**Project 2 – Ball World**

This project will allow you to practice what you have learned about numeric data types (int and double), calculations, arrays and for loops.

Your application will create balls that move around on the screen, changing their direction when they encounter obstacles. You will have access to two predefined classes, BallWorld and BallBot, to help you accomplish your tasks. The ball world is the window within which your ball bots will move. It is represented by an object of type BallWorld. Each ball in the ball world is represented by an object of type BallBot.

To begin, copy the folder, “Project 2 – Ball World”, from the shared drive to your working directory. In that folder you will find a BlueJ project file, package.bluej. Open that file. BlueJ will start. You should see a BlueJ project window. The project does not contain any class files but it does contain a library that you will need to complete the project.

**Activity 1 – A simple world with a single moving ball**

Create a new class, BallRunner, in BlueJ. Declare a run method for BallRunner, the same way you did in activity 6 of project 1. In the body of the run method, you will declare and initialize your ball world by calling the constructor of the BallWorld class. The constructor’s signature is:

*public BallWorld(int width, int height)*

The width and height parameters determine the size of the window in pixels. To initialize an object of type BallWorld, you need to use the new operator and the BallWorld constructor. A width and height of 200 pixels will work fine but feel free to experiment with different values if you wish. Assign the result to a local variable of the run method, called ballWorld.

Next you will create a ball to move around in your ball world. Your balls are objects of type BallBot. To create an object of type BallBot you need to know the signature of the BallBot constructor:

*public BallBot(BallWorld ballWorld, TGPoint startPoint,* ***double*** *startHeading,* ***int*** *radius)*

Lets look at each of the parameters to the BallBot constructor in more detail:

1. ballWorld is simply the object returned by the first line of your run method, described above.
2. startPoint is the location within the ball world window where your ballBot is placed to begin its travels. It is of type TGPoint. An object of type TGPoint has two fields, x and y. These values are specified in the coordinate space of the ball world. x = 0 and y = 0 refer to the point at the center of the ball world window. Positive values of x refer to the right side of the window. Negative values of x refer to the left side. Positive values of y refer to the top half of the window. Negative values refer to the bottom half. TGPoint objects are created by calling the TGPoint constructor. The signature of the constructor is:

*public TGPoint(double x, double y)*

1. startHeading defines the direction the ballBot will move, initially. Its values are interpreted as compass headings; 0 refers to up (north). A value of 90 indicates the ballBot will move to the right (east). startHeading = 180 indicates it will move down (south) and 270 indicates it will move to the left (west). Any value between 0 and 360 is valid and, since startHeading is of type double, the value can have a decimal or fractional part, too.
2. radius is the size of the ballBot, as measured by the radius of the circle that represents it on the screen.

So, now you know enough to write the second line of your run method. Declare a local variable, ballBot, and initialize it with the new operator and BallBot constructor. To do that you need to create a new object of type TGPoint and pass it to the BallBot constructor along with ballWorld and values for startHeading and radius. Here are some recommended values, but feel free to experiment:

1. startPoint: x= 0, y = 0*.*
2. startHeading : 0 (north).
3. radius: 25.

Once you write these two statements in the body of your run method you should be able to compile and run your application. A window should appear with a black circle in the middle. That is your ballBot. Try it!

Now it is time to make your ballBot move. A ballBot is a bit like a mazeBot from Project 1. In particular, it provides you with two familiar methods, moveForward and canMoveForward. moveForward takes no parameters and returns no result. It simply moves your ball forward; that is in the direction that it is currently heading. Unlike a mazeBot, though, when you tell a ballBot to move forward it only moves one pixel. How far did the mazeBot move in Project 1?

Unlike mazeBot, the canMoveForward method of a ballBot object takes one parameter; the ballWorld. Here is the signature:

**public** **boolean** canMoveForward(BallWorld ballWorld)

Like MazeBot it returns a boolean. That boolean value has a different meaning for mazeBots and ballBots. Here is the way to interpret the return value of the ballBot.canMoveForward method. If it returns false then it is touching one of the window boundaries (left, right, top or bottom). If it returns true then it is far enough away from all those boundaries to safely move forward and still be visible on the screen. The ballBot needs the ballWorld parameter to figure out whether to return true or false.

