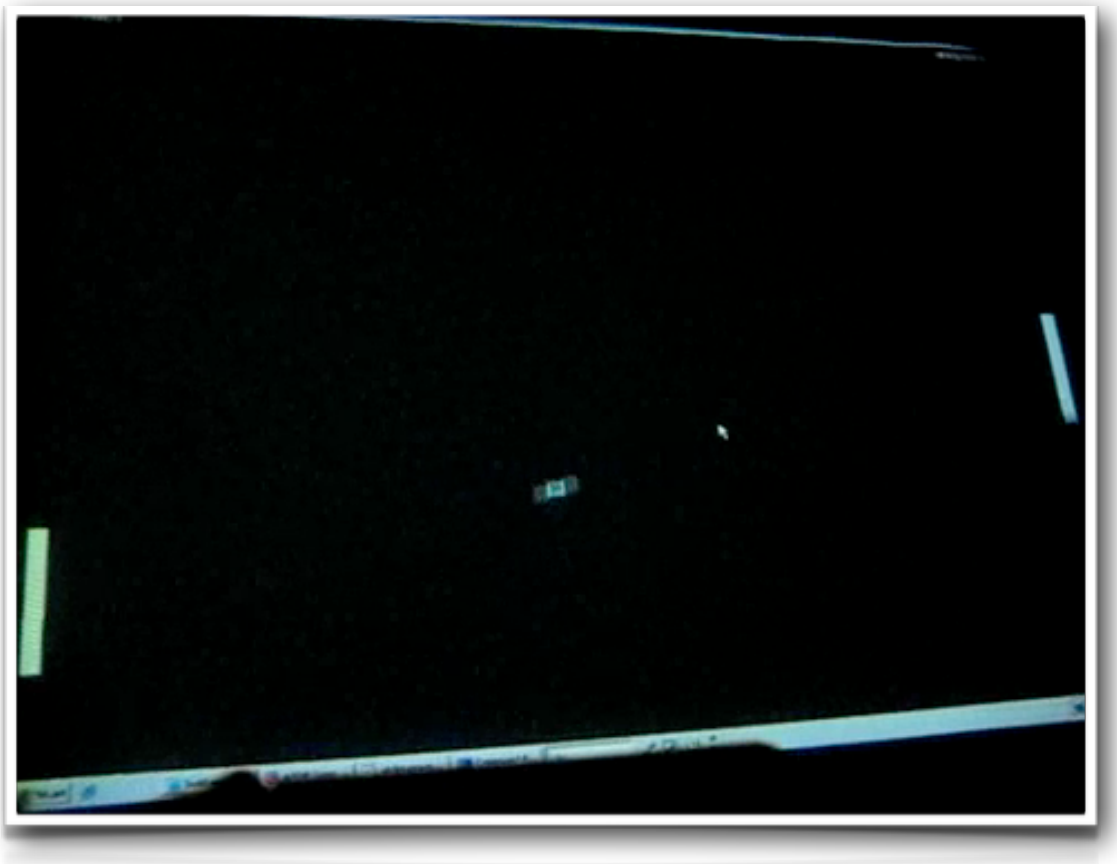
PROJECT #1: PONG



DOCUMENTATION

Instructor: Jonathan Berney • telephone: 510.461.1610 • email: jberney@gmail.com • java.webhop.org

Table of Contents

I. Introduction 1

Description 1

History 1

Interesting Links 1

II. Application Programming Interface (API) 2

The Pong API 2

The "game" Package And Associated Subpackages 2

The "game.pong" Package And Associated Subpackages 5

III. Downloading and Running Pong 7

Downloading Pong 7

Running Pong with Default Input 7

Running Pong with Custom Input 8

IV. The Assignment 9

Project #1: Due Monday, July 20, 2009 @ 12:00am (Midnight) 9

Victory against the Wobbly PaddleLogic 9

In-Class Pong Tournament on July 20th 9

V. Tutorial #1: DoNothing 10

VI. Tutorial #2: Center 11

VII. Tutorial #3: FollowHeight 12

VIII. Tutorial #4: Lazy 13

IX. Tutorial #5: Wobbly 15

I. Introduction

Description

Pong is a 2D sports game which simulates table tennis. Each of two players control a paddle by moving it vertically. The paddles are used to hit a ball back and forth. The goal of the game is to earn more points than your opponent. Points are earned when the balls gets by your opponent's paddle.

