Overview

Assignment Goals

- Learn to instantiate and use methods from provided classes based on their documentation.
- Understand the difference between static and non-static class methods.
- Organize your own code into classes for modularity and code re-usability.
- Become more familiar with the architecture of real-time simulations and games.
- Gain experience using ArrayList collections of different object types.
- Create a basic computer game that can be expanded with many fun new features.

Description

In this programming assignment, you will implement a new game called Portal Snake. This game combines elements from the classic game of Snake (aka Nibble) with the modern puzzle game Portal. In this game, you maneuver a snake to collect apples that extend the length of its body. You win by collecting all of the apples on a given level. However colliding with any rocks or with your own body will end your game with a loss. The twist on this classic Snake gameplay, is that levels also contain pairs of portals (blue and orange) that teleport your snake from one part of the screen to another.

Program Details

Game Architecture

Modern computer games tend to be relatively complex programs. To help manage this complexity, their code is often divided between highly reusable engine code, and more game specific game logic code. Game engine code performs functionality that is used in almost every game: drawing text and graphics to the screen, detecting keyboard and mouse input events, and performing collision calculations between objects. Game logic on the other hand is typically only used in a single game: collecting $200 when passing go in Monopoly, collecting a power pill in PacMan, and hitting a mine to lose in Minesweeper. For this assignment, you will implement the game logic for PortalSnake. The game engine code however, is being provided via three classes: Application, Engine, and GraphicObject.

`Application` *class*

The main method in the skeleton Game class is already implemented to call the Application's static startEngineAndGame() method. This starts the game Engine, instantiates an instance of your Game class, then keeps both running through as many games as you are patient enough to play. The boolean parameter that is passed to this method controls whether the Engine will log messages to the console about GraphicObjects being created, destroyed, changing types, and colliding. You can turn this logging off by passing false instead of true to the startEngineAndGame() method.

`Engine` *class*

The Engine class is responsible for processing user input, moving around game objects, and drawing them all to the screen. However there are only four static methods in this class that you will need to access for this assignment:

- `public static int getWidth()`
- `public static int getHeight()`

The getWidth() and getHeight() methods return the size of the game window in pixels.

- `public static boolean isKeyPressed(String key)`
- `public static boolean isKeyHeld(String key)`

The isKeyPressed() and isKeyHeld() methods provide access to user input. The isKeyHeld() method returns true for as long as a key continues to be held down, however the isKeyPressed() only returns true only once per complete press and release of a particular key. You can check most of the keys on your keyboard by passing these methods a String containing the letter or number label of the key that you wish to check. You can also also pass this method the String names: TAB, ENTER, SHIFT, SPACE, LEFT, UP, RIGHT, or DOWN to check the status of keys that are not labeled with a single character.

`GraphicObject` *class*

The GraphicObject class represents a single graphical and spatial component within your game. You will need to create new instances of these objects with the public constructor:

- `public GraphicObject(String type, float x, float y)`

The type of each GraphicObject is a string containing either: HEAD, BODY, ROCK, DEAD, APPLE, BLUE_PORTAL_X, or ORANGE_PORTAL_X (where X can be substituted with any portal name). The type of each GraphicObject determines what it looks like when drawn by the Engine. This constructor also initializes the position of the game object in pixels relative to the upper-left corner of the game window. X positions go from zero along the left edge of the window to Engine.getWidth() along the right edge, and Y positions go from zero along the top edge of the window, to Engine.getHeight() along the bottom edge.

- `public void destroy()`

When you are done using a GraphicObject, you must call it's destroy() method to remove it from the Engine. This stops it from being moved or drawn, and prevents it from colliding with any other GraphicObjects in the future.

- `public String getType()`
- `public void setType(String type)`
- `public float getX()`
- `public void setX(float x)`
- `public float getY()`
- `public void setY(float y)`
- `public float getSize()`
- `public void setSize(float size)`
- `public float getDirection()`
- `public void setDirection(float direction)`
- `public float getSpeed()`
- `public void setSpeed(float speed)`

- `public GraphicObject getLeader()`
- `public void setLeader(GraphicObject leader)`
- `public GraphicObject getFollower()`
- `public void setFollower(GraphicObject follower)`

These are some of the GraphicObject classes accessors and mutators. The type, x, and y properties were described above. Each GraphicObject also has a size, which is the diameter of an object: used internally for collision detection. The speed and direction can be used to set a GraphicObject in motion. The speed is in pixels per update (so start with small values <10), and the direction an angle measure in degrees where 0=right, 90=up, 180=left, 270=down. And finally, the leader and follower properties are used cause one GraphicObject to follow another. Every GraphicObject can have at most one leader (which they follow) and one follower (which follows them). To establish a following relationship between two GraphicObjects you can call either the setLeader() method of one GameObect or the setFollower() method of the other (calling both is redundant).

