/\*\*

\* File: Angelfish.java

\*

\* Description: Angelfish swim across the aquarium always moving at a slight

\* angle either upward or downward. When they meet another creature, they

\* reverse direction.

\*

\* @author Randall.Bower

\* @author Gavin Delphia

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Angelfish

{

/\*\* Image representing this angelfish to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this angelfish. \*/

private double x;

/\*\* Current y position of this angelfish. \*/

private double y;

/\*\* Current z position of this angelfish. \*/

private double z;

/\*\*

\* The scale of an angelfish is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the angelfish and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this angelfish is moving; positive values indicate angelfish is

\* moving left-to-right, negative values indicate angelfish is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\*

\* Change in position for swimming at an angle.

\*/

private double delta;

/\*\*

\* This constructor creates a new Angelfish with the given image. It is

\* not possible or desirable to create an Angelfish without an image, so

\* there is no zero-parameter constructor. Angelfish properties are set

\* to random values within the bounds of the aquarium and the delta, or

\* vertical speed, is set to a small value to give a slight angle to

\* the Anglefish's movement.

\*

\* @param image BufferedImage object representing this angelfish.

\*/

public Angelfish( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some angelfish

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All angelfish are in the top 3/4 of tank. Subtract this

// height to ensure the bottom of the angelfish doesn't go below the

// upper two-thirds of the aquarium. Use Math.max to ensure angelfish

// is not off the top of the aquarium after subtracting the height.

this.setY( Math.max( Math.random() \* Aquarium.HEIGHT \* 0.75

- this.getHeight(), 0 ) );

// Start swimming horizontally.

this.delta = 0.0;

// Random value between 0.25 and 0.5.

this.setDelta( Math.random() \* 0.25 + 0.25 );

// With a probability of 0.5, invert delta.

if( Math.random() < 0.5 )

{

this.setDelta( this.getDelta() \* -1.0 );

}

}

/\*\*

\* When an Anglefish meets another creature, it reverses direction.

\*/

public void meet()

{

// Inverting the speed reverses the direction.

this.setSpeed( this.getSpeed() \* -1.0 );

}

/\*\*

\* Angelfish move both horizontally and vertically and turn around

\* when they reach an edge of the aquarium.

\*/

public void move()

{

// Move the swimmer horizontally, based on its speed.

this.setX( this.getX() + this.getSpeed() );

// If the angelfish has moved out of the aquarium's window, it turns

// around and re-enters either at a different depth, front to back.

if( this.getX() < -this.getWidth() || this.getX() > Aquarium.WIDTH )

{

// Inverting the speed turns the swimmer around.

this.setSpeed( this.getSpeed() \* -1 );

// The change in depth will be between 1 and 2 "thicknesses" of

// the angelfish. See comment in Aquarium about depth.

double dz = this.getDepth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move toward the front, but no more than the depth of the aquarium.

this.setZ( Math.min( this.getZ() + dz, Aquarium.DEPTH ) );

}

else

{

// Move toward the back, but no less than the back of the aquarium.

this.setZ( Math.max( this.getZ() - dz, 0 ) );

}

}

// If angelfish is at the top and moving up or at the bottom and

// moving down, invert delta before moving y.

if( this.getY() < 0 && this.getDelta() < 0 ||

this.getY() + this.getHeight() > Aquarium.HEIGHT \* 0.75 )

{

this.setDelta( this.getDelta() \* -1.0 );

}

// Now go ahead and move the y position.

this.setY( this.getY() + this.getDelta() );

}

/\*\*

\* This method paints an angelfish using the given Graphics object and

\* the angelfish's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this angelfish.

\*/

public void paint( Graphics g )

{

// An angelfish's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the angelfish is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the angelfish is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this angelfish, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Angelfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this angelfish.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* Sets the y position of this angelfish, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Angelfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param y New y position for this angelfish.

\*/

public void setY( double y )

{

if( y < -Aquarium.HEIGHT || y > Aquarium.HEIGHT \* 2 )

{

throw new IllegalArgumentException( "y is out of range: " + y );

}

this.y = y;

}

/\*\*

\* Sets the y position of this angelfish, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Angelfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium an angelfish

\* is, which determines the scale to be used when drawing the

\* angelfish image.

\*

\* @param z New z position for this angelfish.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

}

/\*\*

\* Sets the speed of this angelfish, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this angelfish.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Sets the delta, or vertical speed, of this angelfish.

\*

\* @param delta New delta, or vertical speed, for this angelfish.

\*/

public void setDelta( double delta )

{

this.delta = delta;

}

/\*\*

\* Accessor method for the x position of this angelfish. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this angelfish.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this angelfish. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this angelfish.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this angelfish. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this angelfish.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this angelfish. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this angelfish.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this angelfish. This value is based

\* on the z position of the angelfish and will be between 0.0 and 1.0.

\*

\* @return The current scale of this angelfish.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the delta, or vertical speed, of this angelfish.

\*

\* @return The current delta, or vertical speed, of this angelfish.

\*/

public double getDelta()

{

return this.delta;

}

/\*\*

\* Accessor method for the width of this angelfish. It is determined by

\* the current scale of the angelfish and the width of the angelfish image.

\*

\* @return The current width of this angelfish.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this angelfish. It is determined by

\* the current scale of the angelfish and the height of the angelfish image.

\*

\* @return The current height of this angelfish.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this angelfish. It is determined by

\* the current scale of the angelfish and is somewhat arbitrarily set

\* to one-half of the height of the height of the angelfish.