History

Pong is one of the earliest arcade video games. It was created by Allan Alcon, and originally manufactured by Atari in 1972. It quickly became a success and was the first commercially successful video game.

Interesting Links

- Wikipedia entry on Pong: http://en.wikipedia.org/wiki/Pong
- Full-sized Air Hockey Robot: http://www.youtube.com/watch?v=oNEjtVUxyX4&fmt=18
- Intro to Java 08 Pong tournament video: http://www.youtube.com/watch?v=ES6nCUDklys&fmt=18
- xkcd.com comic about Pong and The Matrix film: http://xkcd.com/117/

II. Application Programming Interface (API)

The Pong API

Included in the Pong project zip archive is a set of automatically-generated HTML files which document the available classes, fields, constructors, and methods provided by the Pong game. This section will highlight the most interesting fields and methods of each of the included classes.

The "game" Package And Associated Subpackages

I. Package: game

A. Class: GameObject

The GameObject class is a general representation of a game which consists of playable and non-playable characters. It is responsible for evaluating the logic of playable and non-playable characters, as well as drawing the state of the game to the screen. While the NPCs and PCs in a GameObject are not directly accessible outside of the class itself, there are public methods getNPCs() and getPCs() available when information about the players in the game is required.

1. Fields
    a) protected NPC[] npcs
    b) protected PC[] pcs
2. Methods
    a) public NPC[] **getNPCs**()
    b) public PC[] **getPCs**()

II. Package: game.character

A. Class: NPC

An NPC is a non-playable character. This is any character in a game that is not controlled by player-supplied logic. It has two fields, HEIGHT_SCALE and WIDTH_SCALE, which store the bounding box information about the character which can be used to determine when collisions will occur.

The NPC class also manages the position and velocity (speed) of the character. These fields are not accessible outside of the class and its subclasses, but the getPosition() and getVelocity() methods can provide read access.

Additionally, the NPC class contains methods which are called by the GameObject when it is time to update the state information of the NPC, and when it is time to actually draw the NPC to the screen. These methods will **not** need to be called by students working on the assignment associated with the Pong game.

1. Fields
    a) public static final double **HEIGHT_SCALE**
    b) public static final double **WIDTH_SCALE**
    c) protected double[] position
    d) protected double[] velocity
2. Methods
    a) public double[] **getPosition**()
    b) public double[] **getVelocity**()
    c) public void update()
    d) public void draw()

B. Class: PC *extends NPC*

The PC (PlayableCharacter) class extends the NPC class, and thus inherits all of its protected and public fields and methods.

Additionally, two new fields are provided: one which holds a PlayerLogic object which will determine how the PlaylableCharacter will act given the state of the game, and the other which keeps track of the PlayableCharacter's score.

1. Fields
    a) protected PlayerLogic logic
    b) protected int score

III. Package: game.logic

A. Class: PlayerLogic

The PlayerLogic class is meant to be extended in order to provide instructions to PlayableCharacters. It is constructed with a playerIndex, which can be used to identify the associated PlayableCharacter.
It has one method, called evaluate(). This method receives the present state of the game. The content of this method dictates how the PlayableCharacter should react to this state.

1. Fields
    a) protected int **playerIndex**
2. Constructors
    a) PlayerLogic(int playerIndex)
3. Methods
    a) public void **evaluate**(GameObject game)

The "game.pong" Package And Associated Subpackages

I. Package: game.pong

A. Class: PongGame *extends GameObject*

The PongGame class inherits much of its functionality from the GameObject class. It uses its fields and methods to specifically implement a game of Pong.

For this game, this field of play is defined as follows:

- The top-left of the playing field is the origin, and is position: x = 0.0, y = 0.0
- The bottom-right of the playing field is position: x = 1.0, y = 1.0

II. Package: game.pong.character

A. Class: Ball *extends NPC*

The Ball class represents the Pong ball. It extends the NPC class and inherits the position and velocity properties, as well as the associated accessor methods (getVelcoity(), getPosition()). It provides values for the inherited constants WIDTH_SCALE and HEIGHT_SCALE. It also provides a constant called INITIAL_VELOCITY, which is the speed the ball moves at the start of a new round. The speed of ball increases multiplicatively each time it is hit by a paddle.