For now, you need to know about two more BallBot methods:

***public******double*** *getHeading()*

***public******void*** *setHeading(****double*** *newHeading)*

As the name implies, getHeading is a getter method that returns the direction that the ballBot is currently moving. Similarly, setHeading is a setter method that instructs the ballBot object to turn so that it will move forward in the future in the direction specified by the parameter, newHeading.

Now you know enough to complete activity 1. At the end of your run method, insert a while loop that never exits. What should you put in the boolean test of the while loop to make sure it never exits? The body of the while loop should contain a conditional statement that tests whether the ballBot can move forward. If it can, then simply call ballBot.moveForward. If it cannot, then choose a new heading and instruct the ballBot to move in that direction, using the setHeading method. I recommend adding 90 degrees to the current heading. A valid heading should be between 0 and 360 degrees. Use the mod operator (%) to make sure this is true.

Compile and run your application. Your ballBot should move up until it touches the top of the window, then turn right, etc.

**Activity 2 – Add multiple moving balls**

In activity 2 you will add multiple balls to your world and set them in motion in different, random directions. To accomplish this you will enhance your BallRunner class to include one additional helper method. Then you will add instructions to your run method to complete the activity. In the process you will gain experience working with arrays and for loops.

Let me describe the behavior you will create with your program. Balls will appear at a predefined spot in the ball world window, called the entrance point. They will move from the entrance point in a random direction that you choose. They will move in that direction until they hit a boundary of the ball world window, just like your ball did in activity 1. But, at that point they will change direction randomly rather than simply setting a new heading 90 degrees more than the current heading.

Before you start, rename your run method to activity1. You are doing this simply to save that code so that you can refer to it later if need be.

You will need one helper method, so lets write that now. The method searches through the ballBotArray looking for an empty entry where you may remember a newly created ballBot. Name the method findFreeBallBotIndex. The signature of findFreeBallBotIndex is:

**public** **int** findFreeBallBotIndex(BallBot[] ***ballBotArray***)

As you can see, it takes one parameter; the ballBotArray to search, looking for the index of a free element. It returns an int; the index of a free entry in ballBotArray. The body of findFreeBallBotIndex is simple. It contains a for loop that iterates from 0 to but not including ballBotArray.length. Inside the loop it fetches the entry of ballBotArray at the index specified by the for loop control variable. If it equals null then the entry is free – return the current index. If you iterate through all the indexes and don’t find a null entry then return ballBotArray.length to indicate the array is full.

*OK. Now you are ready to write your new run method. Declare it just as you did in activity 1. First, declare and initialize a local variable to contain a BallWorld object, as you did in activity 1. Second, declare a local variable to contain the entrancePoint and initialize by creating a TGPoint object, as you learned in a recent lecture, using the new operator and the TGPoint constructor. I recommend the point in the middle of the window; i.e. x=0 and y=0. You created a TGPoint in activity 1. Refer to that if you don’t remember exactly how to do it.. Third, declare an array of BallBot objects, name it ballBotArray and initialize it to contain 10 elements*. *Finally, declare a ballRunner object and initialize it using the new operator and the BallRunner constructor.*

*Note: You haven’t actually declared a BallRunner constructor but that is OK. Java will create a constructor that takes no parameters automatically for you. In a later lesson you will learn how to declare a class constructor.*

*The reason we are creating a BallRunner object is because we need an object to call a method. In particular, we need a BallRunner object to call our new helper method, findFreeBallBotIndex.*

Next, just as you did in activity 1, insert **a while loop that never exits**. In the body of the while loop, insert instructions to:

1. Test whether there is room in ballBotArray for an addition ballBot. Do this by calling findFreeBallBotIndex and assigning the return value to a local variable, freeBallBotIndex. If freeBallBotIndex is less than the length of ballBotArray then you have found a free entry. In that case, *create a new ballBot by calling the BallBot constructor and assigning the object returned to the entry of ballBotArray at freeBallBotIndex*. Recall that the constructor for BallBot takes 4 parameters. The first is the **ballWorld**. The second is the **starting point of the ballBot**. **Pass the entrancePoint for this parameter.** The third parameter is the **starting heading**. Set this heading to a **random value between 0 and 360. You learned in class about the method, random, in the Math class that will return a random number between 0 and 1. Simply multiply the return value of Math.random by 360 to get your starting heading.**  The final parameter is the **radius of the circle** that represents the ballBot in the ball world window.
2. After the conditional statement in step 1 you will move all the existing balls. To do this, write a for loop that iterates through all the entries in ballBotArray. The for loop will look like the one in findFreeBallBotIndex but with a different body. It will have a control variable, call it index. It will iterate from 0 to but not including ballBotArray.length. In the body, if the ballBot at the current index is not equal to null then move it in a fashion similar to what you did in activity 1. If you can move the ball forward, do so by calling ballBot.moveForward. If you cannot, then point the ball in a new, random direction, by calling ballBot.setHeading with a random value between 0 and 360. Create that random value just the way you did the starting heading in step 1.

