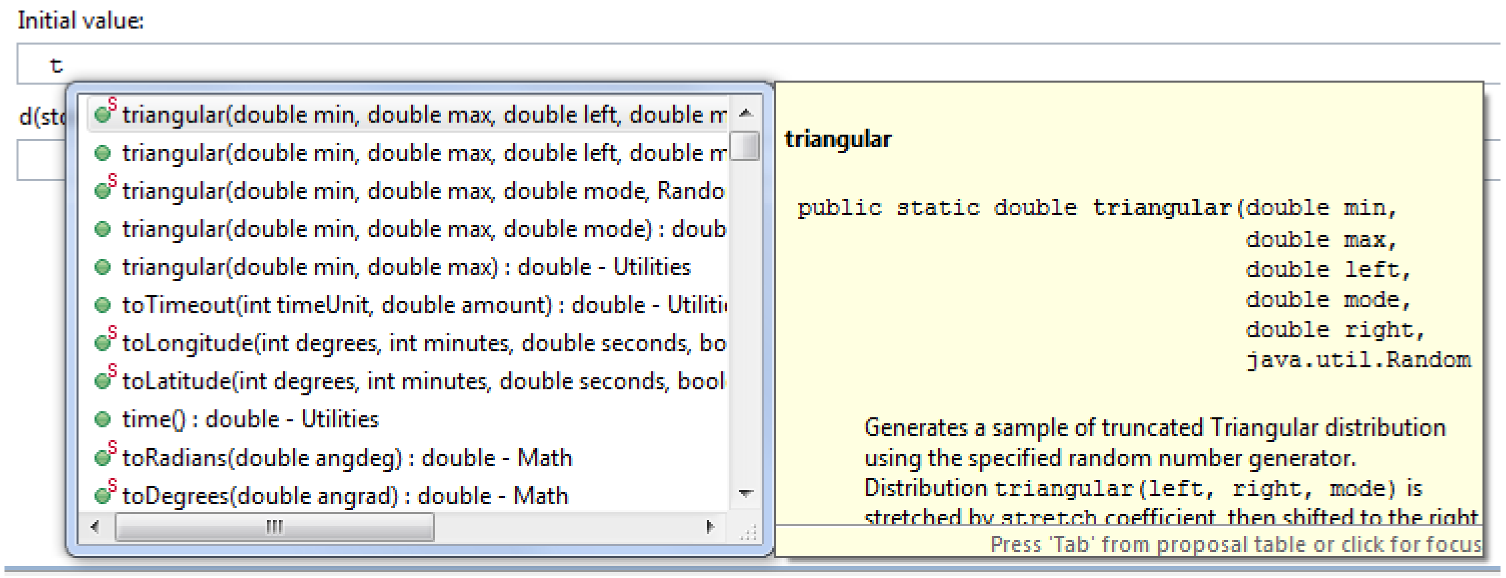
Code Completion Master

AnyLogic supports intelli-sense mechanism. This significantly simplifies typing code since you do not need to type the whole names of functions, variables and parameters. You can use the intelli-sense wizard to insert a variable name or a call of a function.

The wizard looks as a list, containing variables, parameters, and functions. You can simply select the name in the list, and it will be inserted in the expression automatically.

To insert a name using the Code Completion Master

1. Move cursor at the position where you want to place the object name.
2. Press Ctrl+space (Mac OS: Option-space). The wizard listing all model variables and predefined functions appears.


Initial value:
t
d(st
triangular(double min, double max, double left, double m
triangular(double min, double max, double left, double m
triangular(double min, double max, double mode, Rando
triangular(double min, double max, double mode) : doub
triangular(double min, double max) : double - Utilities
toTimeout(int timeUnit, double amount) : double - Utiliti
toLongitude(int degrees, int minutes, double seconds, bo
toLatitude(int degrees, int minutes, double seconds, bool
time() : double - Utilities
toRadians(double angdeg) : double - Math
toDegrees(double angrad) : double - Math
triangular
Press 'Tab' from proposal table or click for focus

```
public static double triangular(double min,
                                double max,
                                double left,
                                double mode,
                                double right,
                                java.util.Random
```

Generates a sample of truncated Triangular distribution using the specified random number generator. Distribution `triangular(left, right, mode)` is stretched by stretch coefficient then shifted to the right

Intelli-sense wizard

3. Scroll to the name you want to add, or type the first letters of the name until it becomes visible in the list.
4. Select the name by clicking. The wizard displays the detailed description of the selected object in the popup text box.
5. Double-click the name to insert it into the equation expression.

