Java for AnyLogic users

It would be nice if any simulation model could be put together graphically, in drag and drop manner. In practice, however, only very simple models are created by using a mouse and not touching the keyboard. As you try to better reflect the real world in the model, you inevitably realize the need to use probability distributions, evaluate expressions and test conditions containing properties of different objects, define custom data structures and design the corresponding algorithms. These actions are better done in text, not in graphics, and therefore any simulation modeling tool includes a textual scripting language.

From the very beginning we did not want to invent a proprietary scripting language for AnyLogic. Moreover, the creation of AnyLogic was significantly inspired by Java, which we think is the ideal language for modelers. On one hand, Java is a sufficiently high-level language in which you do not need to care about memory allocation, distinguish between objects and references, etc. On the other hand, Java is a fully powerful object oriented programming language with high performance. In Java, you can define and manipulate data structures of any desired complexity; develop efficient algorithms; and use numerous packages available from Sun™, Oracle™ and other vendors. Java is supported by industry leaders and as improvements are made to Java, AnyLogic modelers automatically benefit from it.

A model developed in AnyLogic is fully mapped into Java code and, having been linked with the AnyLogic simulation engine (also written in Java), and, optionally, with a Java optimizer, becomes a completely independent standalone Java application. This makes AnyLogic models cross-platform: they can run on any Java-enabled environment or even in a web browser as applets.

A frequently asked question is "How much Java do I need to know to be successful with AnyLogic?" The good news is that you do not need to learn object-oriented programming. The "backbone Java class structure" of the model is automatically generated by AnyLogic. In a typical model, Java code is present in small portions written in various properties of the graphically-created model objects. This can be an expression, a function call, or a couple of statements. Therefore you need to get familiar with the fundamental data types, learn the basics of Java syntax, and understand that to do something with a model object you need to call its function.

This chapter is by no means a complete description of Java language, or even an introduction to Java suitable for programmers. This is a collection of information that will allow you to manipulate data and model objects in AnyLogic models. It is sufficient for a typical modeler. For those who plan to write sophisticated Java code,

use object-orientedness, or work with external Java packages, we recommend learning Java with a good textbook, such as this one:

Bruce Eckel???

Primitive data types

There are about ten *primitive data types* in Java. 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 the AnyLogic engine all real values (such as time, coordinates, length, speed, and 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 either the 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 such as C or C++, they are not interchangeable with numbers, so you cannot treat **false** as 0 or a non-zero number as **true**.

String constants are sequences of characters enclosed between the quotation 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, 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 in a string, you need to write \". For example the string constant "String with \" in the middle" will print as:

String with " in the middle.

To include a backslash in a string, use a double backlash. For example, "This is a backslash: \\" will print as:

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 within the scope of this chapter: 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 by introducing 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(), or if the class Agent is a subclass of ActiveObject, it supports the methods of both of them.

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

Consider an example. Suppose you are working a 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 an entity is called a *class*. Java class definition for the location on a local map can look like this:

```
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 a class, we can write very simple and readable code when working with the map. Consider the following code:

```
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

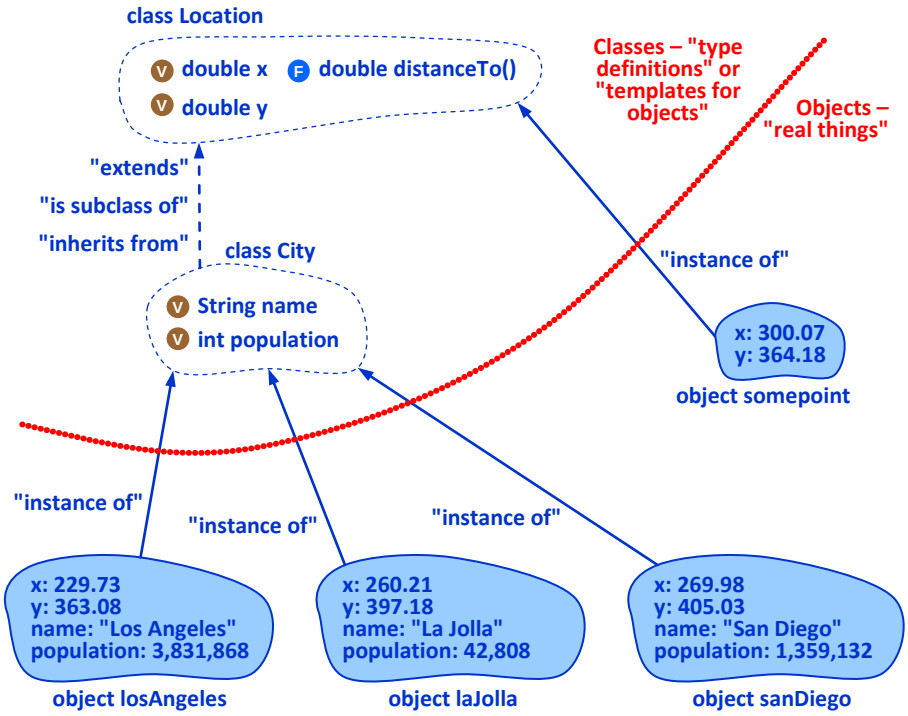
Now suppose some locations in your 2D space correspond to cities. A city has a name and population size. To efficiently manipulate cities we can extend the class **Location** by adding two more fields: **name** and **population**. We will call the new entity **City**. In Java this will look like:

```
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 the code is for finding 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 can still 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 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 parentheses 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 of which you create your models are instances of AnyLogic Java classes. In the Table you will find the Java class names for most model elements you work with. Whenever you need to find out what can you do with a particular object using Java, you should find the corresponding Java class in AnyLogic Class Reference and look through its methods and fields.

Model object	Java class name
Active object (such as **Main**)	Base class: **ActiveObject**
Agent (such as **Person**)	Base class: **Agent** (subclass of **ActiveObject**) Classes: **AgentContinuous2D**, **AgentContinuous3D**, **AgentDiscrete2D**, **AgentContinuousGIS**
Event	Base class: **Event** Classes: **EventCondition**, **EventRate**, **EventTimeout**
Dynamic event	**DynamicEvent**
Statechart	**Statechart**
Transition of a statechart	Base class: **Transition** Classes: **TransitionCondition**, **TransitionMessage**, **TransitionRate**, **TransitionTimeout**
Table function	**TableFunction**
Schedule	**Schedule**
Port	Base class: **Port** Classes of the Enterprise Library: **InPort**, **OutPort**, **OutPortPush**
Statistic collectors: Data set, Histogram Data, …	**DataSet**, **HistogramData**, …
Charts: Bar chart, Plot, Histogram chart, …	**BarChart**, **Plot**, **Histogram**, …
Shapes: Rectangle, Polyline, Group, …	Base class: **Shape** Classes: **ShapeRectangle**, **ShapePolyLine**, **ShapeGroup**, …
Controls: Button, Slider, List box, …	Base class: **ShapeControl** (subclass of **Shape**) Classes: **ShapeButton**, **ShapeSlider**, **ShapeListBox**
Connectivity objects: Text file, Excel file, Database, …	**TextFile**, **ExcelFile**, **Database**, …

Enterprise Library objects: Source, Delay, Queue, Select Output, …	Base class: **ActiveObject** Classes of the Enterprise Library: **Source**, **Queue**, **Delay**, **SelectOutput**, …
Entities and resource units in process models	Classes of the Enterprise Library: **Entity**, **ResourceUnit**
Experiments: Simulation, Parameter variation, Optimization, …	Base class: **Experiment** Classes: **ExperimentSimulation**, **ExperimentParamVariation**, **ExperimentOptimization**
AnyLogic model GUI	**Presentation**, **Panel**

Variables (local variables and class fields)

In this section we are considering plain Java variables. Special kinds of variables with additional functionality specific to simulation modeling, such as parameters or dynamic variables are described in other chapters of the book.

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

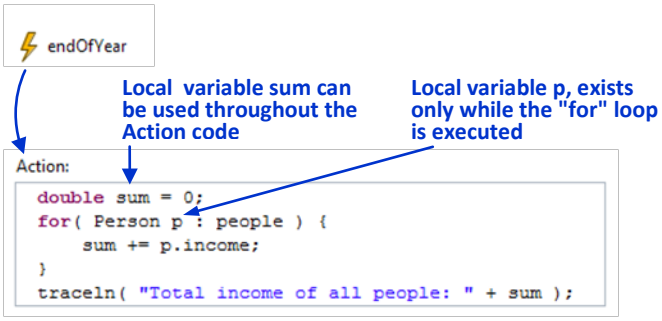
- *Local variable* – an auxiliary temporary variable that exists only while a particular function or a block of statements is executed, or
- *Class variable* (or *class field* – more correct term in Java) – a variable that is present in any object of a class, and whose lifetime is the same as the object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

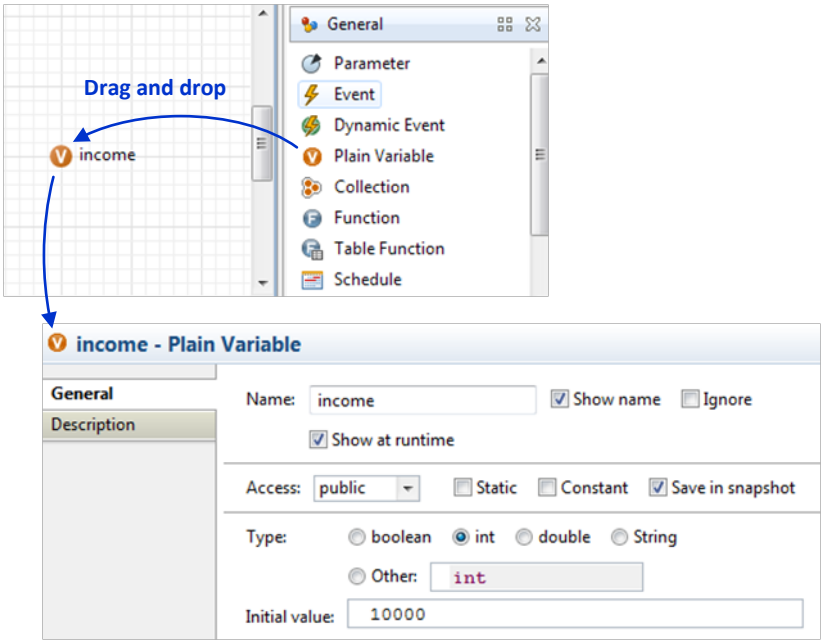
Local variables can be declared and used in AnyLogic fields where you enter actions (sequences of statements), such as **Startup code** of the active object class, **Action** field of events or transitions, **Entry action** and **Exit action** of state, **On enter** and **On exit** fields of flowchart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



Local variables declared in the Action code of an event

Class variables (fields)

Java variables (fields) of the active object class are part of the "memory" or "state" of active objects. They can be declared graphically or in code.



A variable of an active object declared in the graphical editor

To declare a variable of active object (or experiment) class:

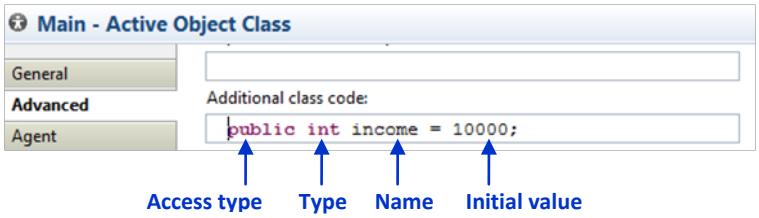
1. Open the **General** palette and drag the **Plain variable** object to the canvas.
2. Type the variable name in the in-place editor or in the variable properties.
3. Choose the variable **Access** type in the **General** page of the variable properties. In most cases you can leave **default**, which means the variable will

be visible within the current model. **public** opens access to the variable from other models, and **private** limits access to this active object 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.

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**.

In the Figure, a variable **income** of type **int** is declared in an active object (or experiment) class. 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 the **Additional class code** field of the **Advanced** property page of the class, as shown in the Figure:



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 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 )
```

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 ActiveObject (a fundamental class of the AnyLogic simulation engine). For your convenience, AnyLogic system functions and most frequently used Java standard functions are available directly through ActiveObject (you do not need to think which package or class they belong to, and can call those functions without any prefixes). Following are some examples (these are only a few functions out of several hundreds; see AnyLogic Class 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:

- ActiveObject getOwner() – returns the upper level active object that embeds this one, if any
- int getIndex() – returns the index of this active object in the list if it is a replicated object
- Experiment<?> getExperiment() – returns the experiment controlling the model execution
- Presentation getPresentation() – returns the model GUI
- Engine getEngine() – returns the simulation engine

If the active object is an agent, more functions are available specific to the particular type of agent, for example:

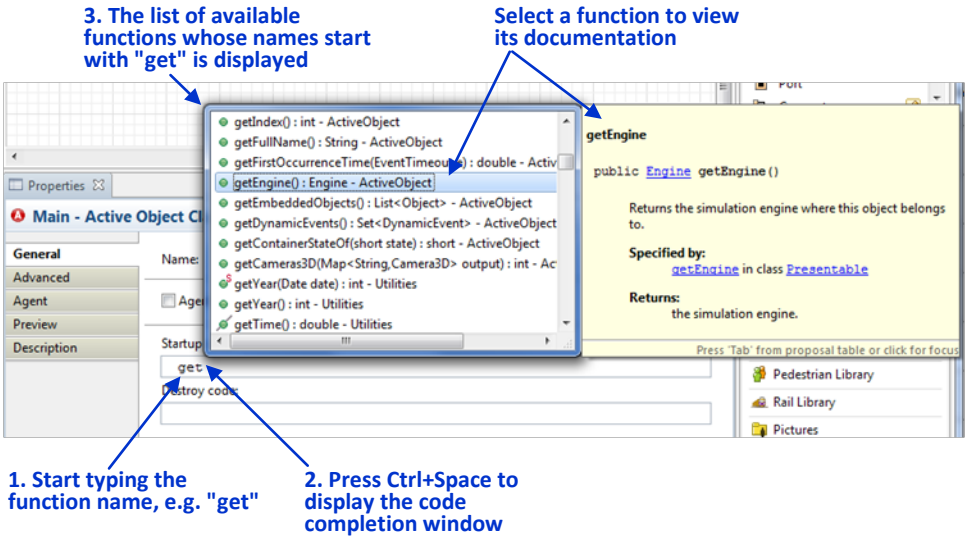
Network and communication-related functions:

- 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 code), are always listed in the code completion window that opens if you press Ctrl+Space (Alt+Space on Mac), as shown in the Figure.