\*

\* NOTE: The depth of an angelfish is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this angelfish.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this angelfish that

\* includes the (x,y,z) position and the scale and speed of the angelfish.

\*

\* @return A string representation of this angelfish.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Bubble.java

\*

\* Description: Bubbles exist in the aquarium and float straight up.

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Bubble

{

/\*\* Image representing this bubble to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this bubble. \*/

private double x;

/\*\* Current y position of this bubble. \*/

private double y;

/\*\* Current z position of this bubble. \*/

private double z;

/\*\*

\* The scale of a bubble is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the bubble and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this bubble is moving.

\*/

private double speed;

/\*\*

\* This constructor creates a new Bubble with the given image. It is

\* not possible or desirable to create a Bubble without an image, so

\* there is no zero-parameter constructor. Bubble properties are set

\* to random values within the bounds of the aquarium, but the speed

\* of all bubbles is the same.

\*

\* @param image BufferedImage object representing this bubble.

\*/

public Bubble( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// All bubbles move at the same speed.

this.speed = 2.0;

}

/\*\*

\* Bubbles move straight up and start over at a random location

\* on the bottom of the aquarium when the reach the top.

\*/

public void move()

{

// Bubbles move straight up, faster than anything else.

this.setY( this.getY() - this.getSpeed() \* 2.0 );

// When a bubble reaches the top, reset it to a random location

// at the bottom of the aquarium.

if( this.getY() < 0 )

{

this.setX( Math.random() \* Aquarium.WIDTH );

this.setY( Aquarium.HEIGHT ); // Bottom of the aquarium.

this.setZ( Math.random() \* Aquarium.DEPTH );

}

}

/\*\*

\* This method paints a bubble using the given Graphics object and

\* the bubble's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this bubble.

\*/

public void paint( Graphics g )

{

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

/\*\*

\* Sets the x position of this bubble, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Bubbles are allowed to

\* float outside this window before being reset.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this bubble.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* Sets the y position of this bubble, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Bubbles are allowed to

\* float outside this window before being reset.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param y New y position for this bubble.

\*/

public void setY( double y )

{

if( y < -Aquarium.HEIGHT || y > Aquarium.HEIGHT \* 2 )

{

throw new IllegalArgumentException( "y is out of range: " + y );

}

this.y = y;

}

/\*\*

\* Sets the y position of this bubble, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Bubbles are allowed to

\* float outside this window before being reset.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a bubble

\* is, which determines the scale to be used when drawing the

\* bubble's image.

\*

\* @param z New z position for this bubble.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

}

/\*\*

\* Sets the speed of this bubble, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed for this bubble.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Accessor method for the x position of this bubble. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this bubble.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this bubble. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this bubble.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this bubble. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this bubble.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this bubble. This value will be set

\* to 4.0 for all bubbles, twice the speed of the fastest creature.

\*

\* @return The current speed of this bubble.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this bubble. This value is based

\* on the z position of the bubble and will be between 0.0 and 1.0.

\*

\* @return The current scale of this bubble.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the width of this bubble. It is determined by

\* the current scale of the bubble and the width of the bubble's image.

\*

\* @return The current width of this bubble.

\*/

public int getWidth()

{

return (int)(this.image.getWidth() \* scale);

}

/\*\*

\* Accessor method for the height of this bubble. It is determined by

\* the current scale of the bubble and the height of the bubble's image.

\*

\* @return The current height of this bubble.

\*/

public int getHeight()

{

return (int)(this.image.getHeight() \* scale);

}

/\*\*

\* Accessor method for the depth of this bubble. It is determined by

\* the current scale of the bubble and is somewhat arbitrarily set

\* to one-half of the height of the height of the bubble.

\*

\* NOTE: The depth of a bubble is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this bubble.

\*/

public int getDepth()

{

return (int)(this.image.getHeight() / 2);

}

/\*\*

\* Creates and returns a string representation of this bubble that

\* includes the (x,y,z) position and the scale and speed of the bubble.

\*

\* @return A string representation of this bubble.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at ("

+ this.x + ", " + this.y + ", " + this.z + "), scale: "

+ this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Clownfish.java

\*

\* Description: Clownfish swim straight across the aquarium. When they meet

\* another creature, for a short time they angle downward if they are moving

\* left-to-right and upward if they are moving right-to-left.

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Clownfish

{

/\*\* Image representing this clownfish to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this clownfish. \*/

private double x;

/\*\* Current y position of this clownfish. \*/

private double y;

/\*\* Current z position of this clownfish. \*/

private double z;

/\*\*

\* The scale of a clownfish is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the clownfish and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this clownfish is moving; positive values indicate clownfish is

\* moving left-to-right, negative values indicate clownfish is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\*

\* Change in position for swimming at an angle.

\*/

private double delta;

/\*\* The amount of time to angle after meeting another creature. \*/

private int angleTime;

/\*\*

\* This constructor creates a new Clownfish with the given image. It is

\* not possible or desirable to create an Clownfish without an image, so

\* there is no zero-parameter constructor. Clownfish properties are set

\* to random values within the bounds of the aquarium and the angle time

\* is set to zero so the Clownfish will begin swimming horizontally.

\*

\* @param image BufferedImage object representing this clownfish.