That should do it! Compile your application. When you run it, 10 balls should appear at the center of the screen, move in random directions and, when they touch a boundary of the ball world, change direction in a random way.

**Activity 3 – Slow down the creation of ballBot objects**

In this activity and the next, you will improve the behavior of your ballBot objects by controlling the rate at which the new objects are created (activity 3) and by changing the direction of ballBot objects that touch each other (activity 4). Our goal for the code we write in activity 3 is to only create a new ballBot when the area around the entrance point that the new ballBot will occupy is not already occupied by an existing ballBot. To achieve this goal you will write two more methods. The first computes the distance between two points, aptly named distanceBetweenPoints. The second method, entranceClear, returns true if the area around the entrance point is clear of other ballBot objects.

The method, distanceBetweenPoints, takes two parameters both of type TGPoint. It returns a double. Here is the method signature:

***public******double***distanceBetweenPoints *(****TGPoint*** *point1,* ***TGPoint*** *point2)*

Implement this method using the well know formula:

Here are some hints to help you implement distanceBetweenPoints:

* Java’s Math class has a method, sqrt, that takes a double as a parameter and returns its square root.
* The simplest way to square a number is to simply multiply it by itself. Alternatively, he Math class has a method, pow, to raise a number to an arbitrary power. The signature of pow is:

**public double** pow(**double** base, **double** power);

* Your code will be more readable if you compute the difference of the x values of the 2 points and store it in a local variable. The same is true of the difference of the y values.

Once you have written the code for distanceBetweenPoints, write the entranceClear method. It’s signature is:

***public******boolean***entranceClear *(****BallBot[]*** *ballBotArray,* ***TGPoint*** *entrancePoint)*

The body of the method, entranceClear, will look a bit like the body of findFreeBallBotIndex. It contains a for loop that iterates over the ballBotArray, just like findFreeBallBotIndex. In the body of the for loop check to make sure that the ballBot at the current index is not null. If it is not, test whether the distance between the ballBot object’s current point and the entrance point is large enough so that the new ball at the entrance point will not touch it. Of course, you will use distanceBetweenPoints to compute the distance. The entrance point is supplied to entranceClear as a parameter, but how do you find the current position of a ballBot? Fortunately, the BallBot class defines a getter method to return it’s current position. It’s signature is:

***public******TGPoint*** *getPoint()*

How small should the distance between entrancePoint and ballBot.getPoint(++) be before you return false? If the distance is less than 2 times the radius of the current ballBot then that ballBot is “in the way”. Return false. You should only return true if all of the ballBot objects in ballBotArray are clear of the entrance; that is, at the end of the method after the for loop completes. You created all you ballBot objects with a radius of 25 so you could use the constant 25 in your test but it is better programming practice to ask the ballBot object how big it is. After all, the radius of the object may change (see activity 5!). The BallBot class has another getter method that will tell you the ballBot object’s radius. Of course, it’ signature is:

***public******double*** *getRadius()*

To summarize, your entranceClear method iterates over all the ballBot objects in ballBotArray. For each ballBot that is not null, if the distance from the entrance to the ballBot is less than 2 times the ballBot’s radius, return false. If the for loop completes without returning false then the entrance must be clear. Return true in this case.

You are now ready to modify your run method to make sure that the entrance is clear before a new ballBot object is created. All you need to do is insert a conditional statement at the beginning of your while loop. The test in that conditional statement is simply a call to ballRunner.entranceClear(). If it returns true you already know what to do. Create a new ballBot! So, just take the instructions that implemented that in activity 2 and move them into the body of the conditional.

Compile and run your application. It should behave similarly to your activity 2 program, except the new ballBot objects will not appear until the previous object moves away from the entrance.

