Running your ink

Quick Start

Note that although these instructions are written with Unity in mind, it's possible (and straightforward) to run your ink in a non-Unity C# environment.

- Download the latest version of the ink-unity-integration Unity package, and add to your project.
- Select your `.ink` file in Unity, and you should see a *Play* button in the file's inspector.
- Click it, and you should get an Editor window that lets you play (preview) your story.
- To integrate into your game, see **Getting started with the runtime API**, below.

Further information

Ink uses an intermediate `.json` format, which is compiled from the original `.ink` files. ink's Unity integration package automatically compiles ink files for you, but you can also compile them on the command line. See **Using inklecate on the command line** in the README for more information.

The main runtime code is included in the `ink-engine.dll`.

We recommend that you create a wrapper MonoBehaviour component for the **ink** `Story`. Here, we'll call the component "Script" - in the "film script" sense, rather than the "Unity script" sense!

```
using Ink.Runtime;

public class Script : MonoBehaviour {

    // Set this file to your compiled json asset
    public TextAsset inkAsset;

    // The ink story that we're wrapping
    Story _inkStory;
```

Getting started with the runtime API

As mentioned above, your `.ink` file(s) are compiled to a single `.json` file. This is treated by Unity as a TextAsset, that you can then load up in your game.

The API for loading and running your story is very straightforward. Construct a new `Story` object, passing in the JSON string from the TextAsset. For example, in Unity:

```
using Ink.Runtime;

...

void Awake()
{
    _inkStory = new Story(inkAsset.text);
}
```

From there, you make calls to the story in a loop. There are two repeating stages:

1. **Present content:** You repeatedly call `Continue()` on it, which returns individual lines of string content, until the

`canContinue` property becomes false. For example:

```
while (_inkStory.canContinue) {
    Debug.Log (_inkStory.Continue ());
}
```

```
A simpler way to achieve the above is through one call to `_inkStory.ContinueMaximally()`. However, in
```

2. **Make choice**: When there isn't any more content, you should check to see whether there any choices to present to the player. To do so, use something like:

```
    if( _inkStory.currentChoices.Count > 0 )
    {
        for (int i = 0; i < _inkStory.currentChoices.Count; ++i) {
            Choice choice = _inkStory.currentChoices [i];
            Debug.Log("Choice " + (i + 1) + ". " + choice.text);
        }
    }

    //...and when the player provides input:

        _inkStory.ChooseChoiceIndex (index);

    //And now you're ready to return to step 1, and present content again.
```

Saving and loading

To save the state of your story within your game, call:

`string savedJson = _inkStory.state.ToJson();`

...and then to load it again:

`_inkStory.state.LoadJson(savedJson);`

Error handling

If you made a mistake in your ink that the compiler can't catch, then the story will throw an exception. To avoid this and get standard Unity errors instead, you can use an error handler that you should assign when you create your story:

```
 _inkStory = new Story(inkAsset.text);

_inkStory.onError += (msg, type) => {
    if( type == Ink.ErrorType.Warning )
        Debug.LogWarning(msg);
    else
        Debug.LogError(msg);
};
```

Is that it?

That's it! You can achieve a lot with just those simple steps, but for more advanced usage, including deep integration with your game, read on.

For a sample Unity project in action with minimal UI, see Aaron Broder's Blot repo.

Engine usage and philosophy

In Unity, we recommend using your own component class to wrap `Ink.Runtime.Story`. The runtime **ink** engine has been designed to be reasonably general purpose and have a simple API. We also recommend wrapping rather than inheriting from `Story`, so that you can expose to your game only the functionality that you need.

Often when designing the flow for your game, the sequence of interactions between the player and the story may not precisely match the way the **ink** is evaluted. For example, with a classic choose-your-own-adventure type story, you may want to show multiple lines (paragraphs) of text and choices all at once. For a visual novel, you may want to display one line per screen.

Additionally, since the **ink** engine outputs lines of plain text, it can be effectively used for your own simple sub-formats. For example, for a dialog based game, you could write:

```
*   Lisa: Where did he go?
    Joe:  I think he jumped over the garden fence.
    * *   Lisa: Let's take a look.
    * *   Lisa: So he's gone for good?
```

As far as the **ink** engine is concerned, the `:` characters are just text. But as the lines of text and choices are produced by the game, you can do some simple text parsing of your own to turn the string `Joe: What's up?` into a game-specific dialog object that references the speaker and the text (or even audio).

