

More AnyLogic & Java Events

Java Types, and Enums

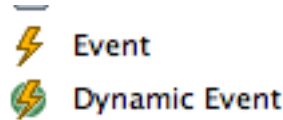
Nathaniel Osgood

CMPT 858

March 15, 2011

Reminder: Rates & Events

- *Rates* and *Timeouts* are associated with types of events in AnyLogic
- Events can also be declared explicitly from the palette



- Dynamic events can have multiple instances
 - Each instance can be scheduled at the same time
 - The instances disappear after event firing
 - Regular (static) events can be rescheduled, enabled/disabled, but can only have one scheduled firing at a time
- There are some subtleties with events

Built-In Events

- In addition to handling occurrence of explicit events, models automatically support “catching” certain “built-in” types of events
- To handle these events, code is inserted into certain handler areas for each of different sorts of classes

Example: Built-In Events (Agent 1)

The screenshot displays the AnyLogic Advanced interface. On the left is a project tree with 'Person' selected. The main workspace shows a statechart for the 'Person' agent with states 'FreeWandering' and 'GoToWater'. A red text box explains the 'Handler' concept. Below, the 'Person - Active Object Class' properties panel is shown, with a red arrow pointing to the 'On Arrival' event handler field.

“Handler”: Code is executed when the specified event (e.g., arrival at a destination, message arrival) occurs.

Person - Active Object Class

General: Space type: Continuous Discrete GIS

Advanced: Environment defines initial location

Agent: Initial coordinates: X: [] Y: []

Parameters: Movement parameters: Velocity: [] Rotation: []

Description: On Arrival: []
On Message Received: `infectionStatechart.receiveMessage(msg);`
On Before Step: []
On Step: []

Example: Built-In Events (Agent 2)

The screenshot displays the AnyLogic Advanced software interface, specifically the 'Person - Active Object Class' configuration window. The main workspace shows a state transition diagram for a person's behavior. The diagram includes a 'FreeWandering' state (yellow oval) with a self-loop labeled 'NewDir'. A transition labeled 'GotThirsty' leads from 'FreeWandering' to a 'GoToWater' state (yellow oval). A transition labeled 'drinkingPeriod' leads from 'GoToWater' back to 'FreeWandering'. The diagram also shows a 'thirsty' variable (yellow oval) and a 'headingRandom' variable (blue oval). The 'Person - Active Object Class' configuration window is open, showing the 'General' tab with the following settings:

- Name: Person
- Ignore:
- Agent: Agent
- Generic: Generic
- Startup Code:
- Destroy Code:

The left sidebar shows the project hierarchy, including 'Main', 'Person', 'Simulation', 'Person', 'Person.java', 'Male.java', 'Male', 'Main', and 'Elephant'. The right sidebar shows the 'Model' palette with various shapes and components like Line, Polyline, Curve, Rectangle, Round Rect..., Oval, Arc, Pixel, Text, Image, Group, Button, Check Box, Edit Box, Radio Buttons, Slider, Combo Box, List Box, File Chooser, Progress Bar, CAD Drawing, GIS Map, Connectivity, Enterprise Li..., and More Libraries... The bottom status bar shows 'Problems' and a table with columns 'Description' and 'Loca'.

Example: Built-In Events (Main)

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a statechart for a 'Main' object. The statechart includes a state 'FreeWandering' (represented by a yellow oval) and a state 'GoToWater' (represented by a yellow rectangle). Transitions are labeled with events: 'GotThirsty' (a lightning bolt icon) and 'NewDir'. The 'FreeWandering' state has a self-loop transition labeled 'NewDir'. The 'GoToWater' state has a transition back to 'FreeWandering' labeled 'GotThirsty'. The statechart is connected to a 'behavior' input and several parameters: 'drinkingPeriod', 'thirsty', 'headingRandom', and 'headingToWater'.

The 'Main - Active Object Class' properties window is open, showing the 'Startup Code' field with the following code:

```
environment.deliverToRandom("Infection");
```

The 'Destroy Code' field is empty. The 'General' tab is selected, and the 'Name' field is set to 'Main'. The 'Agent' and 'Generic' checkboxes are unchecked. The 'Ignore' checkbox is checked.

Types in Java

- Types tell you the class of values from which a variable is drawn
- In Java we specify types for
 - Parameters
 - Variables
 - Return values
 - Class Fields
- Typically, we encode information described by elements of one or more different types

Types & Legal Operations

- For a given type, only certain “operators” can be used e.g.
 - e.g. a double precision value can be divided, multiplied, turned into a String etc.
 - A boolean can be tested for truthhood, turned into a String, etc.
 - A (reference to a) string can be used to
 - Extract prefixes or suffixes, find the length, concatenated, etc.
 - An enum’s values can be turned turned into a String, converted to integer, etc.

