SimpleGame (Java Edition)

Developed by: Blake Troutman
Based on SimpleGame.js, developed by Andy Harris

Introduction

SimpleGame (Java Edition) is a system of classes used abstract the process of writing 2D videogames using JavaFX. It is a nearly-complete port of SimpleGame.js into Java.

Getting Started

In order to create a videogame with SimpleGame (Java Edition), implementations for each of the following classes must be used:

- Display.java
- Game.java
- Keyboard.java
- SimpleScene.java
- SpriteSheet.java (Mandatory if Sprite is used)
- Sprite.java (Optional, but generally necessary)

Display.java is the main class, so to compile your SimpleGame projects, run:

```
javac Display.java
java Display
```

However, the user should almost never modify Display.java unless it is to make modifications to the game engine or to change the framerate. **The entry point for developers is Game.java**.

Every Game class definition *must* contain a `public void init()` method and a `public void update(double t)` method.

The `init()` method is similar the `init()` function in SimpleGame.js. This method is run once at the beginning of the application and should be used to initialize member variables for your game.

The `update(double t)` method is similar to the `update()` function in SimpleGame.js. This method is run once every frame and should be used to clear the main scene and update all of the sprite's data. The argument of the update method, `t`, is the frametime of the last frame. This was included because of the AnimationTimer for JavaFX; however, it is not necessary to add any complexity to your code to handle this. Just pass the `t` to the `update()` method of every Sprite object and the sprites will use it internally.

The template code for the **Game** class should appear as below:

```
1  public class Game{
2
3      ////////////////////////////////////////////////////
4      //////////      MANDATORY PROPERTIES     ///////////
5      ////////////////////////////////////////////////////
6
7      public String title = "This is my Game!";
8      public SimpleScene scene = new SimpleScene(1280, 680);
9      public Keyboard k = new Keyboard();
10     ////////////////////////////////////////////////////
11
12
13
14     //declare your variables here, if you want
15
16
17
18
19     public void init(){
20
21         //your code goes here
22
23         //end this method with this.scene.start()
24         this.scene.start();
25
26     }//end init
27
28
29
30     public void update(double t){
31
32         //begin by clearing the scene
33         this.scene.clear();
34
35         //your code goes here
36
37
38     }//end update
39
40
41
42
43
44
45 }//end Game class definition
46
```

There are 3 public properties that the Game class *must* include:

- **`title`** `(String)`: the title that will appear at the top of the window when running the game.
- **`scene`** `(SimpleScene)`: the scene object that will be used for the game.
- **`k`** `(Keyboard)`: the Keyboard handler for receiving and recording user input.

You are free to create additional properties in the Game class beyond the mandatory 3.

What's New in SimpleGame (Java Edition)?

On top of most of the core functionality of the original SimpleGame engine, there are a few new features to SimpleGame (Java Edition).

Bound action: BACKGROUND

In addition to the original bound actions available to Sprites (in order to handle their behavior when passing edges of the screen), a new bound action is available for images that are intended to act as background images: `"BACKGROUND".` If you need a background image that is larger than the screen size and you want it to move around, then this bound action might be what you are looking for. The main functionality of the BACKGROUND bound action is that it will lock the Sprite's x or y position if it is about to expose the frame. In order to use this bounding action, simply instantiate a Sprite and call `sprite.setBoundAction("BACKGROUND").`