**Activity 4 – Bouncing balls**

You have 10 ballBot objects moving in the screen. If they hit a ball world window boundary they “bounce off” in some random direction. New ballBot objects do not appear until there is space for them at the entrance without touching any other ballBot objects. However, as the ballBot objects move around they don’t “bounce off” each other. They pass through each other like ghosts. In activity 4 you will change this behavior.

You will only need to define and implement one new method to complete this activity, but it is a little tricky to get it right. Don’t worry. The instructions will guide you to the best implementation.

The method you will implement is called ballBotToBounceOff and its signature is:

***public******BallBot***ballBotToBounceOff *(****BallBot*** *ballBot,* ***BallBot[]*** *ballBotArray)*

ballBotToBounceOff returns the first ballBot object it finds that the parameter, ballBot, must bounce off of. Under what conditions must the ballBot bounce? At first, you may think that it should bounce if the 2 ballBot objects are touching. We will call that condition, C1. C1 is necessary but not sufficient for our purposes. If the 2 objects are touching then ballBotToBounceOff must return true only if the 2 objects will still be touching if the parameter ballBot moves forward. Otherwise, once two objects touch they will never move again. We don’t want that! We will call that second condition, C2. To summarize, C1 and C2 must be true in order for ballBotToBounceOff to return true.

So, here are the steps that your method, ballBotToBounceOff, must implement:

1. Declare 2 local variables, to improve readability and performance. Name the first point. Initialize it to the value returned by ballBot.getPoint(). Name the second nextPoint. Initialize it to the position of the ballBot parameter assuming it moves forward. Of course, the BallBot class has a method to tell you this. Its signature is:

***public******TGPoint*** *forwardPoint ()*

1. Write a for loop to iterate over all the ballBot objects in ballBotArray. You should be able to do this in your sleep by now!
2. In the body of your for loop, declare a local variable, call it otherBallBot, and initialize it to the element in the ballBotArray indexed by your for loop control variable. Test whether otherBallBot is both not equal null and not equal to the ballBot parameter. If both of those conditions are met then declare a local variable, currentDistance, and initialize it to the distance between ballBot and otherBallBot. If currentDistance is less than or equal to the sum of the radii of ballBot and otherBallBot then condition C1, defined above is met! Now we need to check the value of condition C2. Declare a local variable, nextDistance, and initialize it to the distance between nextPoint and the current position of otherBallBot. If nextDistance is less than or equal to currentDistance then ballBot must bounce off of otherBot. Return otherBallBot!
3. If the for loop exits without finding any otherBallBot to bounce off of then return null.

You are now ready to modify your run method so that your balls will bounce off each other. All the tough work is done by ballBotToBounceOff. Look at the run method from the previous activity. You will find a conditional statement that looks something like this:

**if** (ballBot.canMoveForward(ballWorld)) {

ballBot.moveForward();

}

**else** {

ballBot.setHeading(360.0\*Math.random());

}

Modify that conditional statement. If ballBot.canMoveForward returns true then test whether ballBotRunner.ballBotToBounceOff(ballBot) returns null. If it does, then ballBot really can move forward, as it does in the above sample code. If it returns something other than null then ballBot need to bounce by choosing a new random direction, just as it does when it can’t move forward because it will run into a ball world window boundary.

Compile and test your application.

**Activity 5 – Bonus activity. Have fun.**

In activities 1 through 4 your ballBot objects are all the same size. They are all black. They move at the same speed. It turns out you can change all of these characteristics of your ballBot objects. You already know that the BallBot constructor takes a parameter, radius, which determines its size. The BallBot class also provides methods to get and set the ballBot objects color and speed. Their signatures are:

***public******int*** *getColor()*

***public******int*** *getPixelsPerSecond()*

***public******void*** *setColor(int color)*

***public******void*** *setPixelsPerSecond (int pixelsPerSecond)*

A color is an int between 0 and 31. A pixel per second speed is any non-negative int.

Have some fun by creating ballBot objects that have different values for radius, color and pixelsPerSecond.

**Appendix 1**

**Provided Classes**

**Class BallWorld**

* java.lang.Object
  + BallWorld

public class **BallWorld** extends java.lang.Object

A BallWorld is the window within which ballBots exist. The origin of the coordinate space is at the center of the window. Positive x coordinates are to the right. Positive y coordintes are up (not down as is typical in most window systems.