This approach can be taken even further to text that flexibly indicates non-content directives. Again, these directives come out of the engine as text, but can parsed by your game for a specific purpose:

```
PROPLIST table, chair, apple, orange
```

The above approach is used in our current game for the writer to declare the props that they expect to be in the scene. These might be picked up in the game editor in order to automatically fill a scene with placeholder objects, or just to validate that the level designer has populated the scene correctly.

To mark up content more explicitly, you may want to use *tags* or *external functions* - see below. At inkle, we find that we use a mixture, but we actually find the above approach useful for a very large portion of our interaction with the game - it's very flexible.

Marking up your ink content with tags

Tags can be used to add metadata to your game content that isn't intended to appear to the player. Within ink, add a `#` character followed by any string content you want to pass over to the game. There are three main places where you can add these hash tags:

Line by line tags

One use case is for a graphic adventure that has different art for characters depending on their facial expression. So, you could do:

```
Passepartout: Really, Monsieur. # surly
```

On the game side, every time you get content with `_inkStory.Continue()`, you can get a list of tags with `_inkStory.currentTags`, which will return a `List<string>`, in the above case with just one element: `"surly"`.

To add more than one tag, simply delimit them with more `#` characters:

```
Passepartout: Really, Monsieur. # surly # really_monsieur.ogg
```

The above demonstrate another possible use case: providing full voice-over for your game, by marking up your ink with the audio filenames.

Tags for a line can be written above it, or on the end of the line:

```
 # the first tag
# the second tag
This is the line of content. # the third tag
```

All of the above tags will be included in the `currentTags` list.

Knot tags

Any tags that you include at the very top of a knot:

```
 === Munich ==
# location: Germany
# overview: munich.ogg
# require: Train ticket
First line of content in the knot.
```

...are accessible by calling `_inkStory.TagsForContentAtPath("your_knot")`, which is useful for getting metadata from a knot before you actually want the game to go there.

Note that these tags will also appear in the `currentTags` list for the first line of content in the knot.

Global tags

Any tags provided at the very top of the main ink file are accessible via the `Story`'s `globalTags` property, which also returns a `List<string>`. Any top level story metadata can be included there.

We suggest the following by convention, if you wish to share your ink stories publicly:

```
 # author: Joseph Humfrey
# title: My Wonderful Ink Story
```

Note that Inky will use the title tag in this format as the `<h1>` tag in a story exported for web.

Choice tags

Tags can also be applied to choices. Depending where the tag is placed inside the ink line, the tag will appear on both the choice and the content generated by that choice; on just the choice; or just on the output:

```
 * A choice! #a tag on both choice and content
* [A choice #a choice tag not on content ]
* A choice[!] which continues. #a tag on output content only
```

You can use all three in the same ink line:

```
 * A choice #shared_tag [ with detail #choice_tag ] and content # content_tag
```

The choice tags are stored inside the `List<string> tags` on the choice object.

Advanced: Tags are dynamic

Note that the content of a tag can contain any inline **ink**, such as shuffles, cycles, function calls and variable replacements.

```
{character}: Hello there! #{character}_greeting.jpg

I open the door. #suspense_music{RANDOM(1, 4)}.mp3
```

Jumping to a particular "scene"

Top level named sections in **ink** are called knots (see the writing tutorial). You can tell the runtime engine to jump to a particular named knot:

```
_inkStory.ChoosePathString("myKnotName");
```

And then call `Continue()` as usual.

To jump directly to a stitch within a knot, use a `.` as a separator:

```
_inkStory.ChoosePathString("myKnotName.theStitchWithin");
```

(Note that this path string is a *runtime* path rather than the path as used within the **ink** format. It's just been designed so that for the basics of knots and stitches, the format works out the same. Unfortunately however, you can't reference gather or choice labels this way.)

Setting/getting ink variables

The state of the variables in the **ink** engine is, appropriately enough, stored within the `variablesState` object within the `story`. You can both get and set variables directly on this object:

```
_inkStory.variablesState["player_health"] = 100

int health = (int) _inkStory.variablesState["player_health"]
```

Read/Visit counts

To find out the number of times that a knot or stitch has been visited by the ink engine, you can use this API:

```
_inkStory.state.VisitCountAtPathString("...");
```

The path string is in the form `"yourKnot"` for knots, and `"yourKnot.yourStitch"` for stitches.