1. Fields

    a) public static final double **INITIAL_VELOCITY**

B. Class: Paddle *extends PC*

The Paddle class represents a Pong paddle. It extends the PC class and inherits all of its fields and methods, including the logic field.

The most notable extension of the PC class is the MAX_VELOCITY property, which limits how far a paddle can move per unit of time.

To make for more interesting logic strategies, collisions with the ball work as follows:

- **If the ball strikes the paddle at its midpoint, it will be deflected with entirely horizontal velocity.**
- **If the ball strikes the paddle above its midpoint, it will be deflected at an upward angle, with a maximum of 45 degrees when the ball hits the very top tip of the paddle.**
- **If the ball strikes the paddle below its midpoint, it will be deflected at a downward angle, with a maximum of 45 degrees when the balls the very bottom tip of the paddle.**

1. Fields

    a) public static final double **MAX_VELOCITY**

III. Package: game.pong.logic

A. Class: PaddleLogic *extends PlayerLogic*

The PaddleLogic class extends the PlayerLogic class. It inherits the playerIndex field and the evaluate() method.

Additionally, it provides for convenience the index of enemy Paddle. It also has a property of type double called target. The target determines to where the associated Paddle will move as fast as it is able. Once the Paddle has reached its target, it will stay there until the value of the target changes. The value of the target is meant to be modified in the evaluate() method based on an evaluation of the current state of the PongGame.

1. Fields

    a) protected int **enemyIndex**

    b) protected double **target**

III. Downloading and Running Pong

Downloading Pong

The Pong project zip archive is available on the course website at http://java.webhop.org/. Download the zip file, and unzip it to your disk in order to access the project files.

Running Pong with Default Input

Included within the project files are build scripts for Microsoft Windows and Mac OS X. It is required that the Java Development Kit (JDK) is installed in order to build and run your work. If this is not yet installed, please refer to the instructions provided at http://java.webhop.org/jdk/.

For Windows, run the batch file by double-clicking build-win32.bat.

For Mac OS X, run the shell script by double-clicking build-macosx.sh. Your Finder will need to be setup in order to run .sh files with the Terminal Application. If you double-click the shell script and it opens in TextEdit, do the following:

1. Select the build-macosx.sh file in the Finder.
2. Get Info by going to the File Menu and selecting Get Info (or hold the Command key and type 'I').
3. Locate the section called "Open with:". If the section is hidden, click the arrow to the left of the text.
4. In the drop-down menu, select the "Other…" option.
5. Locate Terminal.app. This can be done either by manually navigating to /Applications/Utilities/Terminal.app, or using the Spotlight functionality in the file navigation window to search for the app.
6. In order to select Terminal.app, you may need to select "All Applications" in the Enable field.
7. Click the "Add" button.
8. In the "Open with:" section of the Get Info window, click the "Change All" button.

It should now be possible to simply double-click shell files from the Finder in order to build and run your project.

By default, the Paddle on the left side will be controlled by a built-in logic class called Wobbly. The Paddle on the right is controllable with keyboard input. The up and down arrows make the Paddle move accordingly.

Running Pong with Custom Input

Once you have begun to write your own PaddleLogic class(es), it will be necessary to edit your build script in order to build and run your work.
Open your build script in a text editor (Notepad.exe for Windows, TextEdit for Mac).

There are two important lines: the line that begins with javac (this compiles your work), and the line that begins with java (this runs the game).

The first line is commented by default. Uncomment it by removing the first character of the line (a colon for the Windows build script, and a pound sign for the Mac build script).

Next, at the very end of the line is a piece of text that reads custom/jberney/*java. This tells the compiler to compile all java files in the custom/jberney folder. Your package name will be something other than jberney -- change it to whatever you would like. Keep in mind the naming conventions for packages: keep them short, and all lowercase. You will place your PaddleLogic extensions in this location.

The second line is uncommented, and runs the program. The last two parts of this line specify what PaddleLogic to load for the left and right sides respectively. By default, the values are Wobbly and Keyboard. If the second value is omitted, it is filled in by Wobbly automatically. These are the only two values for which you do not need to write the entire package path.

If my package name was jberney, and I had a PaddleLogic extension class called FollowHeight (located at custom/jberney/FollowHeight.java), I might replace one of these two symbols with “jberney.FollowHeight” in order to make my PaddleLogic class control one of the Paddles.