Optionally you can change the key combination that invokes code completion in **AnyLogic preferences** dialog, on the **Key combinations** page.

Some tips for using code assist

- You can use the mouse or the keyboard (Up Arrow, Down Arrow, Page Up, Page Down, Home, End, Enter) to navigate and select lines in the list.
- If you select a line in the content assist list, you can view Javadoc description for that line in the window opened to the right.

- Clicking or pressing Enter on a selected line in the list inserts the selection into the editor.
- You can access specialized content assist features inside Javadoc comments.

Comments

Comments in Java are used to provide additional information about the code that may not be clear from the code itself. Even if you do not expect other people to read and try to understand your code, you should write comments for yourself, so that when you come back to your model in a couple of months you will be able to easily remember how it works, fix, or update it. Comments keep your code live and maintainable; writing good comments as you write code must become your habit.

This is an example of a useless comment:

```
client = null; //set client to null - useless comment
```

It explains things obvious from the code. Instead, you can explain the meaning of the assignment:

```
client = null; //forget the client - all operations are finished
```

In Java there are two types of comments: *end of line comments* and *block comments*. The end of line comment starts with double slash `//` and tells java to treat the text from there to the end of the current line as comment:

```
//create a new plant
Plant plant = add_mills();
//place it somewhere in the selected region
double[] loc = region.findLocation();
plant.setXY( loc[0], loc[1] );
//set the plant parameters
plant.set_region( region );
plant.set_company( this ); //we are the owner
```

AnyLogic Java editor displays the comments in green color, so you can easily distinguish them from code. Block comment is delimited by `/*` and `*/`. Unlike the end of line comment, the block comment can be placed in the middle of a line (even in the middle of an expression), or it can span across several lines.

```
/* for sales staff we inlclude also the commission part
  * that is based on the sales this quarter
  */
amount  = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 )
    doAudit( employee ); /* perform audit for very high payments */
employee.pay( amount );
```

When you are developing a model it may be needed to temporarily exclude portions of code from compilation, for example, for debugging purposes. This is naturally done by using comments. In the expression below the commission part of the salary is temporary excluded from the expression, for example, until the commissions get modeled.

```
amount  = employee.baseSalary /* + commissionRate * employee.sales */ + bonus;
```

If you wish to exclude one or a couple of lines of code, it makes sense to put the end of line comments at the beginning of the line(s). In the code fragment below the line with the function call `ship()` will be ignored by the compiler (note that the line already contains a comment):

```
while( ! backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
    Order order = backlog.getFirst(); //pick the first order in the backlog
    if( order.amount <= inventory ) { //if enough inventory to satisfy this order
//        ship( order ); //ship
        inventory -= order.amount; //decrease available inventory
        backlog.removeFirst(); //remove the order from the backlog
    } else { //not enough inventory to ship
        break; //stop order backlog processing
    }
}
```

If many lines are excluded, it is better to use block comments:

```
/*
while( ! backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
    Order order = backlog.getFirst(); //pick the first order in the backlog
    if( order.amount <= inventory ) { //if enough inventory to satisfy this order
//        ship( order ); //ship
        inventory -= order.amount; //decrease available inventory
        backlog.removeFirst(); //remove the order from the backlog
    } else { //not enough inventory to ship
        break; //stop order backlog processing
    }
}
*/
```

Be careful when using comments to exclude code: you may make undesirable changes to the code nearby . Consider the code fragment below. The original plan was to perform audit for every payment that is over $200,000. The model developer decided to temporary skip the audit and commented out the line with the `doAudit()` function call. As a side effect, the next line became a part of the if statement and the payments will be made only to those employees who earn more than $200,000.

```
amount  = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 )
//    doAudit( employee );
employee.pay( amount );
```

Note that the accident could have been avoided if the modeler had used the block braces with the if statement:

```
amount  = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 ) {
//    doAudit( employee );
}
employee.pay( amount );
```

Naming conventions

Now please take a moment to familiarize yourself with the naming conventions. Names you give to the model objects are important. A good naming system simplifies development process a lot. We recommend you to keep to Java naming conventions throughout the model, not just when writing Java code.

The name of the object should indicate the intent of its use.

