Javacade (v1.0)

- Welcome to Javacade, a retro arcade game engine made in Java using JavaFX! The API described here provides all the tools required for making a classic arcade-style game. The only prerequisite is basic knowledge of Java program structure. However, knowledge of JavaFX can help make some more advanced games.

The arcade is fixed to a resolution of 600px X 600px. It opens to a main menu, from which you can select one of many games to play. Each game has a pre-built pause screen and a configurable main menu screen.

Capabilities of the Javacade API

- Play music and sound effects during gameplay
- Make custom shape sprites and color them with a flat color or gradient
- Animate or manually move, rotate, and scale the sprites
- Add key bindings easily to perform any action, like moving sprites and playing sound effects
- Detect collisions between individual/groups of sprites and perform actions upon collision
- Pick One of 6 Pre-Installed Retro Fonts

1. **Press Start** *(Default)*
2. **DisposableDroidBB**
3. **Manaspace**
4. **Null Pointer**
5. **Pixel**
6. **Yoster Island**

How to Start

Every game starts in its own folder with four main components.

1. The game's primary class, which extends the abstract class **GameRootPane**. These two classes and their requirements are explained in full in the Game Root Pane section of this document.
2. A preview image, `120px X 120px`, to be shown in the game selection menu. If no image is provided, a picture saying "no image" will display instead.
3. The sub-folder "music", which contains the game's main menu music and background music. The two are played automatically, but additional background music can be included and played manually.
4. The sub-folder, **sfx**, which contains all sound effects played in your game.
5. From here, additional classes and folders can be added for more advanced games, but this is all that is required.

Game Root Pane

GameRootPane is an abstract class which all games must extend. The game's logic is configured using four abstract functions. The constructors are used mainly to configure the game by passing in basic parameters and running three optional functions.

Constructors

```
GameRootPane (String gamePackageName, String gameTitle, String localFont, String backgroundMusicFileName, double musicVolume)
```

- Constructs the game by giving the program basic information about the game. Call this constructor in your game's main constructor.
  `@param gamePackageName`: The name of the folder your game is in.
  `@param gameTitle`: the title of your game, as it should be displayed
  `@param localFont`: the font used in this game
  `@param backgroundMusicFileName`: the file name with music file type (.mp3, .wav, etc)
  `@param musicVolume`: the volume of the game's music, 0 to 100.

```
GameRootPane (String gamePackageName, String gameTitle, String backgroundMusicFileName)
```

- Constructs the game the same as above, but with
  `@param localFont`: Press Start
  `@param musicVolume`: 50

It is highly suggested to run these functions in your constructor after invoking the parent constructor

- `void initMenu (double titleSize, double menuElementSize, String menuMusicFileName, Paint bgFill)`
  - Customizes the game's main menu screen
    `@param titleSize`: the point size of the title's font (e.x. 12 point font)

`@param menuElementSize`: the point size of the menu elements' font
`@param menuMusicFileName`: the file name with music file type (.mp3, .wav, etc)
`@param tutorialText`: the text for the "how to play" screen
`@param bgFill`: the color of the main menu's background, defined either by the enumerator `javafx.scene.paint.Color`, or the `radialGradient()` and `linearGradient()` functions in `framework.Util`.

- `void createTimer (double fontSize, boolean enableDropShadow, Pos position)`
  - Optionally enables and places a timer in the game scene
    `@param fontSize`: the point size of the timer's font (e.x. 12 point font)
    `@param fontColor`: the color of the timer's text
    `@param dropShadorColor`: the color of the timer's drop shadow
    `@param position:` the position of the timer, as defined by the enumerator `javafx.geometry.Pos`.

Abstract Functions

- `void update()`
  - This function is run every frame. Put collision checks and network updates within this function.
- `void onPause()`
  - This function is run upon pausing the game. `pause()` all currently animated nodes within this function.
- `void onResume()`
  - This function is run upon resuming the game. `resume()` all currently animated nodes within this function.
- `void onGameStart()`
  - This function is run upon clicking "play" in the game's main menu. INITIALIZE ALL SPRITES WITHIN THIS FUNCTION.
  - Run `setBackground (Paint color)` within this function to set the background for the game itself, if it is different than the game's main menu background.