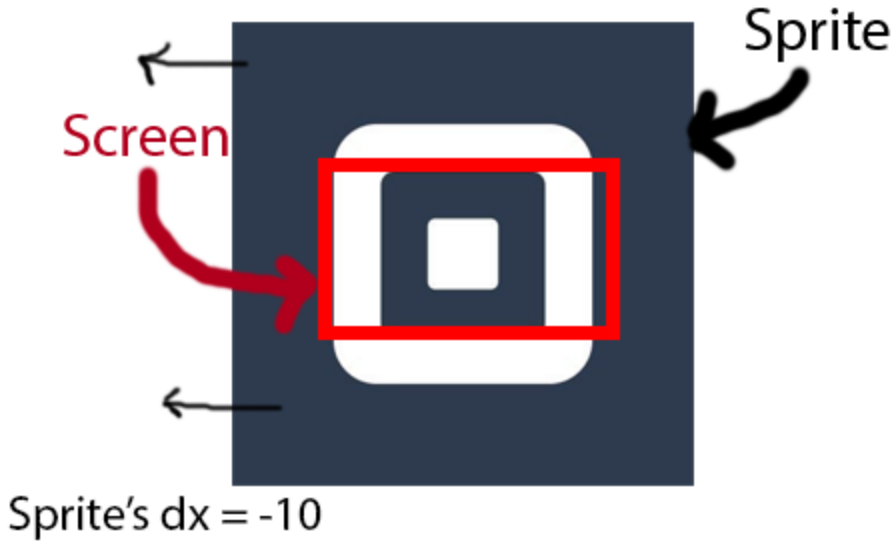


Figure 1 | A sprite with bounding action set to BACKGROUND. Note the sprite's dx = -10 and would normally begin to expose the empty background of the screen if it continues.

Figure 2 | The same sprite with bound action set to BACKGROUND. It cannot move to the left any further, as the bound action pulls it back (the the empty screen behind this sprite is not exposed).

Collision type: HITBOX

The original collision detection mechanism for SimpleGame.js was exclusively bounding box collisions. Bounding boxes are good for their runtime efficiency; however, they can be incredibly imprecise, as they often times return false positives for detecting collisions (especially for Sprites with complicated shapes). To remedy this, SimpleGame (Java Edition) includes an optional hitbox collision mechanism. Hitboxes make up a collection of smaller bounding boxes to provide close runtime efficiency to bounding box collision detection, but allow for greater precision and customization of the collision process. To assign a Sprite to hitbox collision (instead of bounding box), use the `setCollisionType()` method with argument: **"`hitbox`"**:

```
sprite.setCollisionType("hitbox");
```

Until a hitbox is added manually, the default hitbox for the sprite will be the default bounding box. To add hitboxes, use the `addHitbox()` method:

```
sprite.addHitbox(new Double[]{-30.0, 22.0, -70.0, -15.0});
```

The argument for the `addHitbox()` method is an array of type Double (capital "D") with 4 entries:

```
sprite.addHitbox(new Double[]{left-wall, right-wall, top-wall, bottom-wall});
```

Each of the 4 entries is an X or Y value of the location of the different walls (imagining that the origin (0,0) is at the center of the sprite). *left-wall* is the X coordinate of the left wall of the

hitbox, *right-wall* is the X coordinate of the right wall of the hitbox, *top-wall* is the Y coordinate of the top wall of the hitbox, and *bottom-wall* is the Y coordinate of the bottom wall of the hitbox.

You can add as many hitboxes as you want. In order to display the hitboxes (so you can easily see their locations) use the `displayHitboxes()` method:

```
sprite.displayHitboxes();
```

If you are currently displaying hitboxes and you want them to be dynamically turned off, use:

```
sprite.hideHitboxes();
```