Ideally a casual observer should be able to understand the role of the object from its name and, on the other hand, when looking for an object serving a particular purpose, he should be able to easily guess its name.

You can compose a name of several words. Use mixed case with the first letter of each word capitalized. Do not use underscores.

You should never use meaningless names like “a”, “b”, “x23”, “state1”, “event14”. One-character variable names should only be used for temporary “throwaway” variables such as loop indexes. Avoid acronyms and abbreviations unless they are more common than the long form, for example: NPV, ROI, ARPU. Remember that when using the object name in an expression or code you will not need to type it: AnyLogic code completion will do the work for you, therefore multi-word names are perfectly fine.

Keep to one naming system. Names must be uniformly structured.

It makes sense to develop a naming system for your models and keep to it. Large organizations sometimes standardize the naming conventions across all modeling projects.

A couple of important facts: Java is a case-sensitive programming language: Anylogic and AnyLogic are different names that will never match. Spaces are not allowed in Java names.

In the Table below we summarize the naming recommendations for various types of model objects.

Object	Naming rules	Examples

• Java variable
• Parameter of agent
• Collection
• Table function
• Statistics
	First letter can be lowercase or uppercase (here we relax Java conventions), first letter of each internal word capitalized.
	
	Should be a noun.
	
	Use plurals for collections.
	
	Sometimes adding a suffix or prefix indicating the type of the object helps to understand its meaning and to avoid name conflicts. For example, `AgeDistribution` can be a custom distribution constructed from the table function `AgeDistributionTable`.
		`rate`
		`Income`
		`DevelopmentCost`
		`inventory`
		`AgeDistribution`
		`AgeDistributionTable`
		`friends`

• Function	First letter must be lowercase, first letter of each internal word capitalized.
	
	Should be a verb.
	
	If the function returns a property of the object, its name should start with the word “get”, or "is" for boolean return type. If the function changes a property of the object, it should start with “set”.
		`resetStatistics`
		`getAnnualProfit`
		`goHome`
		`speedup`
		`getEstimatedROI`
		`setTarget`

	There are some exceptions created to make the code more compact, such as AnyLogic system functions `time()` and `date()`, or functions `size()` of Java collection.	`inState`
		`isVisible`
		`isEnabled`

• Function argument
• Local variable in the code
	If possible, short, lowercase. If consists of more than one word, first letter of each internal word should be capitalized
	
	Common names for temporary integer variables are `i`, `j`, `k`, `m`, and `n`.
		`cost`
		`sum`
		`total`
		`baseValue`
		`i`
		`n`

• Java constant	All uppercase with words separated with underscores "_".	`TIME_UNIT_MONTH`

Classes:
• Agent type
• User's Java class
• Dynamic event
	First letter *must* be capitalized, first letter of each internal word capitalized as well.
	
	Should be a noun except for process model components that have a meaning of action, in which case it can be a verb.
		`Consumer`
		`Project`
		`UseNurse`
		`RegistrationProcess`
		`HousingSector`
		`PhoneCall`
		`Order`
		`Arrival`

• Agent population, including the library objects	First letter must be lowercase, first letter of each internal word capitalized.
	
	Use plurals for replicated objects.
		`project`
		`consumers`
		`people`
		`doTriage`
		`registrationProcess`
		`stuffAndMailBill`

Dynamic variables:
• Stock
• Flow
• Dynamic variable
	Long multi-word variable names are very common in system dynamics models. As in Java and in AnyLogic spaces are not allowed in names, you should use other ways to separate words. We recommend to use mixed case with the first letter of each word capitalized. The use of underscore "_" is not recommended, although it is allowed.	`BirthRate`
		`Population`
		`DrugsUnderConsideration`
		`TimeToImplementStrategies`