Code completion window shows the functions available in the current context

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 active object you should put the element name followed by dot "." before the function call:
<object>.<method call>

Following are some examples of calling functions of the elements of the current active object (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
- statechart.isStateActive(going) – tests if the state going is currently active in 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**

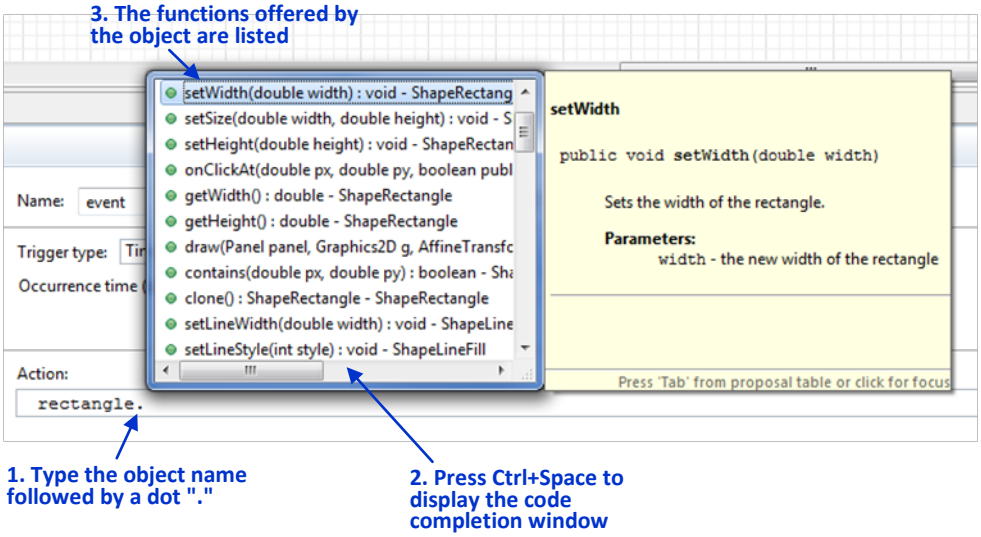
This statement hides or shows the shape depending on the state of the checkbox:

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

Changing parameters and states of embedded active objects:

- **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** on the **General** page of the **source** object properties. To find out the Java names of the parameters, you should open the **Parameters** page of the properties.



Code completion widow showing the functions of a particular object

You can easily find out which functions are offered by a particular object 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.

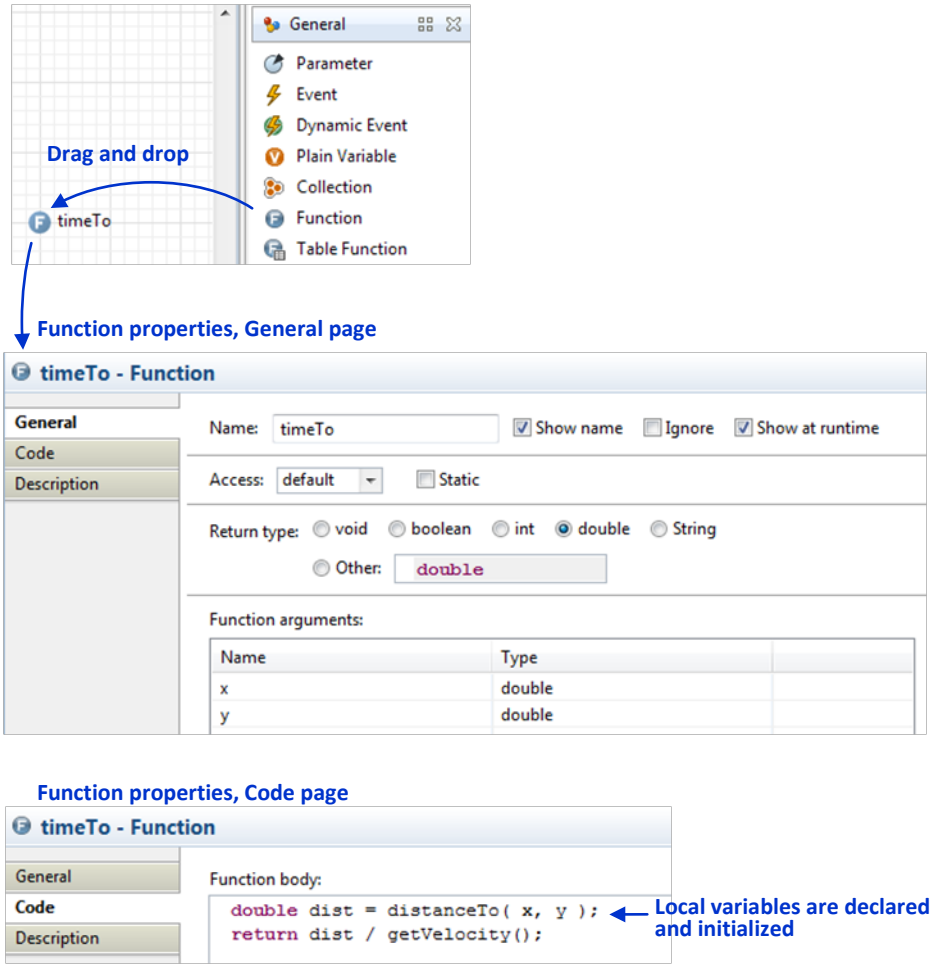
Typically, the list of suggested functions in the code completion window is large, and it makes sense to apply a filter. For example, if you are looking for a function that changes a property of the object, you should type the object name, dot, and "set". The completion suggestions will then be limited to those that start with "set". See Java naming conventions for more tips.

Defining your own function

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

To define a function of an active object (or experiment) class:

1. Open the **General** palette and drag the **Function** object to the canvas.
2. Type the function name in the in-place editor or in the function properties.
3. Choose the function **Access** type in the **General** page of the function properties. In most cases you can leave **default**, which means the function will be visible within the current model. **public** opens the access to the function from other models, and **private** limits the access to this active object only.
4. Choose the function return type. **void** means the function does not return a value. If the return type is not one of the primitive types, you should choose **Other** and enter the type name in the field nearby.
5. If the function has arguments, add them to the **Function arguments** table. You must specify the name and type for each argument. Use the buttons to the right of the table to reorder or delete arguments.
6. Switch to the **Code** page of the function properties and type the function body code (the page may have some auto-generated code for functions returning a value).



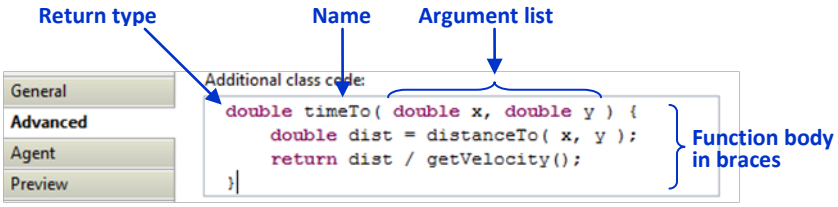
A user-defined function of an active object

In the Figure, the function **timeTo()** is defined for an active object class, which is an agent in continuous 2D space. The function calculates and returns the time needed for the agent to move to a given point (x,y) in the 2D space. The returned value is of type **double**, and the two arguments **x** and **y** are of type **double** as well. In the body code of the function, we declare a local variable **dist** and initialize it with the distance to the point. Then we divide the distance by the velocity of the agent and return the result.

When AnyLogic generates the Java code for the active object class, the function will be mapped to the following fragment of the class code (you can view the code by placing the cursor in the function body and pressing Ctrl+J):

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

Another way of defining a function is writing its full Java declaration and implementation in the **Additional class code** field of the **Advanced** property page of the active object class or experiment, as shown in the Figure. 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.