- `public void moveTo(float x, float y)`
- `public void movePast(GraphicObject dest)`
- `public boolean isCollidingWith(GraphicObject other)`

These last three methods in the GraphicObject class will also be useful in completing this assignment. The moveTo() method moves a GraphicObject to a specific window position. The movePast() method will move a GraphicObject right next to another GraphicObject, so that it is near without colliding into that destination GraphicObject. The moving object's direction property determines which side of the destination GraphicObject it is moved to. For example, if an object's direction is zero, then it will be moved just to the right of the destination GraphicObject. The final method isCollidingWith() is used to determine whether a GraphicObject is colliding with another specific GraphicObject. This method returns true when the two objects are colliding, otherwise it returns false.

The Rules of the Game

`Game` *class*

All of your high level game logic will be organized in the Game class. The provided Application class will create an instance of your game class using the following constructor. And your implementation of this constructor will call either the createRandomLevel() or loadLevel() method depending on the value of the level argument.

- `public Game(String level, int controlType)`
- `public void createRandomLevel()`
- `public void loadLevel(String level)`

Upon instantiation, your Game receives two arguments from the Application: controlType and level. The controlType argument is passed to your Snake to establish how it is maneuvered by the player. The level String contains either the word RANDOM indicating that you should create a random level, or the contents of an entire level file that has been chosen by the player. The createRandomLevel() method should create and randomly position: 20 Rocks, 8 Apples, and 3 PortalPairs. This method should also create ***1 Snake in the center of the screen***. The loadLevel() method is a bit more complex.

The level string that is passed to both the Game's constructor and from there to the loadLevel() method

contains the names and locations of every object that needs to be created at the beginning of a new game. In order to read level data out of this level string, you will instantiate a new Scanner and pass the level string to its constructor. Each object in this level string is on a separate line, and the first thing on each lines is the name of the object's type: Snake, Rock, Apple, or PortalPair. If you find any thing other than one of these names at the beginning of any line, you can ignore that line as either comments or placeholders for future game expansions. For snakes, rocks, and apples: after the name of the object type you will find the x and then y positions that those objects should be created at. For portal-pairs, the name of the object-type is followed by the name labeling that pair of portals, the x and y position of the first portal, and then the x and y position of the second portal. Each of these elements are separated by commas, and they are all contained within a single line. Here is an example of the milestone2 level string:

```
Snake,400,300
Apple,100,100
Rock,700,300
PortalPair,A,400,200,400,500
```

Once a level has been created or loaded, the Game class's update() method is responsible for keeping all of the objects up to date and for enforcing the rules of your game. As long as your Game's update() method returns true, ***the Application class will continue to repeatedly call your Game classes update()*** many times per second. After each of these update() calls, the Application will instruct the Engine to process new user input, to move GraphicObjects according to their current speed and direction, and to draw all GraphicObjects in their new locations. After the player has won or lost the game, this method should return false to let the Application know that the game has ended.

- `public boolean update()`
- `public boolean checkWonNotLost()`
- `public int getScore()`

In addition to implementing update, you will need to implement the checkWonNotLost() method to return true after the player has won the game, and false after the player has lost the game. The Application will use this to show an appropriate message after each win or loss. The getScore() method should return the number of apples eaten by the snake at any time. This is also used by the Application to display the players score.

`Snake` *class*

The Snake class represents the character that is controlled by the player in this game. Really it is only the head of this snake that is directly controlled by the player, and each body segment is set to follow either the head or body segment that is directly in from of them.

- `public Snake(float x, float y)`
- `public void grow()`

Whenever a Snake is instantiated at the specified position x and y, it's head should be given a starting speed of 2 in the direction 90 (up). In addition to this head, your snake should start each game with four body segments. Each time your Snake's grow() method is called, one additional body segment should be added to your snake. Remember that each segment follows the one in front of it, except for the first body segment which follows the head.

