

Rails and trains

The AnyLogic Rail Library allows you to efficiently model, simulate and visualize any kind of rail transportation of any complexity and scale. Classification yards, rail yards of large plants, railway stations, rail car repair yards, subways, airport shuttle trains, rail in container terminals, trams, or even rail transportation in a coal mine can be easily yet accurately modeled with this library.

The Rail Yard library integrates well with AnyLogic [Enterprise](#) and [Pedestrian Libraries](#). This means you can readily combine rail models with the types of models supported by these libraries, namely: auto transport, cranes, ships, passenger flows, warehouses, manufacturing or business processes, and so on.



A Railway station model – Rail Library works together with Pedestrian Library

While the library supports detailed and accurate modeling (dimensions of individual cars, exact topology of tracks and switches, accelerations and decelerations of trains), the simulations it produces are very high performance. This is particularly important when you use the optimizer to identify the best management policies or most efficient operations.

The Rail Library supports 2D and 3D animation of tracks, switches, and rail cars. The 3D Objects palette contains ready-to-use 3D objects for locomotives, several types of freight cars, and passenger cars. Since the Enterprise Library and the Pedestrian

Library also support 3D animation, you can now easily create full dynamic 3D models of subway and railway stations, airport shuttles, or any other system where rail transportation is combined with pedestrian flow (see the Figure).

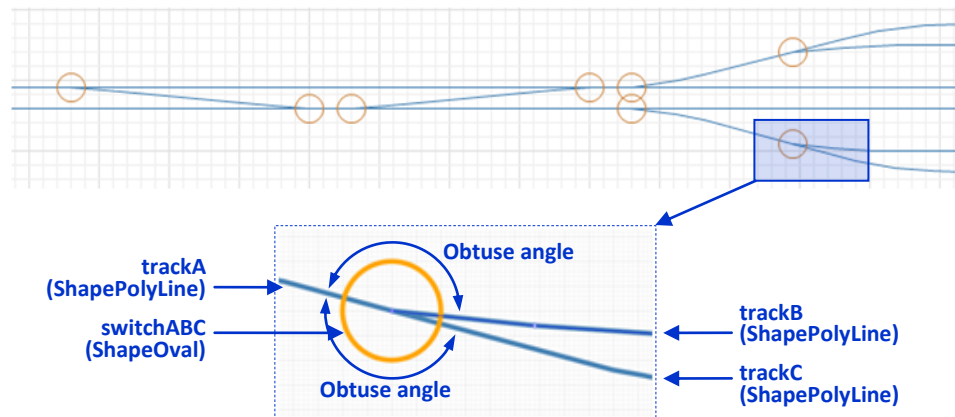
The two main components of a rail model are the [rail topology](#) and the [operation logic](#). We will now discuss these in turn.

Defining the rail topology

The rail topology is specified by a group of shapes representing rail *tracks* and *switches*. Tracks are defined by polylines (**ShapePolyLine**), while switches are defined by circles (**ShapeOval**). Those shapes can either be drawn manually using the AnyLogic graphical editor or [created programmatically](#), for example, by reading the layout from a database or a file.

In a well-defined rail layout:

1. There are no shapes in the group other than polylines (**ShapePolyLine**) and circles (**ShapeOval**)
2. Each circle (switch) must contain exactly three polyline end points (tracks ends) within 2 pixels from the circle center (see the Figure)
3. At each switch there must be at least two obtuse angles out of three between the track ends. The switch determines the routes based on those angles.



A fragment of rail topology near a railway station. Defining a switch

Train collisions at switches are detected automatically and errors are signaled. But note that the Rail Library will not automatically detect track crossings (i.e. places

where two tracks cross each other without a switch) and it is the user's responsibility to make sure there are no train collisions in such places.

The characteristics of the track polylines and circles you use at design time (such as color, line width, or circle size) do not matter; at runtime they will be modified according to the scale settings and the color scheme defined in the **RailYard** object.

The curved track segments should be approximated with multiple point polylines. If a rail yard is drawn manually it may be convenient to have a CAD drawing or an image of the yard as a background, [lock](#) it, and draw the polylines above the image.

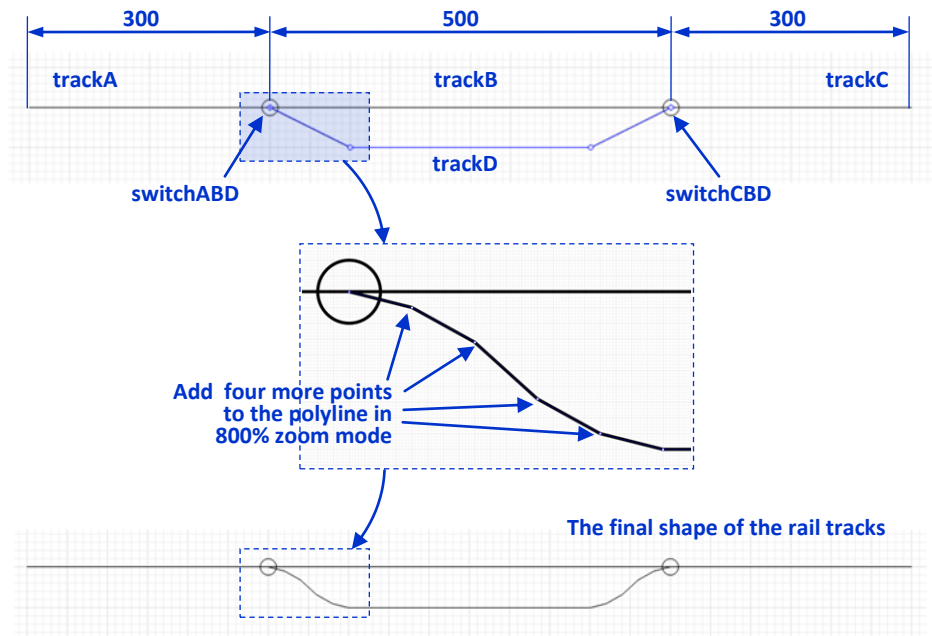
Having drawn the tracks and switches, you should group them, pass the group to the parameter **Rail yard shapes** of the **RailYard** object, and specify the scale in which you have drawn the yard (pixels per meter).

Example: A very simple rail yard

Let us draw a very simple rail configuration such as a main track and a branch line that may be used to wait while another train to passes by.

► Draw a rough sketch of the tracks:

1. Open the **Presentation** palette. Double-click the **Polyline** item and draw a horizontal polyline with one segment of length 300 (click at the start point and double click at the end point). Call it **trackA**.
2. Ctrl+drag the polyline to create a copy. Place the left end of the second polyline at the right end of the first one. Call the new polyline **trackB**.
3. Extend **trackB** to the right so its length is approximately 500 pixels.
4. Ctrl+drag **trackA** again to create another copy of it and place it to the right of **trackB** as its continuation. Call the third polyline **trackC**. Now you have a straight line of length $300+500+300 = 800$ consisting of three polylines.
5. Draw a small circle (with radius, say, 10) with the center at point where **trackA** connects with **trackB** (you may need to select **trackA** to see where it ends). Call the circle **switchABD**. Set the **Fill color** of the circle to **No fill**.
6. Create a copy of that circle and place it where trackB connects with trackC. Call the second circle **switchCBD**.
7. Draw the fourth polyline starting from the center of **switchABD** and ending at the center of **switchCBD**. Let this polyline initially have three segments as shown at the top of the Figure. Call it **trackD**.



Drawing a very simple rail yard

Now let's make the shape of **trackD** more round (closer to what it would be in reality) by adding more points to the polyline.

► Adjust the shape of the trackD to make it more realistic

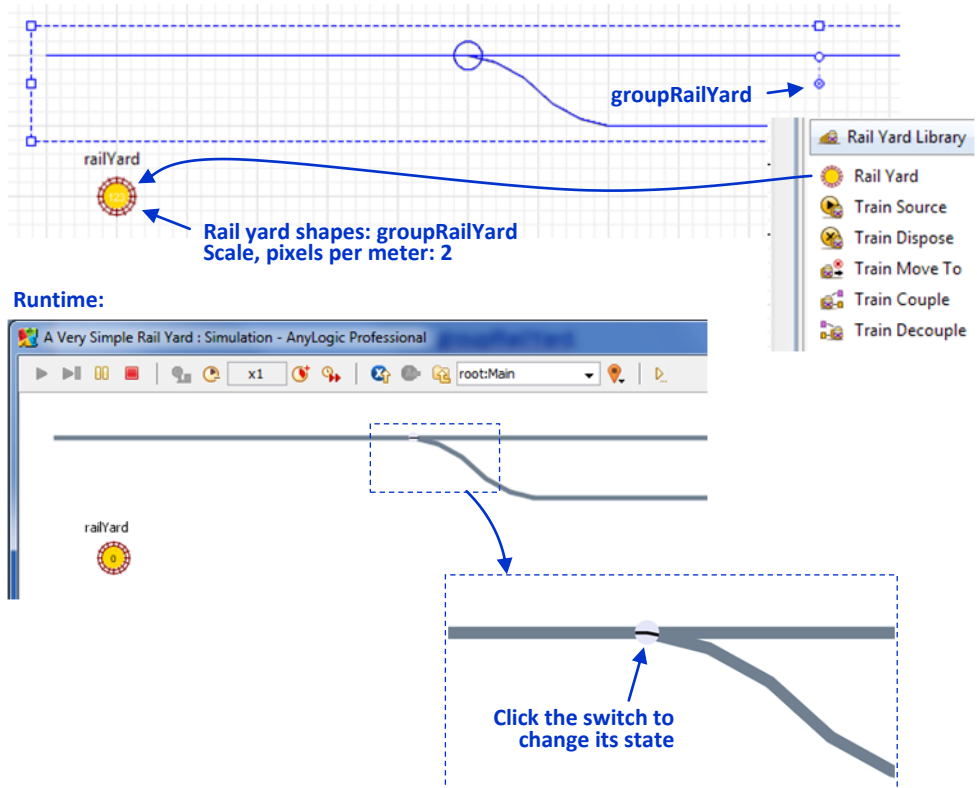
8. Double-click the header of the graphical editor window (it should read **Main**) to extend the editor to a full screen view.
9. Place the mouse cursor at **switchABD**, hold the Ctrl key and rotate the mouse wheel until the picture is zoomed to 800%. You will see the pixel-wise grid and will be able to move the polyline points by one pixel step.
10. Double-click **trackD** to add a point and adjust the point position. Add four points as shown in the middle of the Figure.
11. Adjust the shape of the right end of **trackD** near **switchCBD**.
12. Return to 100% zoom and double-click the Window header again to restore the original windows layout.

Now that you have finished the layout drawing, you should group all those shapes, drop the **RailYard** object to the canvas, and tell it which group contains the rail yard shapes. We will assume the scale of the drawing is 2 pixels per meter.

► Group all shapes and add the RailYard object:

13. Select all the polylines and circles you have drawn.

14. Right-click the selection and choose **Grouping | Create a group** from the context menu. A group is created. Change the name of the group to **groupRailYard**.
15. Open the **Rail Library** palette and drag the **RailYard** object to the canvas.
16. In the General page of **railYard** properties type **groupRailYard** in the **Rail yard shapes (group)** parameter. Remember to use code completion.
17. Set the **Scale, pixels per meter** parameter to **2**.
18. Run the model.
19. Zoom in the animation of the rail yard. Click on a switch to change its state.



Adding the RailYard object. Animation of tracks and switches

Notice that the width and color of the tracks as well as the size and color of the switch circles at runtime are modified according to the settings in the bottom section of **RailYard** object parameters. (As mentioned above, these override the values you may have used in design.)