The same function defined in the Additional class code of the active object

Expressions

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. Parentheses 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 parentheses to explicitly define the order of operations, so you do not have to remember which operation has higher priority.

A discussion on arithmetic expressions should include *integer division*.

The result of division in Java depends on the types of the operands. If both operands are integers, the result will be an 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 row 8 with seat number 7. In the model Seats in a movie theater this technique is used to position the replicated picture of a seat.

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) $\equiv a^b$

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.

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`).

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 an 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 true, we compare that element with the string **"London"**.

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 is the following string expression:

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

which results in a string like this:

```
Number of destinations: 18; the first one is London
```

And once again: you should use the function **equals()** and not the "**==**" operator to compare strings!

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, or any classes. The following expression returns 0 if the backlog contains no orders or returns the amount of the first order in the backlog queue:

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

Conditional operators can be nested. For example, the following code 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:

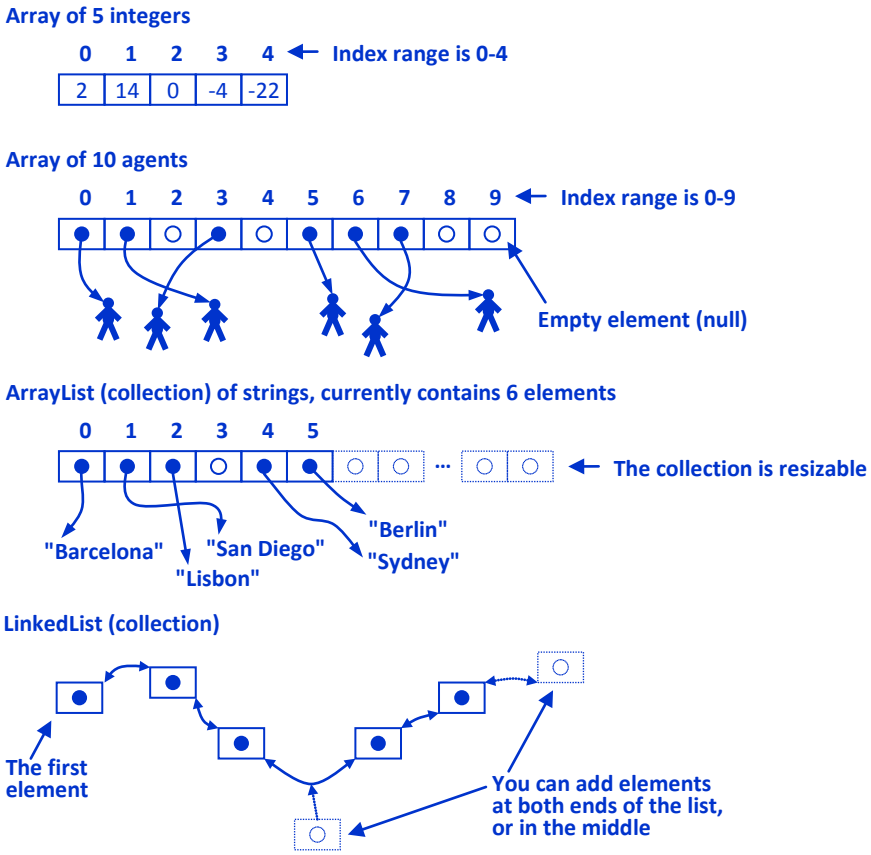
```
trace( "Income: " );
If( income > 10000 ) {
   traceln( "High" );
} else if( income < 1500 ) {
   traceln( "Low" );
} else {
   traceln( "Medium" );
}
```

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; it is described in the chapter on System Dynamics).

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 n[th] element of the **array** can be obtained as **array[n]**. 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. For example, the syntax for obtaining the n[th] 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

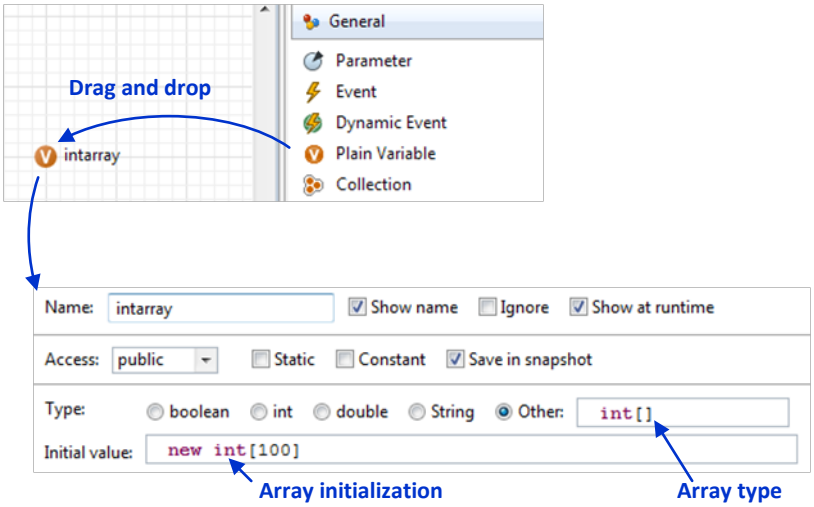
Java arrays are containers with linear storage of fixed size. To create an array you should declare a variable of array type and initialize it with a new array, like this:

```
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[]`, or `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 **Plain 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.



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 can be accessed as:

```
intarray[i]
```

Iteration over array elements is done by index. The following loop increments each element of the array (remember that array indexes are 0-based):

```
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 an "array of arrays". The array initialized as new `double[10][20]` is an 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 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. The structure of a linked list is shown in the Figure.

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 a class Order with the field amount. 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