**Author:**

Tom Malloy, thomasjmalloy@kttk.org

* + ***Constructor Summary***

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**BallWorld**](BallWorld.html#BallWorld-int-int-)(int width, int height)  Public constructor |

* + ***Method Summary***

|  |  |
| --- | --- |
| **All Methods**[**Instance Methods**](javascript:show(2);)[**Concrete Methods**](javascript:show(8);) | |
| **Modifier and Type** | **Method and Description** |
| [**TGPoint**](TGPoint.html) | [**getBottomLeftPoint**](BallWorld.html#getBottomLeftPoint--)()  The minimum x and y coordinates of the BallWorld window |
| [**TGRect**](TGRect.html) | [**getBoundingTGRect**](BallWorld.html#getBoundingTGRect--)()  The bounding box of the BallWorld window |
| [**TGCanvas**](TGCanvas.html) | [**getCanvas**](BallWorld.html#getCanvas--)()  The getter for the AWT component that contains the content of the BallWorld window |
| int | [**getHeight**](BallWorld.html#getHeight--)()  The height of the BallWorld window |
| int | [**getWidth**](BallWorld.html#getWidth--)()  The width of the BallWorld window |

* + - **Methods inherited from class java.lang.Object**

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

* + ***Constructor Detail***
    - **BallWorld**

public BallWorld(int width, int height)

Public constructor

**Parameters:**

width - the width of the window

height - the height of the window

* + ***Method Detail***
    - **getCanvas**

public [TGCanvas](TGCanvas.html" \o "class in <Unnamed>) getCanvas()

The getter for the AWT component that contains the content of the BallWorld window

**Returns:**

the canvas component

* + - **getBoundingTGRect**

public [TGRect](TGRect.html" \o "class in <Unnamed>) getBoundingTGRect()

The bounding box of the BallWorld window

**Returns:**

the bounding box in BallWorld coordinates

* + - **getBottomLeftPoint**

public [TGPoint](TGPoint.html" \o "class in <Unnamed>) getBottomLeftPoint()

The minimum x and y coordinates of the BallWorld window

**Returns:**

the bottom,left point

* + - **getWidth**

public int getWidth()

The width of the BallWorld window

**Returns:**

the width

* + - **getHeight**

public int getHeight()

The height of the BallWorld window

**Returns:**

the height

## Class BallBot

* java.lang.Object
  + [AnimatedSprite](AnimatedSprite.html)
    - BallBot
* **Direct Known Subclasses:**

[CannonBallBotComplete](CannonBallBotComplete.html), [ProjectileBot](ProjectileBot.html" \o "class in <Unnamed>)

public class **BallBot** extends [AnimatedSprite](AnimatedSprite.html)

A BallBot is an animated circle that moves around the BallWorld. The BallBot's size is specified as a radius. Most of its behaviour is implemented by the AnimatedSprite object that is associated with the BallBot through the animatedSprite field.

**Author:**

Tom Malloy, thomasjmalloy@kttk.org

### *Field Summary*

### Fields inherited from class [AnimatedSprite](AnimatedSprite.html" \o "class in <Unnamed>)

[DEFAULT\_PIXELS\_PER\_SECOND](AnimatedSprite.html#DEFAULT_PIXELS_PER_SECOND)

### *Constructor Summary*

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**BallBot**](BallBot.html#BallBot-BallWorld-TGPoint-double-int-)(**[BallWorld](BallWorld.html" \o "class in <Unnamed>)** ballWorld, **[TGPoint](TGPoint.html" \o "class in <Unnamed>)** startPoint, double startHeading, int radius)  Public constructor |
| [**BallBot**](BallBot.html#BallBot-BallWorld-TGPoint-double-int-int-)(**[BallWorld](BallWorld.html" \o "class in <Unnamed>)** ballWorld, **[TGPoint](TGPoint.html" \o "class in <Unnamed>)** startPoint, double startHeading, int radius, int shapeNum)  Public constructor |
| [**BallBot**](BallBot.html#BallBot-BallWorld-TGPoint-TGPoint-int-)([**BallWorld**](BallWorld.html) ballWorld, [**TGPoint**](TGPoint.html) startPoint, [**TGPoint**](TGPoint.html) velocityVector, int radius)  Public constructor |