- `public void updateMoveDirection(int controlType)`

There are three different control types that can be selected by the player and then passed from the Application, to the Game, and ultimately to your Snake via this controlType parameter. For each control type, remember that every snake is always created with speed = 2 in direction = 90 (up).

When controlType is 1, Classic Controls: Each time the left-arrow key is pressed, the snake head's movement direction should be incremented by 90 degrees. Each time the right-arrow key is pressed, the snake head's direction should be decremented by 90 degrees.

When controlType is 2, Analog Controls: For as long as the left-arrow key is held, the snake head's movement direction should be incremented 6 degrees per update. For as long as the right-arrow key is held, the snake head's direction should be decremented 6 degrees per update.

When controlType is 3, Slither Controls: For as long as the spacebar key is held, the snake head's movement direction should be increased 6 degrees per update. Whenever the spacebar key is not held, the snake head's direction should instead be decreased 6 degrees per update.

- `public GraphicObject getHeadGraphicObject()`
- `public void dieIfCollidingWithOwnBody()`
- `public void die()`
- `public boolean isDead()`

These last methods in the Snake class are used to help detect and enforce what happens when the snake collides with other objects. The getHeadGraphicObject() method will allow other objects like the Rock, Apple, and PortalPair to check for collisions between their own GraphicObjects and the Snake's head. However it will be the task of your own snake to both detect and handle collisions between its own head and body via the dieWhenCollidingWithOwnBody() method. The die() method should change the type for all of the snake's GraphicObjects (including the head and all body objects) to DEAD. And the isDead() method should return true after the snake has died, and false beforehand.

`Rock` *class*

The Rock is a relatively simple class. In addition to its constructor, it has the single method killSnakeIfColliding(). As it's name implies, this method should check for collisions between itself and the snake's head. Whenever such a collision is found, the snake should die as a result.

- `public Rock(float x, float y)`
- `public void killSnakeIfColliding(Snake snake)`

`Apple` *class*

The Apple class is only slightly more complex than the Rock class. Instead of simply killing the snake whenever there is a collision, the Apple should remove itself from the game and cause the snake's body to grow. Whenever an apple is eaten in this way, the getEatenBySnakeIfColliding() method should return true to let the Game know that this apple has been eaten and that the player has earned an extra point.

- `public Apple(float x, float y)`
- `public boolean getEatenBySnakeIfColliding(Snake snake)`

`PortalPair` *class*

Whenever the snake's head collides with either end of a PortalPair, they should be instantly transported to past the other end of that PortalPair. They should be moved past rather than on top of this portal so they do not get stuck in an infinite loop of being teleported back and forth forever. The name of each PortalPair can be displayed by appending this name to the end of a graphic object's portal type. For example the portal name "A" can be displayed on a portal by giving it's graphic objects the types: "ORANGE_PORTAL_A", and "BLUE_PORTAL_A".

- `public PortalPair(String name, float blueX, float blueY, float orangeX, float orangeY)`
- `public void teleportSnakeIfColliding(Snake snake)`

Getting Started

1. Begin by reading through the entire project write-up. This will help you understand all that is expected. Attention to detail is very important, so it will definitely pay off to have a solid familiarity with the project write-up.
2. When ready to start working on Milestone 0, create a new project in Eclipse. The name of the project does not matter, but PortalSnake would be a sensible choice.
3. In order to complete this assignment you will need to download the P3.jar library. From Eclipse, you can import this file into your project as a FileSystem. After doing so, you should add this P3.jar file to your build path by right-clicking this file in the PackageExplorer view, and then choosing "Add to BuildPath" from the BuildPath submenu.
4. Next download our Game.java, Snake.java, Rock.java, Apple.java, and PortalPair.java files. After adding these files your new project with the P3.jar file, you should be able to build and run the project from the main() method that is found in the Game class. Upon running this code, you should see a level select menu that will bring you to a blank screen reporting Score: -1 for any level.
5. You will be implementing and submitting these five files for this assignment: Game.java, Snake.java, Rock.java, Apple.java, and PortalPair.java. You must implement all of the methods described in this write-up and use these classes and methods to implement the PortalSnake game. You many not add any additional class files to your implementation.
6. In each of these files, add file headers, class headers, and method headers as described in the CS302 Commenting Guide.
7. Begin programming! And remember to incrementally develop, write code, run and test small sets of modifications as you go. And be sure to check out the Tip Section.