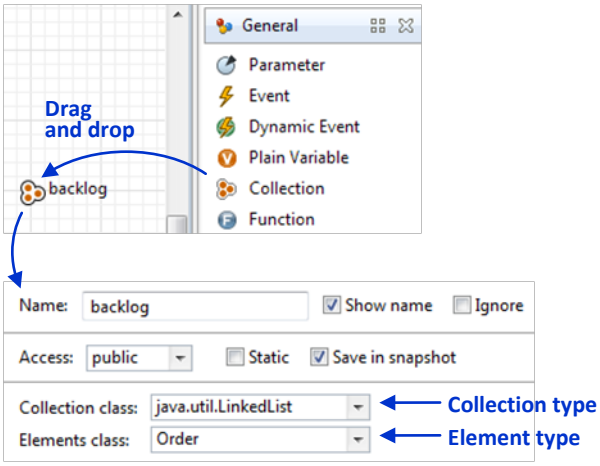
When a new order is received by the distributor it is placed at the end of the backlog:

```
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 declaring collections in active objects and experiments graphically. The **Collection** object is located in the **General** palette. All you need to do is 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.



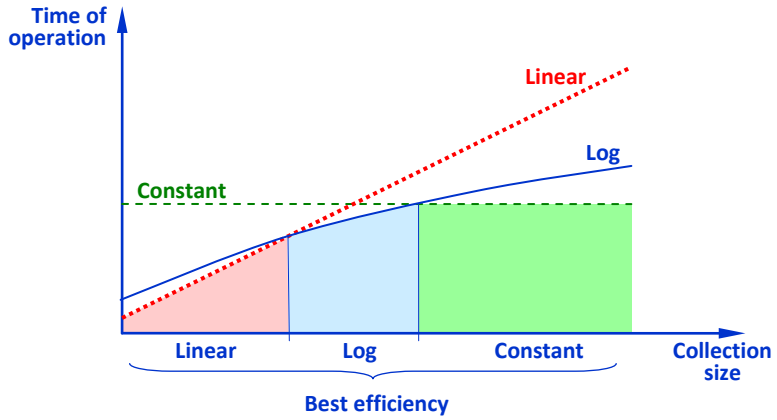
Declaring Java collection graphically

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 following Table 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 a 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 than others

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

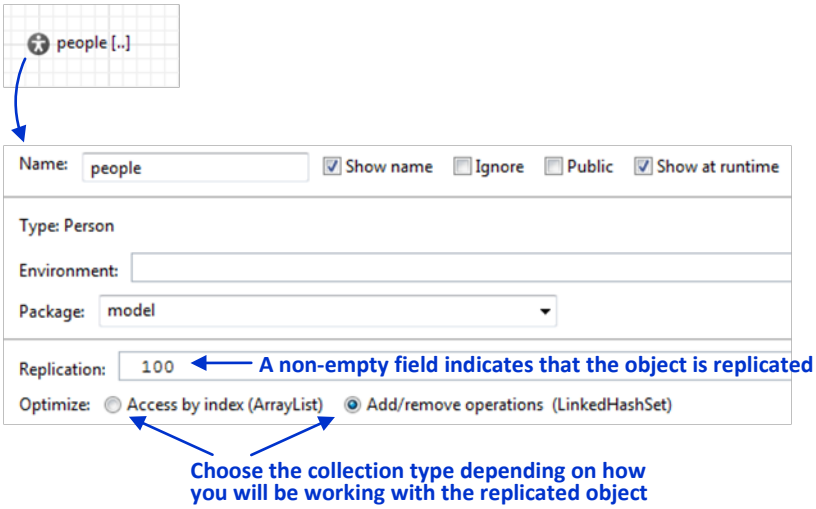
- **HashSet** and **TreeSet** do not support element indexes. For example, it is not possible to get an element at position 32.
- **TreeSet** is a naturally *sorted collection*: elements are stored in a certain order defined by a natural or user-provided comparator.

Replicated active objects are collections too

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

- **ActiveObjectArrayList** – choose this collection type if the set of active objects is more or less constant or if you need to frequently access individual objects by index.
- **ActiveObjectLinkedHashSet** – choose this option if you plan to intensively add new active objects and remove existing ones. For example, if you are modeling 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 at the bottom of the **General** page of the embedded object properties, see the Figure.



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 ) {
  ...
}
```

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 the 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, 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”, or “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, such as: NPV, ROI, and 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 following Table we summarize the naming recommendations for various types of model objects.

Object	Naming rules	Examples
• Plain Java variable • Parameter of active object • Collection • Table function • Statistics • Connectivity object	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". 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.	resetStatistics getAnnualProfit goHome speedup getEstimatedROI setTarget isStateActive 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: • Active object class • Entity class • 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
• Embedded object, i.e. instance of an active object class, 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 • Auxiliary 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 using 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**

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. A *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 the 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>;
```

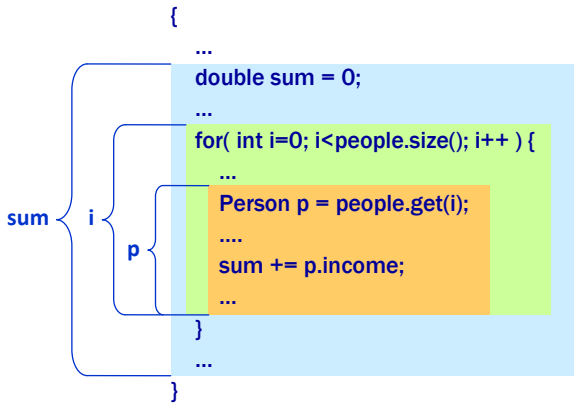
Examples:

```
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 avoiding duplicate names.

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



Code areas where local variables exist and can be used

Function call

Function calls are described in the section Functions. It is worth adding that, even if a function does return a value, it still can be called as a statement if the returned value is not needed. For example, the following statement removes a person from the list of friends:

```
friends.remove( victor );
```

The function remove() returns true if the object being removed was contained in the list. If we are sure it was there (or if we do not care if it was) we can throw away the returned value.

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 with less than 10 clients.

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

The following 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 sections contain more than one statement, they should be enclosed in braces { ... } to become a block, which is treated as a single statement, as shown in the following code. We, however, recommend that you 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 an 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:
   velocity = 10;
   break;
case MOVE_LEFT:
   velocity = -10;
   break;
case STOP:
   velocity = 0;
   break;
case RAISE:
   ...
   break;
case LOWER:
```

```
    ...
    break;
default:
    error( "Invalid command: " + command );
}
```

For loop

In Java there are two forms of **for** loop and two forms of **while** loop. We will begin with the most easy to use so-called "enhanced" form of **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 using this loop because it is more compact and easy to read. It is also 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 object **products** (remember that replicated object is a collection). The following code goes through all products in the portfolio and kills those, where the estimated ROI is less than some allowed minimum:

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

Another example: the following loop counts the number of sold seats in a movie theater. The seats are modeled as Java array **seats** with elements of **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 **for** and **if** statements.

Another, more general, form of **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, this form of loop is the only one applicable here.

Many Enterprise Library objects also offer index-based iterations. For example, this code goes through all entities 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
   Entity e = queue.get(i); //obtain the i-th entity
   if( e.resourceUnits == null || e.resourceUnits.isEmpty() ) { //test
      queue.remove( e ); //remove the entity 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 entity that satisfies our condition, we remove it and we do not need to continue. The break statement is used to exit the loop immediately. If the entity 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 the "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 a **do…while** loop and a **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.

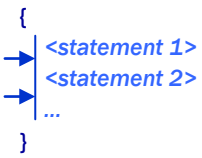
Consider the following example. The local area map is a square 1000 by 1000 pixels. The city bounds on the map are marked with a closed polyline **citybounds**. We need to find a random point within the city bounds. As long as the form of the polyline can be arbitrary we will use the Monte Carlo method, meaning 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 placing the braces on separate lines from the enclosed statements and using *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 a **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 which 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, and 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
}
```

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
```

The AnyLogic Java editor displays the comments in green color, so you can easily distinguish them from code. A 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 necessary 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 temporarily 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, because you may make undesirable changes to the code nearby . Consider the following code fragment. The original plan was to perform an audit for every payment that is over $200,000. The model developer decided to temporarily 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 );
```

