COSC-112-S18
Annabelle Gary, Sylvia Frank, Amy Pass
Final Project Specifications

a-MAZE-ing Maze!!

Some comments that we received from our (many) Beta-testers:

"It was exhilarating!" –Maya '20
"That was more fun than Fifa [the best-selling console title in the world in 2016]" —Jimmy
"It's better than anything I've ever built." –Andrew '19 (Junior Computer Science major)
"This is really hard! I literally can't stop playing." -Katie '21
"No no no, let me play again!" -Oscar '19
"If I didn't have final papers to do, I would play this for hours." -Olivia '21

HIGH LEVEL DESCRIPTION

Our program is a maze game from an aerial perspective. The user is a blue ball running along a grey path which twists and turns *ad infinitum* until the user dies, and the objective is to play for as long as you can and get the highest score. Death can be found in a number of ways—by falling off the path, by running into aliens and failing to kill them, or by hitting a bomb. Your score increases based on how long you play and the number of points you rack up by collecting coins or killing aliens. There are also boosters you can collect, which increase your ammo count, change your speed and size, and put up bumpers on the path so you cannot fall off.

The user controls the marble by using keyboard directional keys ('i'/'j'/'k'/'l' to move up/left/down/right respectively, and 'a' to shoot ammo). The game can be paused/resumed by hitting the 'h' key, which displays a help menu, and the game is started by pressing the spacebar. If you die, you either close the window using the escape key, or restart the game by hitting enter.

The program runs by constantly moving the path down the screen (unless you input keys to make the marble move, it stays in place, so it looks like it's moving up the screen). When you die, the movement stops, and a "You're dead!" message pops up. The help menu pauses the game, and you resume in the same place.

When you die, your score is compared to the current high score, and if you beat it, your score becomes the new high score. This score is written to a file, and the user is told the high score to beat at the beginning of every game.

The general purpose of the game is exactly that—to be a game. The program is a mindless game, intended to capture the user's attention, rendering them incapable of being adequately aware of the passage of time. Our ultimate goal is to decrease world productivity such that we may take control of all major geopolitical institutions and use them to do our bidding. Muah ha ha ha.

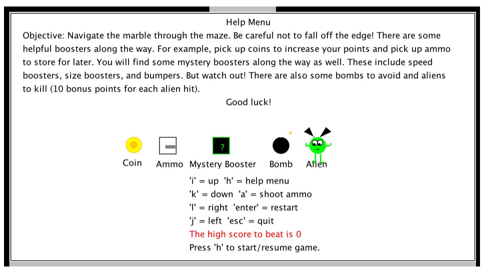
(Just kidding! We want people to have fun, and feel retro when they play this old-school looking game. No ulterior motives whatsoever).

SPECIFIC DESCRIPTION

`public class Game`

- Main class (what the user runs after compiling)

- One instance is created when `<java Game>` is run.
- `extends JPanel`
  - When run, a `JPanel` pops up, which is where the game is played
- `implements KeyListener`
  - When keys are hit, the program responds. More on that below.
- Member variables:
  - NOTE: all these variables are public because they must be accessible from every other class and method, and they are static because there is only one instance of Game
  - `public static final int WIDTH; public static final int HEIGHT`
    - to get passed to the JPanel and paintComponent, to specify size of window that pops up
  - `public static int FPS = 90;`
    - variable specifying Frames Per Second, which is passed to the Runner method and determines how many times per second things are updated.
    - This is not final because we change it after a specified number of points are collected
  - `public static World world;`
    - A public static instance of class World is created so you can access it from any class and any method
  - `public static boolean alive;`
    - The boolean that is passed to the `while` loop in the `Runner` class. Various methods deeper in the game can set this to `false` (which is why it's a static variable
  - `public static boolean hasGameStarted;`
    - starts as `false`, and gets set to `true` when the spacebar is hit
    - begins the `while` loop in `Runner`, which runs the game
  - `public static boolean paused;`
    - gets set to true when 'h' is pressed, which draws the help menu and pauses the runner until the user resumes playing
    - the runner method only runs when `paused` is false
  - `public static boolean helpDrawn;`
    - The boolean used in paintComponent that determines whether the help menu should be drawn or not; this is changed by the user pressing 'h', which pauses the `Runner` method and draws the help menu to the screen



Help Menu

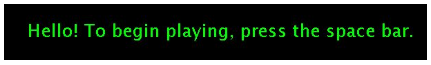
Objective: Navigate the marble through the maze. Be careful not to fall off the edge! There are some helpful boosters along the way. For example, pick up coins to increase your points and pick up ammo to store for later. You will find some mystery boosters along the way as well. These include speed boosters, size boosters, and bumpers. But watch out! There are also some bombs to avoid and aliens to kill (10 bonus points for each alien hit).