\*/

public Clownfish( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some clownfish

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All clownfish are in the top 3/4 of tank. Subtract this

// height to ensure the bottom of the clownfish doesn't go below the

// upper two-thirds of the aquarium. Use Math.max to ensure clownfish

// is not off the top of the aquarium after subtracting the height.

this.setY( Math.max( Math.random() \* Aquarium.HEIGHT \* 0.75 -

this.getHeight(), 0 ) );

// Start swimming horizontally.

this.delta = 0.0;

// Initially not swimming at an angle.

this.angleTime = 0;

}

/\*\*

\* When a Clownfish meets another creature, for a short time they angle

\* downward if they are moving left-to-right and upward if they are

\* moving right-to-left.

\*/

public void meet()

{

// If not already moving at an angle, do so now.

if( angleTime <= 0 )

{

// Random value between 16 and 24. Why 16 and 24? Why not?!

this.angleTime = (int)( Math.random() \* 8 ) + 16;

// The delta, or vertical speed, is set to the same as the horizontal

// speed. The parameter to setDelta uses what is called a ternary

// operator. If necessary, use that magical Internet thingy to find

// more details about its use.

this.setDelta( this.getSpeed() > 0 ? this.getSpeed() : -this.getSpeed() );

}

}

/\*\*

\* A Clownfish moves the same way a swimmer moves, but if it is angling

\* after meeting another creature, each time it moves the angle time

\* is decremented.

\*/

public void move()

{

// Move the swimmer horizontally, based on its speed.

this.setX( this.getX() + this.getSpeed() );

// If the clownfish has moved out of the aquarium's window, it turns

// around and re-enters either at a different depth, front to back.

if( this.getX() < -this.getWidth() || this.getX() > Aquarium.WIDTH )

{

// Inverting the speed turns the swimmer around.

this.setSpeed( this.getSpeed() \* -1 );

// The change in depth will be between 1 and 2 "thicknesses" of

// the clownfish. See comment in Aquarium about depth.

double dz = this.getDepth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move toward the front, but no more than the depth of the aquarium.

this.setZ( Math.min( this.getZ() + dz, Aquarium.DEPTH ) );

}

else

{

// Move toward the back, but no less than the back of the aquarium.

this.setZ( Math.max( this.getZ() - dz, 0 ) );

}

}

// If clownfish is at the top and moving up or at the bottom and

// moving down, invert delta before moving y.

if( this.getY() < 0 && this.getDelta() < 0 ||

this.getY() + this.getHeight() > Aquarium.HEIGHT \* 0.75 )

{

this.setDelta( this.getDelta() \* -1.0 );

}

// Now go ahead and move the y position.

this.setY( this.getY() + this.getDelta() );

// Decrease angle time if necessary.

if( this.angleTime > 0 )

{

this.angleTime--;

// Stop swilling at an angle when angle time runs out.

if( this.angleTime <= 0 )

{

this.setDelta( 0.0 );

}

}

}

/\*\*

\* This method paints a clownfish using the given Graphics object and

\* the clownfish's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this clownfish.

\*/

public void paint( Graphics g )

{

// A clownfish's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the clownfish is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the clownfish is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this clownfish, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Clownfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this clownfish.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* Sets the y position of this clownfish, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Clownfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param y New y position for this clownfish.

\*/

public void setY( double y )

{

if( y < -Aquarium.HEIGHT || y > Aquarium.HEIGHT \* 2 )

{

throw new IllegalArgumentException( "y is out of range: " + y );

}

this.y = y;

}

/\*\*

\* Sets the y position of this clownfish, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Clownfish are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a clownfish

\* is, which determines the scale to be used when drawing the

\* clownfish image.

\*

\* @param z New z position for this clownfish.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

}

/\*\*

\* Sets the speed of this clownfish, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this clownfish.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Sets the delta, or vertical speed, of this clownfish.

\*

\* @param delta New delta, or vertical speed, for this clownfish.

\*/

public void setDelta( double delta )

{

this.delta = delta;

}

/\*\*

\* Accessor method for the x position of this clownfish. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this clownfish.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this clownfish. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this clownfish.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this clownfish. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this clownfish.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this clownfish. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this clownfish.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this clownfish. This value is based

\* on the z position of the clownfish and will be between 0.0 and 1.0.

\*

\* @return The current scale of this clownfish.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the delta, or vertical speed, of this clownfish.

\*

\* @return The current delta, or vertical speed, of this clownfish.

\*/

public double getDelta()

{

return this.delta;

}

/\*\*

\* Accessor method for the width of this clownfish. It is determined by

\* the current scale of the clownfish and the width of the clownfish image.

\*

\* @return The current width of this clownfish.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this clownfish. It is determined by

\* the current scale of the clownfish and the height of the clownfish image.

\*

\* @return The current height of this clownfish.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this clownfish. It is determined by

\* the current scale of the clownfish and is somewhat arbitrarily set

\* to one-half of the height of the height of the clownfish.

\*

\* NOTE: The depth of a clownfish is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this clownfish.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this clownfish that

\* includes the (x,y,z) position and the scale and speed of the clownfish.

\*

\* @return A string representation of this clownfish.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Crab.java

\*

\* Description: Crabs crawl straight forward and backward in the bottom

\* of the aquarium, randomly pausing for a short period. When they meet

\* another creature, they scurry away quickly, moving horizontally at

\* twice their normal speed for a short period.

