Documentation for the BioMotion Lab (BML) Experiment Toolkit

Author: Adam Bebko

# Contents

[Contents 1](#_Toc2241368)

[About 2](#_Toc2241369)

[Overview 2](#_Toc2241370)

[Getting started 4](#_Toc2241371)

[Configuring your experimental design 4](#_Toc2241372)

[Variable options 6](#_Toc2241373)

[Independent variable options: 7](#_Toc2241374)

[Dependent variable options: 9](#_Toc2241375)

[Controlling an experiment **Error! Bookmark not defined.**](#_Toc2241376)

[The Experiment Runner Window **Error! Bookmark not defined.**](#_Toc2241377)

[In-Trial Experimenter Controls 9](#_Toc2241378)

[Structure of an experiment **Error! Bookmark not defined.**](#_Toc2241379)

[Session 10](#_Toc2241380)

[Experiment 10](#_Toc2241381)

[A Note on Coroutines and IEnumerators 11](#_Toc2241382)

[Blocks 12](#_Toc2241383)

[Trial 14](#_Toc2241384)

[Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts 15](#_Toc2241385)

[Debug Mode 16](#_Toc2241386)

[Output 16](#_Toc2241387)

[Logging 17](#_Toc2241388)

[Customizing further 17](#_Toc2241389)

# About

The BML Experiment Toolkit helps you design and run experiments in Unity quickly and iteratively without fussing over coding details. You define your variables and experiment structure, and the toolkit will automatically create a table of trials to run. You can customize what happens before, during, and after a trial, block of trials, or the experiment itself, while not having to worry about the details of setting up and running your experiment.

You configure an experimental design, write custom behavior and functionality, and then the toolkit will automatically construct your experiment, create and run all required trials, and outputs the results to a file.

# Requirements

* Basic coding experience with C# and Unity. At the bare minimum you should know about
  + C#
    - Basic Object-Oriented Programming Techniques
      * Classes, inheritance
      * Methods, functions
      * variables, fields, properties
      * Overriding methods
    - Overloading methods and properties
    - Casting objects to other types.
  + Unity
    - Creating a project, scene, GameObjects.
    - Working with Transforms.
    - Dragging object references in the inspector
    - Difference between assets and objects in the scene.
    - Basic MonoBehaviour scripting.
    - Exposing public variables of MonoBehaviours in the inspector.
    - Unity Coroutines (see main documentation for brief overview)
* Latest version of BML Toolkit. Download here
* Compatible with Unity 2019.1 or later.
* Compatible with SteamVR (although works fine without VR)
* Suggested tools:
  + Visual Studio 2017
    - Visual Studio Unity Tools extension installed.
  + [Highly recommended] JetBrains Resharper Plugin for Visual Studio.
    - Free for students
    - It makes your coding life way easier and saves a ton of time. It keeps your code clean and suggests changes/problems that may come up.
    - Resharper also has its own Unity Extension for unity-specific help.
    - <https://www.jetbrains.com/resharper/>

# Overview

The main structure of an experiment is defined with a variable configuration file. In this file you define your independent, dependent, and participant variables, and any blocking or randomization that needs to occur.

Once the variable structure has been defined, you can write scripts inheriting from the toolkit’s base classes, customizing functions that are called automatically during the experiment. For each “stage” of the experiment (Experiment, Block, Trial), you can define custom behavior that interacts with your Unity scene. Examples of such customization might include a “welcome screen” before the experiment, a “thank you” screen after the experiment, a pause between each block of trials, and instructions before each trial. These customizations can be written very quickly.

Figure 1: Basic structure of an automatically generated experiment. You can customize functionality for all functions except grey ones. You must define the Main() function for your trials to define what should happen in each trial.

The main requirement for setting up an experiment is defining what occurs during a trial. A trial script has access to each of the variables you defined in the config file, so you can set up objects in your scene using values from the independent variable and write results to the dependent variables.

After opening up the main Experiment Runner Window and hitting play in unity, a new experimental session is created, and a window appears prompting you for session details such as participant ID and which block order to use.

The trial structure is displayed to double-check correctness, and then you can begin the experiment. The experiment runs, automatically running the custom code you defined above at the appropriate time.

Steps to Creating an experiment:

1. Create new project, configure it, and import the BML Experiment Toolkit
2. Set up Unity Scene with required GameObjects etc.
3. Create and set up a VariableConfiguration file
4. Create a custom ExperimentRunner class and drag it on a GameObject.
5. Create a custom Trial Class, override at minimum the MainCoroutine() method for custom behavior
6. Point ExperimentRunner to the custom Trial class you created.
7. Open the Runner window, press play in Unity, and run the experiment.

# Getting started

Please check out the tutorials, which should give you a handle on how to get up and running using the toolkit to run your experiments.

* Tutorials are in the documentation folder of the Github Repository
* TutorialExperiment: Creating your first experiment (Start here)
* Advanced trials coming soon.

# Creating a new Experiment

Creating a new experiment is best done in a new project.

1. Some settings are required:
   1. API compatibility level to .NET 4.0
   2. Scripting Runtime Version .Net 4.x Equivalent
2. Import the latest release of the toolkit .unitypackage.

# Configuring your experimental design

To set up your experimental design you need to define your variables. Create a new Config file from the create asset /BML Asset menu. It will appear in your project’s Assets folder. You can also access this menu by right-clicking on a folder in your project and navigate to Create/BML Assets.

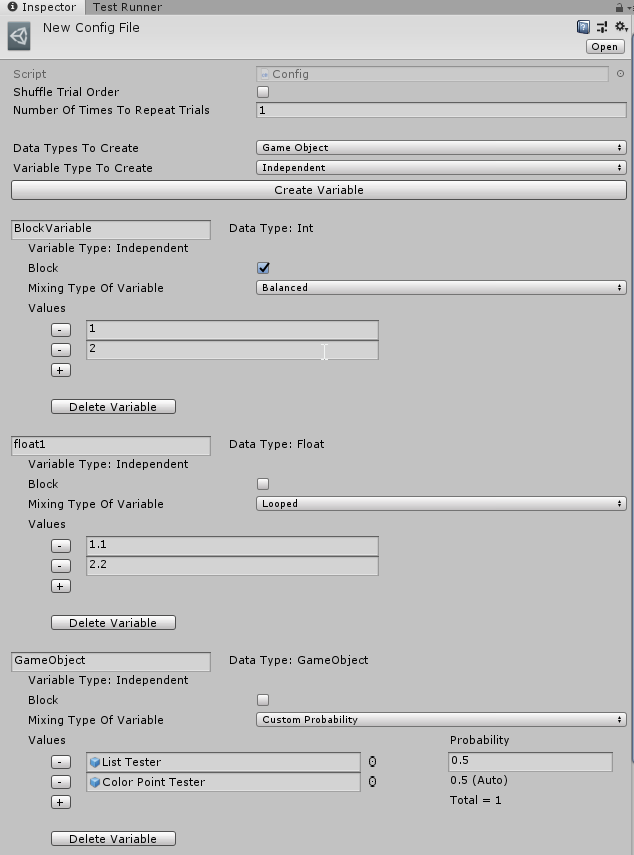


Figure 2: A screenshot of the variable creation inspector.

To add a variable, select your desired data type (int, string, etc.), type of variable (independent, dependent), and then click on Create Variable. You should see your variable added to the list below. Configure your variable by naming it, adding values, selecting whether the variable will be used to create blocks of trials, etc.

To delete a variable from the list click the Delete Variable button

Your changes will be saved automatically.

When complete, drag your config file into your Custom Experiment GameObject Prefab, and it will run.

The power of the configuration files is that you can run the same experiment with multiple saved configurations. For example, you can select different values for your experiment in different config files and drag your desired configuration into the experiment object prior to running participants. This also allows you to iteratively design your experiments while saving previous configurations.

## Variable options

#### Name

You must name your variables. This will be how you access their values within your trials. I recommend using one-word names, or joining words using underscores. Do not use any special characters or punctuation.

#### Datatype

Once created, you cannot modify this variable’s datatype. This lets Unity know what kind of data to expect. To get a value from a variable in a trial you must know its datatype to avoid errors. Currently, supported datatypes include:

* Int
* Float
* String
* GameObject
* Vector3

#### Variable Type

Independent and dependent variables are treated differently. Independent variables define your experimental structure, and dependent variables are values that determined from responses or measurements during each trial

### Independent variable options:

#### Block

Enabling this option will create blocks of trials for each value defined. Useful for counterbalancing or variables that require setup in real life (i.e. participants need to move between values)

#### Mixing Type of Variable

This option will define how the variable is mixed with the other variables when creating your experimental design.

**Balanced:**

This will create trials for every combination of values of each balanced variable. Example: for two 3-value balanced variables, there will be 3x3=9 trials.

|  |  |  |
| --- | --- | --- |
| Trial Number | Balanced Variable 1 | Balanced Variable 2 |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 3 |
| 4 | 2 | 1 |
| 5 | 2 | 2 |
| 6 | 2 | 3 |
| 7 | 3 | 1 |
| 8 | 3 | 2 |
| 9 | 3 | 3 |

**Looped:**

This will loop through the values such that there is an equal number of trials with each value. Lowest common multiple.

Eample1: For a variable with 2 values, and another variable with 4 values, the following table will be created. In this case the lowest common multiple is 4 trials.

|  |  |  |
| --- | --- | --- |
| Trial Number | Looped Variable 1 | Looped Variable 2 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 1 | 3 |
| 4 | 2 | 4 |

Example2: For a variable with 2 values, and another variable with 3 values the following table. In this case the lowest common multiple is 6 trials.

|  |  |  |
| --- | --- | --- |
| Trial Number | Looped Variable 1 | Looped Variable 2 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 1 | 3 |
| 4 | 2 | 1 |
| 5 | 1 | 2 |
| 6 | 2 | 3 |

**Even Probability:**

Each trial will have a randomly selected value for this variable, with the same probability for each value.

Example, A balanced variable with 6 levels, and an even probability variable with 10 values (numbers 1-10)

|  |  |  |
| --- | --- | --- |
| Trial Number | Balanced Variable 1 | Even Probability Variable 2 |
| 1 | 1 | 4 |
| 2 | 2 | 8 |
| 3 | 3 | 9 |
| 4 | 4 | 4 |
| 5 | 5 | 7 |
| 6 | 6 | 2 |

**Custom Probability:**

Each trial will have a randomly selected value for this variable, with a defined probability of being selected. You define the probability to the right of each value (as a decimal). The final probability is automatically calculated to ensure they add up to 1.

Example, A balanced variable with 6 levels, and a custom probability variable with 2 values with the first value having 0.2 probability, and the second value having 0.8 probability

|  |  |  |
| --- | --- | --- |
| Trial Number | Balanced Variable 1 | Custom Probability Variable 2 |
| 1 | 1 | 2 |
| 2 | 2 | 2 |
| 3 | 3 | 2 |
| 4 | 4 | 1 |
| 5 | 5 | 2 |
| 6 | 6 | 2 |

#### Values

Define the levels of each variable. You must define at least one value.

#### Probability

Only visible when custom probability mixing type is selected. This is where you define your probability of each value being picked.

### Dependent variable options:

#### Default value

This is the value given to your dependent variables in case there is no response given, or the variable is not updated in your trial. This can be left blank if desired.

#### Settings

You can modify the default settings here (advanced users only). Copy the default settings, edit them, and drag the new settings object here.

### Participant variable options:

#### Constrain Values

When checked forces values to be among a specified set of values.

#### Values

These are the set of values that are allowed when Constrain Values is checked.

# Starting, Controlling, and Monitoring an Experiment

## Experiment Runner window

Once the experiment’s variables have been configured, the experiment can be run from the Experiment Runner Window. You can open this window from the BML menu. This window can be docked and moved around just like any other editor window.

This is the main display and control for the toolkit. Once play mode is started, you will control and monitor the experiment from here. The window interfaces with a Custom ExperimentRunner class that you define (see below), to notify the unity scene to set up and begin the experiment.

Press play in the unity editor. The unity scene will begin, but the experiment will not yet run. It will prompt you for relevant settings, including picking which block order to use.

When all required settings have been selected, the Experiment controls will be displayed, allowing you to begin the experiment. The experiment will show the trial structure of your experiment and track your progress through the experiment.

## In-Trial Experimenter Controls

During the experiment you can jump between trials by pressing the “go” button next to the trial in the Experiment Runner Window. Any incomplete trials will be revisited at the end of the block. It is not currently possible to jump between blocks. This might be useful if your participant says they made an error. Additional attempts on a trial will be recorded in the output in the “Attempts” column.

The experimenter can also skip between trials using the keyboard using the following default controls:

* Navigate between trials (back, next, etc.) using the WASD keys.
* Skipped trials will be automatically repeated at the end of the current block opf trials.
* Skip trial completely using the X key.
* You can modify the keys through the control settings located in the Data folder.

## Session

The session class defines one time through an experiment. It stores options that affect the whole experiment. You can set options such as debug mode for testing out your experiment, the participant’s ID, block order number and the output file. This class doesn’t need to be modified except for advanced uses.

When first pressing play, the Experiment Runner Window will prompt you for session settings when running your experiment program. By default, the session settings will display your previously-used settings.

Debug mode will simplify your session settings for testing and save your output into a debug file created in your project’s Assets/Debug/debugFile. It can also be used to change functionality in your trials for testing purposes. See the section below on Debug Mode.

You can manually name your output files, or let the toolkit do it for you based on the current date, time, and participant ID.

This is where you choose your desired block order. Or you can have it randomly select the block order for you [Not yet implemented].

# Interfacing the experiment with your unity scene

### Experiment Runner

The ExperimentRunner MonoBehaviour Script is the main connection point between the experimental system and your unity scene. This script interfaces with the Experiment Runner Window to control and set up your experiment.

ExperimentRunner is an abstract class, meaning you need to create your own unique version of it for each experiment. To create a custom ExperimentRunner script, create a new C# script, and have it inherit from ExperimentRunner rather than MonoBehaviour. You will be prompted by visual studio to import the proper namespace.

Now, within this script you can declare public fields for any references to other unity objects in your scene that may be required.

The ExperimentRunner also needs a reference to a variable configuration file (see above). Drag one into the appropriate field in the inspector.

# Defining parts of your experiment (Experiment, Block, Trials)

The experiment is separated into several parts that are nested together

The ExperimentParts type has three subtypes:

Experiment (usually only one)

Creates and runs all blocks

Block

Creates and runs trials in block

Trial

Runs main logic of experiment (stimulus presentation, data collection, etc.)

These parts come pre-defined with all the behaviour you need to run a simple experiment. The only undefined behaviour is what happens during a trial. At the bare minimum you need to create a custom Trial and define the behaviour that occurs in the MainCoroutine method.

When you customize behaviour for your Experiment, Block, Trial (required) scripts, you need to tell the ExperimentRunner to point to those scripts rather than the default. You tell the ExperimentRunner about your custom ExperimentParts using the following procedure:

Inside your custom ExperimentRunnerScript:

public override Type TrialType => typeof(YourCustomTrialScript);

public override Type BlockType => typeof(YourCustomBlockScript);

public override Type ExperimentType => typeof(YourCustomExperimentScript);

doing this will allow you to define custom behaviour in the trials, blocks, and experiments that are automatically created for you by the framework.

To define custom behaviour inside your custom ExperimentParts, the framework automatically calls several methods that you can override to implement your own behaviour.

To learn more about Coroutines, see section below on Coroutines.

In call order:

PreMethod() – Called at the start of the part, useful for setting simpler things up, presenting instructions, calibration etc. Such setup should be done in a single frame.

PreCoroutine() – Called after PreMethod(), useful for setup that spans more than one frame. For example, displaying instructions to the participant for 5 seconds, or until a key is pressed.

MainCoroutine() – The “business” code of the ExperimentPart. For trials this is where the main part of the Trial should be coded (participant responses, any variable setup). This method can only be overwritten within Trials. This is because in Blocks and Experiments, the main method is to pass control to other ExperimentParts (i.e. an Experiment runs its blocks, a Block runs its trials).

PostCoroutine() – Called after MainCoroutine(), useful for cleanup that spans more than one frame.

PostMethod() – Called after PostCoroutine(), useful for cleanup that takes places within a single frame. Useful for writing output data.

For example:

1. using BML\_ExperimentToolkit.Scripts.ExperimentParts
2. **public** **class** MyTrial : Trial {
3. **protected** **override** IEnumerator PreCoroutine() {
4. //Your code here that might last for more than one frame
5. **yield** **return** **null**;
6. }
7. **protected** **override** IEnumerator MainCoroutine() {
8. //Main Trial Code that can last for more than one frame
9. **yield** **return** **null**;
10. }
11. **protected** **override** IEnumerator PostMethod() {
12. //Your code here
13. **//**No yield return needed here
14. }

17. }

#### Accessing other GameObjects from within your experiment code

To access other objects in your unity scene within your custom scripts, you must access the Runner object that is stored in every ExperimentPart. However, the stored Runner is a generic ExperimentRunner, not your custom type. Therefore you must cast it to your custom type to access any objects or variables stored within. Like so:

YourCustomRunnerType customRunnerType = (YourCustomRunnerType)Runner;

## Trial

The code define in your custom Trial class is general for all trials, but the behaviour changes based on the values of your variables. Therefore, you only need to code the behaviour for all trials in one place and set up each trial based on the values of your variables for that trial.

To define the behavior that occurs during a trial, you need to create a script that inherits from the Trial type

1. using BML\_ExperimentToolkit.Scripts.ExperimentParts
2. **public** **class** MyTrial : Trial {
4. }

However, this will give you an error. In most editors you can automatically solve this by right-clicking on MyTrial and selecting “implement missing members”. The Editor just created a constructor method. All it does is forward that job up to the main Trial class (base), so you don’t have to worry about it.

Now you can override the ExperimentPart methods described above to implement custom behaviour.

Remember, that the code defined in your custom Trial script is general for all Trials, but the behaviour changes based on the values of your variables. Therefore, you only need to code the behaviour for all trials in one place and set up each one based on the values of your variables for that trial.

As a rule of thumb, it’s a good idea to follow the following structure:

* In PreMethod():
  + Cast the runner object to your custom type, store in field so it can be used below.
  + Access your independent variables to set up trial.
  + Set up your environment for each trial.
  + Set up stimuli, other variables.
* In PreCoroutine():
  + Show trial-specific instructions.
  + Wait for user to do something before trial starts.
  + Remember to yield return. (see coroutine section in this guide)
* In MainCoroutine():
  + Present stimuli.
  + Collect responses.
  + Take measurements.
  + Remember to yield return. (see coroutine section in this guide)
* In PostCoroutine():
  + Wait for user to do something after trial.
  + Remember to yield return. (see coroutine section in this guide)
* In PostMethod():
  + Finalize any measurements.
  + Write data to your dependent variables.
  + Reset everything in preparation for next trial.

For Trials, after PostMethod() completes, the trial automatically updates the output .csv file with its variables’ values.

#### Access variables within a Trial

Accessing your variables within trials is easy. Their values are stored in a field of the Trial class called “Data”. This Data object will be updated and set to the correct values for that trial automatically by the toolkit.

Let’s say you defined an integer-type independent variable named “MyFirstVariable” with values 1 and 2. To access it from a trial, write the following:

1. 1.  **int** myFirstVariable = (**int**) data[“MyFirstVariable”]

This stores each trial’s value into a new int variable called myFirstVariable which can be manipulated normally like any other variable. You need the (int) at the start to remind C# that your variable is type int.

Now you can modify things for each trial based on the value. For example, you could move an object based on its value:

1. 1.  Vector3 positionToMoveTo = **new** Vector3(myFirstVariable, 0, 0);
2. 2.  someGameObject.transform.localPosition = postionToMoveTo;

Note: See the section on Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts for more information about customizing your trials.

Now, each trial will move someGameObject to the correct position for that trial.

#### Writing output measurements to dependent variables in a Trial

To write responses or results to your dependent variables, write the following

1. 1.  **float** response = 5.6f;
2. 2.  data[“MyFloatDependentVariable”] = response;

Note: Any updated values for the dependent variables will be automatically added to the output CSV file on completion of the trial.

## Block

If you flag any of your independent variables as blocking variables, the toolkit will automatically create blocks for you, and run them without any customization required.

However, If you’d like to customize what happens when a block starts or ends, you can override its ExperimentPart methods. First you’ll need to create a script that inherits from Block:

1. using BML\_ExperimentToolkit.Scripts.ExperimentParts
2. **public** **class** MyBlock : Block {
4. }

However, this will give you an error. In most editors you can automatically solve this by right-clicking on MyBlock and selecting “implement missing members”. This creates a constructor. All it does is forward that job up to the main Block class (base), so you don’t have to worry about it.

To customize behaviour in your Block, overwrite the ExperimentPart methods as desired.

As a rule of thumb, it’s a good idea to follow the following structure:

* In PreMethod():
  + Cast the runner object to your custom type, store in field so it can be used below.
  + Access your independent variables to set up the block (see below for more info)
  + Set up your environment for each block.
* In PreCoroutine():
  + Show Block-specific instructions.
  + Wait for user to do something before Block starts.
  + Remember to yield return. (see coroutine section in this guide)
* In MainCoroutine():
  + No customization allowed (runs trials in block automatically)
* In PostCoroutine():
  + Wait for user to do something after Block.
  + Remember to yield return. (see coroutine section in this guide)
* In PostMethod():
  + Reset everything in preparation for next Block.

For example, in one of my experiments, one of my blocking variables was where the participants stood. So, before each block I gave them instructions to move to a location, and only allowed the program to continue if they were standing in the correct location.

Remember that the code defined in your custom Block class is general for all blocks, but the behaviour changes based on the values of your variables. Therefore, you only need to code the behaviour for all blocks in one place and set up each one based on the values of your variables for that block.

#### Access variables within a Block

Accessing your variables is easy. Their values are stored in a field of the Block class called “data”. This “data” object will be updated and set to the correct values for that block automatically by the toolkit.

Important: Only independent variables flagged as “Block” will be accessible, and an error will be given otherwise.

Let’s say you defined an integer-type independent block variable named “MyIntBlockVariable” with values 1 and 2. To access it from a Block, write the following:

1. **int** blockVariable = (**int**) data[“MyIntBlockVariable”]

This stores each Block’s value for that variable into an int called blockVariable which can be manipulated normally like any other c# variable. You need the (int) at the start to remind c# that your variable is type int.

Now you can modify things for each block based on the value. For example, you could move an object based on its value:

1. Vector3 positionToMoveTo = **new** Vector3(blockVariable, 0, 0);
2. someGameObject.transform.localPosition = postionToMoveTo;

now, each block will move someGameObject to the correct position for that trial.

Note: See the section on Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts For more information on how to customize your Blocks.

## Experiment

To Define custom behavior experiment, you need to create a new script and have it inherit from the Experiment class. You’ll have to import the appropriate namespace

1. using BML\_ExperimentToolkit.Scripts.ExperimentParts
2. **public** **class** MyExperiment : Experiment {
4. }

Customize it similarly to Blocks (described above)

As a rule of thumb, it’s a good idea to follow the following structure:

* In PreMethod():
  + Initialize your experiment (calibration, etc.)
* In PreCoroutine():
  + Show instructions, welcome screen.
  + Wait for user to do something before Experiment starts.
  + Remember to yield return. (see coroutine section in this guide)
* In MainCoroutine():
  + No customization allowed (runs Blocks automatically)
* In PostCoroutine():
  + Wait for user to do something after Experiment.
  + Thank you screen
  + Remember to yield return. (see coroutine section in this guide)
* In PostMethod():
  + Finalize your experiment (confirm calibration still valid, etc.)

# A Note on Coroutines and IEnumerators

Normal loops in normal code keep looping until done, and don’t allow for you to extend code across multiple frames in a unity program.

For example, the following code will hang your program since it never lets the program get to the next frame.

1. **void** NormalKindOfMethod() {
2. **while** (**true**) {
3. //do something
4. }
5. }

What Coroutines and IEnumerators let you do is have a loop, but then pause at certain points to let the rest of the program run. Then, during the next frame of your program, it will pick up where it left off. Therefore, your program will run normally, but the loop is updated each frame.

1. IEnumerator CoroutineMethod() {
2. **while** (**true**) {
3. //do something
5. yield **return** **null**; //continue below this after next frame
7. Debug.Log("this will print to console the next frame");
8. }
9. }

To begin a coroutine method, you cannot call it like a normal method. You need to use a special unity function called StartCoroutine(). This functionality is already written for you in the toolkit. For a Coroutine function, you need only tell Unity where to pause to wait for the next frame using a “yield return” statement. You can call normal methods inside coroutine functions, and everything else behaves as you would expect it to, with the added power of being able to have behavior that occurs over many frames.

In addition to waiting for a frame, you can have unity wait for a specified amount of time before continuing. This is great for displaying instructions. Or for creating time delays in your code. You can have unity display some instructions to a participant, wait for a few seconds, then stop displaying the instructions. Below is an example of displaying instructions at the start of your program for 5 seconds.

1. **protected** **override** IEnumerator Pre() {
3. DisplayInstructions(); //called right away
5. //the rest of your program will run normally while it waits.
6. yield **return** **new** WaitForSeconds(5);
8. //will only get called after 5 seconds
9. StopDisplayingInstructions();
10. }

13. **void** DisplayInstructions() {
14. //your code for displaying
15. }
17. **void** StopDisplayingInstructions() {
18. //your code for stopping to display
19. }

# Accesssing Unity Components and MonoBehaviours from inside your custom experiment/block/trial scripts

The Experiment, Block, and Trial classes reside in the voids of C# and don’t really interact with the unity system at all. For this reason, you need to create a link between your unity project and your custom trial script.

To do this, you need to create references to them in your custom ExperimentRunner class. For example:

1. **public** GameObject gameObjectThatMoves

Then, in your custom Trial script you can reference your ExperimentRunner using the following

1. MyCustomExperimentRunner myRunner = (MyCustomExperimentRunner)Runner;

This line the base ExperimentRunner, but instead as your custom class.

Then you can access the gameobject with:

1. myRunner.gameObjectThatMoves;

This allows you to use your ExperimentRunner object to drag and drop refrences to GameObjects and other things in your unity scene that need to be controlled or accessed during your experiment.

You can extend your custom options class as much as you want to add as much functionality to your trials as you need.

# Debug Mode

You can run your experiment in Debug Mode during testing. This is useful for quickly prototyping changes without having to set everything up each session.

You can run in debug mode from the Experiment runner window (in the BML menu)

Selecting debug mode in the session settings has several advantages.

* It allows you to save output without the possibility of overwriting your real output files
* You can change the functionality of your experiment when debug mode is on.
  + You code special functionality for debug mode using something like the following:
    - Not yet implemented
* Output from debug mode is saved in Assets/BML\_Debug folder.

# Output

By default, the toolkit outputs your experiment results as a .CSV file after the completion of every trial. It automatically numbers your trials by block and by trial number inside each block. It adds a column for each variable including your dependent variables. You can change the names of the automatically added columns in the Data Folder under Settings.

To add a custom column, create a dependent variable of the appropriate type, and set its value in each trial.

The best place to write data to your dependent variables is in the PostMethod() method of your custom Trial script.

# Logging

The BML Experiment Toolkit keeps a log of all sessions. You can access it in the Assets > BML\_ExperimentToolkit > Data Folder

# Customizing further

The package is provided open source, so you can make any modifications you like. If you notice a missing feature, or write something other people might use, consider letting us know so we can add it to the package to be enjoyed by everyone.

# Troubleshooting

1. My MonoBehaviour script is not showing up in the inspector.
   1. Make sure the Class name matches the Filename exactly. MonoBehaviours require this.
2. My custom ExperimentRunner script is not showing up in the inspector. Or I can’t drag it to a GameObject in the scene
   1. Make sure the Class name matches the Filename exactly. MonoBehaviours require this.
   2. Make sure there are not syntax errors in any code in your project. One error will prevent unity from dragging MonoBehaviours to objects.