StoryStylus Scripting Help

Version 0.9.6

Monday, June 29, 2015

One More Story Games, Inc. ©2015

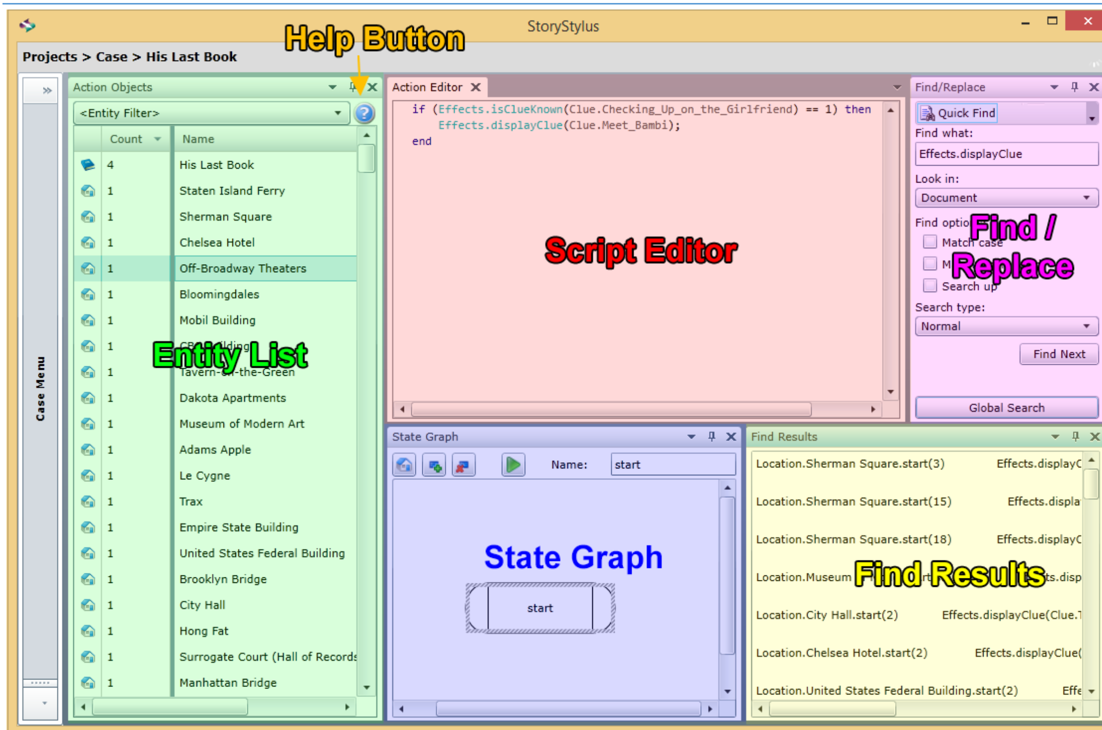
Contents

Versions 3
Scripting User Interface 4
Script Triggers 5
If-Then Scripting Language 6
Intellitype Script Help 8
Handing out Clues to the Sleuths 9
Moving and Timed Events 10
Take/Use Items 12
Combine Items 13
Triggering Different Story Introductions 14
Story Solutions Enabled 15
Story Chapters 16
Case Solutions 18
Advanced Scripting 21

Versions

Version#	Date	Notes
0.9	7 January 2015	• Initial Draft – Missing Sections on Story Chapters, Case Solutions & Advanced Scripting
0.9.5	10 January 2015	• Added section on Case Solutions. • Updated Scripting User Interface screenshot to include help button.
0.9.6	28 June 2015	• Updated section on Case Chapters.

