Guide to making a Text game

Version 1.0 - Text as Actor

The first version is designed to be very similar to your standard Greenfoot project. This means you will be able to integrate it into an existing game, using the Scenario Editor to create and edit the text.

(A) Project Setup

1. Create a new Java Scenario
2. Subclass the “World”, name “TextWorld”
3. Subclass the “Actor”, name “Text”

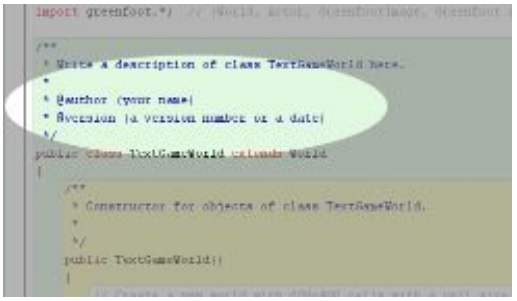
Optionally:

2. Select World image with preferred background color, as it will become the default backdrop.
3. Select an image representing a text-area.

2-3. Set your name and version/date:

A. Double-click each new Subclass

B. Replace the ()s inside at top of the green block



(B) Code Text Actors

1. Double-click the Text-subclass
2. Declare an instance-variable on Text, name “textContent” type “String”
3. Add a constructor “public Text” with “String content” as parameter

4. Declare an instance-variable on Text, type “GreenfootImage”, name “image”
5. Initialize it in constructor, making a new instance with two integer parameters, the desired width and height of your text box.
6. Next in constructor, create another image temporarily: `GreenfootImage txtImg = new GreenfootImage( this.textContent, 20, Color.WHITE, Color.BLACK );` where 20 is font size in pixels, and the two Colors are text-color and text background-color.
7. Then blend the above image on top of the first one: `image.drawImage(txtImg, 0, 0);` where 0’s represent left and top margin.
8. Finally make the blended image appear on screen: `setImage(image);`

(C) Setup first text in Scenario Editor

1. You can now right-click on “Text” in the Scenario Editor class tree.
2. Select “new Text(String …”
3. Place it in middle game area
4. Type your text in popup box (using “\n” as line breaks), exempel “Felicia kattunge”
5. Right-click outside the instance-representation
6. Select “Save the World”

Now run the Game! Texts can be added and deleted as normal Actors.

Version 1.0 solution can be checked out in solution file “MyTextGame1-0.gfar”

Version 1.1 - Animated text!

This version will add a real time “type-writer”-effect to your text!
You will learn how to update a text object, using a standard technique for custom real-time changes in a game

(A) Separate world-updates from screen-refresh

To make game change, bit by bit, Greenfoot automatically calls every World and Actor-instance with an `act()`-method around 25 times a second. A lot of things must take place this 1/25th of a second. Therefore, it's a good idea to comb out code early. Most common is putting logics and calculations inside **update()** and issuing the visible changes on screen inside **refresh()**. (aka paint(), repaint(), draw(), render()...)

1. In Text-class, add methods update() and refresh() using the following signatures:

```
/**
 * Update logics. Start with buffering text...
 */
private void update() {

}
/**
 * Update graphics. Start with the image of the text field...
 */
private void refresh() {

}
```

2. Call both methods from act(). If it doesn't exist in yet, just add

```
public void act() {

}
```

3. Cut-paste the code entered in section v1.0-B step 6-8 into refresh() method body. (the {} part)
4. Hit "Compile" button or `Ctrl-S` / `cmd-S`

If something goes wrong when compiling, it is probably a minor mistake like spelling, misplaced ()- or {}-blocks. Make sure the Scenario Editor still runs the game and shows any Text-instances you create for it. If no text is visible, try creating a "new Text()" in Scenario Editor and remember to "Save the World". (see section 1.0-C)

(B) Introduce a text-buffer, where chars add up one at a time

1. Declare and initialize an instance-variable: `String buffer = "";`
2. Declare and initialize an instance-variable: `int readerIndex = 0;`
3. In update-method, add `buffer += textContent.charAt(readerIndex);`

4. Then increase the index `readerIndex++;`
5. Wrap 3-4 inside an number-comparing if-statement, so it will stop copying chars at the right time.
6. In refresh-method, replace `this.textContent` with `this.buffer`
7. Hit “Compile” button or `Ctrl-S` / `cmd-S`