Good luck!

'i' = up 'h' = help menu
'k' = down 'a' = shoot ammo
'l' = right 'enter' = restart
'j' = left 'esc' = quit
The high score to beat is 0
Press 'h' to start/resume game.

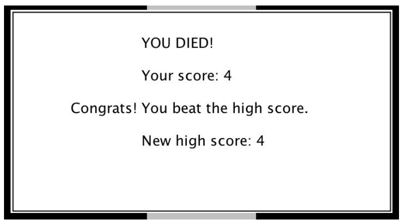
  - `public static boolean pressed;`

    - if any key is pressed, the Runner calls the method which moves the marble
  - `public static boolean ipressed/jpressed/kpressed/lpressed`
    - specifies which directional key is being pressed down (set to `true` in `keyPressed()`, `false` in `keyReleased()`)
    - allows you to move diagonally
    - allows you to shoot ammo while you move
- Constructor
  - One constructor which takes no input, and instantiates all the member variables—the World gets created with the specified WIDTH and HEIGHT, etc.
  - A `keyListener` gets added, `Focus` gets requested, and a `mainThread` is created using the runner so that the `JPanel` thing works.
- Internal Classes
  - `class Runner implements Runnable;`
    - Contains the `run` method, which paints everything to the JPanel if the game has started (`hasGameStarted = true`) and is running (`alive == true, paused == false`)
    - When the game is over (`alive == false`), the `Logger` class writes your score to a file
- Methods
  - `void keyPressed, void keyTyped, void keyReleased`
    - these methods, which come from the `keyListener` interface, specify what the program should do when certain keys are typed.
      - 'i' sets `ipressed = true`
      - 'j' sets `jpressed = true`
      - 'k' sets `kpressed = true`
      - 'l' sets `lpressed = true`
      - 'a' calls the `shootAmmo` method in world
      - 'h' to draw the help menu and pause the `run` method
  - `void paintComponent`
    - if `!hasGameStarted`, write the start message to the screen:



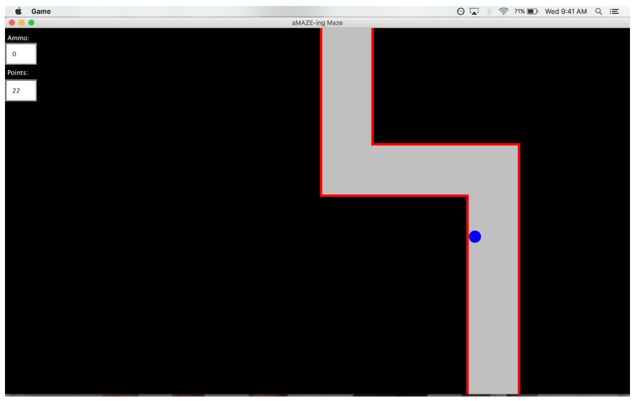
```
Hello! To begin playing, press the space bar.
```

    - if `!alive`, write the death message to the screen:



```
YOU DIED!
Your score: 4
Congrats! You beat the high score.
New high score: 4
```

    - if `hasGameStarted`, call `world.drawToScreen()`, which draws everything aside from the black background to the screen



- `void main(String [] args)`
  - creates an instance of Game, and creates the JFrame.

`public class World`

- One static instance (in Game)
- Member variables
  - `static Marble marble;`
    - an instance of the Marble class; only one is created; this is what the user controls
  - `static int ammoCount;`
    - a counter of the number of ammo you currently have; you increase this by picking up ammo items
    - it's static so you can increase it from the `Item` and `AmmoReleased` classes
  - `static double points;`
    - a counter of the number of points, which increases as time goes on and the user completes actions (killing aliens and collecting coins); this number is drawn to the screen
    - it's static so it can be accessed by `Game`, which writes `points` to the highscore file once you die
  - `static boolean ammoReleased;`
    - this boolean allows for efficiency, so the `updateAmmoReleased` method is only run when there is active ammo on the screen (`ammoReleased` boolean is true when you shoot by hitting 'a')
  - `static Map map;`
    - an instance of the Map class; only one is created; this contains all the paths that show up on the screen
  - `static LinkedList<LinkedList<Path>> visibleScreens;`
    - a collection of the pre-made screens (which are collections of paths) which are currently on the screen
    - There are only ever 2 or 3 in here at a given time, because once one goes off screen it gets removed from `visibleScreens` and another gets generated
  - `static LinkedList<AmmoReleased> ammoActive;`