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Crab

{

/\*\* Image representing this crab to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this crab. \*/

private double x;

/\*\* Current y position of this crab. \*/

private double y;

/\*\* Current z position of this crab. \*/

private double z;

/\*\*

\* The scale of a crab is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the crab and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this crab is moving; positive values indicate crab is

\* moving left-to-right, negative values indicate crab is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\* Amount of time to delay moving when stuttering. \*/

private int delay;

/\*\* The time a crab should scurry after meeting another creature. \*/

private int scurryTime;

/\*\* The speed a crab was moving before it needed to scurry. \*/

private double savedSpeed;

/\*\*

\* This constructor creates a new Crab with the given image. It is

\* not possible or desirable to create an Crab without an image, so

\* there is no zero-parameter constructor. Crab properties are set

\* to random values within the bounds of the aquarium and the

\* scurry time is set to zero so the crap will not be born scurrying.

\*

\* @param image BufferedImage object representing this crab.

\*/

public Crab( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some crabs

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All crabs move from the front to the back of the tank, so their

// y position is determined by their z position. The z position was

// set randomly above, so call the local version of setZ here in

// order to ensure Y is set properly.

this.setZ( this.getZ() );

// A crab is initially moving, so no delay.

this.delay = 0;

// A crab is initially not scurrying, so no scurry time.

this.scurryTime = 0;

}

/\*\*

\* When a crab meets another crab, they scurry away quickly, moving

\* horizontally at twice their normal speed for a short period.

\*/

public void meet()

{

// If it's not already scurrying, it's time to scurry!

if( scurryTime <= 0 )

{

// Randomly set the time the crab will scurry away at double

// speed between 32 and 48. Why 32 and 48? Why not?!

this.scurryTime = (int)( Math.random() \* 16 + 32 );

// Save the current speed for when the crab is done scurrying.

this.savedSpeed = this.getSpeed();

// Invert and double the speed, but do not exceed the speed limit.

if( this.getSpeed() > 0 )

{

this.setSpeed( Math.max( this.getSpeed() \* -2.0, -2.0 ) );

}

else

{

this.setSpeed( Math.min( this.getSpeed() \* -2.0, 2.0 ) );

}

}

}

/\*\*

\* Crabs move forward and backward in the aquarium, randomly pausing

\* for a short period. They also move horizontally when scurrying

\* away from another creature.

\*/

public void move()

{

// A crab only moves if it is not currently delaying.

if( this.delay > 0 )

{

this.delay--;

}

else

{

// Move the crawler forward or backward based on its speed.

this.setZ( this.getZ() + this.getSpeed() );

// If the crab has moved out of the aquarium's window, it turns around.

if( this.getZ() < 0 || this.getZ() > Aquarium.DEPTH )

{

this.setSpeed( this.getSpeed() \* -1 );

}

// If it has gone off the bottom of the aquarium's window, it moves

// randomly to the right or left.

if( this.getZ() > Aquarium.DEPTH )

{

// The change in x position will be between 1 and 2 widths

// of the crab.

double dx = this.getWidth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move right, but no more than the width of the aquarium.

this.setX( Math.min( this.getX() + dx, Aquarium.WIDTH ) );

}

else

{

// Move left, but no less than the back of the aquarium.

this.setX( Math.max( this.getX() - dx, 0 ) );

}

}

// If the crab is scurrying, it needs to move horizontally.

if( this.scurryTime > 0 )

{

// Move the swimmer horizontally, based on its speed.

this.setX( this.getX() + this.getSpeed() );

// If the crab has moved out of the aquarium's window, turns around.

if( this.getX() < -this.getWidth() || this.getX() > Aquarium.WIDTH )

{

this.setSpeed( this.getSpeed() \* -1 );

}

// Decrement scurry time and reset speed to normal if done scurrying.

this.scurryTime--;

if( scurryTime == 0 )

{

this.setSpeed( this.savedSpeed );

}

}

else // If not scurrying, pause about one out of every twenty moves.

{

if( Math.random() < 0.05 )

{

// Randomly set delay between 12 and 16. Why 12 and 16? Why not?!

this.setDelay( (int)( Math.random() \* 4 + 12 ) );

}

}

}

}

/\*\*

\* This method paints a crab using the given Graphics object and

\* the crab's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this crab.

\*/

public void paint( Graphics g )

{

// A crab's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the crab is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the crab is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this crab, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Crabs are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this crab.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* A crab's y position is determined by its z position and is set by setZ.

\*

\* @param y This value is ignored.

\*/

public void setY( double y )

{

// Do nothing.

}

/\*\*

\* Sets the y position of this crab, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Crabs are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a crab

\* is, which determines the scale to be used when drawing the

\* crab's image.

\*

\* @param z New z position for this crab.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// The y position of a lobster must be in the bottom quarter of the

// aquarium. Where in the bottom quarter is relative to the z position.

this.y = Aquarium.HEIGHT \* 0.75 + Aquarium.HEIGHT \* 0.25 \*

this.getZ() / Aquarium.DEPTH;

}

/\*\*

\* Sets the speed of this crab, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this crab.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Sets the delay, or time this stutterer should stutter.

\*

\* @param delay New delay for this stutterer.

\*/

public void setDelay( int delay )

{

this.delay = delay;

}

/\*\*

\* Accessor method for the x position of this crab. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this crab.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this crab. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this crab.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this crab. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this crab.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this crab. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this crab.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this crab. This value is based

\* on the z position of the crab and will be between 0.0 and 1.0.

\*

\* @return The current scale of this crab.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the delay, or time this stutterer should stutter.