Version 1.2 - Handle multiple lines

(A) Approaching the problem

1. Edit a Text-instance so that the text spans to several lines, using "\n" as separator inline a string. (using code or scenario editor)
2. Save, run and observe the behavior.
3. Compare at least two possible solutions
    a. Create extra Lines as subclass to Text?
    b. Create extra Lines programmatically inside Text?
4. Settle for 3.b

(B) Detect line-end automatically

1. In Text class, add another readerIndex for line-number, by refactoring into ***readerCharIndex*** and ***readerLineIndex***.
2. Add another instance-variable, `boolean newLineNeeded = false;`
3. Set it to true inside refresh(), after buffer-image is created. It will be used as a flag.
4. Also add `System.out.println("new line needed");`
5. Wrap 3,4 in an if-statement.
6. Make a condition that is truthy when txtImg grows wider than image. Lookup in the API by typing “`image.`” then `Ctrl-space` / `cmd-space` and selecting suitable method call.
7. Enter Scenario Editor and create a long long text instance without any “\n”. Save the World, then run. The line-end detection works if a debugger window pops up with the message. When it works, remove the log command.

(C) Declare initial values for text dimensions

1. Add following instance variables at beginning of class Text:
    a. `int fontHeight ; //pixels`
    b. `int lineSpacing ; //pixels`
    c. `int containerWidth ; //pixels`
    d. `int containerHeight ; //pixels`
2. Locate and their corresponding values. For each:
    a. Assign each value to their corresponding variable.
    b. Replace the hard-coded value with variable.
    c. Make approximation of lineSpacing to 25% of fontHeight.
3. Add a derived value right after the new variables assignments:

```
int lineTotal = Math.floorDiv(containerHeight, fontHeight+lineSpacing);
```

4. Test it with `System.out.println(lineTotal);` inside constructor.

(D) Add extra buffers

1. Replace `String buffer = "";` with `String[] buffer = new String[lineTotal];`
2. Set `buffer[0] = "";` anywhere in constructor.
3. Replace other occurances of `buffer` /`this.buffer` with `buffer[0]`
4. Test run for errors.

(E) Add increment and condition on new lines

1. Anywhere in update(), create an if-statement, using only newLineNeeded as condition.
2. Move on to the conditional code:
    a. Reset the newLineNeeded flag to false.
    b. Increase readerLineIndex by 1.
    c. Then initialize buffer for the new index with empty string.

(F) Add extra text-images

1. In beginning of refresh(), declare an array of GreenfootImage.
2. Initialize it with length of readerLineIndex + 1. (length start with 1)
3. Loop over the array:
    a. Set each temporary image, using similar code as before.

b. Draw each temporary image on “image”, using similar code as before.
c. Remove the old image creation and drawing.
d. Make sure the correct iteration index is used for identifying correct temporary image as well as buffer.

(G) Offset text-images vertically

1. Inside the loop, declare a variable, vertical offset, using lineSpacing and fontHeight, multiplied by the iterator index.
2. Replace third parameter in “drawImage” with vertical offset. This will push the image down to its line.

(H) Improve line-breaking

Currently, the line breaking can cause words cropped and characters skipped. Solving this is a little tricky!

1. Look at the two if-statements in update() and give them names in comments. The first one is responsibe for buffering another character. The second one is for buffering another line.
2. Inside the first, after adding char to buffer, but before increasing index, add a comment “Check if line-break is needed”. To do this we would like to add another conditional, that is looking ahead to next word and checking if adding it will produce a line that overflows. If so, set the newLineNeeded flag. The old behavior inside refresh() should be removed.
3. Before writing any condition, add the commands that are to be made when next word is found: (similar to the old code in refresh)

```
GreenfootImage txtImgTest = new GreenfootImage(buffer[readerLineIndex]+nextWord,
fontHeight, Color.WHITE, Color.BLACK);

if( txtImgTest.getWidth() > containerWidth ) {
    newLineNeeded = true;
}
```