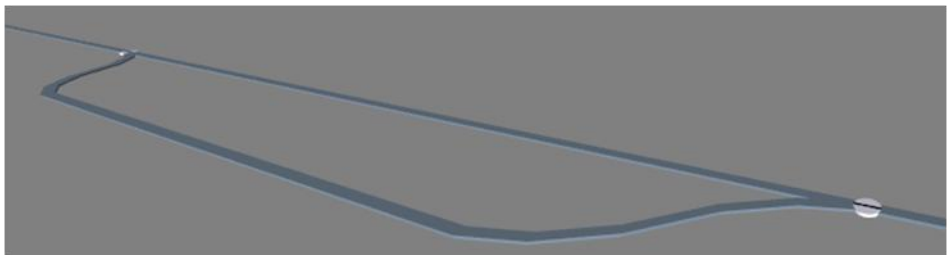
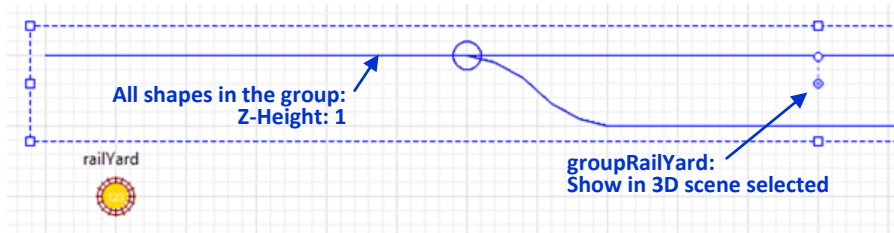
Now the rail yard is fully defined and ready to be used. We will illustrate how to let the trains into the yard later, in the section [Defining the operation logic of the rail yards](#).

3D animation of rail yards

The next thing is to enable 3D animation of the yard. This is done simply by making the group of shapes a 3D group (selecting its **Show in 3D scene** property). There is one thing you should check, though. If the polylines and switches were originally two-dimensional, their Z-height will automatically be set to 10 pixels, which may be undesirable. You should then modify the Z-height.

► **To enable 3D animation of the rail yard:**

20. Click the **groupRailYard** and select **Show in 3D scene** on the **General** page of its properties.
21. Open the **3D** palette and drag the **3D Window** to the canvas below the drawing. Change its size to, say, 1100x250 pixels.
22. Run the model. The tracks and switches have significant Z-height, which we need to fix.
23. Return to the editor. Right-click the group and choose **Select group contents** from the context menu. All rail yard shapes get selected.
24. In the **Advanced** property page of the multiple selection set **Z-Height** to 1.
25. Run the model again and view the 3D picture of the yard.



3D animation of the simple rail yard

Creating rail yards programmatically

For large rail yards it may make sense to read the layout of the yard from an external source and create tracks and switches programmatically instead of drawing them manually.

❖ Example: Creating a rail yard by code

We will assume you have read the yard configuration from an external source and know the coordinates of switches and of all points of tracks (remember that curved tracks should be approximated by polylines). In this example we will create the polylines and circles using AnyLogic API, add them to a group and then dynamically set the group to the **Rail yard shapes** parameter of the **RailYard** object.

► Follow these steps:

1. Open the **Rail Library** palette and drop the **RailYard** object to the canvas at (50, 50). Do not change any of its parameters.
2. Open the **Presentation** palette and drag the **Group** object to, say, (150, 50). Call the group **groupRailYard**.
3. Open the **General** palette and drag the **Function** object to the canvas near the rail yard object. Call the function **createRailYardShapes**.
4. Enter the following code in the **Code** page of the function properties:

```
//create track A
ShapePolyLine trackA = new ShapePolyLine();
trackA.setNPoints( 2 );
trackA.setPoint( 1, 200, 0 );
trackA.setPos( 0, 0 );
groupRailYard.add( trackA );
```

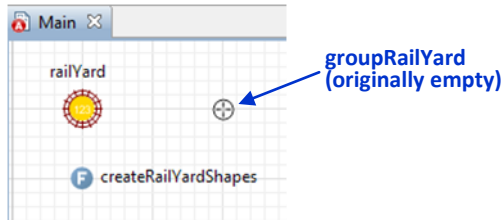
```
// create track B
ShapePolyLine trackB = new ShapePolyLine();
trackB.setNPoints( 2 );
trackB.setPoint( 1, 400, 0 );
trackB.setPos( 200, 0 );
groupRailYard.add( trackB );
```

```
// create track C
ShapePolyLine trackC = new ShapePolyLine();
trackC.setNPoints( 3 );
trackC.setPoint( 1, 100, 50 );
trackC.setPoint( 2, 400, 50 );
trackC.setPos( 200, 0 );
groupRailYard.add( trackC );
```

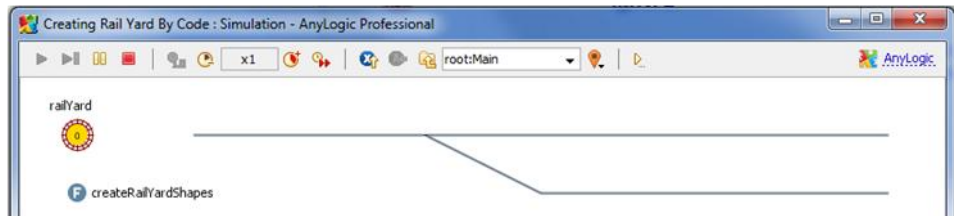
```
//create switch at the end of trackA
ShapeOval switchABC = new ShapeOval();
switchABC.setRadius( 10 );
switchABC.setPos( 200, 0 );
groupRailYard.add( switchABC );
```

```
//refresh the group parameter of the RailYard
railYard.set_railYardShapes( null ); //just in case there were other shapes before
railYard.set_railYardShapes( groupRailYard );
```

5. Click anywhere in the canvas to display the active object properties. On the General page type the following in the **Startup code** field:
`createRailYardShapes();`
6. Run the model. The three tracks and a switch appear at (150,50).



Runtime:



A rail yard created by code

In this model, we did not initially provide a group to the **RailYard** object, so the creation of tracks and switches was delayed. On startup, the function `createRailYardShapes` executes. It creates all shapes, adds them to the group, and then sets the group to the **RailYard**. When the `railYardShapes` parameter is changed, **RailYard** (re)builds the yard.

Please note that the position of the shapes is relative to the group where they are included. The polyline point coordinates are relative to the start point of the polyline, therefore 0 point always has coordinates (0,0).

Java class Track

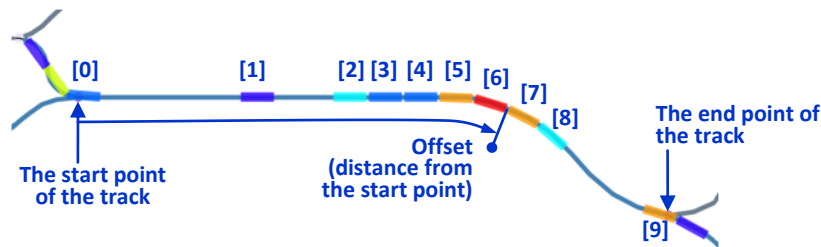
For each polyline in the group of rail yard shapes (in other words, for each continuous track of the yard with no switches in the middle), the **RailYard** object creates a Java object of class `Track`. You will not typically need to use the objects of class `Track` and their methods: most references to tracks in the Rail Library process flowcharts are done by the polyline names; in addition, the **RailYard** object has a number of functions

like `isTrackEmpty()`, or `getCarOnTrack()`, which also accept polyline names. However, in some cases the **Track** API might be useful.

To obtain the **Track** object that corresponds to a particular polyline you should call the function `getTrack()` of the **RailYard** object, for example:

```
Track track123 = railYard.getTrack( polyline123 );
```

The track has a start point (the point 0 of the polyline) and an end point, and therefore has an orientation. The exact position on the track is defined by the distance from the start point of the track, often referred as *offset*.



A track

The track knows about the presence of switches at the either end, if there are any. If there is no switch at one of the sides (an *open-ended track*), and a rail car exits the track at that side, the car leaves the rail yard model.

The track knows all cars that are (fully or partially) located on it, and you can obtain those cars using the **RailYard** object or **Track** object API. The track in the Figure will say that it has 10 cars on it, and will return them by index as shown.

At runtime, the color of an empty track is (optionally) different from the color of the non-empty track. Colors are set up at the bottom section of **RailYard** parameters.

Here are some methods of class **Track** (see [Rail Library Help](#) for the full list):

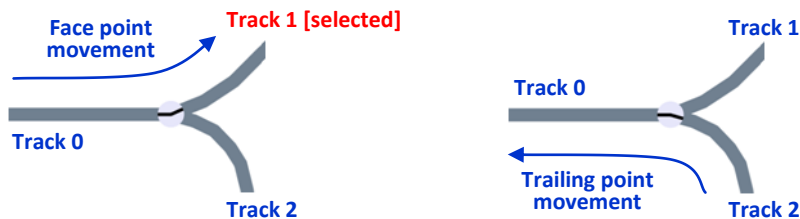
- **boolean isEmpty()** – tests if the track is empty, i.e. there are no cars that are (even partially) on the track. Returns true if the track is empty, false otherwise
- **int getNCars()** – returns the number of cars on the track (including those that are only partially on this track)
- **RailCar getCar(int index)** – returns a car on the track at a given position (**index**) counted from the beginning of the track, or null if there is no such car. All cars count: moving, standing, coupled, and cars that are only partially on this track.
- **double getFreeSpace(boolean fromstart)** – tests the availability of space on the track. If there are no cars on the track, returns **infinity**. If there are cars, returns the distance from the track start or end point (depending on the parameter

fromstart) to the nearest car. If there is a car partially entered or exited the track at a given side, returns a negative value.

- **Switch getSwitch(boolean atend)** – returns the switch (**Switch** object, not the circle shape) at the beginning or at the end of the track, depending on **atend** parameter.
- **double getLength()** – returns the length of the track in meters.
- **ShapePolyLine getPolyline()** – returns the polyline of the track.

Java class Switch

For each circle in the group of rail yard shapes, the **RailYard** object creates a Java object of class **Switch**. **Switch** models a two-way railroad switch that connects three tracks: 0, 1, and 2. Depending on its state, the switch will direct the trains coming from track 0 (*face point movement*) to either track 1 or track 2. The trains coming from track 1 or 2 (*trailing point movement*) will always proceed to track 0 regardless of the state of the switch and will always force the switch to the corresponding state.



A switch

At runtime, the color of a busy switch (the switch with a car over it) is (optionally) different from the color of an idle switch. Colors are set up at the bottom section of **RailYard** parameters. The state of the switch is shown with a thin line drawn over the switch circle. You can toggle the switch state directly by clicking within the circle or via API.

You won't need to use the objects of class **Switch** and their methods too often; the Rail Library object **TrainMoveTo** is capable of setting up the switch states automatically when the train moves along a defined route. In addition, the **RailYard** object has a number of functions like **setSelectedTrack()**, which accept circle names.

However, in some cases the **Switch** API might be useful. To obtain the Switch object that corresponds to a particular circle you should call the function **getSwitch()** of the **RailYard** object, for example:






```
Switch switch123 = railYard.getSwitch( oval123 );
```

Here are some methods of class **Switch** (see **Rail Library Help** for the full list):

- **Track** `getSelectedTrack()` – returns the currently selected track (track 1 or 2).
- **setSelectedTrack(Track track)** – select a given **track** (which should be either 1 or 2). If a car is over the switch, signals error.
- **toggle()** – toggles the selected tracks.
- **Track** `nextTrack(Track from)` – based on the state of the switch, returns the next track, given the switch is approached from a given track **from**.
- **boolean** `isTrailingPoint(Track from)` – tests if movement from a given track **from** through the switch is a trailing point movement or face point. Returns **true** if trailing point, **false** if facing point.
- **Track** `getTrack(int index)` – returns the track connected to the switch with a given **index**.
- **ShapeOval** `getOval()` – returns the oval shape of the switch.