Key Bindings

- `addKeyBinding (KeyAction action)`
  - Adds a key binding to the game. This uses the implicitly defined Interface:

```
public interface KeyAction {KeyCode getKey(); boolean fireOnce(); void action();}
```

- ```
  addKeyBinding(new KeyAction(){
      public KeyCode getKey()
       {return KeyCode.key;}
      public boolean fireOnce()
       {return boolean;}
      public void action ()
       {" take action"}
  });
  ```

  • *KeyCode.key* is an enumerator from the class `javafx.scene.input.KeyCode`, that specifies what key this binding is for

  • *boolean* defines of the key binding should fire once (true), or continue to fire as the key is held down (false)

  • *action* is the code to execute upon firing the key binding.
  ```

Score

- The Score object contains and displays your game score. Multiple Scores can be instantiated.

Constructor

- `Score (GameRootPane parent, double fontSize, Paint fontColor, Color dropShadowColor, Pos position)`
  `@param parent`: the game containing this score counter. Since you should only declare a score counter in your primary file, just pass in `this`.
  `@param fontSize`: the point size of the score counter's font (e.x. 12 point font)
  `@param fontColor`: the color of the score counter's text, defined either by the enumerator `javafx.scene.paint.Color`, or the `radialGradient()` and `linearGradient()` functions in `framework.Util`.
  `@param dropShadowColor` the color of the score counter's drop shadow, as defined only by the enumerator `javafx.scene.paint.Color`
  `@param position`: the position of the timer in the game window, as defined by the enumerator `javafx.geometry.Pos`
- `Score (GameRootPane parent, double fontSize, Pos position)`
  - A simpler constructor. Same as above, but with the colors set to default values
  - `@param fontColor`: Color.BLACK
  - `@param dropShadowColor`: Color.TRANSPARENT
- `void disableScore()`
  - Temporarily disables the score counter after it has been enabled
- `void enableScore()`
  - Re-enables the score counter after it has been disabled
- `void addToScore(double score)`
  - Adds the double passed to the score and updates the score counter
- `void removeFromScore()`
  - Removes the double passed in from the score and updates the score counter
- `double getScore()`
  - Returns a double equal to the score.

Music

- `void setBgMusic (String musicFileName, double volume)`
  - Stops previously playing background music, initializes new background music, and begins playing it.
  - `@param volume` MUST be between 0.0 and 100.0
  - `@param musicFileName`: the exact file name, including type extension (.mp3, .wav, etc)
- `void playSfx (String gamePackageName, String sfxFileName, double volume)`
  - This initializes the sound effect, and subsequently plays it.
  - `@param volume` MUST be between 0.0 and 100.0
  - `@param musicFileName` the exact file name, including type extension (.mp3, .wav, etc)

Pixel Sprite

Constructors

- `PixelSprite (int [][] pixels, double realHeight, double realWidth, String id, Paint... fills)`
  - Creates a pixelated sprite. A 2-D array of integers is passed in to define which pixels to draw. For each element of the array, if it is equal to anything except 0, a pixel will be drawn there. The integers in the array must count up from 0, and the number of colors passed in must be equal to the highest number in the int array.
  - For example, this creates a monochrome T-shaped sprite:
    `int[][] tSprite = new int[][] {{1,1,1,1,1}, {0,0,1,0,0}, {0,0,1,0,0}, {0,0,1,0,0}, {0,0,1,0,0}};`
  - The sprite does not have to be continuous, i.e. the individual pixels do not need to be connected, so:
    `int[][] tSprite = new int[][] {{0,0,1,0,0}, {0,0,0,0,0}, {0,0,2,0,0}, {0,0,2,0,0}, {0,0,2,0,0}};`
    - would be valid, and create a multicolor i-shaped sprite.
  - Despite whatever pixels you specify, the sprite will always have the height `realHeight` and the width `realWidth`.
  - If this sprite is one of a family of sprites, set the id equal to the family's id, to allow for checking collision with all of them simultaneously.
    `@param pixels`: an integer array to define the sprite
    `@param realHeight`: height of the entire sprite, NOT each pixel
    `@param realWidth`: width of the entire sprite, NOT each pixel
    `@param id`: id of this sprite, used in checking collisions
    `@param fills`: the colors of the sprite.
- `void setConstrainToScene(boolean bool)`
  - Set if the sprite is to be prevented from moving outside of the scene or not.
- `void getConstrainToScene()`
  - Returns a boolean representing if the sprite is to be prevented from moving outside of the scene or not.

Collision

- `boolean collided(String id)`
  - Returns a boolean representing if this sprite has collided with ANY other sprite with the specified id. Used to check for collisions with a family of nodes, instead of one specific node.

    `@param id`: the id of the group of nodes for which to check collision
- `boolean collided(Node other)`
  - Returns a boolean representing if this sprite has collided with the ONE other sprite specified
- `Node getCollided(String id)`