4. What's new is "nextWord"! To find next word, use an algorithm that looks ahead: Every space, look for next space. Then save what's between as "nextWord"!
5. Suggested solution:
    a. At beginning of update() add:
    `int charsLeft = textContent.length() - 1 - readerCharIndex;`
    b. Change condition for "buffer another character" to:
    `( charsLeft  > 0 )`
    c. Add condition for "Check if line break is needed" to:
    `( charsLeft > 1 && textContent.charAt(readerCharIndex) != ' ' )`
    d. Then add local block variables:
    ```
    int lookAhead = readerCharIndex + 1;
    boolean nextChar = textContent.charAt(lookAhead);
    ```
    e. Increase lookAhead until whitespace is found:
    ```
    while (nextChar != ' ') {
        nextChar = textContent.charAt(lookAhead++);
    }
    ```
    f. Extract substring from one char ahead of first space till next:
    `String nextWord = textContent.substring(readerCharIndex+1,lookAhead);`
    g. Loop in step (e.) is at risk looking ahead too far, do either:
        i. Always end textContent with " ", add in constructor.
        ii. Add to while-condition: `&& lookAhead < textContent.length()`
6. Optionally refactor tasks inside update().

Version 1.3 - Handle multiple pages

(A) Approaching the problem

1. Make a list of needs as comment inside class TextGameWorld. Suggested requirements:
    a. Each page must support different "themes".
    b. Each page must be able to switch to next on player input.
    c. Each page of text or texts should be stored somewhere.
2. Consider how each feature would be implemented for **one page**:
    a. In TextGameWorld
    b. In act(), available in either TextGameWorld or Text
    c. In TextGameWorld via Scenario Editor

3. Let's stick to TextGameWorld, but there is a problem with creating more than one instance of TextGameWorld. Nothing happens. It's called "Singleton" when only one instance is allowed at a time. The solution in Greenfoot is to create a new Subclass.

(B) Subclass TextGameWorld

1. Create subclasses of TextGameWorld and call them "Page1" and "Page2". Select different background images.
2. Try creating "new Page1()" and "new Page2" in Scenario Editor. It will switch between them like swapping workspaces, but remember to "Save to World" any changes made between swaps!
3. TextGameWorld can handle common code for making the switch happen.
    a. Let's start with a simple next page reference:
    `TextGameWorld nextPage;`
    It can contain any instance of "Page"-class, because they inherit similar class properties.
    b. Next, make a method that executes the change of page:

```
/**
* Instantly switches the game to the world set as next
 */
public void changePage() {
    if(nextPage != null) {
        Greenfoot.setWorld(nextPage);
    }
}
```

    c. Make a public method, a "setter", that actually can initiate nextPage:

```
/**
 * Override when subclass should link to another
 */
public void setNext(){

}
```

    d. This method should be called from constructur after the auto-generated "prepare();", otherwise the Scenario Editor will not work.

(B) Override nextPage

1. Open Code editor for Page1 and Page2
2. On both, add to constructor `super();`
3. On Page1, override the setNext method:

```
@Override
public void setNext(){
    super.nextPage = new Page2();
}
```

4. Test it by opening Scenario Editor, swap to Page1, then right-click on the game-area and select “Inherited from TextGameWorld” -> “void changePage()”
   It should change page temporarily, until game is “reset”.

(C) Add interactivity

1. Add `public act() {}` to TextGameWorld, if not already present.
2. In it, add logic for keyboard input, here using space-key:

```
if(Greenfoot.isKeyDown("space")) {
    wasSpaceDown = true;
} else if (wasSpaceDown) {
    wasSpaceDown = false;
    changePage();
}
```

3. At start of class TextGameWorld, add `boolean wasSpaceDown = false;`

   In Greenfoot, the keyboard API is focused on holding keys down. This code logic is mimicing what happens when a mouse-button is “clicked”. To learn other key-identifiers or input methods, see Greenfoot API docs.

(D) Customizing Text theme on different Pages

Now that a “Page” can vary in theme, the “Text” might need some adjustment as well.

1. In class Text, add at top `Color fontColor = Color.WHITE; //default color`
2. Copy-paste the whole constructor next to it. (before or after)
3. In one constructor, add second parameter `Color color`

4. Inside same constructor, set `fontColor = color;`
5. In refresh(), replace `Color.WHITE` with `fontColor`.
6. For a transparent background, replace `Color.BLACK` with `new Color(0,0,0,0)`. In order to not see through to previous updates, also:
7. Before all drawImage in refresh(), add `image.clear();`
8. When creating new Text in Scenario Editor, optionally create the object with the color parameter. It must be formatted by example:
    a. `greenfoot.Color.CYAN`, or
    b. `new greenfoot.Color(0,255,255,255)` (=red, green, blue, opacity)