Introduction to Reactive Blocks

This set of exercises is meant as a second option to the standard Reactive Blocks tutorials. It is only the first step towards a more interactive and immersive tutorial, but these exercises will hopefully provide a better introduction to UML Activities and working with Reactive Blocks.

We also hope you will take the time to answer a few simple questions about the exercises when you are finished, so we can use your feedback to improve these exercises. You can fill out the paper form, or answer here. Additional comments are also welcome!

This assignment is made up of 7 smaller exercises, where each exercise introduces some new elements and concepts. For each exercise, you need to design a program that uses the new elements introduced (as well as some of the previous ones). The new concepts are briefly described in the exercises, but if you need more information, you can always check out the reference pages (http://reference.bitreactive.com/). Please let me know if you feel any information is incomplete, missing or even incorrect. You only need to think about modeling the system, all *operations* are already implemented.

In addition to the basic exercises, there are some challenge exercises that are a little more difficult to complete. These are completely optional, but may provide some useful additional insight into how you can utilize these concepts in an application. Some of these might even be fun to complete.
If you get stuck, there are some general tips on the last page.

Preparation

1. Sign up for *Reactive Blocks* (http://blocks.bitreactive.com/login/signup.html)
    - Select the "Free plan"
    - In "About yourself", write "TTM4115".

2. Install the *Reactive Blocks SDK* (http://reference.bitreactive.com/install_arctis)

3. Open the *Reactive Blocks Perspective* (http://reference.bitreactive.com/doc/the_arctis_perspective)

4. Import the tutorial project
    - Join the *TTM4115 Tutorial Experiment* team (http://blocks.bitreactive.com/#/group/Gcm7ufo7fb67h0i8a)
    - Click *Import new libraries and examples* in the *Blocks* window
    - Select *no.ntnu.item.tutorials* and click *Finish*

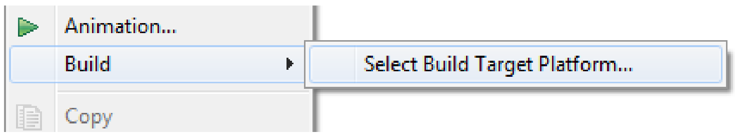
✓ no.ntnu.item.tutorials 1.0.0 (13) no.ntnu.item.tutorials

5. Use the provided blocks to complete the exercises

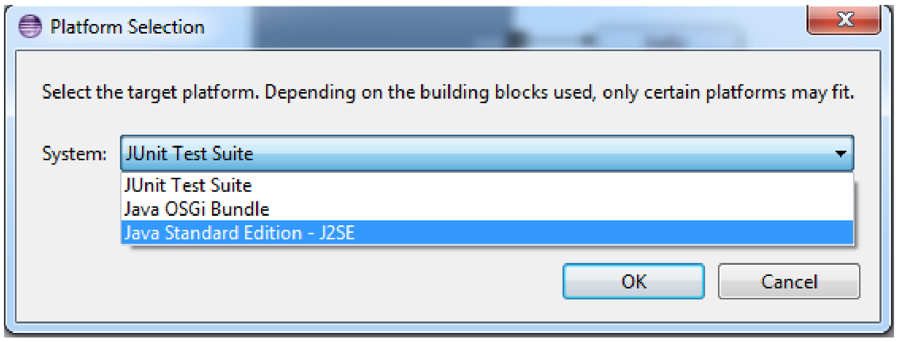
Building and running the exercises

When you have completed modeling the exercise, you may want to run it in order to see if it does what it's supposed to. To build and run an exercise block, do the following steps:

1. Right click -> *Build* -> *Select Build Target Platform...*



2. Select *Java Standard Edition - J2SE*



Platform Selection

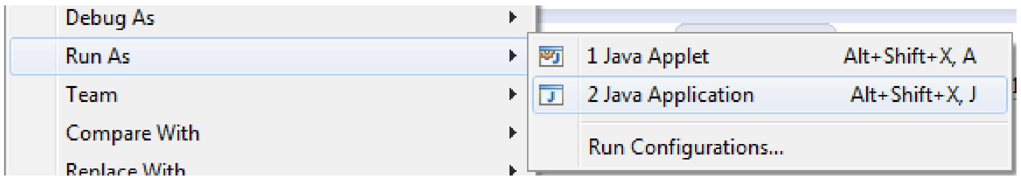
Select the target platform. Depending on the building blocks used, only certain platforms may fit.