Scripting User Interface



Here is a screenshot of the scripts panel from StoryStylus highlighting the five main panel regions:

- Entity List – Lists all of the entities in your story which can have script attached to them. Also, authors can sort by the number of scripts attached to an entity (see states below) making it easier to distinguish between entities that have scripts and those that do not.
- Script Editor – The text editor where entity scripts are loaded and where authors can edit and tweak what happens when sleuths interact with the entities in their stories.
- State Graph – Initially an entity has no scripts associated with it. Authors add states to entities, and each state has a script for that entity. Usually you only need one state per entity, but sometimes authors can achieve very complex scripting behaviour by organizing their entity scripts into finite state machines. This is covered in more detail in the advanced scripting section at the end of this document. For now, if an entity has no scripts, then you can click the add state button and then not worry about any of the advanced state functionality.
- Find/Replace – Sometimes authors will want to see every time a particular code effect is used, or perhaps tweak one parameter in every case it is used. Entering in the search text in the find text box and clicking “Global Search” will reveal every instance of that text in the game script code.
- Find Results – The results of a global search are listed in this panel. Clicking on a list entry will automatically load that script into the editor panel, and select that found section of text.

- Help Button – Provides an instant link to the most current PDF version of this file.

Script Triggers

The key concept of scripting in StoryStylus is that when a player does something, then if there is a script attached to that action, then the script can alter the different outcomes of that action.

Action	Entity	Script Type	Possible Outcomes
A sleuth visits a location looking for clues.	Location	Default	• A clue is discovered • Sleuth is sent to another location. • A person is revealed at that location. • Another location is revealed.
A sleuth attempts to take an item.	Item	Take	• Prevent the item from being taken. • A person is moved into the current location. • The item "breaks", is removed from the location and a new "broken" version of that item replaces it.
A sleuth uses an item.	Item	Use	• A door is unlocked and can now be used. • A clue is discovered. • A hover is revealed.
A sleuth combines an item with another item.	Item	Combine	• Items are removed from the sleuth inventory, and replaced with a new item.
A sleuth attempts to go through a door.	Door	Default	• Sleuth is prevented from moving through the door. • A clue is revealed.
A sleuth examines part of a room.	Hover	Default	• A clue is revealed. • A door is now usable. • An item is placed in the current location.
A sleuth visits a location in a given neighbourhood.	Neighbourhood	Default	• Reveals a new location on the map. • Sleuth is prevented from entering that neighbourhood.
A sleuth moves from one location to another.	Case	Move	• Keeps track of how many turns have elapsed since the start of the story. • Triggers a clue after a number of turns. • Prevents the sleuth from moving. • Triggers the next story chapter.
A sleuth starts a case with a special profession.	Case	Intro	• Issue a different story introduction.
A sleuth thinks they have enough clues to solve the case.	Case	Solve Conditional	• Prevent solve screen. • Allow solve screen.
A sleuth answers all questions to submit their solution.	Case	Solve	• Assign a case result based on sleuth's answers.

If-Then Scripting Language

The scripting language that is used is called LUA. The great advantage of this language is that it allows you to create complicated logic rules to handle a great number of different "what if" situations.

A conditional statement checks the "conditions" of the game story, and does one thing, or another based on that condition:

```
if ( check_something_is_true ) then
      do something
end
```

The format here is the requirement of the if-then followed by the end statement to show where the conditional statements ends. You need the "end" in case you want to do more than one thing as a result of "check_something_is_true":

```
if ( check_something_is_true ) then
      do one thing
      do a second thing
end
```

A more complicated version of the if-then logic is to something else if the "check something is true" doesn't work. So we get a case of do one thing OR do something else:

```
if ( check_something_is_true ) then
      do something
else
      do other thing
end
```

And you can check more than two options by using elseif:

```
if ( check_something_is_true ) then
      do something
elseif (check_other_thing_is_true) then
      do other thing
else
      do last thing
end
```

In this way we can have three (or more) options take place. If one thing is true then do option 1, or if another thing is true then do option 2, otherwise do option 3.

The condition of “check_something_is_true” doesn’t have to be one thing you are checking. You can combine multiple conditions by using the “and”, “or”, & not to come up with any sort of complex logic that you require.

Either check can be true:

```
(check_true) or (check_other_is_true)
```

Both checks must be true:

```
(check_true) and (check_other_is_true)
```

Intellitype Script Help

The main way that you interact with the game in a script is to use an Effect call. In the code library we include a number of useful functions in the Effects such as “is a given clue known to the sleuths?”, or “move a person to a specific location”. Don’t worry about trying to remember all of the different effects calls, we have a built-in help system to help you learn the language.