### *Method Summary*

|  |  |
| --- | --- |
| **All Methods**[**Instance Methods**](javascript:show(2);)[**Concrete Methods**](javascript:show(8);) | |
| **Modifier and Type** | **Method and Description** |
| boolean | [**canMoveForward**](BallBot.html#canMoveForward-BallWorld-)(**[BallWorld](BallWorld.html" \o "class in <Unnamed>)** ballWorld)  Determine if a call to moveForward will encounter a BallWorld boundary. |
| [**TGPoint**](TGPoint.html) | [**deltaPointForward**](BallBot.html#deltaPointForward--)()  Compute the distance that will be traveled forward if moveForward is called. |
| [**TGPoint**](TGPoint.html) | [**forwardPoint**](BallBot.html#forwardPoint--)()  Compute the new position that will result if moveForward is called. |
| int | [**getRadius**](BallBot.html#getRadius--)()  Getter for radius field |

### Methods inherited from class [AnimatedSprite](AnimatedSprite.html" \o "class in <Unnamed>)

[accelerate](AnimatedSprite.html#accelerate-TGPoint-), [addAnimatedMotion](AnimatedSprite.html#addAnimatedMotion-double-double-), [deltaPointForward](AnimatedSprite.html#deltaPointForward-double-double-), [deltaPointPerSecond](AnimatedSprite.html#deltaPointPerSecond-double-double-), [distanceForward](AnimatedSprite.html#distanceForward-double-), [getBoundingTGRect](AnimatedSprite.html#getBoundingTGRect--), [getHeading](AnimatedSprite.html#getHeading--),[getHeadingInRadians](AnimatedSprite.html#getHeadingInRadians--), [getHeadingTowards](AnimatedSprite.html#getHeadingTowards-TGPoint-), [getLastScheduledAnimationAction](AnimatedSprite.html#getLastScheduledAnimationAction--), [getPixelsPerSecond](AnimatedSprite.html#getPixelsPerSecond--), [getPoint](AnimatedSprite.html#getPoint--), [getRectangle](AnimatedSprite.html#getRectangle-TGPoint-), [getSprite](AnimatedSprite.html#getSprite--),[getVelocityVector](AnimatedSprite.html#getVelocityVector--), [hide](AnimatedSprite.html#hide--), [isAnimated](AnimatedSprite.html#isAnimated--), [localPlayAudioFile](AnimatedSprite.html#localPlayAudioFile-java.lang.String-), [moveForward](AnimatedSprite.html#moveForward--), [pendown](AnimatedSprite.html#pendown--), [penup](AnimatedSprite.html#penup--), [setColor](AnimatedSprite.html#setColor-int-), [setHeading](AnimatedSprite.html#setHeading-double-), [setPixelsPerSecond](AnimatedSprite.html#setPixelsPerSecond-double-),[setPoint](AnimatedSprite.html#setPoint-TGPoint-), [setShape](AnimatedSprite.html#setShape-SpritePixels-), [setVelocityVector](AnimatedSprite.html#setVelocityVector-TGPoint-), [show](AnimatedSprite.html#show--), [signalError](AnimatedSprite.html#signalError--), [signalSuccess](AnimatedSprite.html#signalSuccess--), [spin](AnimatedSprite.html#spin--), [startMoving](AnimatedSprite.html#startMoving-double-), [stopAnimating](AnimatedSprite.html#stopAnimating--), [stopMoving](AnimatedSprite.html#stopMoving--)

### Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### *Constructor Detail*

#### BallBot

public BallBot([BallWorld](BallWorld.html" \o "class in <Unnamed>) ballWorld, [TGPoint](TGPoint.html) startPoint, double startHeading, int radius)

Public constructor

**Parameters:**

ballWorld - the world (window) within which the BallBot exists

startPoint - the point in BallWorld that the new BallBot appears

startHeading - the initial forward direction of the BallBot

radius - the radius of the BallBot

#### BallBot

public BallBot([BallWorld](BallWorld.html" \o "class in <Unnamed>) ballWorld, [TGPoint](TGPoint.html) startPoint, double startHeading, int radius, int shapeNum)

Public constructor

**Parameters:**

ballWorld - the world (window) within which the BallBot exists

startPoint - the point in BallWorld that the new BallBot appears

startHeading - the initial forward direction of the BallBot

radius - the radius of the BallBot

shapeNum - the shape of the sprite. Change for debugging purposes to, for instance, Sprite.ARROW