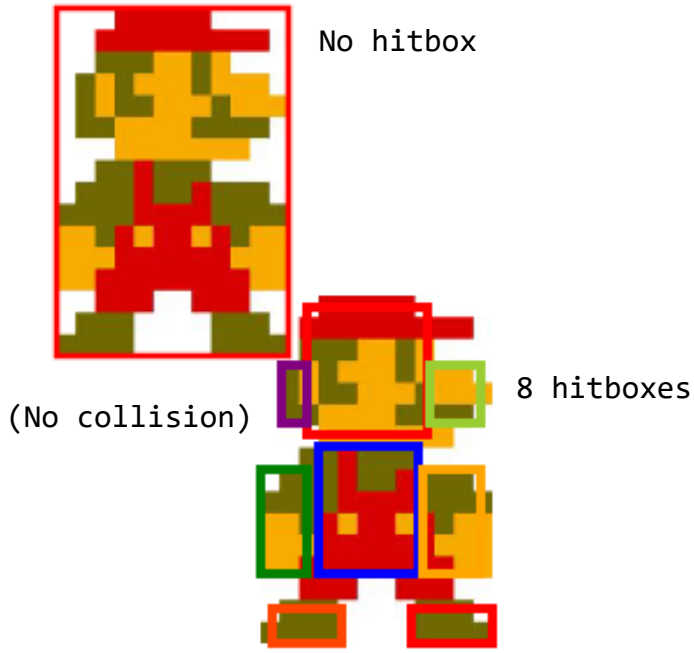
If you still have the reference to a specific hitbox and you want to remove it from the sprite, use:

```
sprite.removeHitbox(Double[] hitboxToRemove);
```

Lastly, to change the line width of the hitboxes (which only affects the display, not the hitboxes themselves) use:

```
sprite.setHitboxDisplayWeight(double weight);
```

The default display weight is 2.5.



A note about the collision mechanism: in SimpleGame (Java Edition) there are currently 2 collision mechanisms — "bounding box" and "hitbox". The real concern is with the defined behavior when a Sprite with one type of collision mechanism collides with a Sprite that has a different collision mechanism. To define these behaviors, imagine that `s1` is a Sprite and `s2` is a Sprite and `s1.collidesWith(s2)` is run:

- If `s1` is set to "bounding box" and `s2` is set to "hitbox", then `s1` assesses bounding box collision with the full bounding box of `s2` (not its hitboxes).
- If `s1` is set to "hitbox" (but does not have hitboxes added) and `s2` is set to "bounding box", then `s1` creates a temporary hitbox (mimicking the bounding box) for itself and for `s2`. Full bounding box collision is then evaluated.
- If `s1` is set to "hitbox" and has hitboxes added to it, while `s2` is set to "bounding box", `s1` builds a temporary hitbox (mimicking the bounding box) just for `s2`, and checks collisions against its own hitboxes with the bounding box of `s2`.
- If `s1` and `s2` are both set to "hitbox", then each combination of their hitboxes is checked for collisions.
- If `s1` and `s2` are both "bounding box", then full bounding box collision is used.

SpriteSheets and Animations

Sometimes, you will want to use animated sprites in your game. SimpleGame (Java Edition) accomplishes this with sprite sheets. The format for sprite sheets in this game engine is that every row is a *state* (facing forward, facing right, etc.) and every column is a *frame* of the given state:

Author: bagzie

Source: https://opengameart.org/content/bat-sprite

In order to make animated sprites, create an instance of the SpriteSheet class. SimpleGame (Java Edition) attempts to simplify this process whilst enabling the user to have optional extended functionality. Assuming that the previous image is called “bat.png”, we can create the animated sprite by creating the spritesheet, then passing that spritesheet instance to the sprite instead of an image file name:

```
SpriteSheet(String filename, int rows, int maxCols)
Sprite(SimpleScene scene, SpriteSheet ss)
```

For the SpriteSheet constructor, the arguments are defined as follows:

- **filename:** the name of the path to the file for the image
- **rows:** the number of rows (states) in an image
- **maxCols:** the maximum number of columns that appears in any given state (the SpriteSheet has the ability to handle images in which different states contain different amounts of frames).

If you want to pass in an animated SpriteSheet, but you want to scale it to a specific size, you can use the following Sprite constructor:

```
Sprite(SimpleScene scene, SpriteSheet ss, int newWidth, int newHeight)
```

By default, the SpriteSheet is set to play (animate) the very first state. Of course, your animations may contain many states and you will want to switch between them, customize them, pause them, etc. The SpriteSheet class has additional functionality for using and customizing your animations.