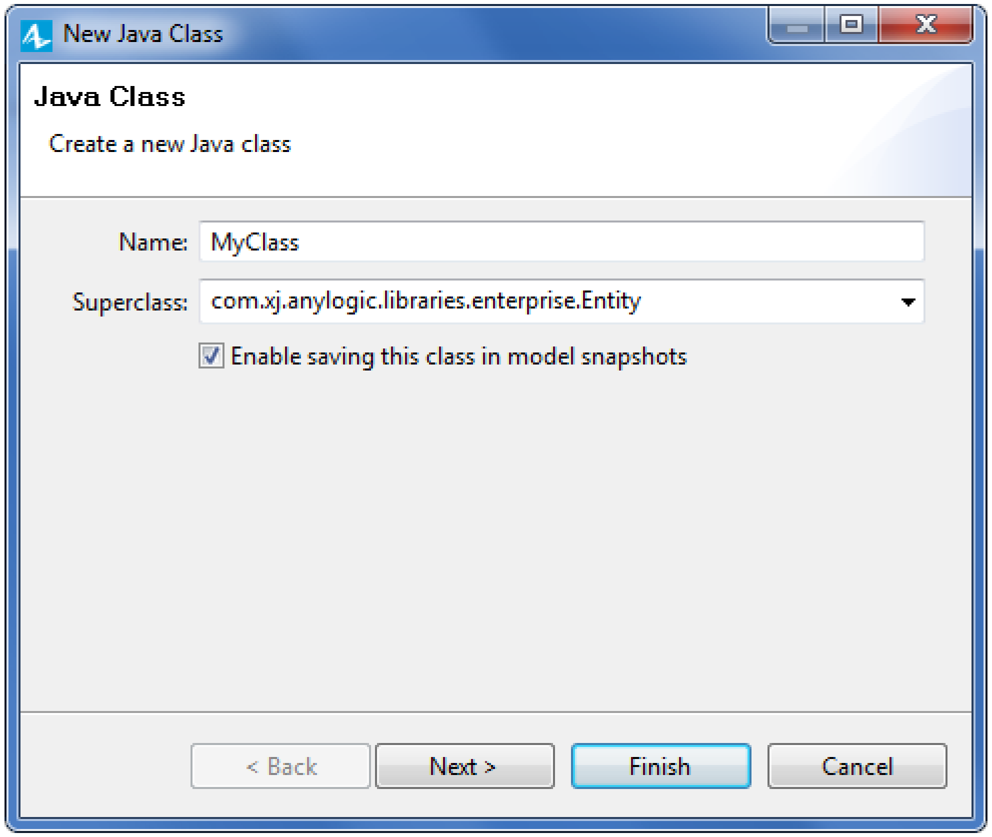
• Events (not dynamic)
• Statechart
• States
• Transition
	First letter can be lowercase or uppercase, first letter of each internal word capitalized.	`overflow`
		`at8AMeveryDay`
		`purchaseBehavior`
		`InWorkForce`
		`discard`

Adding Java Classes

AnyLogic allows the user to create his/her own Java classes in the model with any required functionality.

To add a Java class

1. In the **Projects** view, right-click (Mac OS: Ctrl+click) the model item you are currently working with, and choose **New > Java Class…** from the popup menu.
2. The **New Java Class** wizard is displayed.