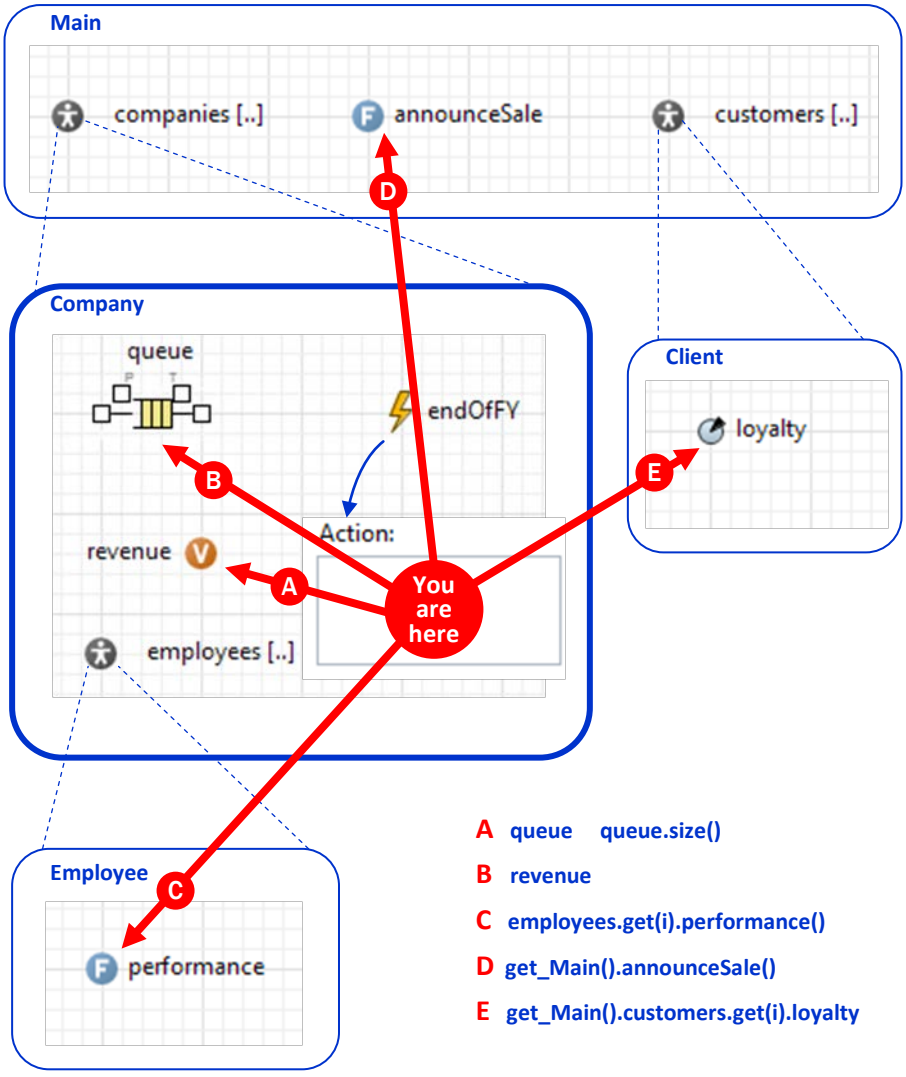
Note that the accident could have been avoided if the modeler had used the block braces with the `if` statement:

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

Where am I and how do I get to…?

In AnyLogic you are not writing the full code of Java classes from the beginning to the end. Instead you are entering pieces of code and expressions in numerous small edit boxes in the properties of various model elements. Therefore it is important to always

understand where exactly are you writing the code (which class and method it belongs to), and how can you access other model elements from there.



Options for collection type of a replicated object

Most of the code you write when you develop models is the code of an active object class. More precisely it is the code of one of the methods of the active object class. No matter whether you are defining an action of an event, setting a parameter of an embedded object, or writing a startup code – you can assume you are writing the code of the current active object class. Therefore the following rules apply (see the Figure):

- The model elements of the same active object class are accessed simply by their names (because they are the fields of the same class). Say you are in the **Action** field of the event **endOfFY** of class **Company**. To access the embedded object **queue** you simply write **queue**. To increase the plain variable **revenue** you write **revenue += amount**. And to restart the event **endOfFY** from its own action you should write **endOfFY.restart(year())**.
- To access an element of an embedded object, you should put dot "." after the embedded object name and then write the element name. For example, to obtain the size of the **queue** object you write **queue.size()**. If the embedded object is replicated, its name is the name of a collection of objects and you should specify exactly which one you want to access. To call the function **performance()** of the employee number 247 among **employees** of the company you should write: **employees.get(246).performance()**.
- To access the container of the current object (the object where this object is embedded), you can call **get_*<class of the container object>*()**. For example, if an object of class **Company** is embedded into **Main**, you should write **get_Main()** to get to **Main**, from within **Company**. Consequently, to call the function **announceSale()** of **Main** you write **get_Main().announceSale()**. There is a function **getOwner()** that also returns the container object, but its return type is **ActiveObject** – the base class of **Main**, **Company**, etc. You will need to cast it to **Main** before accessing the **Main**-specific things: **((Main)getOwner()).announceSale()**. **getOwner()** is useful when we do not know where exactly the current object is embedded.
- To access a "peer" object (the object that shares the container with the current one) you should first get up to the container object and then get down to another embedded object. To access a customer's loyalty from a company in the Figure, we need first to get to **Main** and then to **Client** (with particular index): **get_Main().customers.get(i).loyalty**.

These rules naturally extend to the whole hierarchical structure of the AnyLogic model. You can access anything from anywhere. We, however, recommend developing your models in a modular way as much as possible so that the objects know minimum internals of other objects and setup, and communication is done through parameters, ports, message passing, and calls of "public" functions.

Viewing Java code generated by AnyLogic

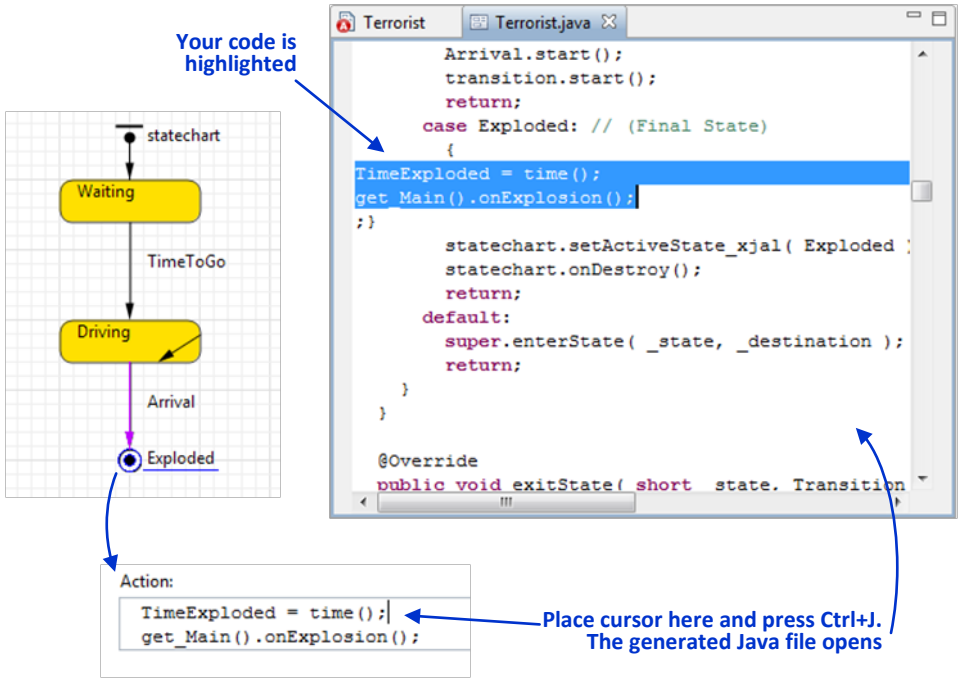
Sometimes it may be useful to see the full Java code generated by AnyLogic to better understand the context your own code is put into. For example, the model compilation may end up with an error, which is not easy to track down by looking just at the user's

code field where the error message is pointing (such errors are frequently caused by missing "{" or "}", missing or wrongly used semicolons, side effect of debug comments, etc). In AnyLogic, you can view the full Java code (.java files) of active object classes and experiments, and easily find the place where your own portion of code is located.