All effects calls start with “Effects.”, so the syntax of using an effect call is:

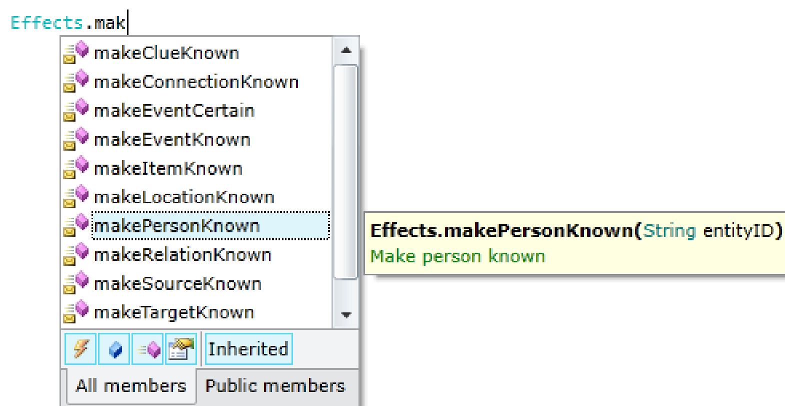
```
Effects.<particular_call>( parameters )
```

If you hover your cursor over an effect call, you can get a general description of how it works and what you can expect:

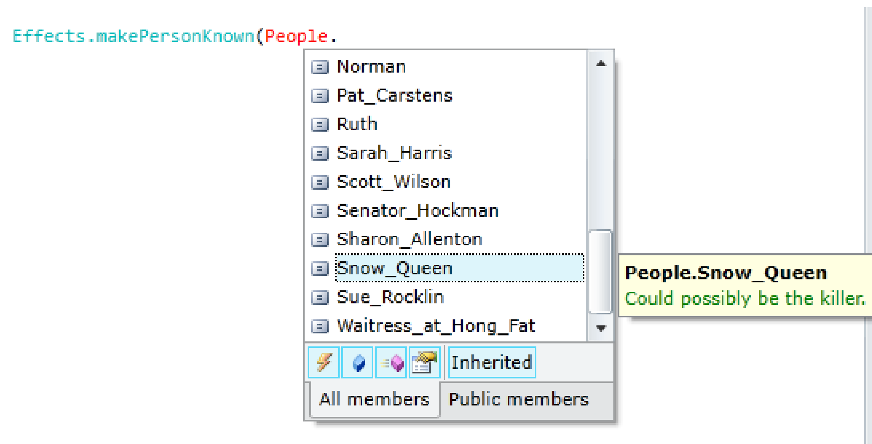
```
Effects.makeRelationKnown(Relation.dontolynn);
Effects.makeRelationKnown(Relation.dontopat);
```

Effects.makeRelationKnown(String entityID**)**
Make a relation known

If you type “Effects.” then you can get a complete list of every effects call you can use in your scripts:



And if you need help figuring out which parameters you need to use, then when you’re typing you can get a list of all of the different entities you’ve created in your case and their descriptions:



Handing out Clues to the Sleuths

Probably the most important effect used in StoryStylus is the “make clue known” call. When this effect is called, a clue is revealed to the players. This line will reveal the clue “Why_Indeed” to the players:

```
Effects.makeClueKnown(Clue.Why_Indeed);
```

A clue can only be revealed once. If this script was triggered a second time, then nothing would happen. A common example of a clue script is to attach it to a location so that when a sleuth goes to that location, they get a clue.

The second most useful effects call is the “is clue known” call. If a clue is known to the player, then it gives a value of 1, otherwise it gives a value of 0.

The “==” is the coding way to check if something is equal to a particular value, so (1 + 1) == 2.

To put it all together, you can have different clues get triggered by using the if-then-end structure:

```
if (Effects.isClueKnown(Clue.First_Clue) == 1) then
    Effects.makeClueKnown(Clue.Second_Clue);
else
    Effects.makeClueKnown(Clue.First_Clue);
end
```

In this way, if you already know the “First_Clue” then you learn the “Second_Clue”. If you don’t know the “First_Clue”, then you learn it. In this way, you can visit the same location twice and get two different results.