System: JUnit Test Suite

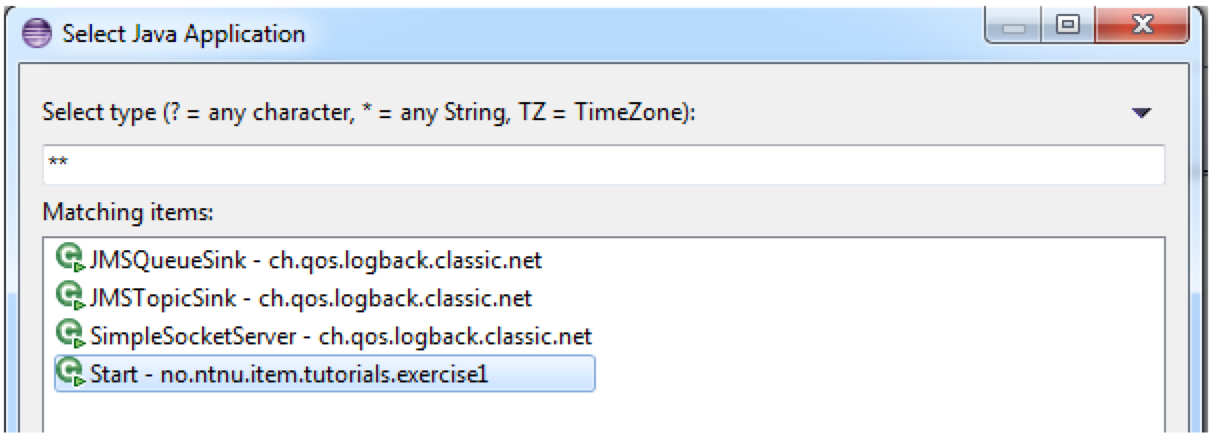
JUnit Test Suite
Java OSGi Bundle
Java Standard Edition - J2SE

3. Click *OK* without changing anything. This will generate a new executable project.

4. Right click the newly generated project (*no.ntnu.item.tutorials.exerciseX_exe*) in the *Package Explorer* and select *Run As* -> *Java Application*



5. Select *Start - no.ntnu.item.tutorials.exerciseX* and click *OK*



Select Java Application

Select type (? = any character, * = any String, TZ = TimeZone):

**

Matching items:

JMSQueueSink - ch.qos.logback.classic.net
JMSTopicSink - ch.qos.logback.classic.net
SimpleSocketServer - ch.qos.logback.classic.net
Start - no.ntnu.item.tutorials.exercise1

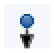
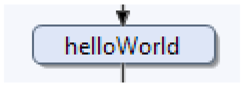
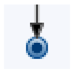
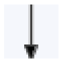
Exercise 1 - Hello world!

Task: Create a “Hello world!” program.

Introducing concepts: Tokens and control flow
In *Reactive Blocks*, an application is executed by creating one or more *control tokens* that is passed through the various *blocks* and *nodes* in the program in a series of *steps*. Whenever a token passes through a node, some application logic is performed, and the token continues to the next node. For example, if this node is an *Operation*, the application will run the code associated with the operation.

When the application is first started, one token is created in each of the *Initial Nodes* present. The token then moves along the *Edge* connected to the *Initial Node* to the next element. Finally, when any token reaches an *Activity Final* node, all tokens stop flowing and the application is terminated.

New elements/nodes

	**Initial Node:** generates a *token* when the program first starts.
	**Operation:** an operation is performed when a *token* passes through it.
	**Activity Final:** when a *token* reaches this node, the program terminates.
	**Edge:** connects nodes, so *tokens* may pass between them.

Information about operations
- **helloWorld**: prints “Hello world!” to the console.

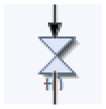
Challenge exercise: Create a program that prints one message, then a second message, and finally the first message again.

Exercise 2 - Delays

Task: Create a program that displays a light, sets its color to red, then changes it from red to green after 5 seconds. Make sure you have time to see that the light has changed before the program terminates.

Introducing concepts: Activity steps and stable positions
As mentioned in exercise 1, applications are executed by passing tokens through nodes in a series of steps. Most of the time, a token will pass through multiple nodes in a *single* step, but some nodes, such as the *timer*, will put a token in a *stable position*, ending the current step. In the case of the *timer*, a new step will be initiated when the set time has expired, and the token will continue to the following nodes.

New elements/nodes

	**Timer:** keeps incoming tokens in a stable position for the duration of the *timer*, then starts a new activity step when it expires.