The playback speed of an animation can be adjusted in a couple of different ways. At the low level, every state has its own playback speed. So if you want the playback speed of a specific state, use:

```
public void setSpeed(int state, double speed)
```

Note, the states of the SpriteSheets are indexed starting at 0, so the first row is 0, second row is 1, etc.. If you want to set the playback speed of the current state, you can just use:

```
public void setSpeed(double speed)
```

Finally, if you want to set the speed for *all* states in the animation, you can use:

```
public void setGlobalSpeed(double speed)
```

You might, at some point, want to pause your animation on a specific frame and/or then resume the animation later. You can use the `pause()` method to pause the animation on whatever state and frame it is set to, and you can use the `play()` method to allow the animation to progress forward at its given speed.

If you want to reset the frame of the state back to the first frame (or any other for that matter), you can use:

```
public void setFrame(int frame)
```

Note, frames are also indexed at 0, so the very first frame of any given state will be indicated by 0. There is also a similar method for changing the state of a SpriteSheet:

```
public void setState(int state)
```

It’s understandable that you might want to address the different states by custom names, so this engine has implemented the necessary functionality for you to do that. Each SpriteSheet can be

given an ArrayList<String> object containing the names for each state, though using this is completely optional. You can instantiate and define your ArrayList<String> object in your Game object, then pass it to the SpriteSheet with:

```
public void setMap(ArrayList<String> names)
```

Then, you can use the overloaded setState() method to use these mappings (note, the mappings are NOT case sensitive):

```
public void setState(String stateNameFromTheListYouJustPassedIn)
```

Some sprite sheets have different amounts of frames for each state. This game engine allows you to customize for that as well. To set the number of frames in a specific state, use:

```
public void setFramesPerState(int state, int frames)
```

Of course, you would normally only use this method in your init() function as it initializes a SpriteSheet. However, this can be a bit cumbersome, and some SpriteSheets are very popular, such as the Universal LPC Sprite Sheet format (http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/#). So, the SpriteSheet class includes a preset for this format (and the ability for you to fairly easily add your own presets if you want to modify the engine a bit). To instantiate a SpriteSheet that use the Universal LPC format, then automatically set its number of frames for every state along with its list of String mappings, use:

```
spriteSheet.setPreset(“LPC”);
```

This will initialize a SpriteSheet to be in the proper structure for this popular sprite sheet format. (Note: this preset does not support Universal LPC Sprite Sheet images that include oversized weapons. For this functionality, cut the sprite sheet into 2 parts as they are in different formats. Then, swap out the SpriteSheet objects with the `Sprite.setSpriteSheet()` method to use both SpriteSheets on a single Sprite.)

Class Documentation

Most of the methods are the same as the original SimpleGame.js, but a few things are slightly different because of the different technology used (JavaFX). Here is the complete object dictionary for SimpleGame (Java Edition):

SimpleScene

SimpleScene parallels the Scene object from the original SimpleGame.js library. This name change was due to a namespace issue. However, much of the original functionality of the Scene object is no longer needed, so SimpleScene contains noticeably less methods.

Return type	Method name	Description
(constructor)	SimpleScene()	Default constructor for SimpleScene. Creates a Canvas of width = 640, height = 480
(constructor)	SimpleScene(int width, int height)	Overloaded constructor for SimpleScene. Creates a Canvas with the specified width and height.
int	getWidth()	Returns the width of the Canvas.
int	getHeight()	Returns the height of the Canvas.
boolean	isVisible()	Returns true is the SimpleScene is set as visible. (If it's not visible, the Canvas is not shown on the screen).
void	show()	Sets the SimpleScene to be visible.

void	hide()	Sets the SimpleScene to be *not* visible.
void	start()	Starts the SimpleScene. (Call this method at the end of your init() implementation)
void	stop()	Halts the SimpleScene. If you want to end your game, call this method.
boolean	getState()	Returns true if the SimpleScene is running.
void	clear()	Clears the screen with a giant white rectangle.