\*

\* @return The current delay for this stutterer.

\*/

public int getDelay()

{

return this.delay;

}

/\*\*

\* Accessor method for the width of this crab. It is determined by

\* the current scale of the crab and the width of the crab's image.

\*

\* @return The current width of this crab.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this crab. It is determined by

\* the current scale of the crab and the height of the crab's image.

\*

\* @return The current height of this crab.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this crab. It is determined by

\* the current scale of the crab and is somewhat arbitrarily set

\* to one-half of the height of the height of the crab.

\*

\* NOTE: The depth of a crab is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this crab.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this crab that

\* includes the (x,y,z) position and the scale and speed of the crab.

\*

\* @return A string representation of this crab.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Lobster.java

\*

\* Description: Lobsters crawl straight forward and backward in the bottom

\* of the aquarium. Each time they move forward or backward in the aquarium,

\* they also move slightly left or right with a probability of 0.25. Thus,

\* half of the time they only move forward or backward and not horizontally.

\* When a lobster meets another creature, it pauses for a short period.

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Lobster

{

/\*\* Image representing this lobster to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this lobster. \*/

private double x;

/\*\* Current y position of this lobster. \*/

private double y;

/\*\* Current z position of this lobster. \*/

private double z;

/\*\*

\* The scale of a lobster is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the lobster and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this lobster is moving; positive values indicate lobster is

\* moving left-to-right, negative values indicate lobster is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\* Amount of time to delay moving when stuttering. \*/

private int delay;

/\*\*

\* This constructor creates a new Lobster with the given image. It is

\* not possible or desirable to create an Lobster without an image, so

\* there is no zero-parameter constructor. Lobster properties are set

\* to random values within the bounds of the aquarium.

\*

\* @param image BufferedImage object representing this lobster.

\*/

public Lobster( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some lobsters

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All lobsters move from the front to the back of the tank, so their

// y position is determined by their z position. The z position was

// set randomly above, so call the local version of setZ here in

// order to ensure Y is set properly.

this.setZ( this.getZ() );

// A lobster is initially moving, so no delay.

this.delay = 0;

}

/\*\*

\* When a lobster meets another creature, it pauses for a short time.

\*/

public void meet()

{

// Don't delay again if already delayed. (This would happen if two

// lobsters are on top of each other, repeatedly delaying.

if( this.getDelay() <= 0 )

{

// Randomly set delay between 8 and 12. Why 8 and 12? Why not?!

this.setDelay( (int)( Math.random() \* 4 + 8 ) );

}

}

/\*\*

\* Lobsters crawl straight forward and backward in the bottom

\* of the aquarium. Each time they move forward or backward in the aquarium,

\* they also move slightly left or right with a probability of 0.25. Thus,

\* half of the time they only move forward or backward and not horizontally.

\*/

public void move()

{

// A lobster only moves if it is not currently delaying.

if( this.delay > 0 )

{

this.delay--;

}

else

{

// Move the crawler forward or backward based on its speed.

this.setZ( this.getZ() + this.getSpeed() );

// If the lobster has moved out of the aquarium's window, turns around.

if( this.getZ() < 0 || this.getZ() > Aquarium.DEPTH )

{

this.setSpeed( this.getSpeed() \* -1 );

}

// If it has gone off the bottom of the aquarium's window, it moves

// randomly to the right or left.

if( this.getZ() > Aquarium.DEPTH )

{

// The change in x position will be between 1 and 2 widths

// of the lobster.

double dx = this.getWidth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move right, but no more than the width of the aquarium.

this.setX( Math.min( this.getX() + dx, Aquarium.WIDTH ) );

}

else

{

// Move left, but no less than the back of the aquarium.

this.setX( Math.max( this.getX() - dx, 0 ) );

}

}

// If the lobster moved horizontally, then move it vertically as

// described above.

if( this.getDelay() <= 0 )

{

// Switch with a random value of [0,3].

switch( (int)( Math.random() \* 4 ) )

{

case 0 : // Move left, but make sure to stay in the aquarium.

this.setX( Math.max( this.getX() - this.getSpeed(), 0 ) );

break;

case 1 : // Move right, but make sure to stay above the bottom.

this.setX( Math.min( this.getX() + this.getSpeed(),

Aquarium.WIDTH - this.getWidth() ) );

break;

default : // Do nothing to the horizontal position.

break;

}

}

}

}

/\*\*

\* This method paints a lobster using the given Graphics object and

\* the lobster's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this lobster.

\*/

public void paint( Graphics g )

{

// A lobster's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the lobster is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the lobster is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this lobster, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Lobsters are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this lobster.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* A lobster's y position is determined by its z position and is set by setZ.

\*

\* @param y This value is ignored.

\*/

public void setY( double y )

{

// Do nothing.

}

/\*\*

\* Sets the y position of this lobster, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Lobsters are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a lobster

\* is, which determines the scale to be used when drawing the

\* lobster's image.

\*

\* @param z New z position for this lobster.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// The y position of a lobster must be in the bottom quarter of the

// aquarium. Where in the bottom quarter is relative to the z position.

this.y = Aquarium.HEIGHT \* 0.75 + Aquarium.HEIGHT \* 0.25 \*

this.getZ() / Aquarium.DEPTH;

}

/\*\*

\* Sets the speed of this lobster, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this lobster.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Sets the delay, or time this stutterer should stutter.