Another important code you will need to know later for Case Solutions is “~=”. This checks to see if one thing does not equal another thing: (1+1) ~= 3. So you can check to see if a clue is NOT KNOWN, and then show a particular clue:

```
if (Effects.isClueKnown(Clue.First_Clue) ~= 1) then
    Effects.makeClueKnown(Clue.First_Clue);
end
```

So if a clue isn’t known then show that clue. But this isn’t really useful because makeClueKnown will not show anything if the clue IS KNOWN, so in this case you may as well just make a clue known and don’t worry if the clue is already known because nothing will happen.

Moving and Timed Events

There three distinct events that happen when a sleuth moves to a location:

1st Move Event	Case	Move	• Keeps track of how many turns have elapsed since the start of the story. • Triggers a clue after a number of turns. • Prevents the sleuth from moving. • Triggers the next story chapter.
2nd Move Event	Neighbourhood	Default	• Reveals a new location on the map. • Sleuth is prevented from entering that neighbourhood.
3rd Move Event	Location	Default	• A clue is discovered • Sleuth is sent to another location. • A person is revealed at that location. • Another location is revealed.

When a sleuth double clicks on a location to move, the three events move events occur in order from the case move script, then the neighbourhood default script for the new location, and then finally the location default script for the new location. If any one of these scripts calls a “stop sleuth movement” call, then the sleuth will stop moving. For example, if the case move script (1st move event) stops the sleuth, then the other two move scripts will not trigger.

Let’s say that you want to perform timed events in your story. You want to keep track of how many times the player has moved, and on special turn numbers you want to stop the sleuth from moving, and reveal specific clues.

To be able to do this, you would create a flag for your story to keep track of the current turn number. You would create a script flag called currentTurn, set the initial value of the flag to 1, set the minimum value to 1, and the maximum value to 100 (or some other large number).

Then you would add the following script to case move:

```
Effects.incrementFlag(Flag.currentTurn, 1);


if(Effects.getFlag(Flag.currentTurn) == 4) then
      Effects.displayClue(Clue.Fourth_Turn_Clue);
      Effects.stopSleuthMovement();
end


if(Effects.getFlag(Flag.currentTurn) == 9) then
      Effects.displayClue(Clue.Nineth_Turn_Clue);
      Effects.stopSleuthMovement();
end
```

Here's how this script code works. Every time that the player moves, the currentTurn flag is incremented by 1. When the currentTurn flag is equal to 4 or 9, then a particular clue is displayed and the sleuth's movement is halted.

You could even have the multiple flags used to keep track of the time passing after several events. You would just put conditionals (using another script flag) to see if the particular events had occurred, increment the different timer flags, and then perform the conditional check like before:

```
if(Effects.getFlag(Flag.triggerEvent_A) == 1) then
      Effects.incrementFlag(Flag.turns_Since_Event_A, 1);
end


if(Effects.getFlag(Flag.turns_Since_Event_A) == 10) then
      Effects.displayClue(Clue.post_Event_A_Clue);
      Effects.stopSleuthMovement();
end
```

This script checks to see if an "EventA" has been triggered, and then 10 turns after that the "post_Event_A_Clue" is displayed to the player. A good use for a script like this could be talking to a person, and then later on the person calls the player back to tell them something that they forgot from their initial conversation.

Take/Use Items

A take item state machine is triggered when a sleuth tries to take an object. Usually a sleuth will have no problem with taking an item and putting it into their inventory. If a stopTakeItem effect is called in a take script, then the sleuth will fail to take the item and be unable to put it in their inventory.