Information about operations

- **displayLight**: displays a “light” image in the application window.
- **setLightRed**: sets the color of the light to red.
- **setLightGreen**: sets the color of the light to green.

Challenge exercise: Create a program that simulates two traffic lights, one for cars and one for pedestrians.

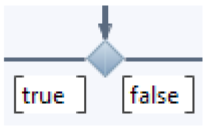
- The car light should start at green, and the pedestrian light at red.
- After 10 seconds, change the car light to yellow.
- After 2 more seconds, change the car light to red.
- Wait another second, then change the pedestrian light to green.
- Give the pedestrians 10 seconds to walk, then change their light back to red.
- Wait 1 seconds, then change the car light to yellow.
- Finally, after 2 more seconds, change the car light to green.
- Let the lights shine for a short time before terminating the program.

Exercise 3 - Choices

Task: Create a program that generates a random number between 0 and 200, and prints "Small!" if the number is smaller than 100, and "Big!" if the number is greater than 100.

Introducing concepts: Object flow and Decisions
In addition to providing a means of controlling the logic of the application, tokens may carry with them a single *object* (this can be any regular Java object). This data can be used by following decisions or operations to perform a part of the application logic. Most nodes, like timers and *decisions*, will pass the object on to the next node, but operations will not. Decisions will additionally check the value of the object, and select an outgoing edge depending on this value.

New elements/nodes:

	**Decision:** reads the incoming data object, and chooses an outgoing edge depending on their *guards* and the value of the data.
	Object flow/edge: *object flows* are like regular *edges* (flows), but tokens traveling along these edges also carry an object that may be used by the receiving node.

Information about operations:

- **generateNumber**: generates a random number (int) between 0 and 200.
- **isNumberBig**: takes a number (int) as input, and returns *true* if the number is greater than 100, and *false* if not.
- **big**: prints "Big!" to the console.
- **small:** prints "Small!" to the console.

Challenge exercise: Create a program that generates a random number between (and including) 1 and 8, and then uses a binary search tree to find out which number it was. This exercise might require the use of variables and the Set/Get Variable Actions (read more at http://reference.bitreactive.com/doc/decisions_and_guards).

Exercise 4 - Looping

Task: Create a program that generates a random number between 0 and 200 until that number is greater than 100 (big).

Introducing concepts: Loops, flow breaks and Merge
Sometimes, we want our application to repeat the same operation or sequence several times without adding many instances of the operation (imagine 100 repeats or more). In this case, we can create a loop in our model, inserting the token back at the start of the sequence with a *merge* node. The token will then pass through the same sequence again, repeating it.

However, a token may not pass through the same sequence more than once in a single *step*. In order to avoid this, we have to make sure the token reaches a *stable position* somewhere in the loop, letting it finish the current step, and then start a new one to run the sequence again. We previously saw that a timer will put tokens in a stable position, but in many cases we don't want to add a timer delay before starting a new step. In this case we may use a timer with *zero* delay, also known simply as a *flow break*.

New elements/nodes:

◆	**Merge:** merges two or more edges into a single edge, passing on tokens from any of the inputs to the output within the same step.
⧗	**Flow break:** a flow break is a timer with zero delay, used to divide a flow into more than one step without adding delay.

Information about operations:

- **generateNumber**: generates a random number (int) between 0 and 200.
- **isNumberBig**: takes a number (int) as input, and returns *true* if the number is greater than 100, and *false* if not.

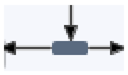
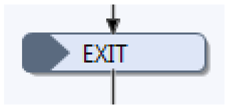
Exercise 5 - User input and parallelism

Task: Create a program that changes the color of a light every time you press one button, and exits when you press another. It should be possible to change the light an arbitrary number of times before exiting (even zero).

Introducing concepts: Forks and Events
A timer is not always the appropriate solution for delaying execution of a step. Sometimes we don't know how long we should wait, as it may depend on something random or outside the program (e.g. user input). In these cases, we use the *Event Reception* node, which puts incoming tokens in a *stable position* until the appropriate *Event* is received by the program.

In many cases, we want our application to perform more than one task at the same time. We may want to perform some computations while waiting for user input, or maybe receive user input from more than one source. To make this possible, we need more than one token flowing in each step. There are essentially two ways of doing this: add more than one *Initial Node*, or place a *Fork* node where you want to add parallelism. The latter is typically more appropriate if you only want parallelism for certain steps.