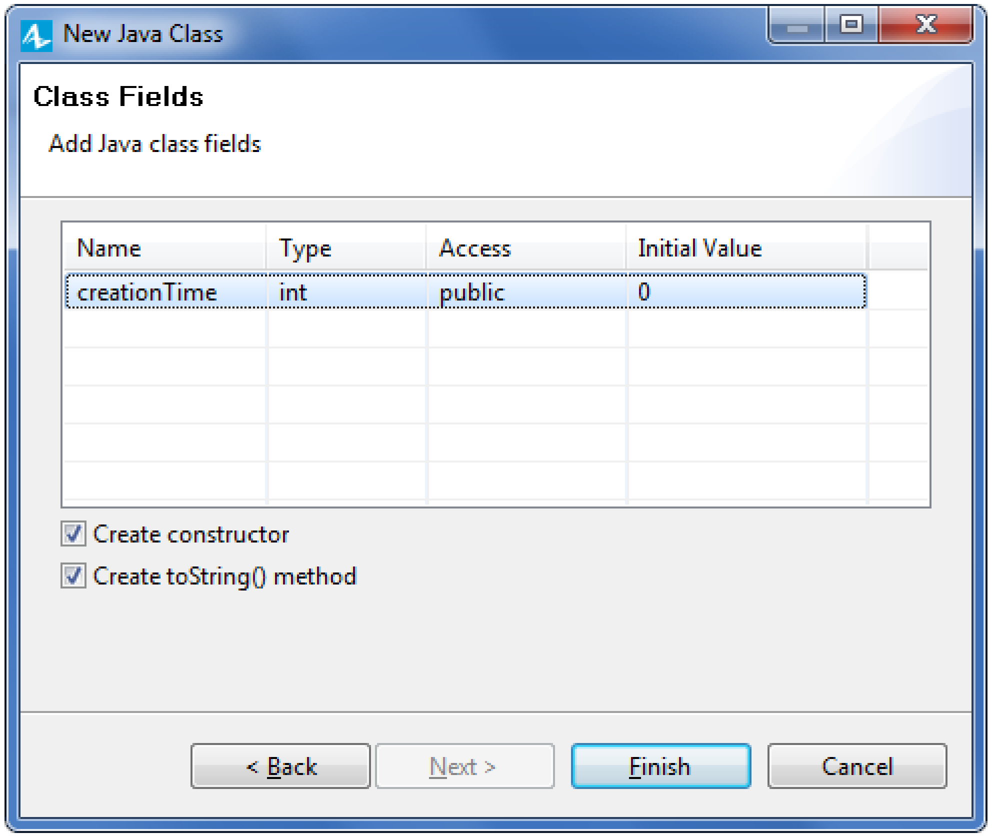


Name: MyClass

Superclass: com.xj.anylogic.libraries.enterprise.Entity

[✓] Enable saving this class in model snapshots

3. On the first page of the wizard, specify the name of the new Java class in the **Name** field and optionally type in the superclass name in the **Superclass** edit box.
4. Click **Next** to go to the next page of the wizard.