The best way to use the take script is to check if a set of circumstances exists then call stopTakeItem to prevent the item from being taken. Then if these circumstances change, then allow the item to be taken. In the example below, this is a script that is attached to a Police Cap object. If the cap item and the person "Officer Brady" are in the same location (Café Perk), then the police officer character stops the player from being able to take the Police Cap.

```
if(Effects.isItemInLocation(Item.PoliceCap, Location.Cafe_Perk) == 1 and
Effects.isPersonKnown(Person.Officer_Brady) == 1 and
Effects.isPersonInLocation(Person.Officer_Brady, Location.Cafe_Perk) == 1)
then

      Effects.displayClue(Clue.Brady_Stops_You);

      Effects.stopTakeItem();

end
```

Here is even an example of a similar script where the players have not yet met the Officer Brady character. In this situation the players will not see the Brady character, but will only see the Police Cap item. But they will still be unable to put the cap into their inventory.

```
if(Effects.isItemInLocation (Item.Doughnut, Location.Cafe_Perk) == 1 and
Effects.isPersonKnown(Person.Officer_Brady) == 0 and
Effects.isPersonInLocation(Person.Officer_Brady, Location.Cafe_Perk) == 1)
then

      Effects.displayClue(Clue.Meet_Officer_Brady);

      Effects.stopTakeItem();

end
```

To resolve this situation, perhaps the players have to use a "doughnut" (or other suitable bribe item) to get Officer Brady to move from the Café Perk location. So this is the script we would place in the use item for the doughnut object:

```
if(Effects.isPersonInLocation(Person.Officer_Brady, Location.Cafe_Perk) ==
1 and Effects.isThisTheRightLocation(Location.Cafe_Perk))  then

      Effects.displayClue(Clue.Officer_Brady_Leaves);

      Effects.movePerson(Person.Officer_Brady, Location.Doughnut_Shop);

end
```

If both Officer Brady and the player are at the café, then we move Brady to the Doughnut Shop location, and give a clue that says that Brady has left the café. Now the player will be able to get the Police Cap item and have it in their inventory.

Combine Items

A combine item script allows a player to combine two or more objects together to create a new item in their inventory. For example, you could have multiple scraps of a torn will that need to be combined together to create an assembled will document.

The combine script needs to only be set on one of the combine items. The program will check all items that have been placed in the combine screen, and if any of them have a combine script, then that script will be triggered.

Here is an example of a combine script for combining two parts of a dagger (a blade and a hilt) to make a complete weapon.

```
if(Effects.isItemInCombineArray(Item.Dagger_Blade) == 1 and
Effects.isItemInCombineArray(Item.Dagger_Hilt) == 1 and
Effects.getItemCombineArrayLength() == 2)  then
	Effects.removeFromInventory(Item.Dagger_Blade);
	Effects.removeFromInventory(Item.Dagger_Hilt);
	Effects.putItemInInventory(Item.Dagger_Full);
	Effects.displayClue(Clue.Dagger_Assembled);
end
```

There are two parts of this script: the conditional check, and then the code that does the actual combine.

In the combine script, you remove the combine items from the sleuth inventory and then put back into the sleuth inventory a new item that represents the combined whole. In the example above, the blade and hilt objects are removed from the inventory and replaced with the full dagger. Also in the example above, there is a clue that gives the sleuth some information about the successful combination of the dagger.

In the conditional check, there are multiple logical checks that need to be true in order for the item combine to actually occur. So the combine is composed of multiple checks that are combined into a collection of “and” statements (i.e. if ALL conditions are true):

```
	if(A and B and C) then
		<combine code>
	end
```

All but the last conditional check is to see if the player has selected the correct objects to be combined. Combining the dagger blade with a baseball bat will not have the same result as the blade with the hilt. So in the code above, you check that the blade and the hilt are in the combine array (the combine interface) when the player clicks on the COMBINE button. The last check counts the number of items that have been combined (i.e. can only be 2). This prevents players from putting their entire inventory in the combine window and hoping that some correct combine “recipe” combination is present.

Triggering Different Story Introductions

Here is an example of a case introduction script:

```
if (Effects.isSleuthOfProfession( Profession.Lawyer ) == 1) then
      Effects.setIntroduction(Introduction.Lawyer_Introduction);
end
```

In this script, we check to see if one of the starting sleuths is of profession type Lawyer, and if so, then the starting introduction becomes Lawyer_Introduction. If one of the sleuths is not a lawyer, then the introduction that will be displayed is whatever was set as the default in the case information panel.

