



BLOCKLY

Distributed Systems Laboratory

Sándor Bácsi
2020.



INTRODUCTION

OBJECTIVE

The aim of this lab is to get to know [Blockly](#), which is client-side library for the programming language JavaScript for creating block-based visual programming languages.

PREREQUISITES

- HTML and Javascript editor (e.g. Visual Studio Code)
- The starter project
- [Block Factory](#) for creating custom blocks

OVERVIEW OF THE STARTER PROJECT

- `blockly_compressed.js`: JS dependencies of Blockly
- `blocks_compressed.js`: Contains built-in blocks
- **blocks.js**: We will store the definition of the custom blocks in this .js file.
- `en.js`: English text representation of the built-in blocks
- **generator_stub.js**: We will implement the code generation logic of the custom blocks in this .js file.
- **index.html**: The starter HTML page. We will put our own blocks in the toolbox in this file.
- `javascript_compressed.js`: JS dependencies of Blockly javascript code generation

During the laboratory, we will edit the files in **bold**.

INTRODUCTION OF BLOCK FACTORY

We will use [Block Factory](#) to create our own custom Blocks. We can easily build custom blocks by plugging field, inputs and other blocks in the editor.

- **Editor**
 - **Input category**:
 - **Value input**: A value socket for horizontal connections.
 - **Statement input**: A statement socket for enclosed vertical stacks.
 - **Dummy input**: For adding fields on separate row with no connections. Alignment options (left, right, centre) apply on to multi-line fields.
 - **Field category**: We can assign fields to the inputs (e.g. an input field for the user to enter a numeric input)
 - **Type**: We can customize the connections of the blocks by using types.
 - **Colour**: We can customize the color of the corresponding block.

- **Preview:** We can preview and test our custom block in this window.
- **Block definition:** The code representation of our own custom block. We will copy the **JavaScript** representation in the **blocks.js** file.
- **Generator stub:** We have to complete the generator stub with our own code generation logic. We will use string concatenation for implementing the code generation logic, which is not the most elegant way of code generation, but this will be sufficient for our during the lab.

TASKS

1. BASIC BLOCKS

1.1. PRINT BLOCK

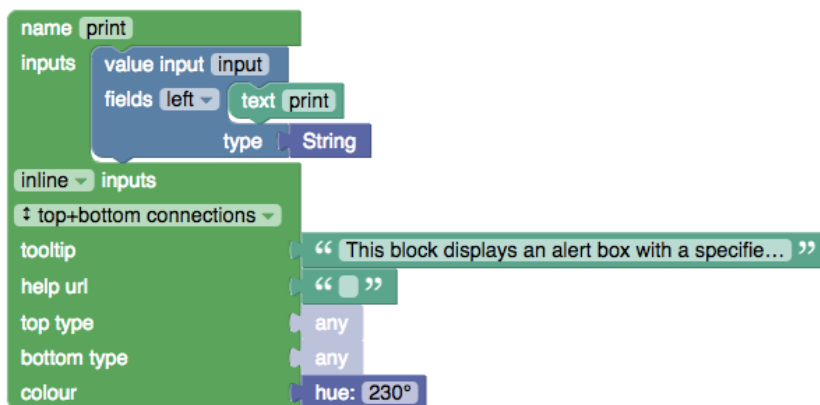
Create blocks that can be used to print text in a JavaScript Alert box. We have to create a block which has String inline value input and another block in which the string parameter of the Alert call can be specified.



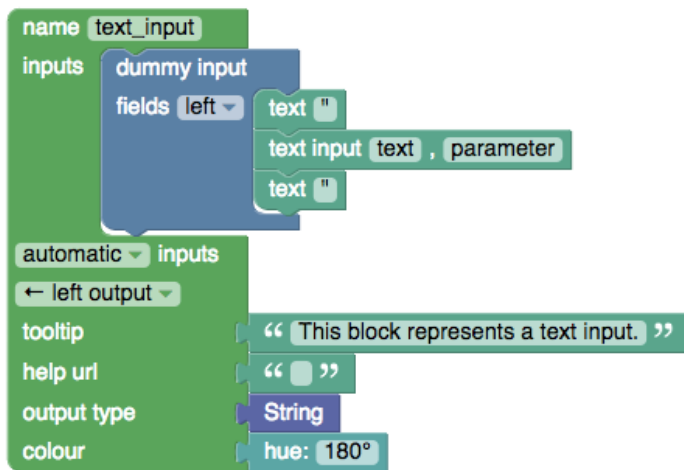
Solution

We have to create the following blocks in Block Factory:

Print block:



Text block:



blocks.js:

```
Blockly.Blocks['print'] = {
  init: function() {
    this.appendValueInput("input")
      .setCheck("String")
      .appendField("print");
    this.setInputsInline(true);
    this.setPreviousStatement(true, null);
    this.setNextStatement(true, null);
    this.setColour(230);
    this.setTooltip("This block displays an alert box with a specified message and an OK button.");
    this.setHelpUrl("");
  }
};
```

```
Blockly.Blocks['text_input'] = {
  init: function() {
    this.appendDummyInput()
      .appendField("")
      .appendField(new Blockly.FieldTextInput("text"), "parameter")
      .appendField("");
    this.setInputsInline(true);
    this.setOutput(true, "String");
    this.setColour(180);
    this.setTooltip("This block represents a text input.");
    this.setHelpUrl("");
  }
};
```

generator_stub.js:

```
Blockly.JavaScript['print'] = function(block) {
  var value_input = Blockly.JavaScript.valueToCode(block, 'input', Blockly.JavaScript.ORDER_ATOMIC);
  var code = 'alert('+value_input +');\n';
  return code;
};
```

```
Blockly.JavaScript['text_input'] = function(block) {
  var text_parameter = block.getFieldValue('parameter');
  var code = "" + text_parameter + "";
  return [code, Blockly.JavaScript.ORDER_ATOMIC];
};
```

Please note that in the **text_input** block we have to use the **Blockly.JavaScript.ORDER_ATOMIC**.

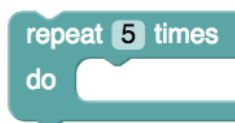
index.html: Add the new blocks to the toolbox.

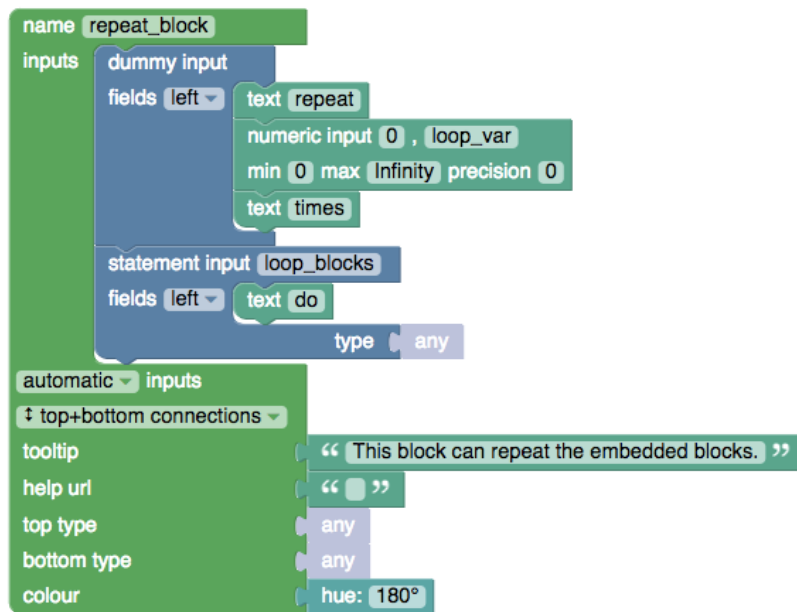
```
<category name="Basic blocks" colour="180">
  <block type="print"></block>
  <block type="text_input"></block>
</category>
```

Let's try the new blocks. Create the following code in Blockly and click Run. (We can display the generated code by clicking on the Show Code button.)

**1.2. REPEAT BLOCK**

Create a repeat block, that can be used to repeat the embedded blocks. The preview of the block should be the following:

**Solution****Block Factory:**



blocks.js:

```
Blockly.Blocks['repeat_block'] = {
  init: function() {
    this.appendDummyInput()
      .appendField("repeat")
      .appendField(new Blockly.FieldNumber(0, 0), "loop_var")
      .appendField("times");
    this.appendStatementInput("loop_blocks")
      .setCheck(null)
      .appendField("do");
    this.setPreviousStatement(true, null);
    this.setNextStatement(true, null);
    this.setColour(180);
    this.setTooltip("This block can repeat the embedded blocks.");
    this.setHelpUrl("");
  }
};
```

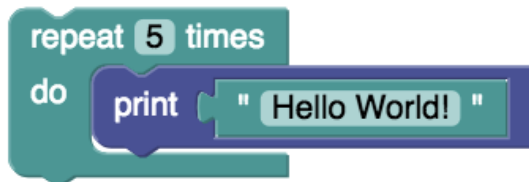
generator_stub.js:

```
Blockly.JavaScript['repeat_block'] = function(block) {
  var loop_var = block.getFieldValue('loop_var');
  var statements_loop_blocks = Blockly.JavaScript.statementToCode(block, 'loop_blocks');
  var code = 'var repeats = 0;\n' +
    'while (repeats < ' + loop_var + ') {\n' +
    statements_loop_blocks +
    'repeats++;\n' +
    '}\n';
  return code;
};
```

index.html: Add the repeat block to the Basic block category.

```
<block type="repeat_block"></block>
```

Let's try our new repeat block. For example, we can repeat printing some text five times.