IV. The Assignment

Project #1: Due Monday, July 20, 2009 @ 12:00am (Midnight)

You will extend the PaddleLogic class to create a class which evaluates the state of a Pong game and makes strategic choices in order to defeat an opponent PaddleLogic.

Victory against the Wobbly PaddleLogic

In order to receive an **A** on this assignment, you must design an extension of the PaddleLogic class that **invariably defeats the Wobbly PaddleLogic class**. This will be accomplished by implementing an evaluate() method which receives the state of the PongGame, and makes strategic decisions about how to get the ball past your opponent.

Submissions which are only able to defeat Wobbly **on occasion** will receive a lower grade depending on the rate of success.

Submissions which **never** defeat Wobbly will receive a **failing** grade.

No credit will be given for late work, so please complete your work on time.

In class we will discuss some strategies and some implementation details of these strategies, but this project is very much about each student's own individual creativity. While we will discuss some possible solutions to this interesting exercise, students should in no way feel constrained to follow all of the guidelines given in the tutorials in class. This is a fairly open-ended problem with many possible solutions, and it should certainly be interesting to see how we each approach the problem!

In-Class Pong Tournament on July 20th

After all of the submissions from the students are in, we will hold an elimination-style tournament, pitting the students' bots up against one another. The winner will receive an automatic A+ on the assignment, and will get to see his or her bot face off against those designed by the TAs, Instructor, and anyone else who has made a solution of their own, including Intro to Java 08 Alumni and other interested friends of the class.

V. Tutorial #1: DoNothing

As a first exercise, we will write an extension of the PaddleLogic class which does nothing in its evaluate() method. By default, the PaddleLogic's target property is 0. As a result, the Paddle will move to position zero (the top of the screen) and stay there forever, usually letting the ball by. This class will lose nearly always, but is a good way to get started nevertheless.

First, we must define the package name. My package name will be jberney, yours should be something else:

```
package jberney;
```

Next, we import some important classes. The GameObject class is needed so that we can refer to it while evaluating its state, and the PaddleLogic class is needed so that we can extend it:

```
import edu.berkeley.atdp.java.game.GameObject;
import edu.berkeley.atdp.java.game.pong.logic.PaddleLogic;
```

Now, we begin our class definition. We will call this class DoNothing, and state that it extends the PaddleLogic class:

```
public class DoNothing extends PaddleLogic {
```

Next, we will write the class's constructor. It takes the index of the player as an argument. We can pass this to the superclass (PaddleLogic) with the super() constructor:

```
    public DoNothing(int playerIndex) {
        super(playerIndex);
    }
```

Finally, we write our very simple implementation of the evaluate() method and end our class:

```
    public void evaluate(GameObject game) {
        // Going up!
    }

}
```

Save this file to custom/yourpackagename/DoNothing.java, make the required changes to your build script, and try it out!

VI. Tutorial #2: Center

For our next exercise, we will write a very simple bot which sets it target to the center of the screen.

Begin by declaring the **package name** and **imports** as we did in the previous tutorial.
Now, we begin our class definition. We will call this class Center:

```
public class Center extends PaddleLogic {
```

The constructor should look fairly similar to the last exercise:

```
    public Center(int playerIndex) {
        super(playerIndex);
    }
```

Now we make our target the center of the screen and we are done:

```
    public void evaluate(GameObject game) {
        this.target = 0.5;
    }

}
```

Our Center class extends the PaddleLogic class. Even though we have no declared an instance variable called target, it has been inherited from the PaddleLogic class, and we can use it as though it were our own.

Save and run this bot. It should stay in the center, unlike the DoNothing class.

VII. Tutorial #3: FollowHeight

For our next exercise, we will learn how to access information from the GameObject so that we can make some decisions about how to act.

From this point on, it is assumed that the structure of these classes is clear, **and we will only discuss the content of the evaluate() method**. Note that any external classes we refer to will need to be imported. Refer to the JavaDoc API documentation for class paths. For instance, for this class we will need to import the NPC class. This is accomplished with the following statement:

```
import edu.berkeley.atdp.java.game.character.NPC;
```

For this bot, we will be looking into the provided GameObject in order to determine the current height (y-position) of the ball. We will simply make this value our target. The result is that we will line our Paddle up with the ball to the best of our ability at all times.