With multiple checks an author could have several different introductions for a case. One story could have a completely different starting set of clues depending on the type of sleuth character being played.

```
if (Effects.isSleuthOfProfession( Profession.Lawyer ) == 1) then
      Effects.setIntroduction(Introduction.Lawyer_Introduction);
      return;
end
if (Effects.isSleuthOfProfession( Profession.Doctor ) == 1) then
      Effects.setIntroduction(Introduction.Doctor_Introduction);
      return;
end
```

Here there are two different introductions, one for lawyers, and another for doctor sleuths. We put a return line after setting the introduction so that no further script code is called. If there were two sleuths playing together, and one was a doctor, and the other was a lawyer character, then the lawyer introduction would be given and NOT the doctor introduction. Once the check is done for the lawyer and it matches, then the introduction for the story is set as the lawyer_introduction, and then the case introduction code stops. The second check for the doctor sleuth is never performed.

One other noteworthy point is that case introductions can only be set in the introduction script for that case. If you try to make this call in any other script, then it will not work. If no other introduction is set for the case, then the default introduction will be set and then this will remain for the rest of the story.

Story Solutions Enabled

Most mystery stories will end with the players entering into the solve screen to answer several skill-testing questions. If they have figured out the mystery, then they will correctly answer all of the questions correctly. But perhaps you don't want to allow the players to enter into the solve screen until they have reached a later stage of the story: i.e. you don't want the players to close the introduction and then randomly entering in answers to try to "skip to the end" of your mystery game. In this case, you can check to see if enough turns have passed, or if an important clue has been discovered, and then the player can go to the solve screen to try to correctly answer the mystery questions.

The type of script is a solve conditional script for a case. This is triggered when a player attempts to go into the solve screen by clicking on the solve button off of the sleuth menu. If the effect "quizConditionsHaveBeenMet" gets called, then the quiz will trigger as normal.

Here is an example of a solve conditional script:

```
if (Effects.isClueKnown(Clue.You_Know_Enough_To_Solve) == 1) then
      Effects.quizConditionsHaveBeenMet();
end
```

In the above example, if the clue "You Know Enough to Solve" is known to the sleuths, then the quiz will trigger when the player clicks on the Solve button; otherwise, the player has not yet met the conditions to solve the case.

If the author set up a case move script that was tracking how many turns have elapsed (see move example above), then the quiz can be triggered after enough turns have elapsed.

```
if(Effects.getFlag(Flag.currentTurn) > 50) then
      Effects.quizConditionsHaveBeenMet();
end
```

In this example, the player sleuths can not solve the mystery until more than 50 turns have elapsed.

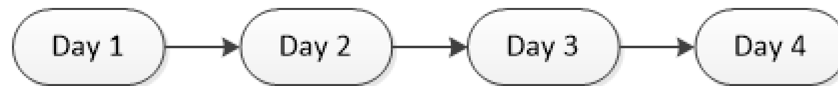
Remember that the only way that this solve conditional script will work is if you have the following line hooked up in the case move script, and have created a scripting flag called currentTurn:

```
Effects.incrementFlag(Flag.currentTurn, 1);
```

Otherwise the solve conditional script will not know when to trigger the conditions for the solve quiz.

Story Chapters

Longer stories will tend to be organized into chapters, which can also help to organize the flow of a branching narrative. When you create a series of chapters you must first specify which of these will be the starting chapter, only one chapter can be checked as the starting chapter. Then as you create new chapters you can specify the default chapter that will follow. Here is an example of a linear flow of chapters:



Chapter Name	Next Chapter	Start Chapter	End Chapter
Day 1	Day 2	Y	N
Day 2	Day 3	N	N
Day 3	Day 4	N	N
Day 4	<NONE>	N	Y

You can always check to see the current chapter of the story, and then do different things using the isCurrentChapter effect:

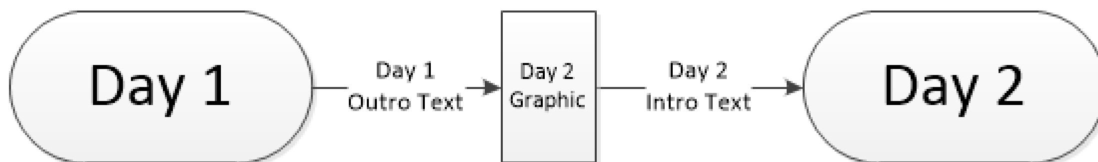
```
if(Effects.isCurrentChapter(Chapter.Day_1) == 1) then
    -- Do chapter 1 stuff.
end
```