1.3. ADDING VARIABLES CATEGORY

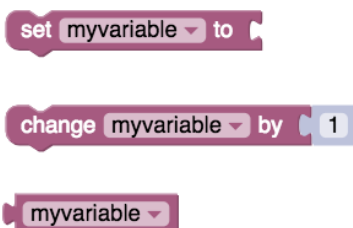
Blockly has a built-in JavaScript variable category, which uses the **Blockly.JavaScript.variableDB**. With the help of these built-in blocks we can easily create and modify new variables in Blockly. In the next exercise we will create a block for a simple for loop, where we will need to handle a loop variable. Add the built-in variables category to the toolbox.

Solution

Index.html: Add the built-in variables category to the toolbox.


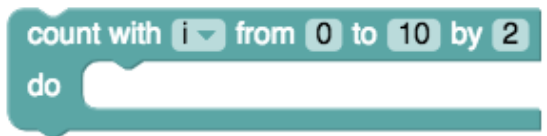
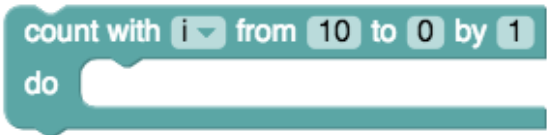
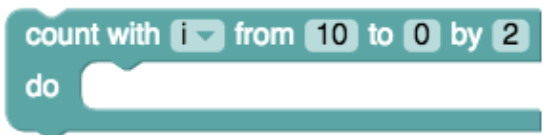
```
<category name="Variables" colour="%{BKY_VARIABLES_HUE}" custom="VARIABLE"></category>
```

Let's try the variables category and its blocks. For example, we can create a new variable and try the *set* and *change* block.



1.4. SIMPLE FOR LOOP (INDIVIDUAL TASK: PRINTSCREEN AND CODE)

Create a custom block which represents a simple *for loop*. Use the **variable field** for the loop variable of the for loop. The preview of the for loop and the generated code should be the following:

	<pre>for (i=0; i<=10; i++) { }</pre>
	<pre>for (i=0; i<=10; i+=2) { }</pre>
	<pre>for (i=10; i>=0; i--) { }</pre>
	<pre>for (i=10; i>=0; i-=2) { }</pre>

Please note that only non-negative numbers can be specified at the **by** parameter. We can determine the traversing direction of the loop variable from the **from** and **to** fields.

Testing: Create a test program in Blockly, which prints numbers in JS Alert box from 0 to 10.

Hint: Use the Show Code button to display and "debug" the generated code.

2. VR BLOCKS

Now, it is time to do something more interesting with Blockly. We will create custom blocks for certain features of a web VR framework. We will use [A-Frame](#), which is a web framework for building virtual reality experiences.

Hint: We can use the arrow key to move in the A-Frame window that appears on the right.

2.1. NUMBER BLOCK

In order to be able to use our blocks easily with a parameter of type Number in the future (we can also assign a number to a value input), let's create a block that can return a value of type number as left output. This will be needed when creating the different A-Frame 3D objects. The preview of the block should be the following:

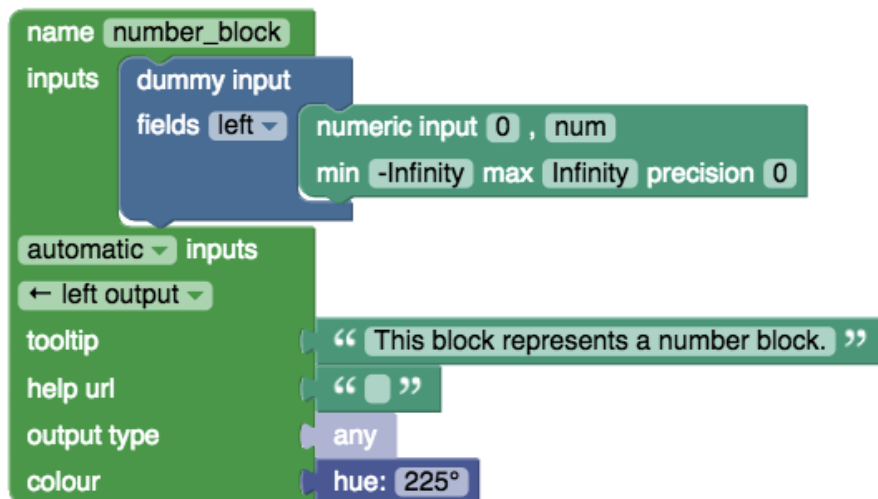


Solution

Index.html: Add a new category (VR Blocks) in the toolbox.

```
<category name="VR blocks" colour="225">
```