Class Fields

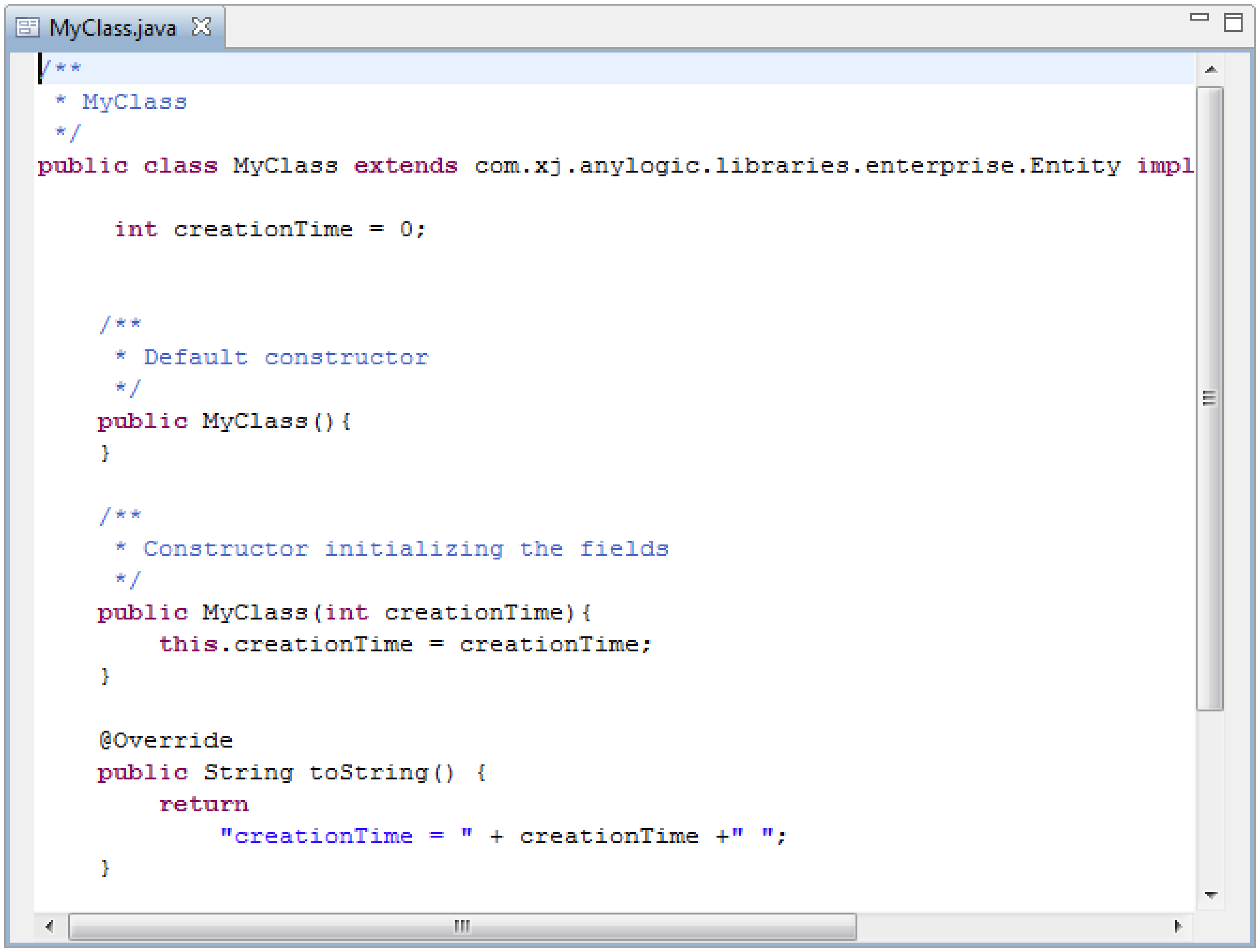
Add Java class fields

Name	Type	Access	Initial Value
creationTime	int	public	0

☑ Create constructor

☑ Create toString() method

5. On the second page of the wizard, specify Java class fields. Fields are specified in the table, each class field is defined in the separate row. Enter the type of the field in the **Type** cell, name of the field in the **Name** cell and optionally name the access modifier in the **Access** cell and specify the initial value in the **Initial value** cell.
6. Using **Create constructor** and **Create toString() method** check boxes, specify whether you want default class constructor and toString() method to be created automatically.
7. Click **Finish** to complete the process. You will see the code editor for the created class opened.


MyClass.java

```
/**
 * MyClass
 */
public class MyClass extends com.xj.anylogic.libraries.enterprise.Entity impl

     int creationTime = 0;


    /**
     * Default constructor
     */
    public MyClass(){
    }

    /**
     * Constructor initializing the fields
     */
    public MyClass(int creationTime){
        this.creationTime = creationTime;
    }

    @Override
    public String toString() {
        return
            "creationTime = " + creationTime +" ";
    }
```

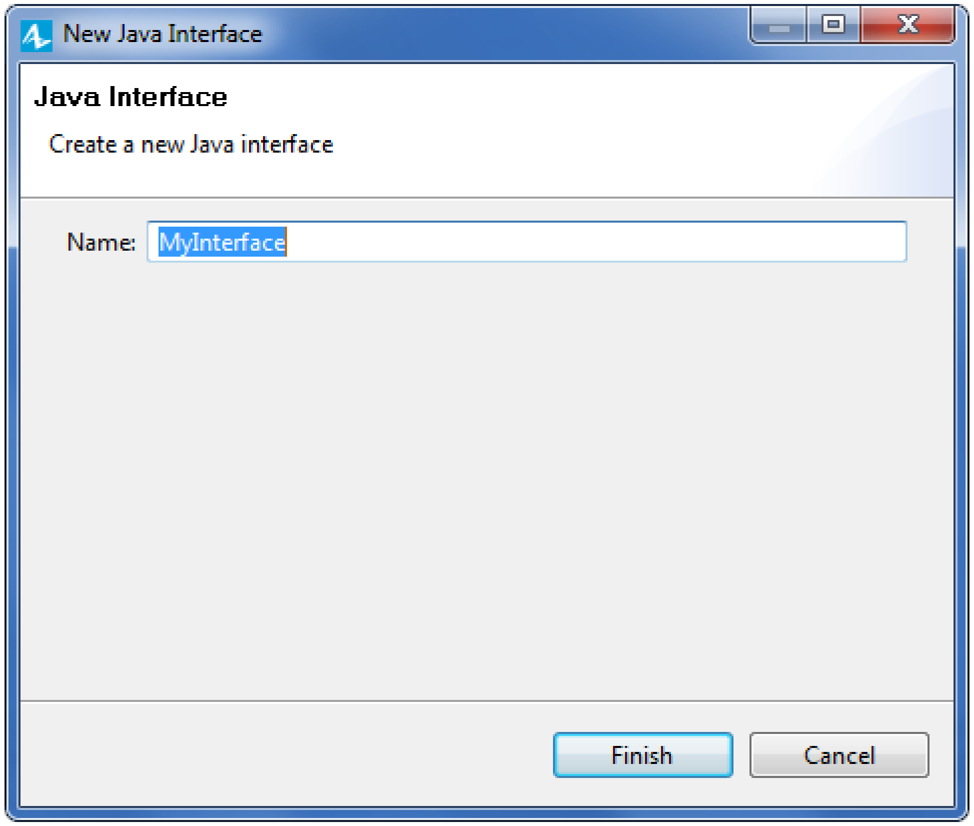
Adding Java Interfaces

AnyLogic allows the user to add Java interfaces to a model.