If you need to trigger a transition from one chapter to another then you call chapterGotoNext:

```
Effects.chapterGotoNext();
```

When you flow from one chapter to another, there are three events that occur:

1) the outro text of the previous chapter is displayed
2) the new chapter graphic is displayed
3) the intro text of the new chapter is displayed

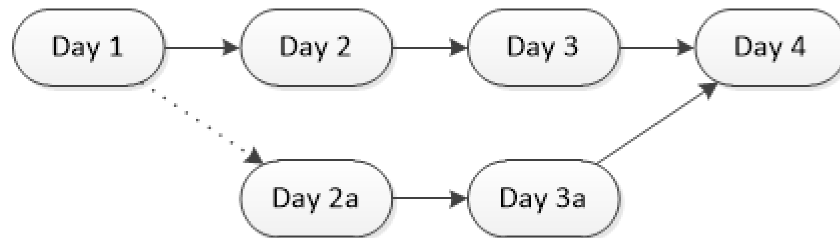


The resulting chapter from the chapterGotoNext call will always be the default next chapter. If you want to have a different resulting chapter then you call chapterSetNext:

```
Effects.chapterSetNext(Chapter.Day2_Other);
```

Then when another chapterGotoNext is called, the player will see the Day1 outro text, but then will instead see the graphic and intro text for the OTHER Day 2.

Let's say you wanted to have a branching chapter narrative in that during day 1, a player could cause an event that would lead to a different day 2 and day 3. In this you would create two extra chapters for this new branching flow.



Chapter Name	Next Chapter	Start Chapter	End Chapter
Day 1	Day 2	Y	N
Day 2	Day 3	N	N
Day 2a	Day 3a	N	N
Day 3	Day 4	N	N
Day 3a	Day 4	N	N
Day 4	<NONE>	N	Y

So on Day 1, you would call a chapterSetNext to Day2a if this special event took place. Then all you need to do is trigger chapterGotoNext (say whenever a player clicks on their bed) and the story will branch as the diagram above.

There is another chapter call that you can call make if you want to set a different next chapter and perform a transition all in one call:

```
Effects.chapterSetNextAndGoto(Chapter.Day_2a);
```

One special thing to note is that as soon as you trigger a chapter transition with either a chapterSetNext or a chapterSetNextAndGoto, the only time when a new chapterSetNext can be called is after the new chapter has been introduced. Any other calls to set a new chapter will not work during this period. So make sure you've done the chapterSetNext before you call chapterGotoNext.

Case Solutions

Case solution scripts are triggered when a player thinks they have solved the mystery. At that point the players clicks the solve button in the sleuth menu. This starts a quiz that determines if the player has grasped the subtle nuances of the solution to the story. Once the player has answered all of the question and confirmed their solution, this starts the case solution script. The job of this script is to assess the player's answers, and then assign a case solution based on those answers.

There are three key effects calls that are used in the case solution script:

- getQuestionAnswer – if the question was a "pick an option" multiple choice (only one answer can be selected), then this function will return the number of the option that the player selected: 1 for the 1st option, 2 for the 2nd option, and so on. So this example will return the option that was checked for the question named Question 3.

```
Effects.getQuestionAnswer(Question.Question_3);
```

- getCheckboxAnswer – if the question was a "pick multiple options" multiple choice (more than one answer can be checked), then this function will return a 1 if a particular option was selected, and 0 if the answer was not checked. This effect requires two parameters: the first specifies which question you are checking, and the second specifies which answer number (1 to the number of options) you want to know if it was checked. For example, this code will give you a 1 if the second option on Question 4 was checked (and 0 if it wasn't checked in the player's answer).

```
Effects.getCheckboxAnswer(Question.Question_4, 2);
```

  This way you can check if the player checked the second and fourth options, but didn't check any of the other options in the question.