Variable observers

You can register a delegate function to be called whenever a particular variable changes. This can be useful to reflect the state of certain **ink** variables directly in the UI. For example:

```
_inkStory.ObserveVariable ("health", (string varName, object newValue) => {
    SetHealthInUI((int)newValue);
});
```

The reason that the variable name is passed in is so that you can have a single observer function that observes multiple different variables.

Running functions

You can run ink functions directly from C# using `EvaluationFunction`.

You can pass the expected arguments for the ink function, if any.

If the ink function has a return value, it will be returned by EvaluationFunction. You do not need to Continue() over any text lines that may exist in the function; it runs to the end. Any content is written to the textOutput parameter, with a line break between each line.

```
var returnValue = _inkStory.EvaluationFunction("myFunctionName", out textOutput, params);
```

External functions

You can define game-side functions in C# that can be called directly from **ink**. To do so:

1. Declare an external function using something like this at the top of one of your **ink** files, in global scope:

   ```
   EXTERNAL playSound(soundName)
   ```

2. Bind your C# function. For example:

```
        _inkStory.BindExternalFunction ("playSound", (string name) => {
            _audioController.Play(name);
        });
```

There are convenience overloads for BindExternalFunction, for up to four parameters, for both generic `System.Func` and `System.Action`. There is also a general purpose `BindExternalFunctionGeneral` that takes an object array for more than 4 parameters.

3. You can then call that function within the **ink**:

   ```
   ~ playSound("whack")
   ```

The types you can use as parameters and return values are int, float, bool (automatically converted from **ink**'s internal ints) and string.

Alternatives to external functions

Remember that in addition to external functions, there are other good ways to communicate between your ink and your game:

- You can set up a variable observer if you just want the game to know when some state has changed. This is perfect for say, changing the score in the UI.
- You can use tags to add invisible metadata to a line in ink.
- In inkle's games such as Heaven's Vault, we use the text itself to write instructions to the game, and then have a game-specific text parser decide what to do with it. This is a very flexible approach, and allows us to have a different style of writing on each project. For example, we use the following syntax to ask the game to set up a particular camera shot:

```
>>> SHOT: view_over_bridge
```

Actions v.s. Pure functions

Warning: The following section is subtly complex! However, don't worry - you can probably ignore it and use default behaviour. If you find a situation where glue isn't working the way you expect and there's an external function in there somewhere, or if you're just plain curious, read on...

There are two kinds of external functions:

- **Actions** - for example, to play sounds, show images, etc. Generally, these may change game state in some way.
- **Pure functions** - those that don't cause side effects. Specifically 1) **it should be harmless to call them more than once**, and 2) **they shouldn't affect game state**. For example, a mathematical calculation, or pure inspection of game state.

By default, external functions are treated as Actions, since we think this is the primary use-case for most people. However, the distinction can be important for subtle reasons to do with the way that glue works.

When the engine looks at content, it may look ahead further than you would expect *just in case* there is glue in future content that would turn two separate lines into one.

However, external functions that are intended to be run as actions, you don't want them to be run prospectively, since the player is likely to notice, so for this kind we cancel any attempt to glue content together. If it was in the middle of prospectively looking ahead and it sees an action, it'll stop before running it.

Conversely, if all you're doing is a mathematical calculation for example, you don't want your glue to break. For example:

```
The square root of 9
~ temp x = sqrt(9)
<> is {x}.
```

You can define how you want your function to behave when you bind it, using the `lookaheadSafe` parameter:

```
public void BindExternalFunction(string funcName, Func<object> func, bool lookaheadSafe=false)
```

- **Actions** should have `lookaheadSafe = false`
- **Pure functions** should have `lookaheadSafe = true`

Fallbacks for external functions

When testing your story, either in Inky or in the ink-unity integration player window, you don't get an opportunity to bind a game function before running the story. To get around this, you can define a *fallback function* within ink, which is run if the `EXTERNAL` function can't be found. To do so, simply create an ink function with the same name and parameters. For example, for the above `multiply` example, create the ink function:

```
=== function multiply(x,y) ===
// Usually external functions can only return placeholder
// results, otherwise they'd be defined in ink!
~ return 1
```

Multiple parallel flows (BETA)

It is possible to have multiple parallel "flows" - allowing the following examples:

- A background conversation between two characters while the protagonist talks to someone else. The protagonist could then leave and interject in the background conversation.
- Non-blocking interactions: you could interact with an object with generates a bunch of choices, but "pause" them, and go and interact with something else. The original object's choices won't block you from interacting with something new, and you can resume them later.