Defining the operation logic of the rail model

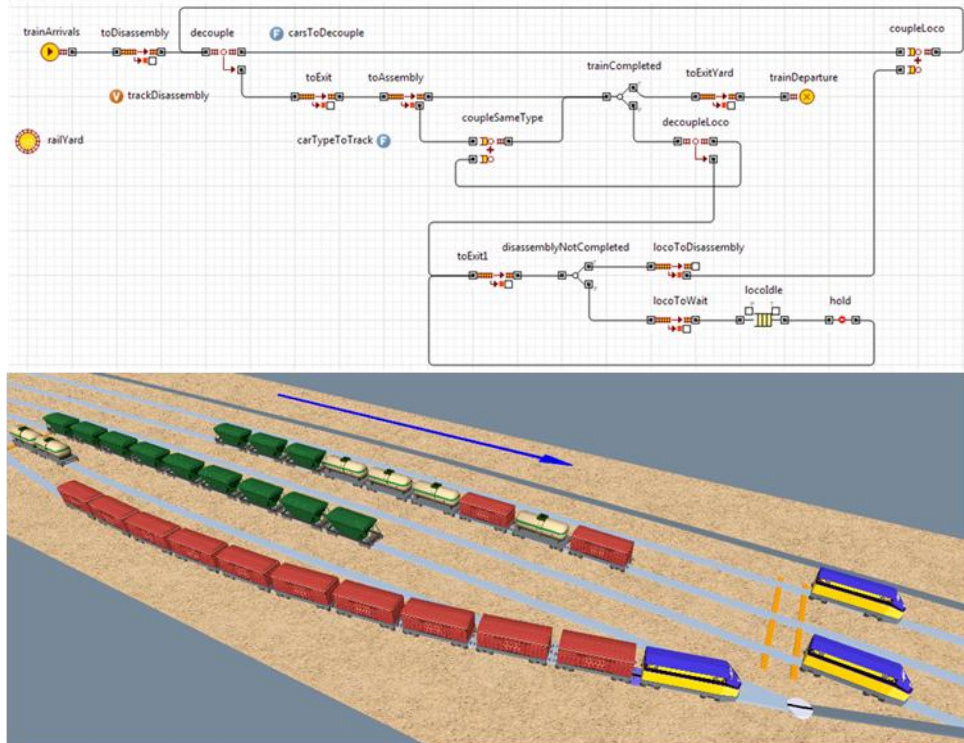
The operational logic definition in the Rail Library is based on an easy-to-use process flowchart methodology. The entity moving in a rail yard flowchart is a **train** – any sequence of coupled rail cars. There are five flowchart objects in the library that describe all the actions specific to trains:

Object	Description
 TrainSource	Creates trains, performs initial setup and puts them in the yard. Starts any rail yard process flowchart. Supports several types of arrivals scheduling.
 TrainDispose	Removes trains from the model; either those that have exited the yard via an open-ended track, or by "deleting" a train that is still on a track.
 TrainMoveTo	Controls movement of trains. Can calculate routes and set switch states as the train goes along the route. Supports acceleration and deceleration.
 TrainCouple	Couples two trains that "touch" each other into one train. Has two queues and supports coupling of trains on several tracks simultaneously and independently.
 TrainDecouple	Decouples cars from the incoming train and creates a new train from those cars. Supports extreme cases when 0 or all cars are decoupled.

These five objects can be mixed in a flowchart with objects from the [Enterprise Library](#), such as **Delay**, **SelectOutput**, **Hold**, **Seize**, **Release**, **Queue**, etc. The latter are used to define time delays, make decisions, and manage sharing of the rail yard's resources – tracks and switches.

For example, if a part of the yard should be locked to allow a train to pass through, you may associate a resource with it. Then a train that enters that part would need to seize the resource, and the other trains would wait in the queue of the **Seize** object. For the same purpose you can use the **Hold** object and the pair **RestrictedAreaStart/End**.

The **SelectOutput** object can be used in the rail yard process flowcharts to choose between the different process branches, and **Delay** can naturally model the stops durations or duration of operations such as coupling/decoupling or loading/unloading.



A simple classification yard and the flowchart describing its operation

The rail yard operation flowcharts created with AnyLogic Rail Library tend to be very compact. For example, the full logic of a classification yard where arriving trains with different types of cars get disassembled and trains containing cars of the same type

are assembled can be defined by a flowchart with less than 20 objects as shown in the Figure.

Example: Train stop

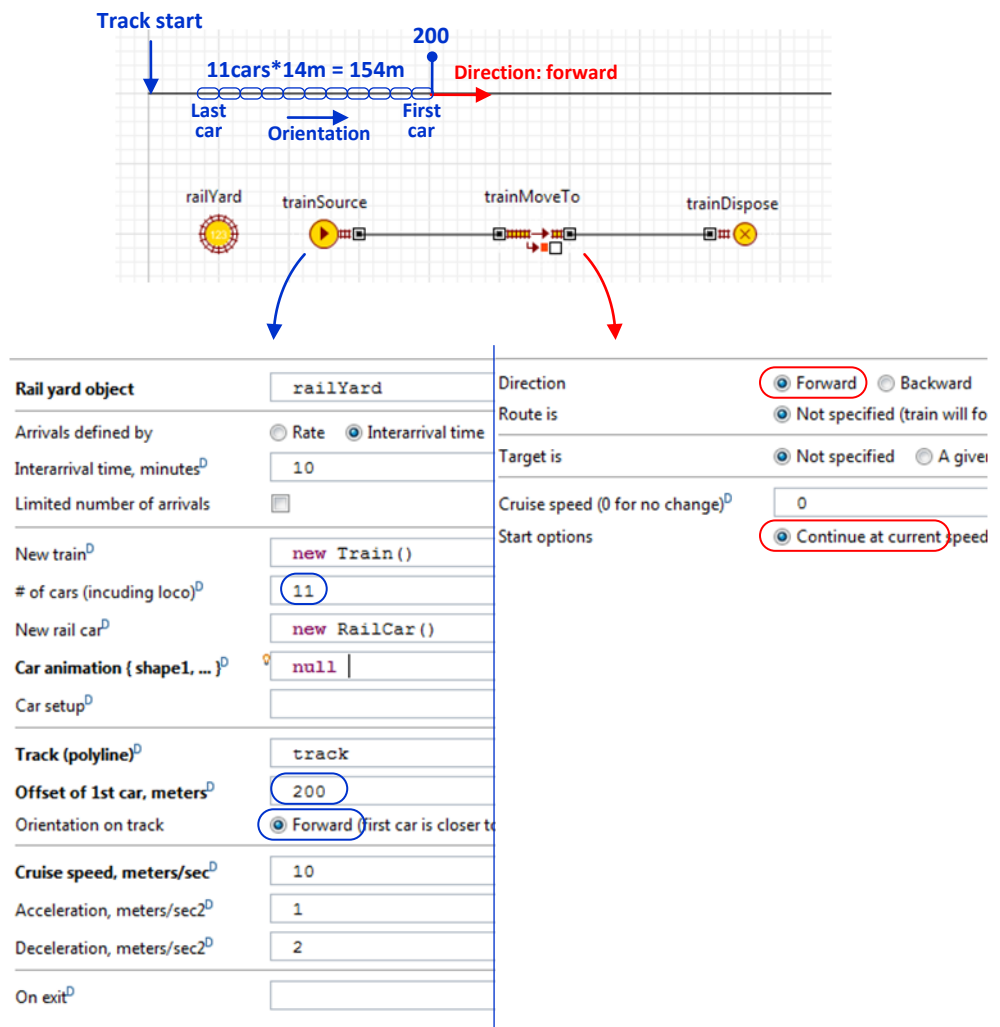
We will now create a very simple model with one straight rail track and no switches. Passenger trains will stop in the middle of the track for one minute and then will continue moving in the same direction.

► Create a train and let it move along the track:

1. Draw a straight polyline with one segment from (0,50) to (1100, 50). Call it **track**.
2. Right-click the polyline and choose **Grouping | Group** from the context menu. A group containing just that polyline is created. Call it **groupRailYard**.
3. Open the **Rail Library** palette and drag the **RailYard** object to, say, (50, 100).
4. Set the parameter **Rail yard shapes** of the **railYard** to **groupRailYard**.
5. From the same palette drag the three objects: **TrainSource**, **TrainMoveTo**, and **TrainDispose**. Place them in a sequence and connect as shown in the Figure (to connect two ports double-click one of them, drag the connector to the other port and click there).
6. Select **trainSource**. Set the following parameters of that object:
Rail yard object: **railYard**
Track (polyline): **track**
Offset of 1st car, meters: **200**
Cruise speed, meters/sec: **10**
7. Click on the model item (the topmost one) in the **Project tree**. Check that the **Time units** parameter is set to **minutes** in the **General** page of its properties.
8. Run the model. Until the model time is 10, nothing happens. At 10 (and then at 20, 30, etc.) a train appears at the beginning of the track, moves to the right and exits the track.

Look at the parameters of **trainSource**.

We have set up the **trainSource** to create a train in our rail yard and place it on the **track** with the front side of the first car located 200 meters from the beginning of the track. The train will have forward orientation relative to the track (parameter **Orientation on track**), so the rest of train will be between the beginning of the track and point 200. At the time of creation, the train must be fully on the track, so you need to make sure there is enough space. By default, a train created by **TrainSource** has 11 cars, and the default length of a **rail car** is 14 meters. As $11 \cdot 14 = 154 < 200$, there is enough space.



The simplest train process flowchart: train appears and moves along the track

The default arrival schedule of **TrainSource** is one train every 10 minutes, so at times 10, 20, 30 and so on. Remember, the time units are meters. Our **trainSource** will create a new train at the same place, and we need to make sure the previous train has moved and gotten out of the way.

The *cruise speed* that was specified in **trainSource** (10 meters per second, which equals 36 km/h) will be the default speed for train movement.

The train has a *current velocity* (the actual velocity at which it moves at the current moment of time). At the time of creation by **TrainSource** the current velocity of a train

is set to its cruise speed as if the train has just been moving with the cruise speed. This is done to enable immediate continuation of movement at the same velocity. You can change the velocity at any time by calling the function `setVelocity()` of the train.

Now look at the parameters of `trainMoveTo`. You will see that the **Start options** is set to **Continue at current speed**, and the **Direction** is **Forward**, which means the train will appear in the model moving to the right at 36 km/h. No target and no routing options are specified in `trainMoveTo`, so the train will naturally exit the `track` at its right end and the train entity will exit the `trainMoveTo` object and will be consumed by `TrainDispose`.

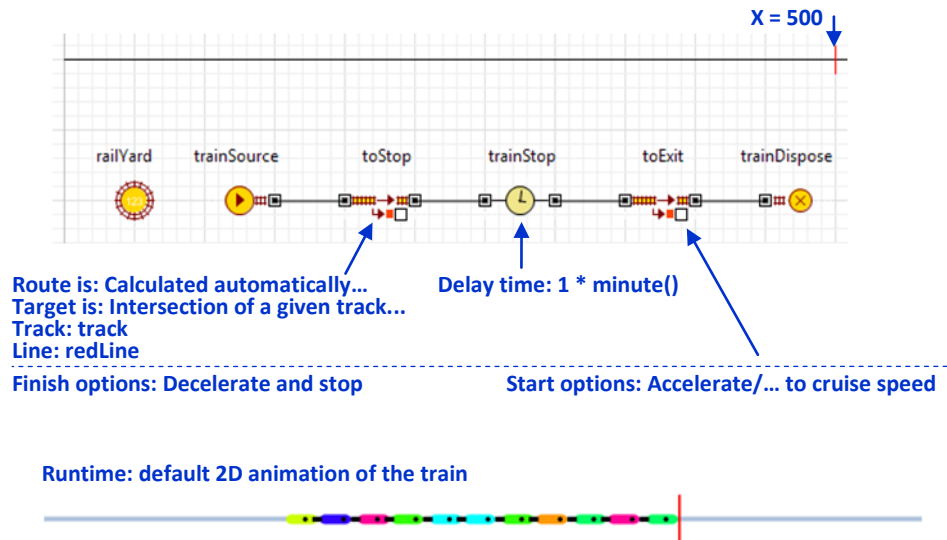
Now we will add a short train stop.

► **Add the train stop**

9. Open the Presentation palette, double-click the line object and draw a short vertical line crossing the track at X=500. Call the line `redLine`.
10. Select `trainMoveTo` object and set the following parameters:
Route is: Calculated automatically from current to target track
Target is: Intersection of a given track and a line
Track: `track`
Line: `redLine`
11. Run the model. You will see that now trains disappear once they reach the red line.
12. Modify the flowchart. Disconnect the `trainDispose` object from `trainMoveTo` and move it to the right to give space for two new objects.
13. Rename the `trainMoveTo` object to `toStop`.
14. Open the **Enterprise Library** palette and drop the **Delay** object in the flowchart after `toStop`. Call the object `trainStop`. Set its parameter **Delay time** to **1 * minute()**
15. Open the **Rail Library** palette again and drop another **TrainMoveTo** object between `trainStop` and `trainDispose`. Call it `toExit`. Connect the ports of all objects.
16. Run the model again. Now the train stops at the red line, waits there for one minute, and then continues to the right end of the track.