Sprite

Return type	Method name	Description
(constructor)	Sprite()	Default constructor for Sprite class. (Not intended for use)
(constructor)	Sprite(SimpleScene scene, String imageName, int imgWidth, int imgHeight)	Single-image constructor for Sprite class. This is the constructor from the original SimpleGame.js. Use this if you want to easily make a non-animated Sprite. [scene] is the SimpleScene that the Sprite will be drawn to, [imageName] is the path to the image file, and [imgWidth]/[imgHeight] are the dimensions that you want the Sprite to stretch/shrink the image to.
(constructor)	Sprite(SimpleScene scene, String imageName)	Intelligent single-image constructor. Creates non-animated Sprite with the image at the path defined by [imageName] and is drawn to SimpleScene [scene]. The width and height of the Sprite is automatically set to the height and width of the image.
(constructor)	Sprite(SimpleScene scene, SpriteSheet ss)	Animated constructor. Use this constructor to build a Sprite that

		uses an animated SpriteSheet. [scene] is the SimpleScene that the Sprite will be drawn to and [ss] is the SpriteSheet that you want the Sprite to use.
(constructor)	Sprite(SimpleScene scene, SpriteSheet ss, int w, int h)	Animated-resize constructor. Use this constructor if you want to create an animated Sprite that is stretched/shrunk to different dimensions. [scene] is the SimpleScene that the Sprite is drawn to, [ss] is the SpriteSheet that the Sprite will use, and [w]/[h] are the width and height (respectively) of the stretched/shrunken Sprite.
int	getWidth()	Returns the width of the Sprite.
int	getHeight()	Returns the height of the Sprite.
long	getX()	Returns the x location of the Sprite.
long	getY()	Returns the y location of the Sprite.
double	getDX()	Returns the Sprite’s dx value (dx = velocity projected onto the x-axis)
double	getDY()	Returns the Sprite’s dy value (dy = velocity projected onto the y-axis)
boolean	isVisible()	Returns true if the Sprite is visible. Sprites will be drawn to the screen and be collidable only if they are visible.
double	getSpeed()	Returns the overall speed of the Sprite by calculating it with dx and dy.
double	getMaxSpeed()	Returns the Sprite’s maximum set speed.
double	getImgAngle()	Returns the current angle of the Sprite’s image (in degrees, offset

		by 90 (0 degrees is straight up, 90 degrees is an angle to the right))
double	getSpeedScale()	Returns the Sprite's speed scaling factor.
String	getCollisionType()	Returns the current setting for the Sprite's collision mechanism. Default is "bounding box". "hitbox" is also a supported option.
boolean	getHitboxDisplay()	Returns true if the Sprite is configured to display its hitboxes when drawn.
int	getNumHitboxes()	Returns the number of hitboxes that the Sprite currently holds.
double	getHitboxDisplayWeight()	Returns the current line weight setting for the Sprite's hitboxes (when drawn)
SpriteSheet	getSpriteSheet()	Returns a reference to the SpriteSheet that the Sprite uses (all Sprites, even non-animated ones, use a SpriteSheet).
void	setWidth(int w)	Sets the width of the Sprite to [w].
void	setHeight(int h)	Sets the height of the Sprite to [h].
void	changeImage(String imgFile)	Changes the Image of the Sprite to the Image specified by the path, [imgFile]. Note, this does not reset the height and width of the image nor does it change row/column properties of the SpriteSheet. Only use this method if the dimensions of the image will be the same. Otherwise, make a new SpriteSheet and swap that out instead.
void	setImage(String imgFile)	Alias for changeImage(String imgFile). It does the exact same thing.
void	setX(long xpos)	Sets the x position of the Sprite to [xpos].
void	setY(long ypos)	Sets the y position of the Sprite to [ypos].
void	setPosition(long xpos, long ypos)	Sets the x position of the Sprite to [xpos] and the y position of the Sprite to [ypos].