\*

\* @param delay New delay for this stutterer.

\*/

public void setDelay( int delay )

{

this.delay = delay;

}

/\*\*

\* Accessor method for the x position of this lobster. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this lobster.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this lobster. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this lobster.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this lobster. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this lobster.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this lobster. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this lobster.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this lobster. This value is based

\* on the z position of the lobster and will be between 0.0 and 1.0.

\*

\* @return The current scale of this lobster.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the delay, or time this stutterer should stutter.

\*

\* @return The current delay for this stutterer.

\*/

public int getDelay()

{

return this.delay;

}

/\*\*

\* Accessor method for the width of this lobster. It is determined by

\* the current scale of the lobster and the width of the lobster's image.

\*

\* @return The current width of this lobster.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this lobster. It is determined by

\* the current scale of the lobster and the height of the lobster's image.

\*

\* @return The current height of this lobster.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this lobster. It is determined by

\* the current scale of the lobster and is somewhat arbitrarily set

\* to one-half of the height of the height of the lobster.

\*

\* NOTE: The depth of a lobster is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this lobster.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this lobster that

\* includes the (x,y,z) position and the scale and speed of the lobster.

\*

\* @return A string representation of this lobster.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Seahorse.java

\*

\* Description: Seahorses swim across the aquarium in an up and down,

\* zig-zag motion. Seahorses ignore all other creatures.

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Seahorse

{

/\*\* Image representing this seahorse to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this seahorse. \*/

private double x;

/\*\* Current y position of this seahorse. \*/

private double y;

/\*\* Current z position of this seahorse. \*/

private double z;

/\*\*

\* The scale of a seahorse is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the seahorse and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this seahorse is moving; positive values indicate seahorse is

\* moving left-to-right, negative values indicate seahorse is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\*

\* Change in position for swimming at an angle.

\*/

private double delta;

/\*\*

\* The seahorse will zigzag equally above and below this position.

\*/

private double startY;

/\*\*

\* This constructor creates a new Seahorse with the given image. It is

\* not possible or desirable to create an Seahorse without an image, so

\* there is no zero-parameter constructor. Seahorse properties are set

\* to random values within the bounds of the aquarium and the starting

\* y position is saved so the Seahorse can zig-zag above and below this

\* line. The Seahorse's delta, or vertical speed, is set to the same

\* as it's horizontal speed.

\*

\* @param image BufferedImage object representing this seahorse.

\*/

public Seahorse( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some seahorses

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All seahorses are in the top 3/4 of tank. Subtract this

// height to ensure the bottom of the seahorse doesn't go below the

// upper two-thirds of the aquarium. Use Math.max to ensure seahorse

// is not off the top of the aquarium after subtracting the height.

this.setY( Math.max( Math.random() \* Aquarium.HEIGHT \* 0.75 -

this.getHeight(), 0 ) );

// Start swimming horizontally.

this.delta = 0.0;

// Save the starting position so the Seahorse knows when to zig or zag.

this.startY = this.getY();

// Set the delta, or vertical speed, to the same as the horizontal speed.

this.setDelta( this.getSpeed() );

}

/\*\*

\* Seahorses move across the screen horizontally, but also vertically

\* in a zig-zag pattern.

\*/

public void move()

{

// Move the swimmer horizontally, based on its speed.

this.setX( this.getX() + this.getSpeed() );

// If the seahorse has moved out of the aquarium's window, it turns

// around and re-enters either at a different depth, front to back.

if( this.getX() < -this.getWidth() || this.getX() > Aquarium.WIDTH )

{

// Inverting the speed turns the swimmer around.

this.setSpeed( this.getSpeed() \* -1 );

// The change in depth will be between 1 and 2 "thicknesses" of

// the seahorse. See comment in Aquarium about depth.

double dz = this.getDepth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move toward the front, but no more than the depth of the aquarium.

this.setZ( Math.min( this.getZ() + dz, Aquarium.DEPTH ) );

}

else

{

// Move toward the back, but no less than the back of the aquarium.

this.setZ( Math.max( this.getZ() - dz, 0 ) );

}

}

// If seahorse is at the top and moving up or at the bottom and

// moving down, invert delta before moving y.

if( this.getY() < 0 && this.getDelta() < 0 ||

this.getY() + this.getHeight() > Aquarium.HEIGHT \* 0.75 )

{

this.setDelta( this.getDelta() \* -1.0 );

}

// Now go ahead and move the y position.

this.setY( this.getY() + this.getDelta() );

// The size of the zigzags is equal to one-half of the seahorse's height.

if( Math.abs( this.getY() - this.startY ) > this.getHeight() / 2.0 )

{

this.setDelta( this.getDelta() \* -1.0 );

}

}

/\*\*

\* This method paints a seahorse using the given Graphics object and

\* the seahorse's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this seahorse.

\*/

public void paint( Graphics g )

{

// A seahorse's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the seahorse is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the seahorse is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this seahorse, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Seahorses are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this seahorse.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* Sets the y position of this seahorse, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Seahorses are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param y New y position for this seahorse.

\*/

public void setY( double y )

{

if( y < -Aquarium.HEIGHT || y > Aquarium.HEIGHT \* 2 )

{

throw new IllegalArgumentException( "y is out of range: " + y );

}

this.y = y;

}

/\*\*

\* Sets the y position of this seahorse, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Seahorses are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a seahorse

\* is, which determines the scale to be used when drawing the

\* seahorse's image.