  - Returns the node with the given id that this sprite has collided with. If it has not collided with any node with the specified id, this will return null. If you want to delete nodes on collision, use this function to get the node, then run
- `removeSprite(node)`
  - in your game's `update()` function

    `@param id`: the id for the group of nodes for which to check collision
- `Node getCollided(Node other)`
  - Returns the node specified if this sprite has collided with it. Otherwise, returns null.

Manual Movement

- These functions augment the sprite's position in a variety of ways. The effect always occurs immediately after running the function.
- `void scale(double scaleBy)`
  - Scales the sprite by a factor of `scaleBy`
- `translate(double x, double y)`
  - Translates the sprite by `x` horizontally and `y` vertically. Cannot translate sprites outside of the scene.
- `void moveTo(double x, double y)`
  - Moves the sprite to the specified coordinates.
- `void rotate(double deg)`
  - Rotates the sprite clockwise by the specified degrees from its current rotation
- `void rotateAbsolute(double deg)`
  - Rotates the sprite to the specified degrees clockwise from the positive y-axis
- `void rotateToNearest90()`
  - Rotates the sprite to the nearest 90 degree angle, squaring it up with the scene

Animated movement

- Performs the same actions as the functions above, but smoothly, over a period of time, instead of instantly.
- Each animation's length is specified by `timeInMs`
- The animation can occur only once, or cycle back and forth indefinitely, as specified by the boolean `cycle`
- If the animation must be completed before any other action must be made, set `allowInterrupt` to false. If a manual augmentation should stop the animation, set `allowInterrupt` to true
- Lastly, all animations can be paused and resumed, with their respective `pause()` and `resume()` functions. If you want to pause all animations, use `pause()` and `resume()`, without the suffix.
- `void scaleAnimation (double scaleBy, double timeInMs, boolean cycle, boolean allowInterrupt)`
  - `void pauseScale()`
  - `void resumeScale()`
- `void translateAnimation (double x, double y, double timeInMs, boolean cycle, boolean allowInterrupt)`
- `void moveToAnimation (double x, double y, double timeInMs, boolean cycle, boolean allowInterrupt)`
- `void pauseTranslate()`
- `void resumeTranslate()`
- `void rotateAnimation (double deg, double timeInMs, boolean cycle, boolean allowInterrupt)`
- `void rotateAbsoluteAnimation (double deg, double timeInMs, boolean cycle, boolean allowInterrupt)`
- `void rotateToNearest90Animation (double timeInMs, boolean cycle, boolean allowInterrupt)`
  - `void pauseRotate()`
  - `void resumeRotate()`

Adding sprites to the game

- `void addSprite (PixelSprite sprite)`
  - Adds a sprite to the scene.
- `void addSprite (PixelSprite sprite, double x, double y)`
  - Adds a sprite to the scene, at the specified `x` and `y` coordinates
- `void removeSprite (PixelSprite sprite)`
  - Removes a sprite from the scene

Util

- This class cannot be instantiated. It contains utility functions used in every class
- `Label styleLabel(String fontFileName, double fontSize, Paint fontColor, Color dropShadowColor, boolean clickable, String text)`
  - Returns a label styled to the font, size, color, drop shadow, and text specified. If `clickable` is `true`, when the mouse is over it it will appear as if the label can be clicked. If `false`, the mouse will not change
- `Label styleLabel (String fontFileName, double fontSize, boolean clickable, String text)`
  - Returns the same label as the above function, but with the default text color of black and no drop shadow
- `LinearGradient linearGradient (boolean horizontal, Color... colors)`
  - Returns a linear gradient that can be used in place of a flat color in any function
  - `@param horizontal` whether the gradient is horizontal (true), or vertical (false)
  - `@param colors`: the colors in the gradient. Each is given equal weight.
- `RadialGradient radialGradient (double centerX, double centerY, Color... colors)`
  - Returns a radial gradient that can be used in place of a flat color in any function
  - `@param centerX`: the x coordinate, 0.0 to 1.0, that the gradient is centered
  - `@param centerY`: the y coordinate, 0.0 to 1.0, that the gradient is centered
  - `@param colors`: the colors in the gradient. Each is given equal weight.
- `playSfx (String gamePackageName, String sfxFileName, double volume)`

- Plays the specified sound effect. Should not be widely used, as this function is also included in the `GameRootPane` class

Advanced Functions (Do not use unless you know what you're doing)

- `Media getMusic (String gamePackageName, String musicFileName)`
  - Returns a Media of the specified music file
- `ImageView getImage (String gamePackageName, String imageFileName)`
  - Returns an ImageView from the specified image file.
- `Font getFont (String fontFileName, double size)`
  - Returns a Font from the specified font file.
- `Media getSfx (String gamePackageName, String sfxFileName)`
  - Returns a Media of the specified sound effect file.
- `double clamp(double value)`
  - Returns a double, clamped between 0.0 and 1.0