void	setDX(double ndx)	Sets the dx value of the Sprite to [ndx] (dx = the velocity projected onto the x-axis).
void	setDY(double ndy)	Sets the dy value of the Sprite to [ndy] (dy = the velocity of the Sprite projected onto the y-axis)
void	setChangeX(double ndx)	Alias for setDX(double ndx).
void	setChangeY(double ndy)	Alias for setDY(double ndy).
void	changeXby(double tdx)	Adds [tdx] to the Sprite's x position.
void	changeYby(double tdy)	Adds [tdy] to the Sprite's y position.
void	setSpeed(double speed)	Sets the speed of the Sprite to [speed] and recalculates dx and dy based on the current direction and new speed.
void	setMaxSpeed(double speed)	Sets the maximum speed that the Sprite is able to move at.
void	changeSpeedBy(double diff)	Adds [diff] the the Sprite's speed and recalculates the Sprite's dx and dy based on the current direction and new speed if it remains under maxSpeed.
void	setImgAngle(double degrees)	Sets the angle in which the Sprite's image will be displayed (in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))
void	changeImgAngleBy(double degrees)	Adds [degrees] to the current Image display angle.
void	setMoveAngle(double degrees)	Sets the angle in which the Sprite will make movements (in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))

void	changeMoveAngle(double degrees)	Adds [degrees] to the current angle at which the Sprite makes its movements.
void	setAngle(double degrees)	Sets both the image drawing angle and the Sprite moving angle to [degrees] (in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))
void	changeAngleBy(double degrees)	Adds [degrees] to the Sprite's image angle and move angle (in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))
void	turnBy(double degrees)	Alias for changeAngleBy(double degrees)
void	hide()	Sets the Sprite's visibility to false. Sprites with false visibility will not be considered when computing collisions and will not be drawn to the screen.
void	show()	Sets the Sprite's visibility to true. Sprites with true visibility will be considered in collision calculations and will be drawn to the screen.
void	setBoundAction(String action)	Sets the bounding action to [action] for the Sprite. Default value is "wrap", but "wrap", "bounce", "stop", "die", and "background" are also supported. Not case sensitive.
void	setSpeedScale(double ss)	Sets the speed scale of the Sprite to [ss]. When the Sprite's state is updated (each frame), its dx and dy values are multiplied by the speedScale before changing x and y.
void	setCollisionType(String type)	Sets the collision type of the Sprite to [type]. Default is "bounding box", but "hitbox" is also supported. Not case sensitive.
void	displayHitboxes()	Sets the Sprite to draw its hitboxes on each frame.
void	hideHitboxes()	Sets the Sprite to *not* display its hitboxes on each frame. This behavior is the default.

void	addHitbox(Double[] hitbox)	Adds [hitbox] to the ArrayList of hitboxes that the Sprite contains. The format of [hitbox] is {x-left-wall, x-right-wall, y-top-wall, y-bottom-wall} where each value is an x OR a y coordinate in relation to the center of the Sprite as (0,0)
void	removeHitbox(int i)	Removes the hitbox in the Sprite's hitbox list at index [i].
void	removeHitbox(Double[] hitbox)	Removes the reference [hitbox] from the hitbox list of the Sprite.
void	setHitboxDisplayWeight(double weight)	Sets the line width for the hitbox rectangles to (when drawn) to [weight]. The default weight is 2.5
void	setSpriteSheet(SpriteSheet ss)	Sets the SpriteSheet in which the Sprite uses to [ss]
void	update(double t)	Updates the position of the Sprite, progresses the SpriteSheet animation in time, checks the bound actions of the Sprite, then draws the Sprite to the screen. [t] is the frametime of the last frame and is used by the Sprite and SpriteSheet to calculate the correct changes in state.
void	updateState(double t)	Updates the state of the Sprite without drawing it. Updates the position of the Sprite, progresses the SpriteSheet animation in time, and checks the bound actions of the Sprite.
void	draw()	Applies transformations to the Sprite and draws it to the screen. Also draws the hitboxes if displayHitboxes() has been enabled.
void	addVector(double degrees, double thrust)	Adds a motion vector to the Sprite's current motion variables. Similar to adding Newton force vectors. [degrees] is the angle in which the force will be applied and