In the first `TrainMoveTo` object, which is now called `toStop`, we specified the target position of the movement, which is graphically defined by the red line. (Note that the line is *not* a part of the rail yard shapes group.) Having reached the red line the train entity exits the `toStop` object and enters the object `trainStop` of type **Delay**. While the train entity waits in the `trainStop` object, the train does not move. Then, after a one minute delay, the train entity exits `trainStop` and enters the second `TrainMoveTo` (`toExit`) which moves it further.

In this flowchart we mixed the objects from the Rail Library with the objects from the Enterprise Library. This is possible because the **Train** object that is generated by **TrainSource** is a subclass of **Entity** – a generic entity that can be handled by process flowchart objects from the Enterprise Library.



The complete process flowchart for the Train stop model. Default 2D animation.

If, however, you look into the parameters of **toExit**, you will see that **Start options** is set to **Continue at current speed**. So you may ask: why did the train start moving when its speed had been 0 at the train stop? The answer is: we did not ask the train to decelerate and stop during the first movement. Therefore, the train exited **toStop** object with its original velocity (10 meters/sec), which "remained set" during the stop, although the train was not physically moving. This is obviously an unrealistic behavior: the train cannot stop immediately and cannot immediately gain speed. This is not an accurate representation of the physics of trains, though it may be useful for certain models with a higher level of abstraction. Now, however, let's add deceleration before the stop and acceleration afterwards.

► Add acceleration / deceleration

17. Select **toStop** and set **Finish options** to **Decelerate and stop**.
18. Select **toExit** and set its **Start options** to **Accelerate/decelerate to cruise speed**.
19. Run the model. Play with different cruise speed, acceleration and deceleration values (they are set up in **TrainSource**). Try to slow down the simulation.

As a final part of this example, we will add the 3D animation.

► **Add 3D animation**

20. Click the **groupRailYard** and select **Show in 3D scene** on the **General** page of its properties.
21. Right-click the group and choose **Select group contents** from the context menu. All rail yard shapes get selected.
22. In the **Advanced** property page of the multiple selection set **Z-Height** to 1.
23. Open the **3D** palette and drag the **3D Window** to the canvas below the flowchart. Change its size to, say, 1000x250 pixels.
24. Run the model. Move the camera to view the track and the trains better.

The default 3D animation of trains is very schematic. However, you can always use a custom 3D object as animations of your rail cars. Some ready-to-use objects are contained in AnyLogic **3D Objects** palette.

► **Add custom 3D objects for the rail cars**

25. Open the **3D Objects** palette. Drag the **Passenger Car** and the **Locomotive** to anywhere in the canvas.
26. Select **trainSource**.
Set the parameter **Car animation {shape1, ... }** to **{locomotive, passengerCar}**.

Although there are 11 cars in our train, we can provide only two shapes in the list: the last shape will be used for all remaining cars. This is fine in our case: all cars except for the first one are passenger cars.

27. Run the model. View the 3D animation of the train. Notice that the cars heavily intersect.

It happens because the 3D objects for cars have different lengths (as real cars do), but **TrainSource** uses just one default length of the **RailCar**, which is 14 meters. We need to specify the custom lengths of our cars. This can be done in the parameter **Car lengths, meters** of **TrainSource** or in the **Car setup** code.

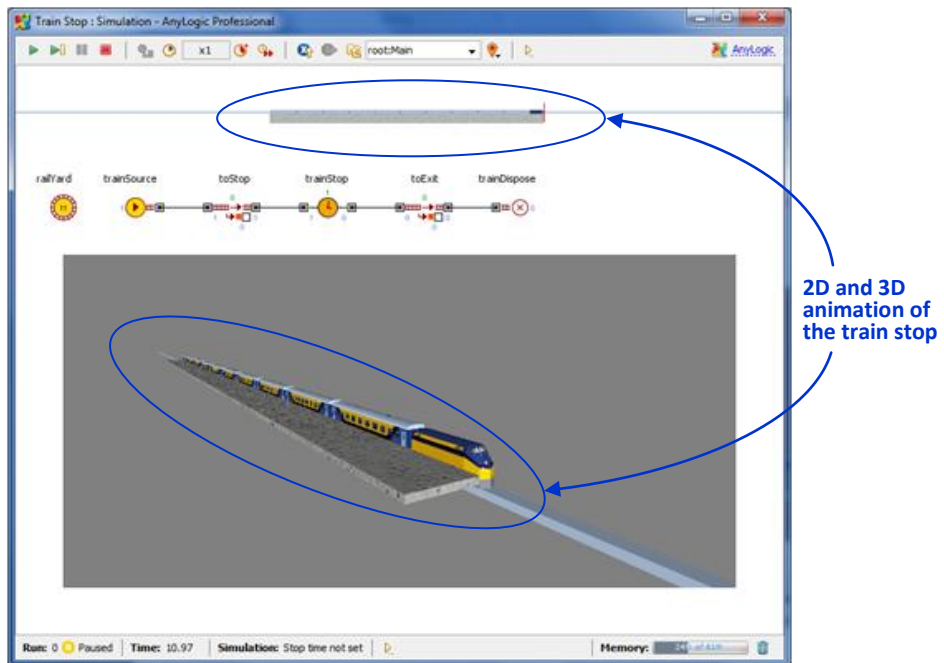
28. Select **trainSource**. Set the parameter **Car lengths, meters** to **{14, 27}**.

In the AnyLogic **3D Objects** palette the locomotive and all types of freight cars have a length of 14 meters, and the passenger car is 27 meters long.

29. As our train is now longer, we need to adjust its initial location so all cars fit on the track. Set the parameter **Offset of 1st car, meters** to: **14 + 27*10 + 10**.
(The last 10 is just for safety: numeric errors can always occur.)
30. Run the model.

► **Draw the platform at the train stop**

31. Open the 3D palette and drag **Rectangle** to the canvas.
32. Edit the rectangle size and position this way (you may need to change zoom):
X: 265, Y:52, Width: 285, Height: 10. The rectangle should appear just below the track to the left of the red line.
33. Set the **Line color** of the rectangle to **No color** and **Fill color** to **concrete** texture.
34. On the Advanced page of the rectangle properties set:
Z: 2
Z-Height: 1.
35. Run the model again.



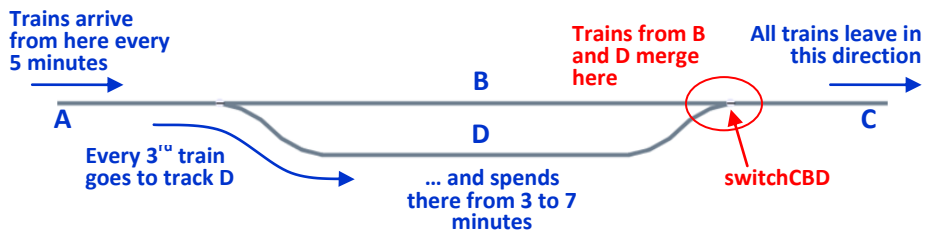
A screenshot of the Train Stop model with an adjusted camera position

Example: Ensuring safe movement of trains

The AnyLogic Rail Library enables you to define rail topology and control the movement of trains but ensuring the safe train movement is the task of the modeler. The library will only detect train collisions at switches (a train arrives at a switch while another train is over the switch) or when a train collides with another train moving or standing on the same track.

In this example, we will show how you can use AnyLogic Enterprise Library resources to model a simple safety control system. The implementation of the safety control in a real rail yard will contain various communication protocols between the train and the dispatchers, traffic lights, and other elements that are outside the Rail Library's scope. Here we are only suggesting one of the ways to map train management policy to AnyLogic modeling language.

We will use the rail layout from the model [A very simple rail yard](#) considered earlier in this chapter. Assume trains are arriving from the left every 5 minutes. Every third train goes to the lower track, stays there from 3 to 10 minutes and then continues in the same direction. All other trains just move along the main track from left to right. As the trains merge at the [switchCBD](#), collisions are possible and we need to manage the traffic to avoid the collisions.



The problem definition for the example Ensuring safe movement of trains

First we will define the traffic without the safety control.

► Create the rail configuration and set up the traffic without safety control:

1. Repeat the steps 1-19 of the example [A very simple rail yard](#) (or just copy the group of the rail yard shapes and the [railYard](#) object from there).
2. Open the Presentation palette and draw two short vertical lines: [redLine](#) crossing the [trackB](#) at X=800, and [blueLine](#) crossing [trackD](#) at X=750 (see the Figure).
3. Open the **Rail Library** palette and drag the **TrainSource** object to the canvas at, say, (150,250). Set the following parameters of the trainSource:
Rail yard object: [railYard](#)
Interarrival time, minutes: 5
of cars, including loco: 8
Track (polyline): [trackA](#)
Offset of 1st car, meters: 120
Cruise speed, meters/second: 10
4. Open the **Enterprise Library** palette and drag the **SelectOutput** object to the right of [trainSource](#). Call it [notEachTrthird](#) and change its parameters:

Select True output: If condition is true

Condition: `self.in.count() % 3 != 0`

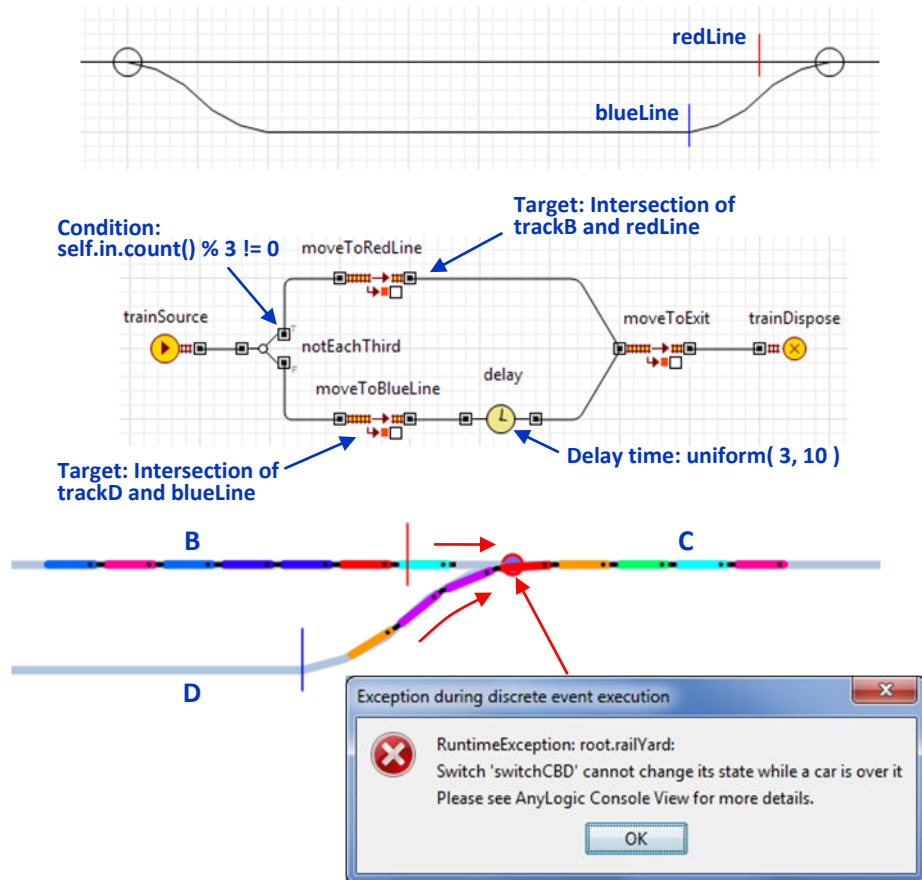
We need to treat each 3rd incoming train differently, and this **SelectOutput** object is here to distinguish between each 3rd train and all other trains. The easiest way to do it is to find out how many trains have entered the **SelectOutput**, divide it by three, and look at the remainder. For every 3rd train the remainder will be 0.

In the dynamic parameters of embedded active objects you can always access the variable `self` – this is the embedded object itself. So, the expression `self.in.count()` written in a dynamic parameter of `notEachTrhird` returns the number of entities that have entered `notEachTrhird` via its `in` port.