First, we declare a new NPC array called npcs, and initialize it with the array of NPCs held by the GameObject:

```
        NPC[] npcs = game.getNPCs();
```

We know that there is only one object in the npcs array (the Ball), so we can create an NPC object and grab it from the array. We can call it anything we like, but we might as well be descriptive and call it ball:

```
        NPC ball = npcs[0];
```

We can extra the position of the ball by calling its getPosition() method as follows:

```
        double[] ballPosition = ball.getPosition();
```

The position array stores the x position at index 0 and the y position at index 1. Since we are only interesting in the height of the Ball, we can extract the value in the array at index 1 and store it in a new double variable called ballY:

```
        double ballY = ballPosition[1];
```

We end our evaluate() method by setting our target to be equal to the y-position of the Ball:

```
        this.target = ballY;
```

This is a very simple bot which just follows the height of the ball. It rarely moves at its maximum velocity before it is defeated, and this is arguably its greatest weakness. It moves only as fast as is required to keep its center aligned with the y-position of the ball.

VIII. Tutorial #4: Lazy

The Lazy bot will build on FollowHeight. We will see how to get information about our own position. We will also see some ways in which the logic branching (i.e. "if" statements) we've learned about may work its way into our solutions.

Begin the evaluate() method as we did in FollowHeight in order to determine the height of the ball. We will use a similar strategy to find and store our own Paddle's height (measured as the position of the vertical center of our Paddle). We grab the GameObject's PCs and store them in an array of PCs. We grab the PC at position "playerIndex" in the PC array to get the object representing our Paddle. Then we find our Paddle's position (and y-position) as we did for the Ball:

```
        PC[] pcs = game.getPCs();
        PC player = pcs[playerIndex];
        double[] playerPosition = player.getPosition();
        double playerY = playerPosition[1];
```

The strategy we would like to implement at this point is the following:
if (we will hit the ball from where we are currently) then do not move
else follow the height of the ball

Clearly this can be accomplished with an if statement. The question is, how do we form the condition in Java code?

We need to figure out how to say the boolean statement ("we will hit the ball from where we are currently") in Java. We will have to represent this mathematically.

If the absolute distance between the Ball and the center of our Paddle is less than the closest the two objects can be without a collision, then we will hit the ball from where we are currently. Let's represent each of these values mathematically, one at a time.

First, the distance between the Ball and the center of our Paddle is simply the absolute value of the difference between their y-positions. We can write this as:

```
        double distanceToBall = Math.abs(ballY - playerY);
```

The maximum separation between the Paddle and Ball where they will just barely not collide is half the height of the Paddle plus half the height of the Ball. Try to prove this to yourself if you do not see why this is, or ask a TA or the instructor if this is still not clear. This can be written in code as:

```
        double maxDistance = (Ball.HEIGHT_SCALE + Paddle.HEIGHT_SCALE) / 2.0;
```

Now we must check if the distance to the Ball is less than the maxDistance variable. If it is, we would like to stay where we are, so we just set our target to be our current position:

```
        if(distanceToBall < maxDistance) {
            this.target = playerY;
        }
```

If we are not currently in a spot where we are going to hit the ball, we follow the old FollowHeight logic in order to get ourselves there:

```
        else {
            this.target = ballY;
        }
```

Whether this bot performs better than FollowHeight is not immediately clear. But we have learned several important things in this tutorial, such as how to get information about our own Paddle and how to access and use the HEIGHT_SCALE properties of Balls and Paddles. Most importantly, we have seen how to formulate a strategy, represent it with mathematics, booleans, and logic branches, and write a Java implementation of our strategy.

IX. Tutorial #5: Wobbly

Sometimes simple randomness is a good way to make a bot more interesting. We can achieve some level of pseudo-randomness with the Math class's random() method. When we call Math.random(), we receive a random double value between 0 and 1.

The Wobbly class is a very simple addition to the logic of FollowHeight. After we set the value of this.target, we add to it a random double value between -0.1 and +0.1. We can accomplish this through an algebraic scaling of the output from Math.random():

```
        this.target += (2 * Math.random() - 1) / 10.0;
```

Originally the range of values is from 0 to 1. We multiply by 2 to increase the range to 0 to 2. Then we subtract 1 to center our values between -1 and +1. Finally, we divide by 10 in order to make the range -0.1 to +0.1.

The end result is a constant jitter in the position of our Paddle, which leads to some unpredictable hits. This can throw many of the bots we've constructed so far for a bit of a loop!