		[thrust] is the magnitude of that force. ([degrees] is in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))
boolean	collidesWith(Sprite sprite)	Uses the Sprite's collision mechanism to evaluate a collision with [sprite]. Returns true if there is a collision, false if there is not. If the current Sprite has "hitbox" set as its collision mechanism, but [sprite] does not, a temporary hitbox (matching the bounding box) is created and used for [sprite].
double	distanceTo(Sprite sprite)	Returns the distance between the center of the Sprite and the center of [sprite].
double	angleTo(Sprite sprite)	Returns the angle (in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.)) from the center of the current Sprite to the center of [sprite].
void	report()	Prints status information about the Sprite.
double	degToRad(double deg)	Returns the radian form of [deg] (where [deg] is in degrees, with 90 degree offset (straight up is 0 degrees, to the right is 90 degrees, etc.))

SpriteSheet

SpriteSheet is a new class that represents data about an image file and manages the different states of that image in order to facilitate animated Sprites. Every Sprite contains a SpriteSheet in SimpleGame (Java Edition).

Return Type	Method Name	Description
(constructor)	SpriteSheet()	Default constructor of the SpriteSheet class. Not intended for use.
(constructor)	SpriteSheet(String file)	Non-animated constructor of the SpriteSheet class. Initializes an Image at the path [file] and sets the states and frames both to the value of 1.

(constructor)	SpriteSheet(String file, int rows, int maxCols)	Regular overloaded constructor of the SpriteSheet class. Initializes an Image at path [file], and sets the SpriteSheet's number of states to [rows] and the maximum number of frames that appears to [maxCols]. From these 2 numbers, the SpriteSheet calculates the height and width of each frame.
(constructor)	SpriteSheet(String file, int w, int h, int rows, int maxCols)	Specialized overloaded constructor for SpriteSheet class. Initializes an Image at path [file], sets the SpriteSheet's number of states to [rows] and the maximum number of frames that appears to [maxCols], and sets the visible width of the SpriteSheet to [w] and the visible height of the SpriteSheet to [h]. This enables the user to only use a section of an image as a SpriteSheet.
void	step(double t)	Uses the last frametime [t] and the speed of the current state to evaluate whether or not the SpriteSheet needs to update the frame it displays and updates the frame accordingly. This method is intended for the Sprite class to use internally. The user should not use this method. Instead, use play() and pause().
void	setPreset(String preset)	Customizes properties of the SpriteSheet based on the value [preset]. Currently, the only installed preset is "LPC" for the Universal LPC SpriteSheet format.

Image	getImage()	Returns a reference to the Image used by the SpriteSheet.
double	getFullWidth()	Returns the full width of the Image that the SpriteSheet observes.
double	getFullHeight()	Returns the full height of the Image that the SpriteSheet observes.
double	getWidth()	Returns the width of a single frame of the SpriteSheet.
double	getHeight()	Returns the height of a single frame of the SpriteSheet
int	getNumStates()	Returns the number of states in the SpriteSheet.
int	getMaxFrames()	Returns the maximum number of frames that appears in any of the given states of the SpriteSheet. (This is used to calculate the width of each frame).