Please refer to Interfaces section of Java online tutorials for more information on Java interfaces.

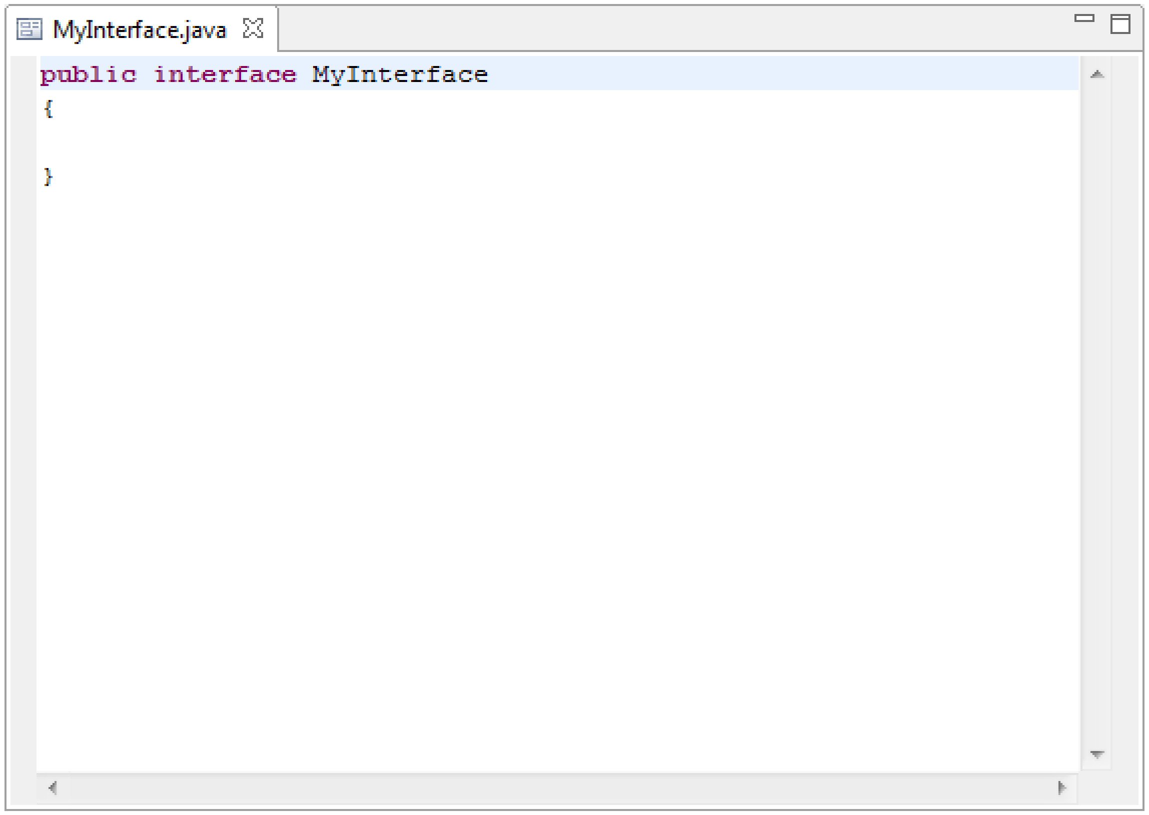
To add a Java interface

1. In the **Projects** view, right-click (Mac OS: Ctrl+click) the model item you are currently working with, and choose **New > Java Interface…** from the popup menu.
2. The **New Java Interface** dialog box is displayed.



Name: MyInterface

3. Specify the name of the new Java interface in the **Name** field and click **Finish** to complete the process.
4. You will see Java editor opened prompting you to write Java code for the just defined interface.



```
public interface MyInterface
{

}
```

To use an interface, you write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

To make agent class implementing an interface

1. Select the agent type in the **Projects** view.
2. In the **Advanced Java** section of the **Properties** view, type the interface name in the **Implements (comma-separated list of interfaces)** field.

To make Java class implementing an interface

1. Double-click the Java class in the **Projects** view to open its code in the Java editor.
2. Complete the first code line containing the class name with the string `implements <InterfaceName>`:

```
public class MyClass implements Animatable
{ ...
```