To be able to view the generated Java code you need to compile the model first.

To view your code in the context of the generated Java code:

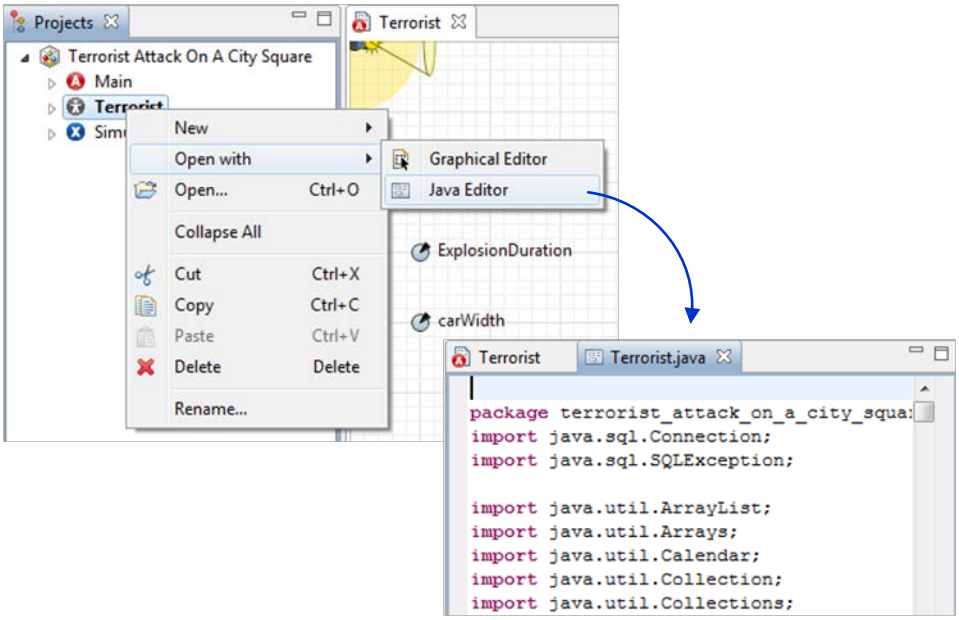
1. Open the text edit field where you entered your code (for example, the **On enter** code of an Enterprise Library object, **Action** code of a statechart transition, dynamic field for the X coordinate of a shape, etc).
2. Press Ctrl+J. The Java file editor opens in read-only mode with your code highlighted.



Opening the generated Java file and locating a particular portion of your code

To view the full Java file generated for an active object class or experiment:

1. Right-click the active object class or experiment in the **Projects** tree and choose **Open with | Java editor** from the context menu.



Opening the Java file generated for a particular active object class

Note that the generated Java files are for viewing purposes only. You cannot edit them in AnyLogic and should not try to edit them externally; the changes will not get back to the model.

Creating your Java classes within AnyLogic model

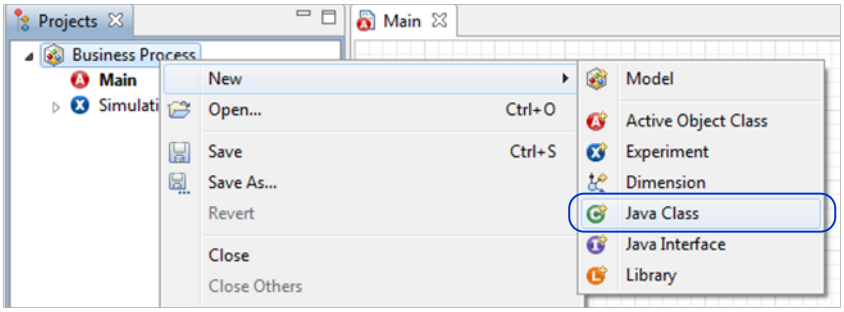
In AnyLogic you can create your own Java classes and include them in the model. This may serve various purposes. For example:

- To define entities in process models with additional fields and/or functions
- To create auxiliary data structures, such as **Location** class considered earlier
- To include classes borrowed from somewhere else in source code form and used in the model, such as problem-oriented optimization algorithms

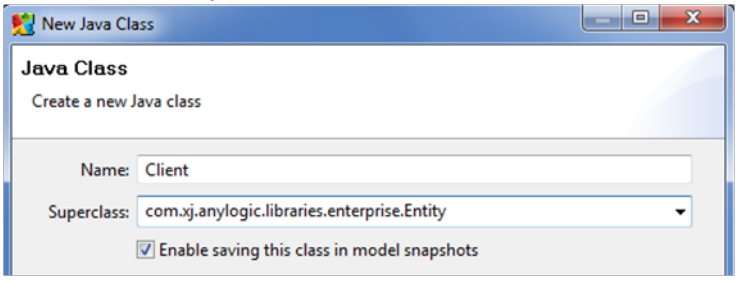
AnyLogic includes a wizard for creation of Java classes and a Java editor with syntax highlighting and code completion (the functionality is inherited from Eclipse).

As an example, we will create a Java class **Client** to be used in a business process model. **Client** will have a **boolean** field **vip** to indicate its importance, a field **complexity** of type **double** that will affect the time needed to complete the current operation requested by the client, and the field **satisfaction** where we will store the client's satisfaction level after the process is completed. As the objects of class **Client** will flow in the business

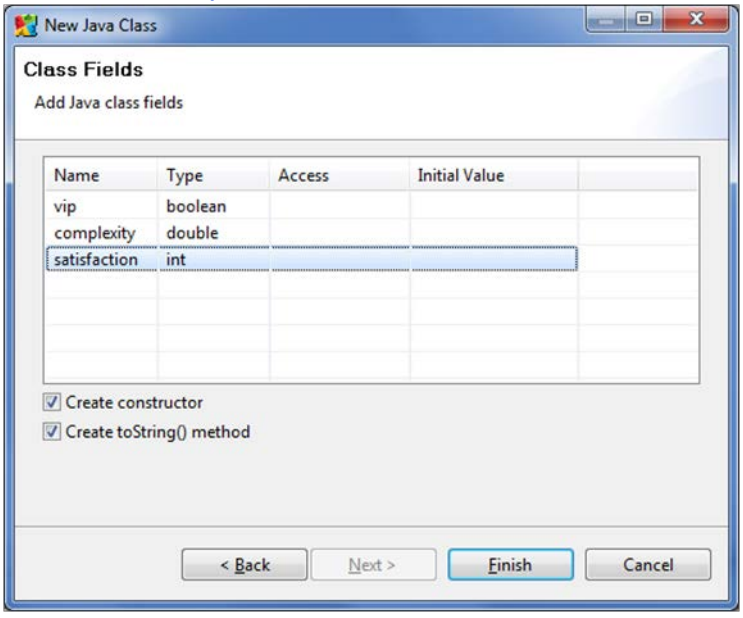
process as entities, they must have all properties of entities. Therefore we will declare the class **Client** as a subclass of AnyLogic class **Entity**.