Java Primitive Types

- These are built-in to the Java language
- Primitive types in Java are the following
 - boolean
 - double
 - short (small integer)
 - int
 - char
 - byte
 - long
 - float

Non-Primitive Types

- Most types we used are not primitive types
- These are defined either
 - In our code
 - In the standard Java libraries

Why Types?

- Like specifying dimensions for an object (e.g. L, L³/T), specifying types lets us
 - Know what we're dealing with (so we know what to do with it)
 - Avoid making a silly mistake
 - e.g. attempting to divide a number by a (reference to) a Person
 - Absent types, it is likely that these mistakes wouldn't be identified until runtime
 - If we don't happen to test that portion of the program, we won't be aware of the error
 - With types, we can discover these errors when we are building the program -- during our "Build"

Type Coercion (“Casting”): Why

- Sometimes we have something that is a member of one type, but that can be logically converted to another type
- Examples:
 - We have a double-precision value and we wish to convert it instead to an integer (by dropping fractional component)
 - We have an integer (or a double, char, boolean, etc.) and wish to convert it to a string
 - (Subtyping) We have an `ActiveObject` that we know is a `Person` and wish to treat it as a `Person`

Type Coercion (“Casting”): How

- To “cast” a value v in one type to another type, the following syntax is used

`(TargetType) v`

- Examples:

`println((String) age)`

`((Female) item).stateChart.isStateActive(Pregnant)`

`((int) age) + 1`

Parameterized Types

- Sometimes a type (A) is defined in terms of another type “(B)”
 - This allows the definition of A to take & give back information specific to type B
 - e.g. methods take an A as a “parameter”, or return a B, etc.
- Common example: Collections depending on the type of their content
- We say that the type A is “parameterized by” type B
- We can describe such “Parameterized Types” using Java “Generics”
 - Syntax used: `A`

Examples of Parameterized Type (Generics)

- A resource pool depending on what resources are included
(ResourcePool<MyResourceUnit>)
- An “array list” (like an extensible vector) depending on the type of the elements
(ArrayList<Person>)
- Hypothetical: A Pair defined in terms of the first and second element
 - Pair<String, Double>

Example of a Parameterized Type

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a process flow diagram with steps: release, moveToExit, networkExit, and sink. A 'network' resource is connected to three sub-resources: doctor, procRoom, and scope. The 'doctor' resource is highlighted with a blue box.

The Properties window for the selected 'doctor' resource is shown below the diagram. It is titled 'doctor - NetworkResourcePool' and contains the following configuration:

- General:** Name: doctor, Show Name, Ignore, Public, Show At Runtime,
- Parameters:** Type: NetworkResourcePool<T extends Resource>, Generic parameters: ResourceUnit
- Package:** com.xj.anylogic.libraries.enterprise
- Resource type:** Moving
- Capacity defined:** Directly, By home shape, By table over time
- Capacity*:** 5
- Speed:** 10
- ^New resource unit:** new ResourceUnit()
- ^On new unit:** (empty field)
- ^On seize:** (empty field)

The left sidebar shows a project tree with folders for Ophthalmology Department, MainPhase1, MainPhase2, and MainPhase3, each containing sub-elements like Ports, Embedded Objects, and Presentation. The bottom left shows a Problems window with a table for Description.

Enums: Why

- Often we desire in our models to encode categorical information
- We can certainly encode such information using integers (or shorts, etc.)
 - e.g.
 - Male=0, Female=1
 - Province: NL=0,NB=1,PEI=2,QC=3,etc.
- Example using variables

```
int sex
int province
```

Problem: This is fragile

– We could easily mistake a value “0” as encoding either Males or Newfoundland/Labrador

– e.g.

- Reversing order of parameters given to a method, or entered into a file
- Assigning value for one to another, due to a poorly named values

• e.g.

sex=province

Enums in Java

- Enums let us
 - Give names to such information
 - Refer to the names in our code
 - Convert the names (where necessary) into their associated values
 - Compare names
 - Define operations on names

Simplest Examples

- `enum Sex { Male, Female };`
- `enum Province { NL, NB, PEI, QC, ON, MB, SK, AB, BC};`
- Variables using enum:
 - `Sex sex`
 - `Province province`
- Causes error: `sex=province`