The API is relatively simple:

- `story.SwitchFlow("Your flow name")` - create a new Flow context, or switch to an existing one. The name can be anything you like, though you may choose to use the same name as an entry knot that you would go on to choose with `story.ChoosePathString("knotName")`.
- `story.SwitchToDefaultFlow()` - before you start switching Flows there's an implicit default Flow. To return to it, call this method.
- `story.RemoveFlow("Your flow name")` - destroy a previously created Flow. If the Flow is already active, it returns to the default flow.
- `story.aliveFlowNames` - the names of currently alive flows. A flow is alive if it's previously been switched to and hasn't been destroyed. Does not include default flow.
- `story.currentFlowIsDefaultFlow` - true if the default flow is currently active. By definition, will also return true if not using flow functionality.
- `story.currentFlowName` — a string containing the name of the currently active flow. May contain internal identifier for default flow, so use `currentFlowIsDefault` to check first.)

Working with LISTs

Ink lists are a more complex type used in the ink engine, so interacting with them is a bit more involved than with ints, floats and strings.

Lists always need to know the origin of their items. For example, in ink you can do:

```
~ myList = (Orange, House)
```

...even though `Orange` may have come from a list called `fruit` and `House` may have come from a list called `places`. In ink these *origin* lists are automatically resolved for you when writing. However when work in game code, you have to be more explicit, and tell the engine which origin lists your items belong to.

To create a list with items from a single origin, and assign it to a variable in the game:

```
var newList = new Ink.Runtime.InkList("fruit", story);
newList.AddItem("Orange");
newList.AddItem("Apple");
story.variablesState["myList"] = newList;
```

If you're modifying a list, and you know that it has/had elements from a particular origin already:

```
var fruit = story.variablesState["fruit"] as Ink.Runtime.InkList;
fruit.AddItem("Apple");
```

Note that single list items in ink, such as:

```
VAR lunch = Apple
```

...are actually just lists with single items in them rather than a different type. So to create them on the game side, just use the techniques above to create a list with just one item.

You can also create lists from items if you explicitly know all the metadata for the items - i.e. the origin name as as well as the int value assigned to it. This is useful if you're building a list out of existing lists. Note that InkLists actually derive from `Dictionary`, where the key is an `InkListItem` (which in turn has `originName` and `itemName` strings), and the value is the int value:

```
var newList = new Ink.Runtime.InkList();
var fruit = story.variablesState["fruit"] as Ink.Runtime.InkList;
var places = story.variablesState["places"] as Ink.Runtime.InkList;
foreach(var item in fruit) {
    newList.Add(item.Key, item.Value);
}
foreach (var item in places) {
    newList.Add(item.Key, item.Value);
}
story.variablesState["myList"] = newList;
```

To test if your list contains a particular item:

```
fruit = story.variablesState["fruit"] as Ink.Runtime.InkList;
if( fruit.ContainsItemNamed("Apple") ) {
    // We're eating apple's tonight!
}
```

Lists also expose many of the operations you can do in ink:

```
list.minItem    // equivalent to calling LIST_MIN(list) in ink
list.maxItem    // equivalent to calling LIST_MAX(list) in ink
list.inverse    // equivalent to calling LIST_INVERT(list) in ink
list.all    // equivalent to calling LIST_ALL(list) in ink
list.Union(otherList)      // equivalent to (list + otherList) in ink
list.Intersect(otherList)  // equivalent to (list ^ otherList) in ink
list.Without(otherList)    // equivalent to (list - otherList) in ink
list.Contains(otherList)   // equivalent to (list ? otherList) in ink
```

Using the compiler

Precompiling your stories is more efficient than loading .ink at runtime. That said, it's a useful approach for some situations, and can be done with the following code:

```
// inkFileContents: linked TextAsset, or Resources.Load, or even StreamingAssets
var compiler = new Ink.Compiler(inkFileContents);
Ink.Runtime.Story story = compiler.Compile();
Debug.Log(story.Continue());
```

Note that if your story is broken up into several ink files using `INCLUDE`, that you will need to use:

```
var compiler = new Ink.Compiler(inkFileContents, new Compiler.Options
{
    countAllVisits = true,
    fileHandler = new UnityInkFileHandler(Path.GetDirectoryName(inkAbsoluteFilePath))
});
Ink.Runtime.Story story = compiler.Compile();
Debug.Log(story.Continue());
```