#### BallBot

public BallBot([BallWorld](BallWorld.html" \o "class in <Unnamed>) ballWorld, [TGPoint](TGPoint.html) startPoint, [TGPoint](TGPoint.html) velocityVector, int radius)

Public constructor

**Parameters:**

ballWorld - the world (window) within which the BallBot exists

startPoint - the point in BallWorld that the new BallBot appears

velocityVector - the initial heading and speed of the new BallBot

radius - the radius of the BallBot

### *Method Detail*

#### getRadius

public int getRadius()

Getter for radius field

**Returns:**

radius of this ballBot

#### deltaPointForward

public [TGPoint](TGPoint.html" \o "class in <Unnamed>) deltaPointForward()

Compute the distance that will be traveled forward if moveForward is called.

**Returns:**

a TGPoint that contains the distance in the x and y direction that will be moved

#### forwardPoint

public [TGPoint](TGPoint.html" \o "class in <Unnamed>) forwardPoint()

Compute the new position that will result if moveForward is called.

**Returns:**

the new position

#### canMoveForward

public boolean canMoveForward([BallWorld](BallWorld.html" \o "class in <Unnamed>) ballWorld)

Determine if a call to moveForward will encounter a BallWorld boundary.

**Returns:**

true if this ballBot can move forward without hitting a boundary

## Class TGPoint

* java.lang.Object
  + TGPoint
* **All Implemented Interfaces:**

java.lang.Cloneable

public class **TGPoint** extends java.lang.Object implements java.lang.Cloneable

A TGPoint is a virtual point, a point in TurtleSpace. In TurtleSpace, 0.0,0.0 is at the center of the graphics canvas, like the way coordinate spaces are represented in Algebra. Likewise, TGPoints have appoximated real number coordinates. TGPoints may or may not be visible based upon the current size of the graphics canvas window of TG. Methods are provided to map the point's coordinates from TurtleSpace to an Image's x and y indicies.

TurtleGraphics now works entirely with TGPoints - points in TurtleSpace. Early versions of TG used int primitive values for X and Y coordinates. This approach led to too many visual problems caused by the propagation of rounding off values. TGPoint isolates actual implementation of points to one place which can thus be more easily controlled.

### *Field Summary*

|  |  |
| --- | --- |
| **Fields** | |
| **Modifier and Type** | **Field and Description** |
| double | [**x**](TGPoint.html#x)  This TGPoint's X coordinate. |
| double | [**y**](TGPoint.html#y)  This TGPoint's Y coordinate. |

### *Constructor Summary*

|  |
| --- |
| **Constructors** |
| **Constructor and Description** |
| [**TGPoint**](TGPoint.html#TGPoint--)()  Default TGPoint is [0,0] which is the center of TurtleSpace. |
| [**TGPoint**](TGPoint.html#TGPoint-double-double-)(double x, double y)  Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided double values for x and y coordinates. |
| [**TGPoint**](TGPoint.html#TGPoint-float-float-)(float x, float y)  Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided float values for x and y coordinates. |
| [**TGPoint**](TGPoint.html#TGPoint-int-int-)(int x, int y)  Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided int values for x and y coordinates. |
| [**TGPoint**](TGPoint.html#TGPoint-long-long-)(long x, long y)  Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided long values for x and y coordinates. |