New elements/nodes:

	**Fork:** when a token is received on this node, it creates a token on each c its outputs (two or more) within the same step.
	**Event Reception:** Like *timers*, *Event Receptions* are stable positions marking the end of an activity step. However, instead of waiting for a timeout, a new activity step is initiated when the named *event* is received

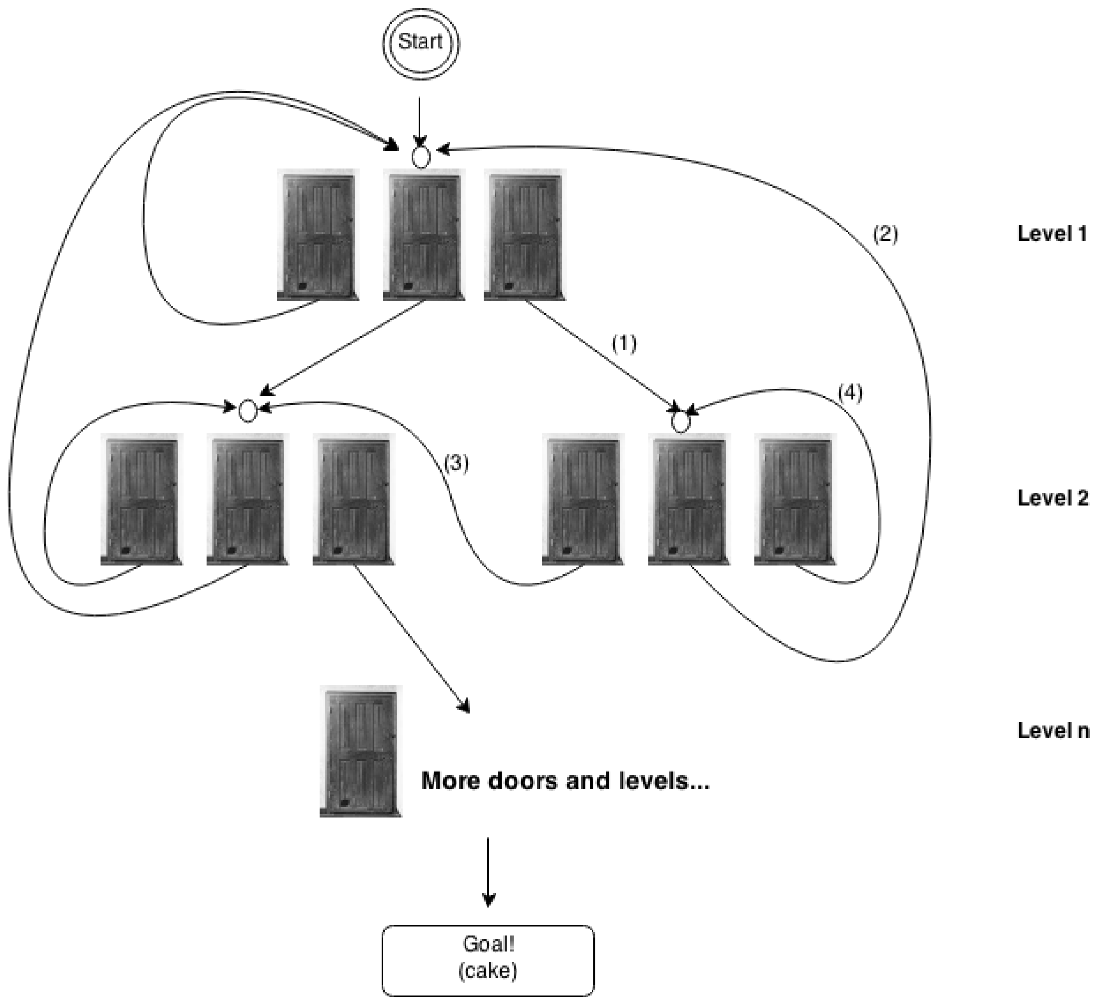
Information about operations:

- **showLight**: displays the light in an application window.
- **showButtons**: displays the "Change color" and "Exit" buttons in an application window.
- **changeColor**: changes the color of the light.

Challenge exercise: Implement *The Door Game* (see next page).

The Door Game

The door game is a simple game where the purpose is to find the correct way through a set of doors to the goal, without knowing where each door leads. The player starts at level 1, and must then advance through the levels by opening doors until the final level (and the goal) is reached. At each step, the player is presented with 3 doors; blue, red and green. Depending on which door the player chooses, it will take him/her one step forward (1), one step back (2), one step to the side (3), or nowhere (4).