5. Continue the flowchart by adding two **TrainMoveTo** objects after the two outputs of `notEachTrhird`.
6. Rename the **TrainMoveTo** at the T (**true**) output port of `notEachTrhird` to `moveToRedLine` and set the following parameters:
Route is: Calculated automatically...
Target is: Intersection of a given track and a line
Track: `trackB`
Line: `redLine`
7. Similarly, rename the **TrainMoveTo** at the F (**false**) port to `moveToBlueLine` and set the following parameters:
Route is: Calculated automatically...
Target is: Intersection of a given track and a line
Track: `trackD`
Line: `blueLine`
8. Add a **Delay** object from the Enterprise Library after `moveToBlueLine` and set its **Delay time** to `uniform(3, 10)`.
9. Add the third **TrainMoveTo** and connect the outputs of `moveToRedLine` and `delay` to its input. Call it `moveToExit`.
10. Finish the flowchart with **TrainDispose**.
11. Run the model. Watch the trains going through the yard. The first several trains may do fine, no collisions occur.
12. Run the model in virtual time mode (as fast as possible). The model will stop almost immediately with an error "Switch 'switchCBD' cannot change its state while a car is over it".

As long as the time the trains spend at `trackD` is nondeterministic and can be greater than the time interval between trains (5 minutes), collisions at `switchCBD` will inevitably occur. The error message you are getting (see the Figure) is typical for that

type of collision: while the train exiting **trackD** is still moving over the **switchCBD**, another train from **trackB** approaches the switch and tries to change its state (in this case as a result of trailing point movement).



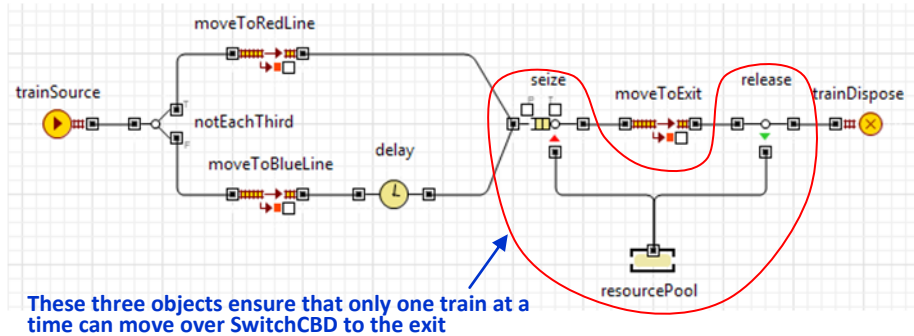
The unsafe train movement flowchart and the collision detected by the library

We will now add a safety management mechanism to our model. The idea is to associate an Enterprise Library resource with the part of the rail yard shared by the conflicting train flows and have trains seize the resource before they enter that part and release it when they exit.

► Add safety control

13. Modify the flowchart by embracing the **moveToExit** object (the one where conflict actually occur) with the pair **Seize / Release**, both connected to a **ResourcePool** as shown in the Figure. Those three objects can be found in the **Enterprise Library** palette. Leave all parameters with at default values.

14. Run the model again. Try virtual time mode. No collisions occur.



The problem definition for the example Ensuring safe movement of trains

Associating **ResourcePool** with a shared section of a rail yard is not the only way of avoiding conflicts. Alternatively you could use the object **Hold** or a pair **RestrictedAreaStart/End**.

The model of the safety control system is now in place. You may have noticed, however, that the movement of trains is not very realistic: they start and stop without acceleration and deceleration, the speed changes from 0 to 10 m/s (cruise speed) immediately. You already know that you can request **TrainMoveTo** to apply acceleration and deceleration. However, in this particular example, trains that move along the main track (A-B-C) should only decelerate before the **redLine** if the exit is seized by another train; otherwise they should just continue going at cruise speed. How can we implement such optional deceleration?

One approach is to determine whether the exit is busy (seized by another train) at some point before the **redLine**. If it is busy, the train will decelerate and stop at the **redLine**, where it will wait for the resource to become available. If the exit is free, the train will immediately seize it and proceed to the exit without deceleration.

► Add acceleration and deceleration of trains:

15. Draw one more line crossing **trackB** at X=700. Change its color to yellow and call it **yellowLine**.
16. Insert two objects between the **true** port of **notEachThird** and the input port of **moveToRedLine**: **TrainMoveTo** and **SelectOutput** as shown in the Figure. Call them **moveToYellowLine** and **exitIsBusy**.
17. Set the following parameters of **moveToYellowLine**:
Route is: Calculated automatically...
Target is: Intersection of a given track and a line

Track: trackB

Line: yellowLine

18. Change the parameters of **exitIsBusy**:

Select True output: If condition is true

Condition: `resourcePool.idle() == 0`

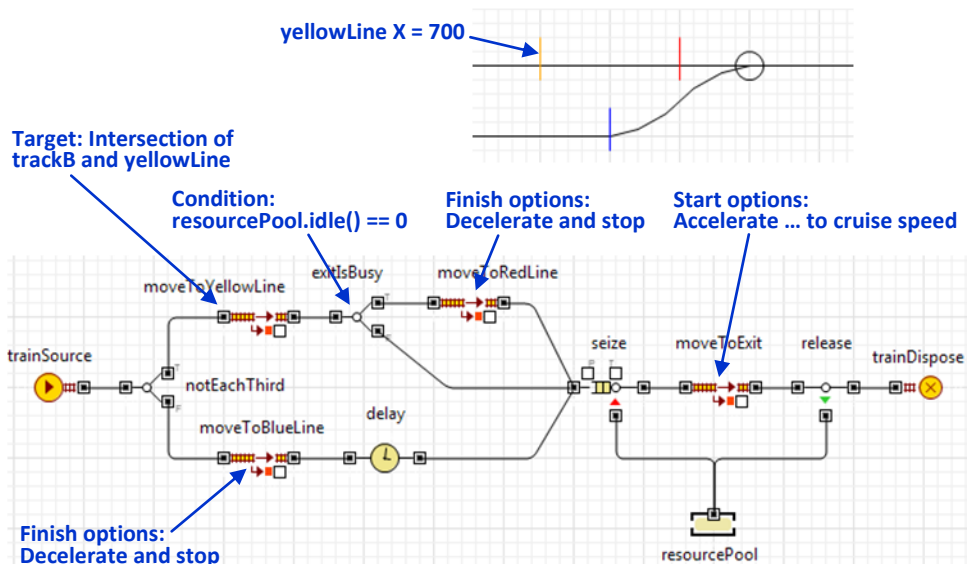
19. Change the start and finish options of the three **TrainMoveTo** objects created earlier:

moveToRedLine: Finish options: Decelerate and stop

moveToBlueLine: Finish options: Decelerate and stop

moveToExit: Start options: Accelerate/decelerate to cruise speed

20. Run the model. You may need to slow down the simulation to see how the trains accelerate.



The flowchart modified to model acceleration and deceleration

To check the state of the exit area of the rail yard in advance, we added an intermediate point on **trackB** marked with the **yellowLine**, and then we divided the movement into two stages: move to the **yellowLine** (without deceleration), and if the exit is idle, seize it immediately and proceed to the exit without deceleration. Otherwise (if the exit is busy), decelerate, stop at the **redLine** and wait there until the exit gets free. Note that:

- Continuous movement can be modeled by multiple **TrainMoveTo** objects put in a sequence (like the movement along the main track A-B-C is modeled by **moveToYellowLine** and **moveToExit**).

- **TrainMoveTo** with the start option **Accelerate/decelerate to cruise speed** will either take a train already going at its cruise speed (in which case acceleration will not be done), or will accelerate a standing train – see [moveToExit](#).
- The same **TrainMoveTo** object can take trains at different locations and bring them to the same or different destinations. For example, a train entering [moveToExit](#) can be either at a red, blue, or yellow line, but will be routed to the exit.

Example: Simple classification yard

In this example, we will begin by creating a very simple classification yard. Arriving trains will contain cars of two types: tanks and box cars. The tanks will be moved to the departure track and left there, and the box cars will continue to the exit. Once the departure track accumulates more than six tanks, they will be driven away by another locomotive. Once again, we will use the rail layout created in the example [A very simple rail yard](#).

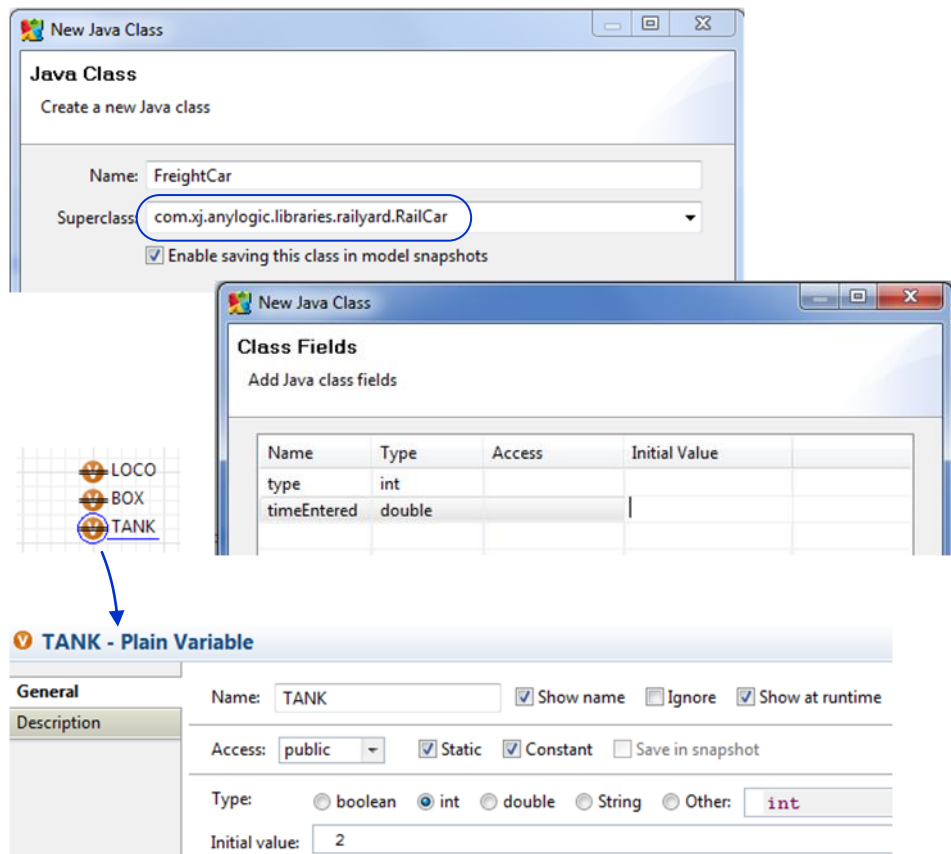
Next, we will create a custom rail car class with information about the car type and with built-in statistics, and we will show how to access the properties of individual rail cars. We will also show how to *couple* and *decouple* rail cars.

► Create a custom class for rail cars:

1. Repeat the steps 1-19 of the example [A very simple rail yard](#) (or just copy the group of the rail yard shapes and the **railYard** object from there).
2. In the **Projects** tree right-click the model (topmost) item and choose **New | Java class**.
3. In the new Java class wizard enter the name of the new class: **FreightCar** and choose the base class **com.xj.anylogic.libraries.railyard.RailCar** (this is available in the drop-down menu). Press **Next**.
4. Add two fields: **type** of type **int** and **timeEntered** of type **double** (see the Figure). Press **Finish**. The Java editor opens for the new class. We do not need to modify the class implementation, so you can close the editor.
5. Return to the editor of **Main** where you have the rails and the **railYard** object.
6. Define three integer constants for three car types. To do it open the **General** palette and drag the **Plain variable** item to the canvas. Call it **LOCO** and set the following properties:
Static: yes
Constant: yes
Type: int
Initial value: 0

7. Ctrl+drag the **LOCO** constant to create a copy. Call the copy **BOX** and set its initial value to **1**.
8. In the same way create the third constant **TANK** with value **2**.

It is Java code convention to capitalize all letters in constants' names. This way you can easily distinguish them from variables.