int	getCurrentState()	Returns the index of the current state of the SpriteSheet (indexed starting at 0).
int	getCurrentFrame()	Returns the index of the current frame of the current state of the SpriteSheet (indexed starting at 0).
double	getSpeed(int state)	Returns the speed of the given [state] of the SpriteSheet.
double	getSpeed()	Returns the speed of the current state of the SpriteSheet.
boolean	isPaused()	Returns true if the SpriteSheet is set to *not* update its frames.
int	getFramesPerState(int state)	Returns the observed number of frames in the given [state] index
ArrayList<String>	getMap()	Returns the list of strings that act as names for each state.
int	getStateIndex(String state)	Returns the index of [state] as found in the SpriteSheet's naming map.
void	setImage(Image img)	Sets the Image of the SpriteSheet to [img]

void	setWidth(int w)	Sets the observed width for each frame to [w]
void	setHeight(int h)	Sets the observed height for each frame to [h]
void	setNumStates(int num)	Sets the number of observed states to [num]
void	setMaxFrames(int frames)	Sets the maximum number of frames in any of the given states to [frames].
void	setState(int s)	Sets the current state to index [s].
void	setState(String s)	Sets the current state the index discovered by the index of [s] in the SpriteSheet's naming map.
void	setFrame(int f)	Sets the current frame to [f]
void	setSpeed(double s)	Sets the playback speed to [s] for the *current state*.
void	setSpeed(int state, double s)	Sets the playback speed to [s] for the *given state* [state].
void	setSpeed(String state, double s)	Sets the playback speed to [s] for the state defined by the SpriteSheet's name mapping for [state].

void	setGlobalSpeed(double s)	Sets the speed for *all states* in the SpriteSheet to [s].
void	pause()	Prevents the SpriteSheet from animating automatically until play() is called.
void	play()	Allows the SpriteSheet to animate itself automatically. By default, the SpriteSheet is playing.
void	setFramesPerState(int state, int frames)	Sets the number of frames that are observed in the given [state] to [frames]
void	setFramesPerState(String state, int frames)	Sets the number of frames that are observed in the state to [frames] (the state is discovered by the SpriteSheet's name mapping for [state])
void	setMap(ArrayList<String> names)	Assigns the list [names] as the string name mappings for each state (so users don't need to index states by integers)

Sound

Sound is an additional class available for implementing sounds.

Return Type	Method Name	Description
(constructor)	Sound()	Default constructor for Sound class. Not intended for use.
(constructor)	Sound(String file)	Initializing constructor for Sound class. Instantiates a Sound object using the audio file located at path [file]
void	play()	Begins/resumes playback of the sound.
void	pause()	Pauses playback of sound (but remembers where you left off)
void	stop()	Stops the sound. The next time play() is called, the sound will start from the beginning.

Keyboard

The Keyboard class is a utility class for reading user input for the Keyboard and mouse. It contains a public array of boolean variables representing the state of keys on the keyboard. The value is true if the key is currently pressed down, false otherwise.

```
public boolean[] keysDown = new boolean[256]
```

The Keyboard class contains many public mappings to make inspection of keysDown easier:

```
//keyboard constants
public final int A = 65, B = 66, C = 67, D = 68, E = 69, F = 70, G = 71,H = 72, I = 73, J = 74, K = 75, L = 76, M = 77, N = 78, O = 79, P = 80, Q = 81, R = 82, S = 83, T = 84, U = 85, V = 86, W = 87, X = 88, Y = 89, Z = 90, LEFT = 37, RIGHT = 39, UP = 38, DOWN = 40, SPACE = 32, ESC = 27, PGUP = 33, PGDOWN = 34, HOME = 36, END = 35, K0 = 48, K1 = 49, K2 = 50, K3 = 51, K4 = 52, K5 = 53, K6 = 54, K7 = 55, K8 = 56, K9 = 57;

//additional keyboard constants
public final int SHIFT = 15, CTRL = 17;
```

So to check the state of the A key in the Game class (with the instance of Keyboard, [k]):

```
k.keysDown[k.A] | returns true if key is pressed down
```

The Keyboard also contains some prototype functionality for mouse data:

```
public double mouseX | current x position of the mouse
public double mouseY | current y position of the mouse
public boolean clicked | true if the mouse is clicked down, false otherwise.
```