- setCaseResult – once you've assessed all of the questions, then use this function to set which solution the player has earned based on their answers.

```
Effects.setCaseResult(Solution.So_Close_But_Still_Wrong);
```

Sometimes a case result can be given based on particular scriptFlag value when the case solution was given: e.g. a silver medal ending if the player took too long to figure out the solution, and a gold medal ending if the current turn script flag is under 40 (i.e. the player took less than 40 moves to reach a solution). You would simply do a check for the value of one of the script flags in the game, and then set the case result (solution) as in the example above.

```
if(Effects.getFlag(Flag.currentTurn) < 40) then
	Effects.setCaseResult(Solution.Just_in_Time);
else
	Effects.setCaseResult(Solution.Too_Late);
end
```

So how do you use these functions in a Case Solution script? Let's say you have a case quiz with five questions. Four of the questions are pick an option (only one correct answer), and one is multiple choice that has two correct answers. There are three endings to the story: one if you get everything right, one if you make a mistake, and then one special hidden ending that occurs if you get all of the pick an option questions correct, get the multiple choice question wrong, but select the fourth option.

Questions	Solutions
Question1 – pick an option – Answer 3. Question2 – pick an option – Answer 1. Question3 – pick an option – Answer 4. Question4 – pick an option – Answer 2. Question5 – multiple choice – Answers 1 & 3 with 4 being a special answer that unlocks the hidden ending.	All_Correct – you got every answer right. Wrong – you got something wrong. HiddenEnding – you got every pick an option answer correct, didn't get the correct answer to the multiple choice question, and you picked option 4 in answering the multiple choice question.

So here's the how we set-up a solution script:

- Handle all answers for the pick an option questions that don't result in correct answers.
    - Very simple conditional not-equal checks.
- Check for the patterns of multiple choice answers that provide a correct answer.
    - Look for the correct pattern of multiple choice options that are both checked and unchecked.
- See if the hidden option has been picked.
- If we're still going, then only the wrong answer remains.

The other thing you need to know is that "~=" checks to see if something does not equal something else. (remember that example from before).

So here is the script for the example above:

```
if(Effects.getQuestionAnswer(Question.Question_1) ~= 3) then
	Effects.setCaseResult(Solution.Wrong);
	return;
end

if(Effects.getQuestionAnswer(Question.Question_2) ~= 1) then
	Effects.setCaseResult(Solution.Wrong);
	return;
end
```

```
if(Effects.getQuestionAnswer(Question.Question_3) ~= 4) then
      Effects.setCaseResult(Solution.Wrong);
      return;
end

if(Effects.getQuestionAnswer(Question.Question_4) ~= 2) then
      Effects.setCaseResult(Solution.Wrong);
      return;
end

if((Effects.getCheckboxAnswer(Question.Question_5, 1) == 1) and
      (Effects.getQuestionAnswer(Question.Question_5, 2) == 0) and
      (Effects.getQuestionAnswer(Question.Question_5, 3) == 1) and
      (Effects.getQuestionAnswer(Question.Question_5, 4) == 0) and
      (Effects.getQuestionAnswer(Question.Question_5, 5) == 0)) then
      Effects.setCaseResult(Solution.All_Correct);
      return;
end

if(Effects.getCheckboxAnswer(Question.Question_5, 4) == 1) then
      Effects.setCaseResult(Solution.HiddenEnding);
      return;
end
Effects.setCaseResult(Solution.Wrong);
```

The first four conditionals check to see if you didn't get the pick and options answers right (not equal to), and if so then you get the Wrong solution and stop checking anything else with the "return" statement.

The big conditional is checking the status of all of the possible multiple choices for question 5. If that question has been answered correctly (1 and 3 checked but none of the others checked) then you get the All Correct solution.

The final conditional checks to see that even though you got question 5 wrong, did you check option 4 in that question. If so then you get the Hidden Ending solution and stop checking anything else with the "return" statement.

Finally, if you're still checking then you got question 5 wrong but didn't check answer 4 so you just get the Wrong solution.

Advanced Scripting

(Coming Soon)