A custom class for rail cars and three constants for three car types

► **Create a simple train flowchart and set up train creation:**

9. Open the **3D Objects** palette and drag three objects from there: **Locomotive**, **Box Car**, and **TankCar**. Set the scale of all three objects to 200% (remember that the **Scale** parameter of the **railYard** is set to 2 pixels per meter).
10. Create a flowchart of three Rail Library objects: **TrainSource**, **TrainMoveTo**, and **TrainDispose**.

11. Set the following parameters of **trainSource**:

Rail yard object: **railYard**

of cars (including loco): **6**

New rail car:

new FreightCar(carindex == 0 ? LOCO : (carindex < 3 ? TANK : BOX), time())

Car animation: { locomotive, tankCar, tankCar, boxCar, boxCar, boxCar }

Track polyline: **trackA**

Offset of 1st car: **100**

Cruise speed, meters/sec: **5**

12. Run the model. At time 10 (and then at 20, 30, and so on) a new train appears on the main track and proceeds to the right.

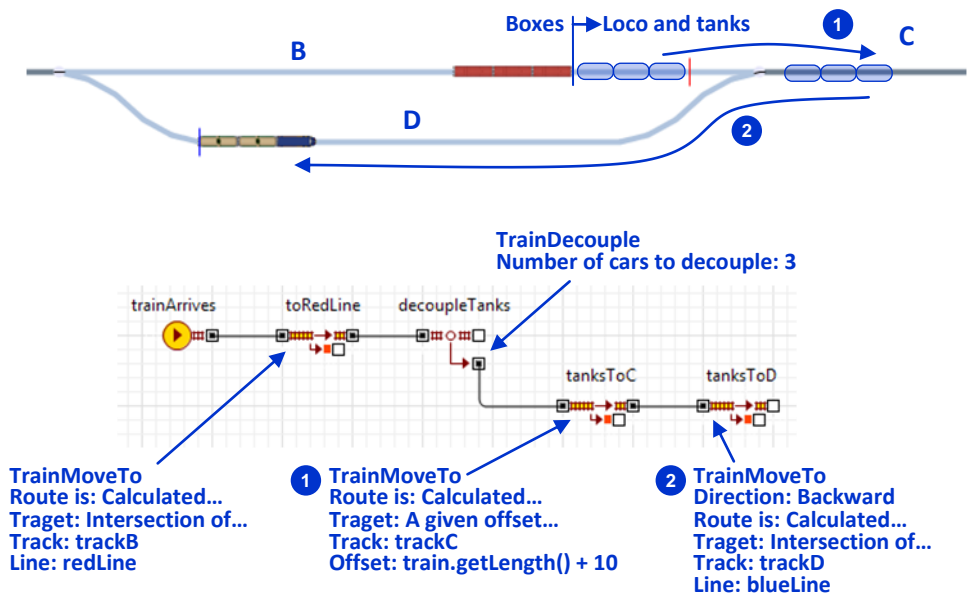
The rail cars in our train are a custom class, **FreightCar**, which is a subclass of **RailCar**. This means that the cars have all of the properties of the base class as well as the properties we defined for **FreightCar**, namely the fields **type** and **timeEntered**. In the field **New rail car** of **TrainSource**, we call the default constructor of **FreightCar** with two parameters: type and time of creation. The type depends on the position of the car in the train, which can be accessed as **carindex**. The first car (position 0) is the locomotive (constant **LOCO**), followed by two tank cars and three box cars. The second parameter is set to the current model time, the function **time()**.

► **Model decoupling of tank cars and moving them to the departure track**

13. Open the **Presentation** palette and draw two short vertical lines: **redLine** crossing the **trackB** at X=800, and **blueLine** crossing **trackD** at X=450 (see the Figure).
14. Modify the flowchart as shown in the Figure.
15. Run the model (as the process flowchart is incomplete, it makes sense to watch the first train only: the following trains will just crash into the box cars standing on track B).

The object after **TrainSource** is **toRedLine** (of type **TrainMoveTo**); it brings the incoming train to the **redLine**. The next object, **decoupleTanks** (type **TrainDecouple**), decouples 3 cars: the loco and two tanks. Next, **tanksToC** takes the decoupled part of the train to the **trackC**. Finally, the last object **tanksToD** pushes the loco and the tanks to **trackD** where they stop at the blue line.

The object **TrainMoveTo** can only calculate the straight routes, i.e. the routes not requiring reverse movement of the train. Therefore, to bring the tanks from B to D we need to use two **TrainMoveTo** objects: one that moves the train forward to C (note that the offset on C is chosen to fit the full train plus 10 meters), and another that moves the train backwards to track D.



The first step of the classification process: tank cars are decoupled and moved to D

In the object **decoupleTanks**, we assumed that any train has only two tank cars after the locomotive. We will now generalize that object so that it will decouple any number of tank cars. To accomplish this, we need to access the contents of the incoming train and look into the properties of the individual cars.

► **Generalize TrainDecouple to handle any number of tank cars**

16. Create a new function with the name **nCarsToDecouple** (Function object is in the **General** palette). Set the **Return type** of the function to **int** and add one parameter **train** of type **Train** (remember that AnyLogic is case sensitive).

17. Write the following code in the body of **nCarsToDecouple**:

```
int n = 1; //the first cars is locomotive
while( ((FreightCar)train.get(n)).type == TANK )
    n++;
return n;
```

18. Set the parameter **Number of cars to decouple** of **decoupleTanks** to:
nCarsToDecouple(train)

19. Run the model. The behavior is the same. Try to modify the parameters of **trainSource** so that the incoming trains have different or variable number of tank cars following the locomotive.

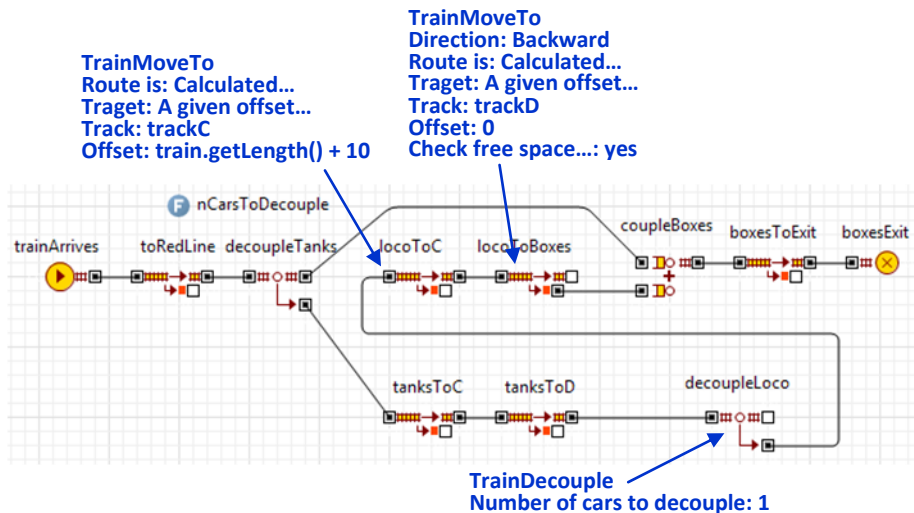
In the code of **nCarsToDecouple**, we access the object of class **Train**. The method **get(n)** returns the rail car with a given index. However, the class of the returned rail car is a

generic class **RailCar**, which we extended in order to include information about the car type. We therefore need to "cast" it to **FreightCar** to access the type field. In the field **Number of cars to decouple** we call the function **nCarsToDecouple**, giving it the current train (**train**) as an argument.

We will now continue the classification process. Having moved the tanks to track D, the locomotive will decouple from them, return to track B, couple with the box cars waiting there, and leave the yard.

► **Model return of locomotive to box cars and their departure from the yard:**

20. Add more objects to the flowchart and set their parameters as shown in the Figure. Leave the parameters of the last three objects **coupleBoxes**, **boxesToExit** and **boxesExit** at their default values. Note that the output port of **locoToBoxes** is different: it is **outHit** port.
21. Run the model. The locomotive now returns to box cars and drives them away. Again, the flowchart is still incomplete and already the second incoming train will not be able to complete the process correctly.



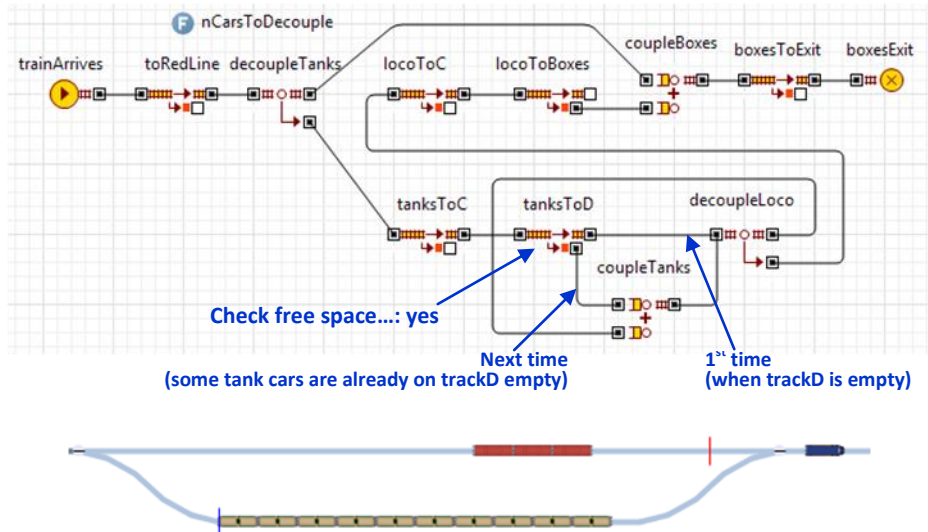
The 2nd step of classification: loco returns to box cars, picks them up, and drives away

Decoupling the locomotive is quite straightforward and can be accomplished in the following steps. First, decouple the first car of the train. Then, return to the box cars on track B, utilizing two moves: **locoToC** and **locoToBoxes**. The parameters of **locoToBoxes** are described as follows. The beginning of the track C (offset 0) is set as the target point. We know that there are box cars on track C, so the loco will inevitably hit them on its way to the target point. To let the locomotive move as close as possible to the

nearest box car and then stop, we selected the parameter **Check free space on target track** of **TrainMoveTo**.

The option **Check free space on target track** allows you to finish movement at the first rail car met on the target track. The train would "touch" the car and stop. This option is mostly used when you are going to couple with that car. Please keep in mind that the free space on the target track is checked *at the time the train starts movement*, so if the situation on the track changes while the train is moving, it may stop at an incorrect position. If the train stops having touched another train, the train entity exits via **outHit** port of **TrainMoveTo**, otherwise the train proceeds to the target position, and exits via **out** port.

Next, look at the object **coupleBoxes** of type **TrainCouple**. It has two input ports for the two parts of the train that will be coupled into one train. When a train arrives to either of the two inputs, **TrainCouple** checks to see whether it "touches" any train waiting at another input. If yes, the trains are coupled into one (more precisely, the train at **in2** is added to the train at **in1**), and the train entity exits **TrainCouple**. Otherwise, the train waits in the queue associated with the input port.



The tank cars are coupled on the departure track

Now we will show how to extend the classification process to couple the tank cars that are moved to **trackD** with other tank cars that are already there.