\*

\* @param z New z position for this seahorse.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

}

/\*\*

\* Sets the speed of this seahorse, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this seahorse.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Sets the delta, or vertical speed, of this seahorse.

\*

\* @param delta New delta, or vertical speed, for this seahorse.

\*/

public void setDelta( double delta )

{

this.delta = delta;

}

/\*\*

\* Accessor method for the x position of this seahorse. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this seahorse.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this seahorse. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this seahorse.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this seahorse. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this seahorse.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this seahorse. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this seahorse.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this seahorse. This value is based

\* on the z position of the seahorse and will be between 0.0 and 1.0.

\*

\* @return The current scale of this seahorse.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the delta, or vertical speed, of this seahorse.

\*

\* @return The current delta, or vertical speed, of this seahorse.

\*/

public double getDelta()

{

return this.delta;

}

/\*\*

\* Accessor method for the width of this seahorse. It is determined by

\* the current scale of the seahorse and the width of the seahorse's image.

\*

\* @return The current width of this seahorse.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this seahorse. It is determined by

\* the current scale of the seahorse and the height of the seahorse's image.

\*

\* @return The current height of this seahorse.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this seahorse. It is determined by

\* the current scale of the seahorse and is somewhat arbitrarily set

\* to one-half of the height of the height of the seahorse.

\*

\* NOTE: The depth of a seahorse is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this seahorse.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this seahorse that

\* includes the (x,y,z) position and the scale and speed of the seahorse.

\*

\* @return A string representation of this seahorse.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}

/\*\*

\* File: Snail.java

\*

\* Description: Snails crawl straight forward and backward in the bottom

\* of the aquarium, ignoring all other creatures. (Life in a shell!)

\*

\* @author Randall.Bower

\* @author YOUR NAME HERE

\*/

import java.awt.Graphics;

import java.awt.image.BufferedImage;

public class Snail

{

/\*\* Image representing this snail to be drawn on the aquarium. \*/

private BufferedImage image;

/\*\* Current x position of this snail. \*/

private double x;

/\*\* Current y position of this snail. \*/

private double y;

/\*\* Current z position of this snail. \*/

private double z;

/\*\*

\* The scale of a snail is determined by its distance from the front

\* of the aquarium window. See note in Aquarium about Aquarium.DEPTH.

\* It is determined by the z position of the snail and is set whenever

\* the z position is changed. It must be between 0.5 and 1.0.

\*/

private double scale;

/\*\*

\* Speed this snail is moving; positive values indicate snail is

\* moving left-to-right, negative values indicate snail is moving

\* right-to-left. Absolute value of speed must be between 0.5 and 2.0.

\*/

private double speed;

/\*\*

\* This constructor creates a new Snail with the given image. It is

\* not possible or desirable to create an Snail without an image, so

\* there is no zero-parameter constructor. Snail properties are set

\* to random values within the bounds of the aquarium.

\*

\* @param image BufferedImage object representing this snail.

\*/

public Snail( BufferedImage image )

{

this.image = image;

// The x, y, and z coordinates are random within the aquarium.

// Subclasses may impose further restrictions.

this.x = Math.random() \* Aquarium.WIDTH - this.image.getWidth();

this.y = Math.random() \* Aquarium.HEIGHT - this.image.getHeight();

this.z = Math.random() \* Aquarium.DEPTH;

// Scale depends on the z coordinate. See comment above.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// Speed is a random value between 0.5 and 2.0.

this.speed = 0.5 + Math.random() \* 1.5;

// Randomly, with 0.5 probability, invert speed so some snails

// are moving left-to-right and some are moving right-to-left.

if( Math.random() < 0.5 )

{

this.speed \*= -1;

}

// All snails move from the front to the back of the tank, so their

// y position is determined by their z position. The z position was

// set randomly above, so call the local version of setZ here in

// order to ensure Y is set properly.

this.setZ( this.getZ() );

}

/\*\*

\* Snails move forward and backward in the aquarium,

\* turning around when they reach an edge of the aquarium.

\*/

public void move()

{

// Move the snail forward or backward based on its speed.

this.setZ( this.getZ() + this.getSpeed() );

// If the snail has moved out of the aquarium's window, it turns around.

if( this.getZ() < 0 || this.getZ() > Aquarium.DEPTH )

{

this.setSpeed( this.getSpeed() \* -1 );

}

// If it has gone off the bottom of the aquarium's window, it moves

// randomly to the right or left.

if( this.getZ() > Aquarium.DEPTH )

{

// The change in x position will be between 1 and 2 widths

// of the snail.

double dx = this.getWidth() \* ( 1 + Math.random() \* 2 );

if( Math.random() < 0.5 )

{

// Move right, but no more than the width of the aquarium.

this.setX( Math.min( this.getX() + dx, Aquarium.WIDTH ) );

}

else

{

// Move left, but no less than the back of the aquarium.

this.setX( Math.max( this.getX() - dx, 0 ) );

}

}

}

/\*\*

\* This method paints a snail using the given Graphics object and

\* the snail's BufferedImage object.

\*

\* @param g The Graphics object to use to paint this snail.