Development Milestones

We suggest that you incrementally develop this program by following these milestones in the order listed below. This order will make it easier for you to check that each milestone is working properly before you move on to the next one. Don't move on to a new milestone until you can convince ourself that the current milestone is working correctly. The further along you get, the harder it will be to detect errors in code you developed earlier.

- ***Milestone 0 (Project Setup, and adding random Rocks by ???)***

After creating a new project with the files listed above, ensure that you can compile and run the provided code, before making any modifications. To become more familiar with creating and using

simple GraphicObject's, start by implementing the constructor in the Rock class. It should create a new GraphicObject with the type ROCK in the designated position. To ensure that this works, goto the Game class and have the Game's constructor immediately call createRandomLevel(). Then implement the createRandomLevel() method to create and randomly position 20 new rocks. When you run the game, and select the RANDOM level, you should see these 20 rocks randomly scattered around.

- ### *Milestone 1 (A Moving Snake)*

Start implementing the Snake's constructor similar to the Rock's to simply create a GraphicObject with the type HEAD, and have the Game class create a single snake from the createRandomLevel() method. Once this is working, you can give the Snake its initial movement speed and direction. Once the snake's head is moving correctly, you can implement the grow() method and use it to add length to your snake. The final step in this milestone is to implement the player's controls for maneuvering the snake. Start by implementing the classic controls (controlType==1), then the analog controls (controlType==2), and then the slither controls (controlType==3). ***In order for these controls to work, your game will have to call your snake's updateMoveDirection() method, every time the Application calls your Game's update() method.***

- ### *Milestone 2 (Snake Interactions: Self, Rocks, and Portals by ???)*

Next implement the PortalPair classes constructor, and create three randomly positioned PortalPairs from the Game's createRandomLevels() method. Now that you have randomly positioned rocks, portals, and a moving snake, it is time to start enforcing some of the rules for collision responses. Implement the snake's die() and isDead() methods, along with the dieIfCollidingWithOwnBody() method. Remember that this method will need to be called every time the Game's update() method is called. After you get this working, implement the Snake's getHeadGraphicObject() method, so that you can then focus on getting the Rock class's killSnakeIfColliding() method working. Once this is working, the last method to implement for this milestone is the PortalPair's teleportSnakeIfColliding() method.

- ### *Milestone 3 (Random Apples, Score, and Game Win/Lose by ???)*

Now you can implement the Apple class. In addition to implementing this class, you'll need the Game class to create several randomly positioned instances of it, and then to repeatedly call the getEatenIfCollidingWithSnake() method. You'll also need to keep track of the number of apples eaten, and use this number to determine when the player has won. ***Be sure that the game ends whether the player wins or loses, and that the Application class is is correctly informed of each.***

- ### *Milestone 4 (Loading Custom Levels by ???)*

As a final step, you will implement the Game class's loadLevel() method. To test this functionality, you will need to create a new folder within your project named levels. You can add the following levels to this folder for testing, and create more of your own with the editor described below. As an added bonus, you can also create another separate folder called images and add the following images to it. These images will then be used to draw your GraphicObjects. After performing enough tests to assure yourself that your code is functioning correctly, remove any test code that does not contribute to the assigned computations of this assignment. Review your code to ensure that you have completely and correctly implemented all of the provided method stubs and sufficiently commented your code.

Submitting Your Work

Submit these source code files only: (be sure that you are not trying to submit any .class files)
Game.java, Snake.java, Apple.java, Rock.java, PortalPair.java

[use similar verbiage from past assignments to complete this section]

Developing your Own Levels

The P3.jar file that you have been using up to this point doubles as a level editor when you double click on it. Once running, you can create and position new game objects under your mouse by pressing s-snake, a-apple, r-rock, or p-portalpair. Once created you can drag objects to new positions with the left mouse button, and delete them with the right mouse button. After your level is arranged as your liking, you can press enter to save a new level file. Rename this level file and move it into your project's levels folder to test, play, and enjoy your new creation.