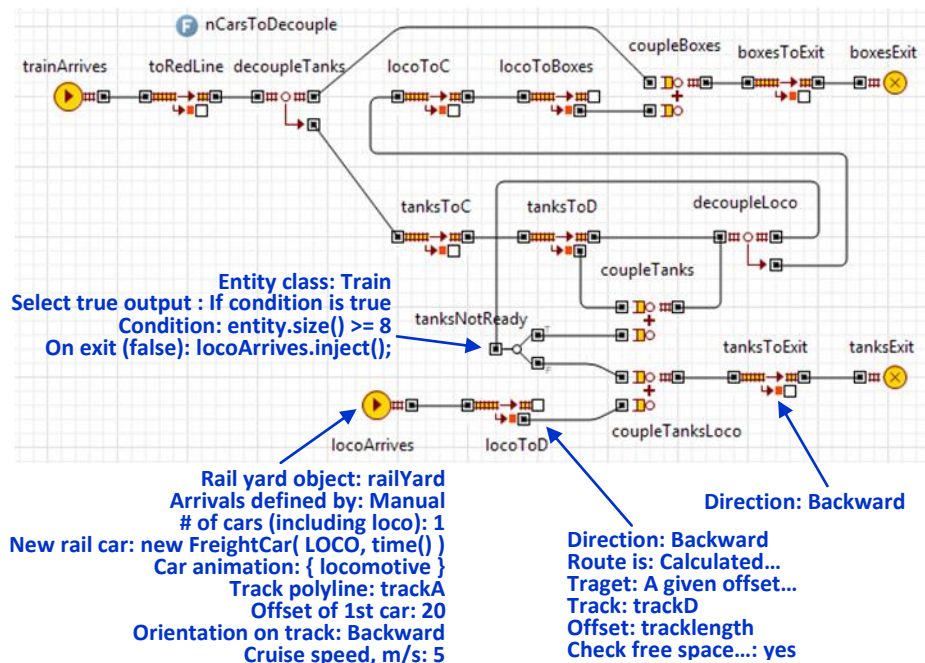
► **Model coupling of tank cars with each other:**

22. Add one more object to the flowchart: **coupleTanks** of type **TrainCouple** and connect it as shown in the Figure.
23. Set the parameter **Check free space on target track of tanksToD** to **yes**.
24. Run the model. The tank cars now accumulate on the departure track and are coupled with each other.

The departure track has limited capacity and the cars that have accumulated on it need to be moved out of the way periodically. Another locomotive will be brought in as soon as eight tank cars have accumulated on the departure track. The locomotive will come from the left-hand side (track A), couple with the tanks, and drive them away back to the left.

► **Model departure of the tank cars**

25. Modify the flowchart as shown in the Figure.
26. Run the model. The tanks are now periodically driven away by a new locomotive and the model can run for an infinite amount of time.



The final version of the classification flowchart

Note that in the flowchart object **tanksNotReady** (of type **SelectOutput**) we need to determine the size of the train, i.e., we need to access the property specific to entities

of class **Train**. As **SelectOutput** is an Enterprise Library object and not a Rail Library object, we need to tell it that entities going through it are of class **Train** – this is done by specifying **Train** in the **Entity class** field. Only after this is completed can we write **entity.size()**.

The arrival of locomotives that drive away the tanks is triggered from the **SelectOutput** object **tanksNotReady**, as seen in the action code of its **On exit (false)** field. Correspondingly, the arrival type of **TrainSource** is set to **Manual**.

There are a couple of additional things to notice in this flowchart. First, although the tank cars are all physically accumulated on track D, in the flowchart they are waiting either in the queue of **coupleTanks** object, or (if there are eight of them) in the queue of **coupleTanksLoco** object. Second, notice the orientation of the locomotive and the departure train. The initial orientation of the locomotive is backward (parameter **Orientation on track of locoArrives**), such that it couples with the tanks at its rear side. After coupling, however, the loco is added to the tanks and not vice versa because it enters the **TrainCouple** object from its lower port. Therefore, the orientation of the departure train is same as orientation of the tank cars, and departure to the left is departure in backward direction – see the parameter **Direction** of the **tanksToExit**.

Now that the classification process is complete, we can collect some statistics. Let us obtain the length of stay of the tank cars in the yard. You may remember that we have recorded the time each car entered the yard by providing the current model time in the constructor of the **FreightCar** class – see the parameter **New rail car** of the **trainArrives** object. Now we will calculate the length of stay at the time the cars are exiting the yard. We will use the field **On exit yard** of the **railYard** object.

► **Add collection of length of stay statistics**

27. Open the **Analysis** palette and drag the **Histogram data** object to the canvas. Call it **lengthOfStay** and set the **Number of intervals** to 20.
28. Select the **railYard** object and set the following parameters:
Rail car class: **FreightCar**
On exit yard:

```
if( car.type == TANK )
    lengthOfStay.add( time() - car.timeEntered );
```
29. Drag the **Histogram** chart from the **Analysis** palette to the canvas.
30. Click **Add histogram data** button on the **General** page of its properties and enter **lengthOfStay** in the **Histogram** field.
31. Run the model. Switch to virtual time model so that statistics is collected faster.

The length of stay appears to be deterministic and takes one of the four values, as shown in the Figure. This is expected: the model itself is a deterministic model, and the variation of length of stay is due to the fact that tank cars are moved to the departure track in four portions of two cars each, so the first two rail cars will wait for a longer time than the last. You may add stochastic elements to the model, such as:

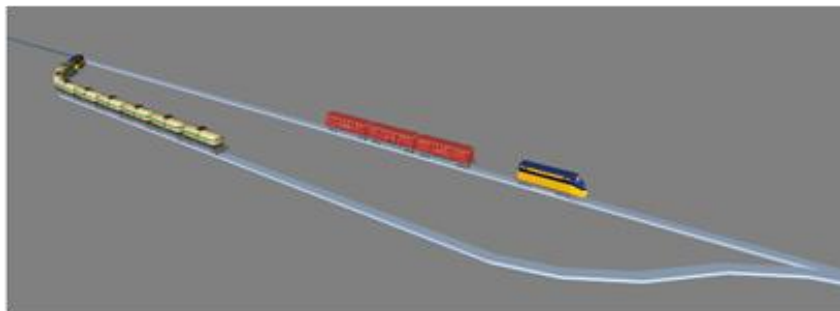
- Variation of arrival times of the original trains or the locomotive
- Stochastic time delays associated with coupling and decoupling
- Cruise speed variations



Collecting length of stay statistics for the tank cars

► Add acceleration, deceleration and 3D animation of the model

32. In all **TrainMoveTo** objects with the exception of **toRedLine** and **locoToD** set **Start options** to **Accelerate/decelerate to cruise speed**.
33. In all **TrainMoveTo** objects without exception set **Finish options** to **Decelerate and stop**.
34. Open the **3D** palette and drag the **3D Window** to the canvas below the flowchart. Change its size to, say, 750x350 pixels.
35. Run the model. Move the camera to find a better viewpoint.



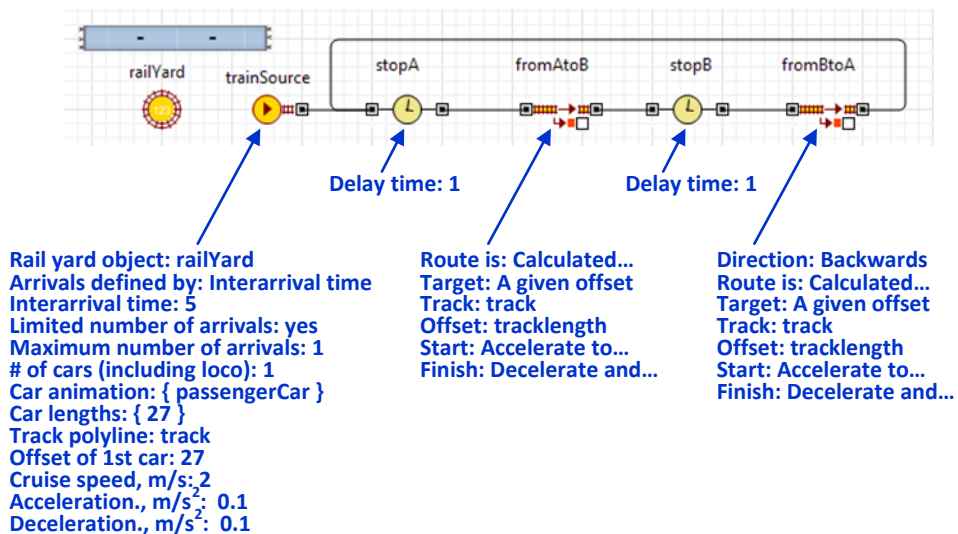
3D animation of the classification yard

❖ Example: Airport shuttle train (featuring AnyLogic Pedestrian Library)

We will now make the Rail Library and the Pedestrian Library work together by modeling a shuttle train that transfers passengers between two airport terminals. The passengers will be modeled using the AnyLogic Pedestrian Library when they are on the platform, and when they are on the train, they will be contained in a **Train** entity. The rail system will be very simple: just one passenger car moving back and forth along one straight track.

► Model the rail part of the system:

1. Draw a straight one-segment polyline from (50,50) to (950,50). Call it **track**. Select its property **Show in 3D scene**.
2. Create a group containing just this polyline and call the group **groupRails**.
3. Drag the **Rail Yard** object from the **Rail Library** palette and type **groupRails** in its **Rail yard** field. Also set the **Scale, pixels per meter** to **5**.
4. Open the **3D objects** palette and drag the **Passenger Car** object anywhere on the canvas. Set the scale of all three objects to 500% (corresponds to **Scale** set to 5 pixels per meter).
5. Create the train process flowchart as shown in the Figure. Note that the delay time "1" means one minute as the model time units are minutes (default setting, see the **General** page of the model properties).
6. Run the model.

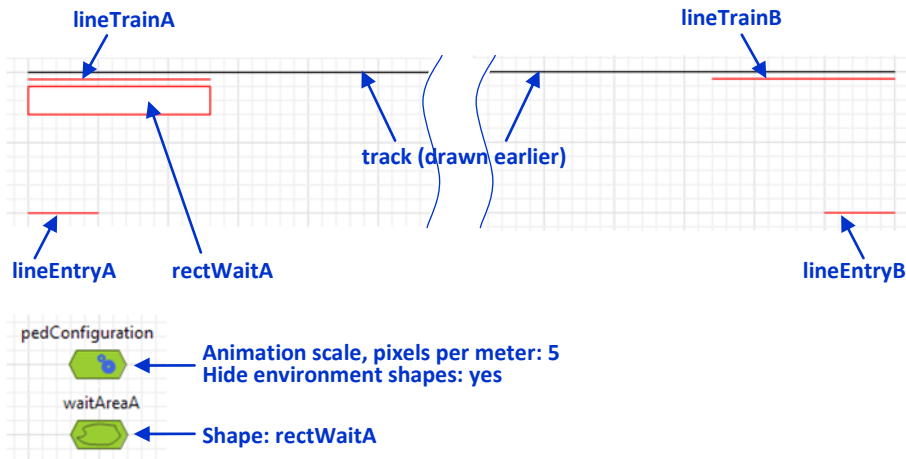


The rail part of the Airport shuttle train flowchart

Note that the **TrainSource** object in this model is used to create only one train that remains in the system. We have prevented any generation of other trains by setting the **Maximum number of arrivals** to 1.

► **Mark up the pedestrian ground and create configuration objects:**

7. Create the markup as shown in the Figure. Use **Line** and **Rectangle** objects from the **Presentation** palette.
8. Drag **PedConfiguration** and **PedArea** objects from the **Pedestrian Library** palette. Call **PedArea** **waitAreaA** and set the properties of these two objects as shown in the Figure.



Markup of the pedestrian ground

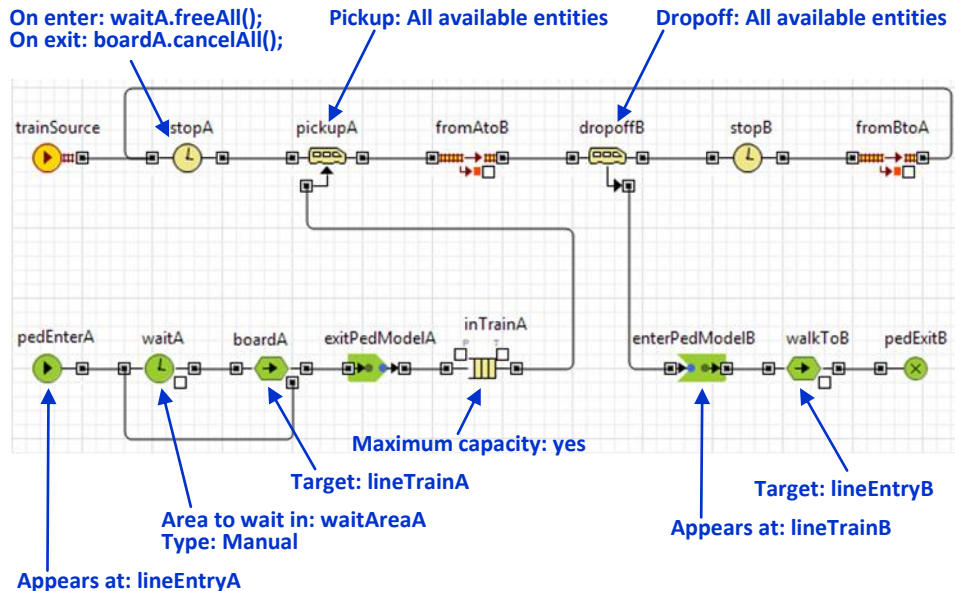
► **Add the pedestrian part of the flowchart:**

9. Insert **Pickup** and **Dropoff** objects into the rail flowchart as shown in the Figure. Set their **Pickup/Dropoff** parameters to **All available entities**.
10. Add the pedestrian flowchart and set the parameters of the objects as show in the Figure.
11. Add the entry/exit action of the **stopA** object as shown.
12. Run the model.