Java class wizard, step 1



Java class wizard, step 2

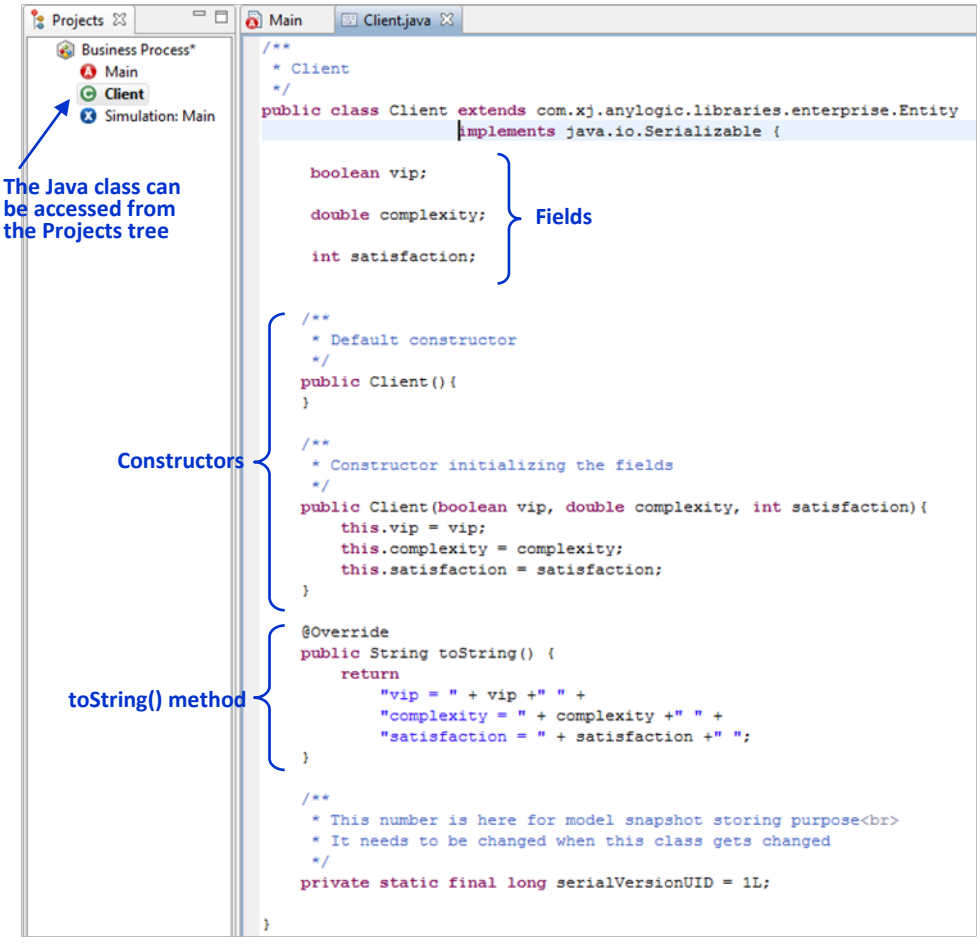


Using AnyLogic wizard to create a new Java class

To create a Java class in AnyLogic model:

1. Right-click the model item in the **Projects** tree (the topmost one) and choose **New | Java class** from the context menu. The **New Java Class** wizard is displayed, as shown in the Figure.
2. Type the name of the class (**Client**) and type or choose the superclass. The class **Entity** of AnyLogic Enterprise Library is among the available choices in the drop-down list. Press **Next**.

Until you use at least one object from the Enterprise Library in the model, the library is not "linked" to the model and its classes are not included in the drop-down list.



Java class created by AnyLogic wizard and opened in Java class editor

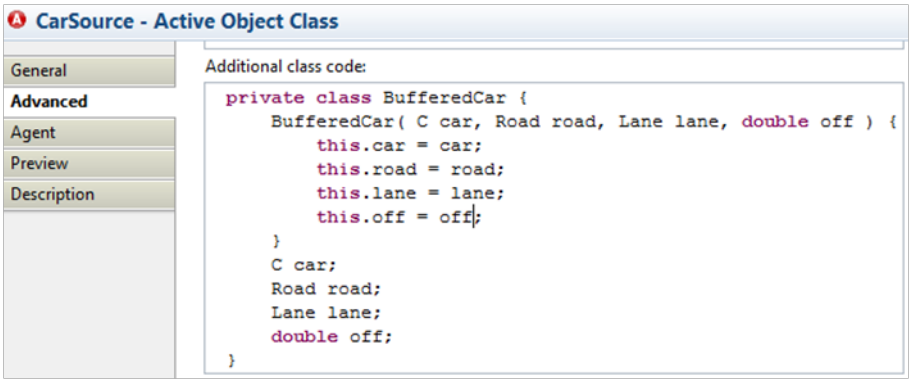
3. In the **Class Fields** page of the wizard enter the three fields that we planned to include in the Client class. Press **Finish**.

When the wizard completes its work, a new Java class is created in the model and opens in the AnyLogic Java class editor, a shown in the Figure. The wizard generates the class declaration and the class body with fields, constructors and some straightforward implementation of the toString() method. You can then edit the class as you wish. The class appears in the **Projects** tree with the green icon. To open the editor, you should double-click the icon. To rename the class simply change its name in the editor and it will change in the **Projects** tree as well.

The Java class created this way belongs to the model package and therefore is in the same Java namespace as the active object and experiment classes. The name of the model package can be found and edited in the **General** page of the model properties.

Inner classes

You can also create Java classes "inside" active object classes. This may make sense when the class is simple and small and used primarily within one active object class. The entire code of the Java class is written in the **Additional class code** field on the **Advanced** page of the active object class properties. For example, in the Figure, the class BufferedCar is defined inside the CarSource active object class where it is used to store cars (entities in AnyLogic Road Traffic library) with some additional information on each car in a buffer.



Defining an inner Java class inside an active object class

In Java, such classes defined inside other classes are called *inner classes*. Should you need to use them outside the "parent" class, you must put the name of the parent class before the name of the inner class, like this: CarSource.BufferedCar.

Linking external Java modules (JAR files)

The external Java code can be linked to the model in compiled form as well. Compiled Java classes are typically distributed in the packaged form in the files with .JAR extension. Numerous JAR files solving various problems can be found on the Internet, both free and commercial.

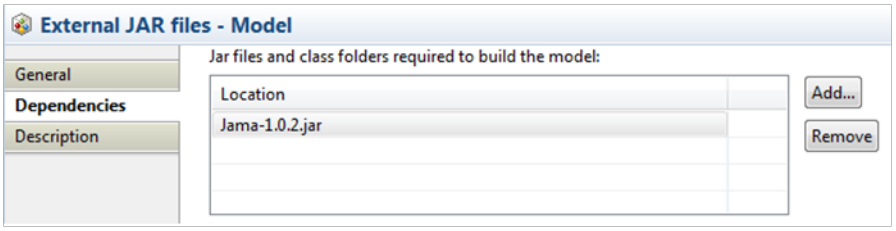
To import an external JAR file (or .class file folder) to the model:

1. Select the model (the topmost) item in the **Projects** tree and open the **Dependencies** page of its properties.
2. Press the **Add** button on the right of the list **Jar files and class folders required to build the model**.
3. In the **Add classpath entry** dialog choose the type of import, enter the file name, and choose whether to copy the file to the model folder (recommended). Press **Finish**.

Once you have imported a JAR file, you can use its classes in the model. The example model External JAR files shows how this can be done using JAMA package – a basic linear algebra package for Java from NIST available for free download.

Please note that when you reference the classes from the imported JAR in your model, their names must be prefixed with the package name. For example, to declare a variable of type Matrix defined in JAMA, you have to write Jama.Matrix where Jama is the imported package name. If you wish to use the class names without prefixes, you must write the "import" statement in the **Import** section of the **Advanced** property page of the active object class where the names are used:

```
import jama.*;
```



Managing the list of external Java modules used in the model