Tips:

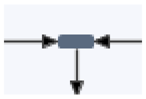
- It's entirely up to you how you create the door structure (i.e. which doors lead where).
- The events must be used with their corresponding operations (e.g. OPEN_DOOR1 and showDoors1).
- The *goBack()* etc… operations provide useful feedback info to the player.
- The events carry with them a *String* object, which is either "red", "green" or "blue".
- Code for 5 sets of doors is already implemented, but you can add more if you want. This will however require editing the Java code.

Exercise 6 - Synchronizing

Task: Create a program that prints a message and then terminates when three buttons have been clicked (in any order).

Introducing concepts: Join
In addition to timers and events, there is a third option to delaying steps. If you have several concurrent flows, with a sequence depending on all of them to be completed, you can use a *Join* node to synchronize the flows. The *Join* node will put incoming tokens in a *stable position* until a token has been received on *all* inputs, and then produce a *single* token on its output (within the same step as the last received token).

New elements/nodes:

	**Join:** The join node produces a *single* token on its output only when a token has been received on *all* its inputs. This synchronizes any incomin flows, forcing all of them to wait for the last token to arrive.

Information about operations:
- **showButton1**: displays the first button.
- **showButton2**: displays the second button.
- **showButton3**: displays the third button.
- **printMessage**: prints a message to the console.

Exercise 7 - Local blocks

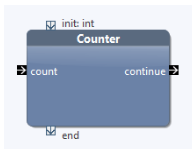
Task: Use the *Counter* block to make the *CoinFlipper* block "flip a coin" 10 times, and then return the result.

Introducing concepts: Local Blocks
When creating bigger applications, it becomes inconvenient and chaotic to have all the logic in a single block. Most of the time, many parts of the program can be separated into their own blocks, allowing for application design by composition of blocks. Separating functionality in this way is generally considered good design, and it is one of the core principles of Reactive Blocks.

So far we have been working with only the *System Block*, which is the top-level block of the application (it contains all other blocks). Inside the *System Block*, we can add an arbitrary number of *Local Blocks*, and these *Local Blocks* may themselves contain other *Local Blocks*. Each *Local Block* has a set of input and output pins, where tokens may enter or leave the block.

One important attribute of *Local Blocks* is the *External State Machine (ESM)*, which serves as a contract between the *Local Block* and its environment, deciding/limiting which signals may be sent or received in any defined *state*. If interaction with a *Local Block* violates its ESM, Reactive Blocks will produce an error. For more information about ESMs, see http://reference.bitreactive.com/doc/esm_basic.

New elements/nodes:

	**Local Block:** A *Local Block* is a reusable component providing part of th application logic. It has input/output pins for tokens, and an *External Stat Machine* defining how it may interact with its environment.

Information about operations:

- **displayResult**: prints the input String to the console.

Information about blocks:

- **Counter**: The Counter block receives an *int* value when it is initialized, and counts down from this number towards zero every time a token is received on the *count* pin. When the counter reaches zero, the block terminates and releases a token on its *end* pin, otherwise a token will be released on its *continueCount* pin.
- **CoinFlipper**: Flips a coin every time it receives a token on its *flip* pin. When it receives a token on its *getResult* pin, it terminates and returns a token carrying the result on its *result* pin.

Challenge exercise: Taxi Order System (TTM4115 Lab Exercise 1).

General tips

- Right click -> *Add…* to add more elements to a block.
- Right click on any element for more information and options.
- Double click on *operations* to see their source code.
- The same operation may be used more than once in a block.
- It is possible to have more than one *Initial Node*.
- The console window is at the bottom of your Eclipse window.
- Red color usually means something is wrong. You can view the error messages under *Analysis* at the bottom of your Eclipse window. Additionally, you can check if there are other problems with Right click -> *Analyze*
- If you want to see how tokens will flow in a program before running it, you can do so with Right click -> *Animation*
- There is often more than one way to solve the exercise!
- You can move the lines of edges around to make it look cleaner.
- In exercise 7, you may encounter the problem that the "Building block … is missing". If this is the case, simply right click the Local Block giving the error message, and select *Set Building Block…* Type the name of the block you want to link to (e.g. "Counter"), select the correct block and press *OK*. You can also simply delete the "missing" block, and drag-and-drop a new one from the *Building Blocks* window.
- If you get stuck, check the reference pages (http://reference.bitreactive.com/) or ask someone for help.
- Please take the time to answer the feedback questions! You can also provide feedback directly to me at oyvinric@stud.ntnu.no.