Once the train arrives at the stop at terminal A, it lets the waiting passengers in by calling the method **freeAll()** of the **waitA** object (of type **PedWait**). Upon closing the doors, the passengers who could not board the train are returned to the wait area by calling the method **cancelAll()** of the object **boardA**. Then they exit the object via the port **ccl** and get back into **waitA**.

The object **exitPedModelA** of type **PedExit** temporarily removes the passengers who boarded the train from the pedestrian ground, i.e. from under control of the

Pedestrian Library. The passengers are then picked up by the train entity and travel with the train to the terminal B, where they unboard (object **dropoffB**), are returned to the control of the Pedestrian Library, and appear again on the pedestrian ground at **lineTrainB**.



The complete flowchart including the pedestrian part

► Add 3D animation:

13. Drag the **Person** object from the **3D Objects** palette and set its Scale to 50% (remember that 100% scale of people objects correspond to the Pedestrian Library's default 10 pixels per meter scale, and our scale is 5 pixels/meter).
14. Click **pedEnterA** object and set its Animation shape to **person**.
15. Click the **track** polyline twice (the first click selects the **groupRails** group and the second – the polyline itself). On the **Advanced** page of its properties set: **Z: -10** and **Z-Height: 1**.
16. Open the **3D** palette and drag **3D Window** to the canvas e.g. below the flowchart. Change its size to, say, 800 x 450.
17. Run the model and view the 3D animation.
18. Draw the platforms. Drag the **Rectangle** object from the **3D** palette, place it at the train stop at terminal A and resize to cover the area where passengers walk to the train. Set its properties:
Line color: No line
Fill color: texture concrete

Z: -2

Z-Height: 2

19. Ctrl+drag the rectangle to create its copy at the stop at terminal B.

20. Run the model again.

By setting the **Z** of the track polyline to -10 and its **Z-Height** to 1 we put the rails below the level 0, which by default is the ground level for pedestrians. Correspondingly, we set up the platforms so that their upper surface is at level 0.



2D and 3D animation of the Airport shuttle train

Java class Train (subclass of Entity)

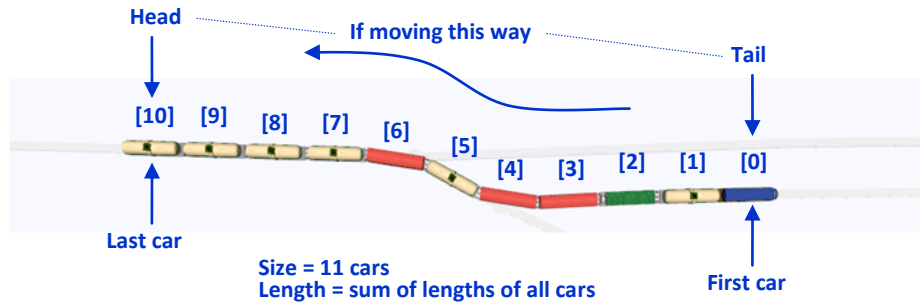
A train is a sequence of one or several rail cars (objects of class **RailCar**) coupled with each other that can move in the rail yard and is controlled by the Rail Library and Enterprise Library flowchart objects.

Trains are created by **TrainSource** objects and must be disposed by **TrainDispose** objects. A train can be split into two trains by **TrainDecouple** and two trains can be combined into one by **TrainCouple**. Trains move in the rail yard under control of **TrainMoveTo** objects. Train class is a subclass of **Entity**, so trains can go through any Enterprise Library objects like **Delay**, **Seize**, **Release**, etc.

The cars in the train are ordered – there will always be the first car and the last car (which are the same in case the train contains only one car), as shown in the Figure. The orientation of the cars at the time of the train creation is the same as that of the train, but later on it can change as a result of coupling/decoupling. The length of the train (the length of the track portion occupied by the train) is equal to the sum of all the car lengths.

The Rail Yard Library does not have a concept of a locomotive, or of any other car type: all rail cars are considered to be equal. Any train can move at any speed, or be coupled/decoupled at any side.

The train has *cruise speed*, *acceleration* and *deceleration* properties. They are initially set up by **TrainSource** but can later on be changed via the Train API. While the train is moving you can change its speed instantly or by applying acceleration and deceleration.



A train

Animation of a train is actually the animation of the rail cars of the train, – there are no specific graphics related to the train.

Here are some methods of class **RailCar** (see **Rail Library Help** for the full list):

Train contents and dimensions

- **int size()** – returns the number of cars in the train
- **double getLength()** – returns the length of the train (the sum of lengths of all cars) in meters
- **RailCar getFirst()** – returns the first car in the train.
- **RailCar getLast()** – returns the last car in the train.
- **RailCar get(int index)** – returns a car with a given index, the first car index is 0, the last car – **size()-1**

Speed and acceleration

- **setCruiseSpeed(double speed)** – sets the cruise speed of the in meters/sec.
- **double getCruiseSpeed()** – returns the cruise speed of the train in meters/sec.
- **setAcceleration(double acceleration)** – sets the acceleration of the train in meters/sec².
- **double getAcceleration()** – returns the acceleration of the train in meters/sec².
- **setDeceleration(double deceleration)** – sets the deceleration of the train in meters/sec².

- **double getDeceleration()** – returns the deceleration of the train in meters/sec².

Movement control

- **boolean isMoving()** – returns true if the train is moving, false if not.
- **RailCar getHead()** – returns the car that is at the head of a moving train (either first or last), null if the train is not moving.
- **RailCar getTail()** – returns the car that is at the tail of the moving train (either first or last), null if the train is not moving.
- **setVelocity(double v)** - sets the new velocity of the train. It applies immediately even if the train is moving. If the train is not moving, just remembers the velocity but does not start the train.
- **double getVelocity()** – returns the velocity of the train in meters / sec. If the train is empty, returns 0.
- **accelerateTo(double speed)** – accelerates or decelerates the train to achieve a given speed. Can only be called while the train is moving.
- **ShapePolyLine getTrack(boolean front)** – returns the track (the polyline) where a given side of the train is currently located (if **front** is true, this is front side, otherwise – rear).
- **double getOffset(boolean front)** - returns the offset of a given side of the train relative to the track start point in meters, 0 is the beginning of the track (if **front** is true, this is front side, otherwise – rear).
- **ShapePolyLine getTargetTrack()** – returns the target track of a moving train, null if target is not set.
- **double getTargetOffset()** – returns the target offset (on the target track) of a moving train in meters, assumes the target is set.
- **double getDistanceToTarget()** – returns the distance from the current position to the target point in meters. Assumes the train is moving along a route (specified manually or calculated automatically) and the target is set.

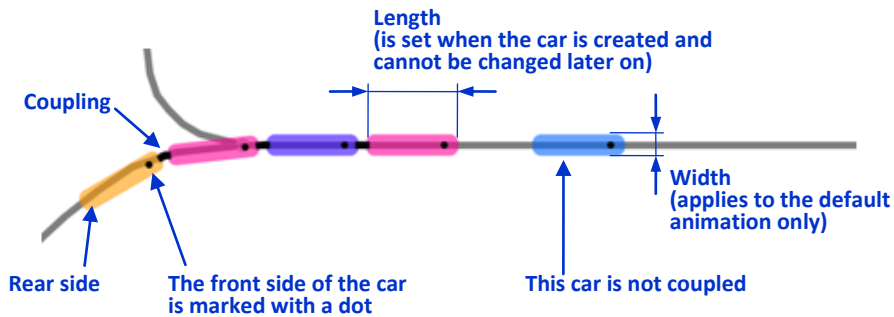
Java class RailCar

In the AnyLogic Rail Library Java class, **RailCar** is the base class for all kinds of rail cars and locomotives. A rail car has dimensions (length and width), can move along a track, and can be coupled and decoupled with another car. A car can have 2D and 3D animation.

Starting with AnyLogic version 6.5.1, rail cars are created by **TrainSource** objects as a part of a train and are fully controlled by trains during their whole lifetime in the rail system. However, the low-level interface to car creation, movement and

coupling/decoupling is still supported by the Rail Library for the purpose of compatibility with earlier models.

From the Rail Library viewpoint there is no difference between a car and a locomotive or between freight and passenger cars, but you can make such a distinction in your model. For example, you can specify different dimensions, assign different animation shapes, or create different subclasses from the **RailCar** class with different properties and methods. And of course, you can associate different behavior logic.



Rail car details explained on default 2D animation

A **RailCar** has length and two sides: front and rear. The length of the rail car can only be set up at the time of its creation (e.g. in the parameter **Car lengths** or if you call **setLength()** in the **Car setup** code of **TrainSource**) and cannot be changed later on. You can get the location (track, orientation on track, and offset) of the either side of the car. You can ask the car to **callback** and execute a custom action at the specified point of a given track. Arbitrary information can be passed to the callback code.

The default 2D animation of the car is a rounded rectangle, the front side of which is marked with a black dot as shown in the Figure. If the car is coupled, the connection to the other car is drawn. If the car is highlighted, a circle will be drawn around it. The default 3D animation is a parallelepiped. The color of the default animations can be customized by calling **setColor()**. You can also assign a custom animation shape for a car in the parameter **Car animation** or by calling **setShape()** in the **Car setup** code of **TrainSource**. The 3D Objects palette contains ready-to-use 3D animations for different types of cars and locomotives.

Please note that the size of the rail car custom animation will not be automatically adjusted to the length of the car, so you need to choose the proper scale of the animation shape to ensure there are no spaces between cars in the train and cars are not drawn one above the other. AnyLogic passenger car 3D object at 100% scale is 27

meters long, and other cars and the locomotive are 14 meters long (assuming one pixel is 1 meter).

Here are some methods of class **RailCar** (see **Rail Library Help** for the full list):

- **setLength(double length)** – sets the length of the car in meters. Can only be done before the car is added to the rail yard.
- **double getLength()** – returns the length of the car in meters.
- **setWidth(double width)** – sets the width of the car in meters. Applies to the default animation only.
- **double getWidth()** – returns the width of the car in meters.
- **setColor(Color color)** – sets the color of the car. Although the color is always stored in a **RailCar** object, it only applies to the default animation.
- **Color getColor()** – returns the color of the car.
- **setShape(Shape shape)** – sets a custom 2D or 3D shape that will be used to animate this car.
- **Shape getShape()** – returns a custom shape that is used to animate the car, or **null**.
- **Train getTrain()** – returns the train the car belongs to, or **null**.
- **Track getTrack(boolean infront)** – returns the track on which a given side of car is currently located (if **infront** is true, this is front side, otherwise rear side).
- **double getOffset(boolean infront)** – returns the offset of a given side of car relative to the track start point in meters: 0 - the beginning of the track (if **infront** is true, this is front side, otherwise rear side).
- **double getX(boolean infront)** – returns the X coordinate of a given side of the car relative to the rail yard group of shapes *in pixels* (if **infront** is true, this is front side, otherwise rear side).
- **double getY(boolean infront)** – same for the Y coordinate.
- **callbackAt(Track track, double offset, Object info)** – requests the car to execute a callback (the code in the field **On at callback** of the **RailYard** object) at the specified point, namely when its side that moves in front reaches a given offset on a given track. You can pass arbitrary information (**Object**) to the callback code to identify what kind of event it is.
- **callbackAt(Track track, ShapeLine line, Object info)** – same as **callbackAt(track, offset, info)** where the offset is the offset of intersection of the track and a given line.