\*/

public void paint( Graphics g )

{

// A snail's speed determines if it is facing left or right.

if( this.speed > 0 )

{

// Positive speed indicates the snail is facing right.

g.drawImage( this.image, (int)x, (int)y,

(int)( x + this.image.getWidth() \* scale ),

(int)( y + this.image.getHeight() \* scale ), 0, 0,

this.image.getWidth(), this.image.getHeight(), null );

}

else

{

// Negative speed indicates the snail is facing left. Notice the

// second and fourth paramters, which are the x-coordinates of the

// image being drawn, are inverted from the right-facing image above.

g.drawImage( this.image, (int)( x + this.image.getWidth() \* scale ),

(int)y, (int)x, (int)( y + this.image.getHeight() \* scale ),

0, 0, this.image.getWidth(), this.image.getHeight(), null );

}

}

/\*\*

\* Sets the x position of this snail, which must be between

\* -Aquarium.WIDTH and Aquarium.WIDTH \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Snails are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* @param x New x position for this snail.

\*/

public void setX( double x )

{

if( x < -Aquarium.WIDTH || x > Aquarium.WIDTH \* 2 )

{

throw new IllegalArgumentException( "x is out of range: " + x );

}

this.x = x;

}

/\*\*

\* snail's must have their own version of this method because a snail

\* cannot directly set its y position. A snail's y position is determined

\* by its z position and is set by setZ.

\*

\* @param y This value is ignored.

\*/

public void setY( double y )

{

// Do nothing.

}

/\*\*

\* Sets the y position of this snail, which must be between

\* -Aquarium.HEIGHT and Aquarium.HEIGHT \* 2. The dimensions of the aquarium

\* are actually a "window" into an aquarium. Snails are allowed to

\* swim or crawl outside this window before turning around to return.

\* The value is still error checked to ensure nothing accidentally

\* escapes the aquarium completely.

\*

\* NOTE: The depth of the aquarium is the distance from the back

\* to the front of the aquarium, not the water depth. Thus, this

\* value indicates how close to the front of the aquarium a snail

\* is, which determines the scale to be used when drawing the

\* snail's image.

\*

\* @param z New z position for this snail.

\*/

public void setZ( double z )

{

if( z < -Aquarium.DEPTH || z > Aquarium.DEPTH \* 2 )

{

throw new IllegalArgumentException( "z is out of range: " + z );

}

this.z = z;

// The z position determines the scale.

this.scale = 0.5 + this.z / Aquarium.DEPTH / 2.0;

// The y position of a snail must be in the bottom quarter of the

// aquarium. Where in the bottom quarter is relative to the z position.

// The y position of a snail must be in the bottom quarter of the

// aquarium. Where in the bottom quarter is relative to the z position.

this.y = Aquarium.HEIGHT \* 0.75 + Aquarium.HEIGHT \* 0.25 \*

this.getZ() / Aquarium.DEPTH;

}

/\*\*

\* Sets the speed of this snail, which must be between 0.5 and 2.0,

\* positive or negative.

\*

\* @param speed New speed position for this snail.

\*/

public void setSpeed( double speed )

{

if( Math.abs( speed ) < 0.5 || Math.abs( speed ) > 2.0 )

{

throw new IllegalArgumentException( "speed is out of range: " + speed );

}

this.speed = speed;

}

/\*\*

\* Accessor method for the x position of this snail. This value will

\* be between -image.getWidth() and Aquarium.WIDTH.

\*

\* @return The current x position of this snail.

\*/

public double getX()

{

return this.x;

}

/\*\*

\* Accessor method for the y position of this snail. This value will

\* be between 0 and Aquarium.HEIGHT. Subclasses may restrict this range.

\*

\* @return The current y position of this snail.

\*/

public double getY()

{

return this.y;

}

/\*\*

\* Accessor method for the z position of this snail. This value will

\* be between 0 and Aquarium.DEPTH.

\*

\* @return The current z position of this snail.

\*/

public double getZ()

{

return this.z;

}

/\*\*

\* Accessor method for the speed of this snail. This value will

\* be between 0.5 and 2.0, positive or negative.

\*

\* @return The current speed of this snail.

\*/

public double getSpeed()

{

return this.speed;

}

/\*\*

\* Accessor method for the scale of this snail. This value is based

\* on the z position of the snail and will be between 0.0 and 1.0.

\*

\* @return The current scale of this snail.

\*/

public double getScale()

{

return this.scale;

}

/\*\*

\* Accessor method for the width of this snail. It is determined by

\* the current scale of the snail and the width of the snail's image.

\*

\* @return The current width of this snail.

\*/

public int getWidth()

{

return (int)( this.image.getWidth() \* scale );

}

/\*\*

\* Accessor method for the height of this snail. It is determined by

\* the current scale of the snail and the height of the snail's image.

\*

\* @return The current height of this snail.

\*/

public int getHeight()

{

return (int)( this.image.getHeight() \* scale );

}

/\*\*

\* Accessor method for the depth of this snail. It is determined by

\* the current scale of the snail and is somewhat arbitrarily set

\* to one-half of the height of the height of the snail.

\*

\* NOTE: The depth of a snail is its "thickness" or the amount of

\* space it occupies front to back in the aquarium.

\*

\* @return The current width of this snail.

\*/

public int getDepth()

{

return (int)( this.image.getHeight() / 2 );

}

/\*\*

\* Creates and returns a string representation of this snail that

\* includes the (x,y,z) position and the scale and speed of the snail.

\*

\* @return A string representation of this snail.

\*/

@Override

public String toString()

{

return this.getClass().getName() + " is at (" +

this.x + ", " + this.y + ", " + this.z + "), scale: " +

this.scale + ", speed: " + this.speed;

}

}