### *Method Summary*

|  |  |
| --- | --- |
| **All Methods**[**Instance Methods**](javascript:show(2);)[**Concrete Methods**](javascript:show(8);) | |
| **Modifier and Type** | **Method and Description** |
| java.lang.Object | [**clone**](TGPoint.html#clone--)()  Creates and returns a copy of this TGPoint. |
| boolean | [**equals**](TGPoint.html#equals-java.lang.Object-)(java.lang.Object obj)  Override Object.equals(Object) so that comparison is based on the x and y fields. |
| double | [**imageX**](TGPoint.html#imageX-double-)(double imageWidth)  Return an Image's index equivilent for this point's X coordinate. |
| int | [**imageX**](TGPoint.html#imageX-double-int-)(double offset, int imageWidth)  Return an Image's index equivilent for this point's X coordinate combined with a provided offset. |
| int | [**imageX**](TGPoint.html#imageX-int-)(int imageWidth)  Return an Image's index equivilent for this point's X coordinate. |
| double | [**imageY**](TGPoint.html#imageY-double-)(double imageHeight)  Return an Image's index equivilent for this point's Y coordinate. |
| int | [**imageY**](TGPoint.html#imageY-double-int-)(double offset, int imageHeight)  Return an Image's index equivilent for this point's Y coordinate combined with a provided offset. |
| int | [**imageY**](TGPoint.html#imageY-int-)(int imageHeight)  Return an Image's index equivilent for this point's Y coordinate. |
| [**TGPoint**](TGPoint.html) | [**otherEndPoint**](TGPoint.html#otherEndPoint-double-double-)(double radians, double length)  Given one end point of a line, its length and heading (in radians), return its other end point. |
| java.lang.String | [**toString**](TGPoint.html#toString--)()  Returns the String representation of this point. |
| double | [**xDoubleValue**](TGPoint.html#xDoubleValue--)()  Returns the X coordinate of this point in double precision. |
| long | [**xLongValue**](TGPoint.html#xLongValue--)()  Returns the X coordinate of this point as a long int. |
| double | [**yDoubleValue**](TGPoint.html#yDoubleValue--)()  Returns the Y coordinate of this point in double precision. |
| long | [**yLongValue**](TGPoint.html#yLongValue--)()  Returns the Y coordinate of this point as a long int. |

### Methods inherited from class java.lang.Object

getClass, hashCode, notify, notifyAll, wait, wait, wait

### *Field Detail*

#### x

public final double x

This TGPoint's X coordinate.

#### y

public final double y

This TGPoint's Y coordinate.

### *Constructor Detail*

#### TGPoint

public TGPoint()

Default TGPoint is [0,0] which is the center of TurtleSpace.

#### TGPoint

public TGPoint(double x, double y)

Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided double values for x and y coordinates.

#### TGPoint

public TGPoint(float x, float y)

Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided float values for x and y coordinates.

#### TGPoint

public TGPoint(int x, int y)

Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided int values for x and y coordinates.

#### TGPoint

public TGPoint(long x, long y)

Constructs a new TGPoint (a virtual point in TurtleSpace) based upon provided long values for x and y coordinates.

### *Method Detail*

#### equals

public boolean equals(java.lang.Object obj)

Override Object.equals(Object) so that comparison is based on the x and y fields.

**Overrides:**

equals in class java.lang.Object

#### imageX

public double imageX(double imageWidth)

Return an Image's index equivilent for this point's X coordinate. The index reflects a mapping based on a provided Image width.

#### imageX

public int imageX(int imageWidth)

Return an Image's index equivilent for this point's X coordinate. The index reflects a mapping based on a provided Image width.

#### imageX

public int imageX(double offset, int imageWidth)

Return an Image's index equivilent for this point's X coordinate combined with a provided offset. The index reflects a mapping based on a provided Image width.

#### imageY

public double imageY(double imageHeight)

Return an Image's index equivilent for this point's Y coordinate. The index reflects a mapping based on a provided Image height.

#### imageY

public int imageY(int imageHeight)

Return an Image's index equivilent for this point's Y coordinate. The index reflects a mapping based on a provided Image height.

#### imageY

public int imageY(double offset, int imageHeight)

Return an Image's index equivilent for this point's Y coordinate combined with a provided offset. The index reflects a mapping based on a provided Image height.

#### clone

public java.lang.Object clone()

Creates and returns a copy of this TGPoint.

**Overrides:**

clone in class java.lang.Object

#### otherEndPoint

public [TGPoint](TGPoint.html" \o "class in <Unnamed>) otherEndPoint(double radians, double length)

Given one end point of a line, its length and heading (in radians), return its other end point. The idea of rounding deltaX and deltaY to zero when close came from Berkeley Logo - absolutely necessary to make graphics look pretty

#### xDoubleValue

public double xDoubleValue()

Returns the X coordinate of this point in double precision.

#### xLongValue

public long xLongValue()

Returns the X coordinate of this point as a long int.

#### yDoubleValue

public double yDoubleValue()

Returns the Y coordinate of this point in double precision.

#### yLongValue

public long yLongValue()

Returns the Y coordinate of this point as a long int.

#### toString

public java.lang.String toString()

Returns the String representation of this point. The coordinates are enclosed in eliptical brackets and separated with a comma.

**Overrides:**

toString in class java.lang.Object