Block Factory:



blocks.js:

```
Blockly.Blocks['number_block'] = {
  init: function() {
    this.appendDummyInput()
      .appendField(new Blockly.FieldNumber(0), "num");
    this.setOutput(true, "Number");
    this.setColour(225);
    this.setTooltip("This block represents a number block.");
    this.setHelpUrl("");
  }
};
```

generator_stub.js:

```
Blockly.JavaScript['number_block'] = function(block) {
  var number_num = block.getFieldValue('num');
  var code = number_num;
  return [code, Blockly.JavaScript.ORDER_ATOMIC];
};
```

Add the *number* block to the *VR blocks* category:

index.html:

```
<block type="number_block"></block>
```

2.2. CYLINDER BLOCK (INDIVIDUAL TASK: PRINTSCREEN AND CODE)

Create a *cylinder block* that can be used to create a cylinder in the embedded **A-Frame** window. The structure of the block should be the following:



The user can specify the color, height, radius and the position (x,y,z) of the cylinder. We can use the [documentation of the A-Frame](#) to implement the code generation logic behind the *cylinder* block.

Hints:

- The most important part of the documentation: [JavaScript, Events, DOM APIs](#)
- We can use [appendChild\(\)](#) to add an 3D entity to the DOM.

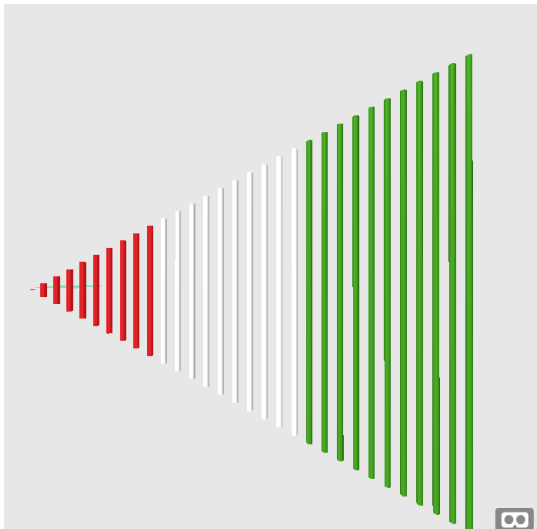
```
var sceneEl = document.querySelector('a-scene');  
var entityEl = document.createElement('a-entity');  
// Do `.setAttribute()`s to initialize the entity.  
sceneEl.appendChild(entityEl);
```

- We can use [setAttribute\(\)](#) to bind the parameters (height, radius, x ,y ,z) of the cylinder to the interface of the cylinder block.
- Use the dependencies of A-Frame in the code generation logic.

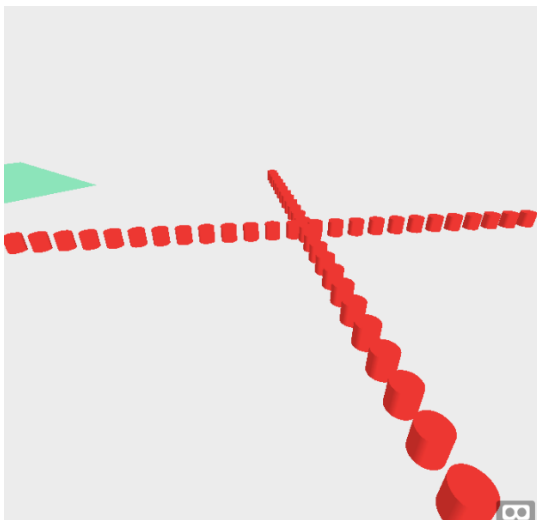
2.3. REPRODUCTION OF CYLINDERS (INDIVIDUAL TASK: PRINTSCREEN AND CODE)

Now, it is time to use visual language to reproduce cylinders with for loops.

a, Create cylinders next to each other, which increase in size and has three different colors. For example:



b, Create an X shape from the cylinders. For example:



c, Create a 3D grid of cylinders (e.g. 10x10x10). The cylinders should be displayed in the grid points:

