In this tutorial, we will learn to implement a simple experiment design using PennController for IBEX. Click here to see the project that we want to reproduce. While this currently serves primarily as a toy example, there’s an empirical question of theoretical interest underlying it, namely: What is the time course of interpreting the agreement morphology in English, in particular the presence or absence of an -s on a verb?

A good exercise when you start designing an experiment is to try to schematize the general flow of events. In this case, we’re aiming for the following components:

1. A welcome screen with
   (i) some introductory text,
   (ii) a text input box to record a participant ID, and
   (iii) a button to start the experiment
2. A series of trials all following the same pattern:
   i. A screen that’s initially blank for a brief period of time
   ii. Two pictures appearing side by side
   iii. A Brief pause
   iv. A sentence unfolds both visually as text and auditorily from a sound file
   v. Wait until one of the two pictures is selected (by either click or button press) and the audio is done playing
   vi. End of trial
3. A final screen with a confirmation link

This is what we will have at the end of this tutorial, but as a first step, we will start with something much more basic: a simple task on a single display screen showing one sentence and two pictures asking for a key press.
Getting started

PennController  April 28, 2019

Setup

Create a new experiment on the PClbex Farm (create an account first if you don’t already have one).

Open your project (click the name of your experiment) and find the text Update from git repo on the page: below it, you should see two input boxes and a Sync button—if you can't see them, just click on the text to unfold.

In the first input box (repo url) enter https://github.com/PennController/TimedPictureSelection Then click on the Sync button. You should see several filenames highlighted in red. It means that you successfully imported the resources that we will be using in this tutorial.

The script editor

Find the Script section. You should see a file named main.js. Open the script editor by clicking on the file.

Your file main.js should be rather empty for now: we will script our experiment from scratch. You will spend most of your time editing this file, so let’s get familiar with the interface of the script editor.

The top bar contains two links: one link to the PennController documentation and one link to the original Ibex manual. They are very helpful, e.g., when you can’t remember the name of a command or what it does.

As you start typing text in the editor, you will see a modular window appear. This is another feature that will help you with the commands: it will suggest command names for autocompletion. Start typing Pe and then press Enter/Return on your keyboard to write PennController in your script.

The autocompletion window is particularly helpful in reminding you of the exact names of the commands, for example which characters are lower- or upper-case. It also gives you very short descriptions of the commands. Once again, use the PennController documentation link if you need more information about the commands.

Finally, the bottom bar contains four buttons:

- The first one, Discard changes (same as the cross button at the top-right corner) closes the editor without saving its content: any changes are permanently lost.
- The second one, Save changes, simply updates the content of your file without closing the editor.
- The third button, Save and close, does the same thing but additionally closes the editor.
- The last button, Save and test, also updates the content of your file and opens your updated experiment in a new window afterward. This is probably the button you will use the most.

Next: Now that you are familiar with the script editor, we can start writing some code Text, Pictures and Keys.
Text, Pictures and Keys

Elements

Elements are the core pieces that PennController works with: they allow you to manipulate and present content such as text, images, audio or videos, but also to structure trials by inserting pauses and, crucially, collecting the participant’s input.

The main parts of PennController trial scripts therefore consist of sequences of commands relating to an element. When you want to show some text, you create a text element and use a command to print the text onto the screen. If you want to add a picture below it, you create an image element and again use a command to print it onto the screen.

With this in mind, consider the following code:

```javascript
newText("The fish swim in a tank which is perfectly round")
.print()
.
newImage("2fishRoundTank.png")
.print()
```

Try it

It’s probably pretty transparent what this does: it creates a text element with the content *The fish swim in a tank which is perfectly round* and prints it onto the screen. After the comma, an image element is created using the file 2fishRoundTank.png (which, like all the other files we’ll use here, was uploaded under Resources during the initial project updated from github) and printed onto the screen as well.

**Note:** You can use the Try it button here and below to test out (and play around with) bits of code without actually working on a full experiment on PCiBex Farm.

Walking through a trial

For integrating this into an experiment on the PCiBex Farm, just copying the script from above and pasting it into your file `main.js` isn’t quite enough. You need to embed this code within the PennController(...) environment, to make it part of a trial created with PennController. So try adding the following to your script, then click *Save and test* and see what happens:

```javascript
+ PennController(
  newText("The fish swim in a tank which is perfectly round")
  .print()
  .
  newImage("2fishRoundTank.png")
  .print()
+
```
You’ll find that this doesn’t work as intended yet. As soon as you open the page, you immediately reach the end of the experiment, with a message saying *The results were successfully sent to the server. Thanks!* *(Note: This problem doesn’t arise if you use the *Try it* environment, because it is configured slightly differently for sake of simple illustration.)*

This illustrates an important aspect of how scripts are executed: they carry on plowing through the sequence of commands until reaching the end. So what happened here is:

1. the trial started by executing the command on the very first line inside `PennController(...): newText("The fish swim in a tank which is perfectly round")`. It created a text element in the cache of the trial. Any command after this line relates to the newly created text element, until we reach a separating comma.
2. The second line, `print()`, thus relates to the very same text element, and even though you couldn’t see it when you tested your script, the text was actually added to the page.
3. Execution went on to the third line of the trial, where it encountered a comma indicating that a new series of commands was coming.
   The line `newImage("2fishRoundTank.png")` created an image element in the cache of the trial from the file `2fishRoundTank.png`. Any command after this line relates to the newly created image element.
4. The next and last line of the trial, `print()`, thus relates to the image element we just created with `newImage`. It adds the image to the page below the text element that was previously added.

And that was it, there were no commands left to be executed for the trial, so the trial ended, and it was the end of the experiment. All of that happened within a few milliseconds.

**Holding the press: Waiting for a key press**

If we want to pause this series of commands getting rapidly executed upon runtime in the order in which they appear in your script, so that your participants can interact with the page, we have to add a command that will halt things. This is where more interactive elements, such as the key element, come in handy. Try adding the lines prefaced with a '+' to your script:

```
1. PennController()
2.   newText("The fish swim in a tank which is perfectly round")
3.     .print()
4.     ,
5.     newImage("2fishRoundTank.png")
6.     .print()
7.     +,
8.     + newKey(" ")
9.     + .wait()
10. )
```

Try it

Up to the second comma (i.e., up to printing the image), everything proceeds exactly as before. But now, when execution of the script reaches `newKey(" ")`, the trial starts listening for a key press on the space bar. The last line, `.wait()`, relates to this key element and causes the execution to **pause until the space bar is pressed**. As a result, execution will not reach the end of the trial until your participants press the space bar, which will give them time to see what is on the page.
Two images, two keys

We now have all we need to ask participants to indicate which of two pictures matches a description by pressing one of two keys. We simply create a second image element in the cache using 1fishSquareTank.png that we print below the first one, and we replace the string " " in the argument of the key element with "FJ", so that it now pays attention to key presses on both the F and the J keys. (Note that the string of keys included here specifies a set of keys that cause the execution of the script to resume.)

```
PennController(
    newText("The fish swim in a tank which is perfectly round")
    .print()
    ,
    newImage("2fishRoundTank.png")
    .print()
    +
    newImage("1fishSquareTank.png")
    .print()
    ,
    newKey("FJ")
    .wait()
)
```

Visual Layout

Although we already have a basic functional version of what we’re aiming for, it would be best if our participants could see smaller renderings of the pictures, side by side.

Resizing elements

The command .settings.size makes it possible to address the first issue:

```
PennController(
    newText("The fish swim in a tank which is perfectly round")
    .print()
    ,
    newImage("2fishRoundTank.png")
    .settings.size(200,200)
    .print()
    +
    newImage("1fishSquareTank.png")
    .settings.size(200,200)
    .print()
    ,
    newKey("FJ")
    .wait()
)
```

Canvas

Now how about positioning the images side by side? A very simple and yet powerful element is the Canvas element: it defines a transparent rectangle surface on the page onto which you can add whatever element (even other canvas) wherever you want (even over other elements on the Canvas, like images). In our case, we simply want a 450x200px surface so we can place the images side by side while leaving a 50px gap between them:
We brought several changes at once here.

- For one, we now named the image elements: so far, we had only used them anonymously, as it were, because any commands operating on them, such as printing them, were executed immediately after creating them. But now, we need to give them names so we can refer back to them within a command that relates to another element, namely a canvas element.

- We also no longer use .print() commands on the image elements directly. Instead, we add the images to the canvas element, using .settings.add, and refer back to the previously created image elements using their names within getImage (a first example of a family of getX commands which serve to retrieve already created elements). Then we print the canvas element as a whole, which now contains the images, and this is how the images end up appearing onto the screen, side by side in the positions defined in the pixel parameter in the .settings.add commands.

The rest remained unchanged. In particular, the last command of the trial is still a .wait on the key element, which ensures that the screen won't be cleared until the F or the J key is pressed.

Next: Data collection basics
Collecting data

PennController automatically logs when a trial starts and when it ends, but you have to explicitly tell it what other information you want to collect. In our case, we are interested in what is captured by the key element. PennController has a command that logs element-related events: `.settings.log`. All we need to do in order to see a line reporting key presses appear in our results file is use this command within the `newKey` block, like this:

```javascript
PennController(
    newText("The fish swim in a tank which is perfectly round")
    .print()
    ,
    newImage("two", "2fishRoundTank.png")
    .settings.size(200, 200)
    ,
    newImage("one", "1fishSquareTank.png")
    .settings.size(200, 200)
    ,
    newCanvas(450, 200)
    .settings.add( 0, 0, getImage("two") )
    .settings.add( 250, 0, getImage("one") )
    .print()
    ,
    newKey("FJ")
    +
    .settings.log()
    .wait()
)
```

Inspecting the results file

To look at the results do the following:

1. Save and Close the `main.js` file.
2. Delete any existing results files on your main experiment project page.
3. Do a full run-through of the experiment to collect data.

After this, you should see two files under **Results** on the main page of your project. (If you can't see them, try clicking the refresh icon next to **Results.**) We will work with the file `results`, which is a comma-separated-value (CSV) file. Click on `results` to see its content.
The first thing to note is that lines starting with # are comments providing information on how and when the data was collected, and on what the values separated by commas in the lines below represent. You don't have to spend time on these for now.

For each time that you've taken the experiment, you will now see 3 lines. The first and the last are the beginning and end of the trial with timestamp information; these are recorded for any PennController trial and say Start and End in one of the columns towards the right end.

In addition, there is a line containing the value Key between the Start and the End lines. Here, the third value from the end, next to KeyPressed, indicates the key that was pressed. The next value (second from the end) is the timestamp of this event. Subtract it from the timestamp of the Start line to get the response time. When you have multiple trials, you can use this method to compare response times in different conditions, e.g., for trials with and without an -s on the verb in our example experiment.

This was a first quick introduction to the results file. As you collect more data, you will want to automatize the analysis process, which is documented later on in the tutorial. For now, let us see how to add another crucial piece of information to our results file, which adds an identifier for the participant to each line of the results.

Next: Participant information.
Welcome/Instruction screen

There are various good reasons to add a preface page (or even several) to your experiment, including:

1. the page can describe and give instructions about the experiment to your participants
2. the page can serve as a consent form
3. the page can ask the participant to type an ID

In PennController, every page corresponds to a trial, and accordingly has the same structure: To add a page before our picture-selection screen, we simply insert a new `PennController(...)` in our script above the one we currently have.

For starters, our welcome trial will simply print some text on the page, and invite participants to click a button to start the experiment. Insert the following after `PennController.ResetPrefix(null)` and before the line `PennController(...)` that you already have in your script:

```java
1. PennController(
2.    defaultText
3.      .print()
4.    ,
5.    newText("<p>Welcome!</p>"
6.    ,
7.    newText("<p>In this experiment, you will have to report which of two pictures matches a description.</p>"
8.    ,
9.    newText("<p>Press the <strong>F</strong> key for the picture on the left, or the <strong>J</strong> key for the picture on the right.</p>"
10.   ,
11.   newText("<p>Click the button below to start the experiment.</p>"
12.   ,
13.   newButton("Start"
14.     .print()
15.     .wait()
16. )
```

Some explanations: the first command block `defaultText.print()` tells the script to implicitly insert the command `.print` immediately below each `newText` command in the trial. (You can set defaults of all sorts for all types of elements following this pattern.) This way, we don’t have to repeat the `.print` command for each of the four text elements that we create next in the welcome trial.
As you can see, we wrapped `<p>` around our texts: these are HTML tags indicating paragraphs, which browsers render with top and bottom spacing, making for a nicer visual layout. The `<strong>` tags print the text in boldface.

The button element is new, but it should be straightforward: parallel to a text element, you create a button in the cache of the trial using `newButton` and print it using `.print`. Parallel to a key element, you pause the execution of the trial (which, in our case, would otherwise immediately reach the end) using the command `.wait`, which in this case listens for a click on the button. Execution resumes when the button is clicked and, since there are no commands left for the trial, the experiment moves on to the next PennController trial, namely the picture-selection screen we created before.

**Collecting IDs**

You typically will want to have some type of ID associated with each unique participant (e.g., for payment/credit attribution and for purposes of data analysis). You can ask them to type in an ID so you can report it on every line in your results file. (There also is a way to automatically extract IDs from URLs, as certain recruitment platforms will generate URLs containing additional information. See the full documentation for more details, and in particular the command `PennController.GetURLParameter`.)

To do this, you simply need to create and print a text input element using `newTextInput` and the command `.print`. But you also want to retrieve its content and append it to the end of each line in your results file. Here is how to do it:

```javascript
PennController(  
  defaultText  
  .print(  
    ,  
    newText("<p>Welcome!</p>")  
    ,  
    newText("<p>In this experiment, you will have to report which of two pictures matches a description.</p>"")  
    ,  
    newText("<p>Press the <strong>F</strong> key for the picture on the left, or the <strong>J</strong> key for the picture on the right.</p>"")  
    ,  
    newText("<p>Please enter your ID and then click the button below to start the experiment.</p>"")  
    ,  
    newTextInput("ID")  
    ,  
    .print(  
      ,  
      newButton("Start")  
      .print(  
        ,  
        .wait(  
          ,  
          newVar("ID")  
          .settings.global(  
            .set( getTextInput("ID") )  
          )  
        )  
        .log( "ID", getVar("ID") )  
      )  
    )  
  )  
)
```

Try it

Note first that we didn’t use `.wait` on the text input: we could have done so, and it would have paused the execution of the trial before adding the button to the page, until the participant presses Enter/Return on their keyboard while typing in the input box.

More crucially, we added some code at the end of the trial. After the Start button is clicked, the script executes line 20: `newVar("ID")`. This creates a var(iable) element in the cache of the trial. The next line, `.settings.global()`, makes this var element available for all subsequent trials.
Line 22, `.set( getTextInput("ID") )` lets the var element store the current content of the text input element. With no lines left to be executed in this PennController trial, the experiment goes on to the picture-selection trial.

Note that the last line we added, `.log( "ID", getVar("ID") )`, appears immediately after the closing parenthesis corresponding to `PennController()` from line 1. Its effect is to add a value to the lines in the results files generated by this welcome trial. The value that will be added to the results file is the one stored in the var element named `ID`.

If you test your experiment through the end and refresh your `results` file, you should see two lines corresponding to your welcome screen: one line for `Start` and one line for `End` (automatically reported by PennController). They should contain one more value than the lines we looked at before: that value is what you typed in the input box.

**Referencing global vars in later trials**

Since we made our var element `ID` global, every trial after the welcome screen can access it. This means that we can also append the command `.log( "ID", getVar("ID") )` to the closing parenthesis of our picture-selection trial, and the `ID` value will appear in the results lines of that trial as well. Here is our complete script so far:
Adding a completion screen

Currently, when your experiment is over, a submission confirmation message appears on the screen, but you might want to have a customized final screen for your experiment.

Again, the idea is to create a new PennController trial, but you'd usually want to make sure it appears after the results are saved (so that participants don't close their browser before data is saved).
You can take manual control over when, during your experiment, the results are sent using the command PennController.SendResults.

For illustration, let's say you want to show a link to the recruitment platform (e.g., for validation of experiment completion) on your final screen. Here is how you could implement this:

```javascript
newKey(
  .settings.log()
  .wait()
)
+ .log( "ID" , getVar("ID") )

PennController.ResetPrefix(null);

PennController(  
defaultText
  .print()
  ,
  newText("<p>Welcome!</p>"
  ,
  newText("<p>In this experiment, you will have to report which of two pictures matches a description.</p>"
  ,
  newText("<p>Press the <strong>F</strong> key for the picture on the left, or the <strong>J</strong> key for the picture on the right.</p>"
  ,
  newText("<p>Please enter your ID and then click the button below to start the experiment.</p>"
  ,
  newTextInput("ID"
    .print()
    ,
    newButton("Start"
      .print()
      ,
      .wait()
      ,
      newVar("ID"
        .settings.global()
        .set( getTextInput("ID") )
      )
    )
  )
)+ .log( "ID" , getVar("ID") )

PennController(  
  newText("The fish swim in a tank which is perfectly round"
    .print()
    ,
    newImage("two", "2fishRoundTank.png"
      .settings.size(200,200)
    ,
    newImage("one", "1fishSquareTank.png"
      .settings.size(200,200)
    ,
    newCanvas(450,200)
      .settings.add(0,0,getImage("two")
      .settings.add(250,0,getImage("one")
```
When execution reaches the end of the picture-selection trial, after the F or the J key has been pressed, the script now executes the line `PennController.SendResults()`, sending the participant’s responses to your results file. Once the results have been saved, the script goes on to the next and final trial, the completion screen, which contains a text element with the HTML `<a>` tag to generate a link.

We don’t want participants to go beyond this screen, so the last commands to be executed in this final PennController trial set up a little trick to make it impossible to move any further: `newButton("void")`, which creates a button element in the cache of the trial, together with `.wait()` pauses the execution until the button is clicked. But since `.print` was never encountered, the button never gets added to the page, and therefore execution never resumes, leaving the text on the page indefinitely.

Next: **Audio, Text unfolding and Clicks**
**Audio, Text unfolding and Clicks**

* PennController  April 28, 2019

---


---

### Audio

PennController makes playing audio simple: just create an audio element in the cache using `newAudio` and play it using the command `.play.

#### Note: the scripts on this page correspond to the content of the picture-selection PennController(...)

```javascript
1. + newAudio("2fishRoundTank.mp3")
2. + .play()
3. +,
4. newText("The fish swim in a tank which is perfectly round")
5. + .print()
6. +,
7. newImage("two", "2fishRoundTank.png")
8. + settings.size(200,200)
9. +,
10. newImage("one", "1fishSquareTank.png")
11. + settings.size(200,200)
12. +,
13. newCanvas(450,200)
14. + settings.add( 0, 0, getImage("two") )
15. + settings.add( 250, 0, getImage("one") )
16. + .print()
17. +,
18. newKey("FJ")
19. + settings.log()
20. + .wait()
```

---

### Try it

You might want more explicit control over what happens to audio play-back relative to potential key presses before the audio is over. Let us consider two approaches: (a) you could decide to stop playback when a key is pressed, or (b) you could decide to wait until playback is over before moving on to the next trial. Both of these require us to do something with the audio element after a key is pressed, in other words you will want to add commands after the key element’s `.wait` command, which will refer to the audio element you created at the beginning of the trial.

#### Stopping playback at key press

In order to refer back to your audio element, you first have to name it. Then you can add a new last block of commands relating to it at the end of your trial by adding a comma and referring back to your audio element using `getAudio`. Thus, after the key press (satisfying the condition imposed by `.wait` on the key element), the...
script evaluates `getAudio("description")` and will interpret the next commands as relating to the audio element that you named `description` (see line 1). If audio playback should stop when a key is pressed before it's over, just add `.stop` as the command on this audio element. After registering the key press, this will stop the audio and then move on in the script to the end of the trial. (If the audio was already finished at the time of the key press, the script will continue in its execution as well.)

Try it

Waiting until audio is over

Alternatively, if you want to wait until playback is over to end the trial, you would use a `.wait` command on the audio:

Try it
The command `.wait("first")` on the audio element will pause the execution, and therefore prevent the trial from ending, until the audio is done playing.

The "first" part in the parentheses of `.wait` helps to ensure proper interaction with key presses both during and after the audio is over. Without it, execution would only resume when the script detects an end-of-playback event. But if the key press occurs after the audio was over, there will be no more such end-of-playback events, and the condition imposed by `.wait` would never be met, pausing the trial indefinitely. The presence of "first" tells the `.wait` command to release execution whenever an end-of-playback event has occurred, either after or before the key press: this way, there will be no pause if the end-of-playback event happened before execution of line 23 and the trial will therefore end immediately after the key press, but if instead the key was pressed before the audio was done playing, execution is paused until the end-of-playback event.

**Text unfolding**

You already know how to *instantly* print a text element onto the page: just use `.print`. If you want it to unfold over time, simply use `.unfold` instead. To align unfolding (roughly) with the auditory playback, we can just time the unfolding relative to the audio file length. The duration of our audio file is approximately 2.6s, so we will use a duration of 2600ms for our `.unfold` command:

```javascript
newAudio("description", "2fishRoundTank.mp3")
  .play()
newText("The fish swim in a tank which is perfectly round")
  .unfold(2600)
newImage("two", "2fishRoundTank.png")
  .settings.size(200,200)
newImage("one", "1fishSquareTank.png")
  .settings.size(200,200)
newCanvas(450,200)
  .settings.add(0,0,getImage("two") )
  .settings.add(250,0,getImage("one") )
  .print()
newKey("FJ")
  .settings.log()
  .wait()
getAudio("description")
  .wait("first")
```

### Clicks

There is one last change we want to bring to our trial structure: in addition to pressing a key, we want to make it possible to click on the pictures to make a choice. This can seem trivial, but we currently pause the execution until selection happens using `.wait` on our key element. How can we instead pause the execution until (a) F or J is pressed, or (b) a picture is clicked?

Since all PennController commands are element-based, ideally we would want to use `.wait` on an element that represents a choice between two options: option (i) a press on the F key or a click on the picture on the left; option (ii) a press on the J key or a click on the picture on the right.
The selector element does just that: it groups elements together and makes them exclusively selectable through clicks or key presses:

```javascript
newAudio("description", "2fishRoundTank.mp3")
  .play()
,
newText("The fish swim in a tank which is perfectly round")
  .unfold(2600)
,
newImage("two", "2fishRoundTank.png")
  .settings.size(200,200)
,
newImage("one", "1fishSquareTank.png")
  .settings.size(200,200)
,
newCanvas(450,200)
  .settings.add( 0, 0, getImage("two") )
  .settings.add( 250, 0, getImage("one") )
  .print()
,  
  // newKey("FJ")
  + newSelector()
  + .settings.add( getImage("two") , getImage("one") )
  + .settings.keys("F", "J")
  .settings.log()
  .wait()
,
getAudio("description")
  .wait("first")
```

We replaced the key element with a selector element, to which we added the two images using `.settings.add( getImage("two") , getImage("one") )` which we respectively associate with the F and J keys using `.settings.keys("F", "J")` (spaces only for sake of transparency for the reader; PCIbex doesn’t care about them).

We kept the `.settings.log` command since we’re still interested in the selection event (which picture was selected, and when selection happened). We also kept the `.wait` command which now applies to the selector element and achieves our goal: it pauses execution until one of the elements added to the selector (that is, one of the two images) is selected, either through a click or through a key press.

Note that as things stand, it’s possible for participants to click multiple times, allowing them to change which picture is selected until the trial is over (here, when the audio play back ends, assuming at least one selection has been made). If you don’t want to allow this, insert a `.settings.once()` command just before `.wait()`, and no further clicks or key presses will alter the participant’s initial choice.

Next: Trial templates & tables
Multi-trial experiments using templates

Most experiments consist of multiple trials, typically varying levels of a factor, e.g., whether or not the inflectional morpheme -s is present or not in our example experiment. So the last step in designing our experiment is to create an entire set of trials.

We could copy and paste the PennController(...) defining our picture-selection trial, making changes along the way to display other sentences and using different audio and image files for the different experimental items. But then we would need to edit every copy whenever we want to change small aspects of the trial structure, e.g., adding a pause of some number of milliseconds at the beginning of each trial. This is cumbersome and seems unnecessary, since the various trials will usually have the exact same structure, apart from some key ingredients.

A more sensible approach is to define a template for the general trial structure and to identify the variable bits in your PennController(...), which can then be supplied from a table. In our case, there are four strings that will change on a trial-by-trial basis:

- "2fishRoundTank.mp3" in newAudio("description", "2fishRoundTank.mp3")
- "The fish swim in a tank which is perfectly round" in newText("The fish swim in a tank which is perfectly round")
- "2fishRoundTank.png" in newImage("two", "2fishRoundTank.png") and
- "1fishSquareTank.png" in newImage("one", "1fishSquareTank.png")

Everything else will be constant from one trial to the other. So we can respectively replace these bits with variable.AudioFile, variable.Description, variable.PluralImageFile and variable.SingularImageFile, and embed PennController(...) within PennController.Template( variable => ... ) to transform it into a template:

```
Note: your script should also still contain the welcome trial above this

PennController.Template(
  variable => PennController(
    newAudio("description", variable.AudioFile)
    .play()
    ,
    newText(variable.Description)
    .unfold(2600)
    ,
    newImage("two", variable.PluralImageFile)
```

}
Save your script and test your experiment. You should now see a series of 4 picture-selection trials. The values for the variables for these trials are supplied by a table, whose structure we turn to next.

### Table

Importing the resources at the beginning of this tutorial from github uploaded various files (such as the image and audio files we already used) to the `Resources` section of the main project page. This also included a file named `fulldesign.csv`. This file is a table created using a spreadsheet editor and saved in a comma-separated-value (CSV) format (i.e., same format as the results file). Below is a rendering of the table:

<table>
<thead>
<tr>
<th>AudioFile</th>
<th>Description</th>
<th>PluralImageFile</th>
<th>SingularImageFile</th>
<th>Item</th>
<th>Group</th>
<th>Ending</th>
</tr>
</thead>
<tbody>
<tr>
<td>1fishSquareTank.mp3</td>
<td>The fish swims in a tank which is perfectly square</td>
<td>2fishRoundTank.png</td>
<td>1fishSquareTank.png</td>
<td>fish</td>
<td>A</td>
<td>-s</td>
</tr>
<tr>
<td>2fishRoundTank.mp3</td>
<td>The fish swim in a tank which is perfectly round</td>
<td>2fishRoundTank.png</td>
<td>1fishSquareTank.png</td>
<td>fish</td>
<td>B</td>
<td>No-s</td>
</tr>
<tr>
<td>1deerDenseWood.mp3</td>
<td>The deer runs in a wood which is extremely dense</td>
<td>2deerSparseWood.png</td>
<td>1deerDenseWood.png</td>
<td>deer</td>
<td>B</td>
<td>-s</td>
</tr>
<tr>
<td>2deerSparseWood.mp3</td>
<td>The deer run in a wood which is extremely sparse</td>
<td>2deerSparseWood.png</td>
<td>1deerDenseWood.png</td>
<td>deer</td>
<td>A</td>
<td>No-s</td>
</tr>
<tr>
<td>1sheepRedPen.mp3</td>
<td>The sheep roams in a pen which is strikingly red</td>
<td>2sheepBluePen.png</td>
<td>1sheepRedPen.png</td>
<td>sheep</td>
<td>A</td>
<td>-s</td>
</tr>
<tr>
<td>2sheepBluePen.mp3</td>
<td>The sheep roam in a pen which is strikingly blue</td>
<td>2sheepBluePen.png</td>
<td>1sheepRedPen.png</td>
<td>sheep</td>
<td>B</td>
<td>No-s</td>
</tr>
<tr>
<td>1mooseNewPark.mp3</td>
<td>The moose walks in a park which is visibly new</td>
<td>2mooseOldPark.png</td>
<td>1mooseNewPark.png</td>
<td>moose</td>
<td>B</td>
<td>-s</td>
</tr>
<tr>
<td>2mooseOldPark.mp3</td>
<td>The moose walks in a park which is visibly old</td>
<td>2mooseOldPark.png</td>
<td>1mooseNewPark.png</td>
<td>moose</td>
<td>A</td>
<td>No-s</td>
</tr>
</tbody>
</table>

The template we created looks for columns with corresponding names in the CSV table based on the specified variables, such as `variable.AudioFile`, (e.g., AudioFile, Description, PluralImageFile and SingularImageFile) and successively uses the values in each row to generate as many trials.

There are two additional columns not corresponding to variables in our template: `Group` and `Ending`. We commonly want a given participant to only see a subset of the various versions of our table, and the `Group`
column controls this: PennController automatically detects it and will generate trials only using the rows with one value in Group, in our case either the A or the B rows every other time the experiment runs (for more on how the group value is chosen on a given run of the experiment, see the full documentation). As a result, even though the table contains 8 rows besides the header row, when you test the experiment you only see 4 trials, either generated from rows 2, 5, 6 and 9 (highlighted in the table above) or generated from rows 3, 4, 7 and 8.

The Ending column specifies the property of our sentences that we care about, i.e., the manipulation of our independent variable, here whether or not the verb ends in the inflectional morpheme -s.

**Tracking trial details**

In order to keep track of all the details for a given trial, we need the lines of our results file to indicate things like what item they correspond to, what condition (presence vs absence of -s) they correspond to and what group (A vs B) they correspond to. In other words, we want to report the values of the table’s Item, Ending and Group columns to the lines in our results file. We simply use the same .log command we used before, referring to the table’s columns using the variable method:

```plaintext
1.  PennController.Template(
2.     variable => PennController(
3.         newAudio("description", variable.AudioFile)
4.            .play()
5.         ,
6.         newText(variable.Description)
7.            .unfold(2600)
8.         ,
9.         newImage("two", variable.PluralImageFile)
10.             .settings.size(200,200)
11.         ,
12.         newImage("one", variable.SingularImageFile)
13.             .settings.size(200,200)
14.         ,
15.         newCanvas(450,200)
16.             .settings.size(200,200)
17.         ,
18.         .print()
19.         ,
20.         newSelector()
21.             .settings.add( getImage("two") , getImage("one") )
22.             .settings.keys( "F" , "J" )
23.             .settings.log()
24.             .wait()
25.         ,
26.         getAudio("description")
27.         .wait("first")
28. )
29.   .log( "ID" , getVar("ID") )
30.   + .log( "Item" , variable.Item )
31.   + .log( "Ending" , variable.Ending )
32.   + .log( "Group" , variable.Group )
33)
```

Save and take the experiment, and you’ll see all the relevant information listed in the lines for each trial in the results file.

Next: **Timers & Randomization**
Adding Timers & Delays

When you tried out the experiment, you probably noticed that the trials came in rapid sequence without any pauses between them, with picture selection (or end of audio) marking the end of the trial, followed by an immediate transition to the next trial. It would probably be a good idea to add some time both at the beginning and at the end of a trial, to create a brief pause between trials. To do this, simply use `newTimer`:

```
Note: your script should also still contain the welcome trial above this
```

```r
PennController.TEMPLATE(
  variable = PennController(
    + newTimer(500)
    + start()
    + wait()
    +
    ,
    newAudio("description", variable.AudioFile)
    .play()
    ,
    newText(variable.Description)
    .unfold(2600)
    ,
    newImage("two", variable.PluarImageFile)
    .settings.size(200,200)
    ,
    newImage("one", variable.SingularImageFile)
    .settings.size(200,200)
    ,
    newCanvas(450,200)
    .settings.add( 0, 0, getImage("two") )
    .settings.add( 250, 0, getImage("one") )
    .print()
    ,
    newSelector()
    .settings.add( getImage("two") , getImage("one") )
    .settings.keys( "F" , "J" )
    .settings.log()
    .wait()
    ,
    getAudio("description")
    .wait("first")
    +
    ,
    newTimer(500)
    + start()
    + wait()
    )
  .log( "ID" , getVar("ID") )
)```
(Note that using a template makes things much easier when we want to update details of our trial structure for a whole sequence of trials!)

Following the general logic of PennController, creating a timer element in the cache does not on its own start the timer, in much the same way that creating a text or an image element does not print it. If you don't want to freeze your experiment, make sure that a timer has been started prior to a .wait command relating to it being executed.

**Randomization**

There are at least two things you might want to randomize in your experiment.

**The images' positions**

Having a static layout for your stimuli, with fixed positions for images of a certain type, is usually not a good idea, unless it was part of your design for some reason. 

In our case, it could be that participants are in general faster to choose any image that appears on the left (maybe English readers tend to process visual information from left to right, leading to reaction time advantage for images on the left) and since our current version always prints the two images on the left, this creates a potential confound that could wrongly lead us to conclude that people are faster to notice the absence of -s than the presence of -s.

To remedy this, we can randomly shuffle the positions of all the elements contained in a selector using the command .shuffle:

```javascript
PennController.Template(
  variable => PennController(
    newTimer(500)
      .start()
      .wait()
    ,
    newAudio("description", variable.AudioFile)
      .play()
    ,
    newText(variable.Description)
      .unfold(2600)
    ,
    newImage("two", variable.PluralImageFile)
      .settings.size(200,200)
    ,
    newImage("one", variable.SingularImageFile)
      .settings.size(200,200)
    ,
    newCanvas(450,200)
      .settings.add( 0, 0, getImage("two") )
      .settings.add( 250, 0, getImage("one") )
      .print()
    ,
    newSelector()
      .settings.add( getImage("two") , getImage("one") )
      .shuffle()
      .settings.keys( "F" , "J" )
  )
)
```
It is important here that `.shuffle` gets executed after the images are printed (through the canvas' `.print` command above) and added to the selector (otherwise, there is nothing to shuffle) but also before the command `.settings.keys` is executed: this way, shuffling happens before key assignment, and whichever image ends up to the left and right will be respectively associated with the F and J keys. If `.settings.keys` were executed before `.shuffle` then the association would take place before the shuffling. (Such a scenario would maybe make sense if you wanted to respectively associate the two and one images with the numeric keys 2 and 1 regardless of where each image ends up on the screen.)

The sequence of trials

Manipulating the order in which the trials are presented requires labeling them first. Note that labels are not unique IDs: two trials can share the same label, which we will make use of.

For all the trials you define with PennController(...), you can specify a label using a string as the first argument: PennController( "label" , ... ).

(Note that some steps in the trial sequence require a slightly different format. For example, the command PennController.SendResults has no pair of parentheses appended to PennController: instead, you can define a label within the parentheses of SendResults: PennController.SendResults( "label" ).

To gain explicit control over the order of the overall trial sequence, you can specify your own order using the command PennController.Sequence:
newButton("Start")
.print()
.wait()
,
newVar("ID")
.settings.global()
.set( getTextInput("ID") )
)
.log( "ID" , getVar("ID") )

PennController.Template(
variable => PennController("experiment",
newTimer(500)
.start()
.wait()
,
newAudio("description", variable.AudioFile)
.play()
,
newText(variable.Description)
.unfold(2600)
,
newImage("two", variable.PluralImageFile)
.settings.size(200,200)
,
newImage("one", variable.SingularImageFile)
.settings.size(200,200)
,
newCanvas(450,200)
.settings.add( 0 , 0 , getImage("two") )
.settings.add( 250 , 0 , getImage("one") )
.print()
,
newSelector()
.settings.add( getImage("two") , getImage("one") )
suffle()
.settings.keys( "F" , "J" )
.settings.log()
.wait()
,
getAudio("description")
.wait("first")
,
newTimer(500)
.start()
.wait()
)
.log( "ID" , getVar("ID") )
.log( "Item" , variable.Item )
.log( "Ending" , variable.Ending )
.log( "Group" , variable.Group )
)

PennController.SendResults("send")

PennController("final",
newText("<p>Thank you for your participation!</p>")
.print()
,
newText("<p><a href='https://www.put.your/platform/confirmation/link.here'>Click here to validate your participation.</a></p>")
.print()
,
newButton("void")
.wait()
)
Our command `PennController.Sequence( "welcome", randomize("experiment"), "send", "final" )` does not change the order much here, since it still corresponds to the order of trial elements in the script. However, the additional specification of `randomize("experiment")` makes sure that all the trials labeled `experiment` (all the table-generated trials, see line 32) will be run in a random order.

The list of commands like `randomize` that you can use within `Sequence` (which come from the basic Ibex setup, and are not special to PennController) are listed in the Ibex manual.

Next: **Data analysis in R**
Starting with a clean slate

You probably tested your experiment at multiple steps along the way, at which your were recording different pieces of information in the results file. Now that you have a final design, scroll down to Results to delete the files results and raw_results.

Then take your experiment at least twice, entering different IDs, to generate some results. Refresh the Results folder (click on the circular arrow) to see the new results files appear.

Save your results file in a folder that you will be able to access from an R script (click on the eye to open it in a new tab which you can save).

Load your results file in R

Add the function below to your R script so you can read your results file in R as a CSV file, and automatically name the columns using the comments lines:

```r
read.pcibex <- function(filepath, auto.colnames=TRUE, fun.col=function(col,cols){cols[cols==col]<-paste(col,"Ibex",sep=".");return(cols)}{  
  n.cols <- max(count.fields(filepath,sep=" "),na.rm=TRUE)
  if (auto.colnames){
    cols <- c()
    con <- file(filepath, "r")
    while ( TRUE ){
      line <- readLines(con, n = 1, warn=FALSE)
      if ( length(line) == 8 ) {
        break
      }
      m <- regmatches(line,regexexec("^\# (\d+)\. (.+)\.$",line))[[1]]
      if (length(m) == 3) {
        index <- as.numeric(m[2])
        value <- m[3]
        if (index < length(cols)){
          cols <- c()
        }
        if (is.function(fun.col)){
          cols <- fun.col(value,cols)
        }
        cols[index] <- value
      }
    }
  }
}
```
Then all you need to do to read your results file is:

```r
results <- read.pcibex("results")
```

(assuming your saved your results file in your R working directory under the name `results`).

You can then take a look at the first lines of your table, and their column names, using `head(results)`.

By default, when the R-function above finds more than one column with the same label in the comments of the results file, it suffixes the first column name with `.ibex`. Since the native IBEX format already inserts two columns in the results file described as `Item` and `Group`, and assuming you used `.log("Item", variable.Item)` and `.log("Group", variable.Group)` on your table-generated trials, this behavior ensures that the ibex-native columns (which appear first) are named `Item.Ibex` and `Group.Ibex` while the columns you added using the `.log` commands are indeed named `Item` and `Group`.

## Compare RTs

You can easily take a look at response times using the `dplyr` package:

```r
require(dplyr)
results %>%
  filter(Ending %in% c("No-s","-s") & (Parameter == "Selection" | Value == "Start")) %>%
  mutate(Accurate = rep(Value[Parameter=="Selection"], each=2)) %>%
  group_by(Accurate, Ending, Group, ID) %>%
  summarise(RT = mean(EventTime[Parameter=="Selection"] - EventTime[Value="Start"]), N = length(Value)/2)
```

Our filter only keeps rows which report `No-s` or `-s` (the welcome-trial rows) and corresponding to an image selection or to the start of a trial.

The function `mutate` adds a column `Accurate` indicating whether the `Value` for the `Selection` rows matches the condition identifier encoded in `Ending`.

We then group our rows by `Accurate` and `Ending` to get a clear idea of the effect of our manipulation (presence vs absence of `-s`) and also by Group and ID to get individual observations.

Finally we report the mean differences between the `EventTime` of the `Selection` minus the `Start` rows, along with the number of observations (two rows per trial, `Start` and `Selection`, call for a division by 2).