        - This is a collection of the current `AmmoReleased` on the screen, so you know how many are active; the `updateAmmoReleased` method runs through this LinkedList to update each `AmmoReleased`
    - `static Random rand;`
        - instance of random, which we use to pass to different methods so they randomize this (e.g. `generateNextItem`)
    - `static boolean bumpersOn;`
        - a boolean which tells the marble if bumpers are on or not; it is turned on or off by the `Bumpers` class, when the user picks up a Bumpers booster.
        - if `!bumpersOn`, the marble checks if it's dead by seeing if it's on a path; otherwise, it checks for bumpers, which means it can't go off the map
    - `static LinkedList<Item> itemsActive;`
        - a collection of the items (of class `Item`) that are on the screen at any given time; in the World constructor, a new item is generated and added to the list
- Constructor
    - takes no inputs; only one
    - instantiates all the member variables; generates a new item and adds it to `itemsActive`
    - adds `Map.upcomingPath` to `visibleScreens`, which is a few straight paths in a row to get the user started
- Methods
    - `drawToScreen(Graphics g);`
        - called in `Game.paintComponent` if `Game.alive` and `game.hasGameStarted`
        - calls all the draw methods for the member variables (`path, marble, item, points, ammoCount, ammoReleased, helpMenu)` and passes the Graphics to each
            - each of these calls the draw method for the actual instance of each member variable
                - eg. `drawMarble (Graphics g){marble.draw(g);}`
    - `nextFrame(double time);`
        - called in the `Runner` of `Game`
        - calls all the update methods for the `World` member variables (eg. `updateMarble`, `updateItem`, etc)
            - each of these methods calls each individual member variable's update method
                - eg. `updateMap(){map.update();}`
    - `moveMarble();`
        - gets called if `Game.pressed` is true (gets set to true when a key is pressed)
        - checks if i, j, k, and/or l is pressed based on booleans in `Game`, and calls corresponding method from `marble` which changes `marble` position accordingly
    - `shootAmmo();`
        - gets called from `Game` when the 'a' key is pressed

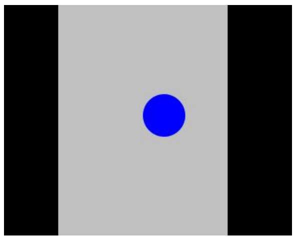
        - if `ammoCount > 0`, then ammo is shot out of the marble

`public class Map`

- One instance (in World)
- Member variables
    - `public static LinkedList<Path> initialScreen;`
        - A linked list consisting of paths. This linked list contains the first paths that appear on screen in a new game (6 straight paths); i.e. it is only used once
    - `public static LinkedList<Path> Screen1;`
    - `public static LinkedList<Path> Screen2;`
    - `public static LinkedList<Path> Screen3;`
    - `public static LinkedList<Path> Screen4;`
        - Map 1 through 4 are linked lists consisting of paths. Each "map" consists of a number of paths that start and end at the same place. The idea is that each "map" feeds into the next randomly, creating an infinite series of paths.
    - `public static LinkedList<LinkedList<Path>> allScreens;`
        - allScreens is a linked list of linked lists of paths; it will contain Map 1 through 4
    - `public static Random rand;`
        - An instance of random that is used elsewhere in the class to generate random ints
- Constructor:
    - One constructor which takes no input, and instantiates all the member variables
    - Adds 6 straight paths to `initialScreen`
    - Calls the `addMaps();` method
- Methods
    - `public static void addMaps()`
        - adds `initialScreen` and Map 1 through 4 to `allScreens`
    - `public void update()`
        - Method gets called every frame
        - Checks to see if the last path if the last path in the 0th screen in `visibleScreens`
    - `public static void draw(Graphics g)`
        - Loops through all of the paths in `World.visibleScreens` and draws them
    - `public static LinkedList<Path> getScreen1(LinkedList<Path> curr)`
        - Creates a new `LinkedList<Path>` and adds a series of `Paths` to it so that the first instance in the list's 'x' value is the same as the that of the last instance; then it returns the `LinkedList<Path>`
    - `public static LinkedList<Path> getScreen2(LinkedList<Path> curr)`

        - Creates a new `LinkedList<Path>` and adds a series of `Path`s to it so that the first instance in the list's 'x' value is the same as the that of the last instance; then it returns the `LinkedList<Path>`
    - `public static LinkedList<Path> getScreen3(LinkedList<Path> curr)`
        - Creates a new `LinkedList<Path>` and adds a series of `Path`s to it so that the first instance in the list's 'x' value is the same as the that of the last instance; then it returns the `LinkedList<Path>`
    - `public static LinkedList<Path> getScreen4(LinkedList<Path> curr)`
        - Creates a new `LinkedList<Path>` and adds a series of `Path`s to it so that the first instance in the list's 'x' value is the same as the that of the last instance; then it returns the `LinkedList<Path>`
    - `public static LinkedList<Path> generateNext()`
        - Permutes a random integer between 1 and 4 and then returns the corresponding screen

`public class Marble`



- One instance (in World)
- Member variables:
    - `Pair position;`
        - an instance of the `Pair` class; holds the marble's 'x' and 'y' values
    - `int diameter;`
        - the width of the marble
        - temporarily changed by the ChangeSize booster
    - `Color color;`
        - the color of the marble (Color.BLUE)
    - `double XposIncrement;`
        - the change in 'x' value when `moveRight()` and `moveLeft()` are called; the change in 'y' value then `moveUp()` and `moveDown()` are called
    - `double YposIncrement;`
        - the change in 'y' value when `moveRight()` and `moveLeft()` are called
- Constructor:
    - One constructor which takes no input, and instantiates all the member variables

- Internal Classes
    - `class Pair`
        - Member Variables:
            - `public double x;`
            - `public double y;`
        - Constructor:
            - takes in `double initX` and `double initY`
            - sets `x` to `initX` and `y` to `initY`
- Methods
    - `public void draw(Graphics g)`
        - draws the marble based on its `x` and `y` values and `diameter`
    - `public void moveUp()`
        - if the marble has not exceeded the upper bound of the limit of its movement (`Game.HEIGHT / 4`) then the position is decremented by `XposIncrement`
    - `public void moveDown()`
        - if the marble has not exceeded the lower bound of the limit of its movement (`Game.HEIGHT / 4 * 3`) then the position is incremented by `XposIncrement`
    - `public void moveRight()`
        - the marble's `x` value is incremented by `XposIncrement`; the marble's `y` value is incremented by `YposIncrement`
    - `public void moveLeft()`
        - the marble's `x` value is decremented by `XposIncrement`; the marble's `y` value is incremented by `YposIncrement`
    - `public static void checkForBumpers()`
        - based on the `Path` returned in `checkPath()`, it ensures that the marble does not exceed the limits of the path by incrementing or decrementing the marble's `x` and `y` values accordingly
    - `public static void checkDead`
        - based on the `Path` returned in `checkPath()`, it sets `Game.alive` to false if the marble's `x` or `y` values exceed the limits of the path
    - `public static Path checkPath()`
        - uses the marble's `x` and `y` values to determine which instance of `Path` from `World.visibleScreens` the marble is in

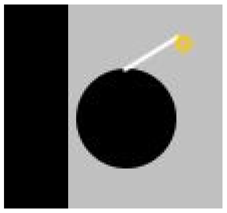
`class Item`

- There are many instances of Item subclasses, but no instances of Item
    - new instances of Item get created in updateItem in World through the generateNext method
- Item passes a number of variables and methods to its children:
- Member variables:
    - `public int x, public int y;`

            - the initial location on screen of the item
    - `static Random rand;`
        - used to randomize whether changeSize and changeSpeed increase or decrease
        - used to randomize time until the next item appears
    - `public static final double originalTimeUntilNextItem = 4;`
        - items will appear a minimum of 4 seconds apart from each other
    - `public double timeUntilNextItem;`
        - originalTime plus random number of seconds to generate the next item
    - `public boolean activated;`
        - used in each Item (turned on in `update()` if the marble passes over the booster; turned off in `deactivate()`)
    - `public boolean passed;`
        - set to `true` in `update()` if the x/y of marble and booster match up (if the marble passes over the booster), `false` otherwise
    - `public boolean drawn;`
        - boolean that tells draw method whether or not an item should be drawn to the screen (if `true`, draw, else don't)
        - initialized to `true`; set to `false` if item subclass has `pickUp()` method
- Constructor:
    - For Item and for every Item subclass, there is only one constructor. This is because the only thing an Item needs to be initialized is a location, as every other variable it might need is static (eg. World.ammoCount, World.points).
    - The constructor takes a location and initializes all the above variables
- Methods:
    - `public void update();`
        - called from `World.updateItem()`, which is called from `World.nextFrame()` (which is called every frame in Game)
        - moves `Item` down the screen at the same rate the `Map` is moving (so it looks like `Item` is staying in place)
        - checks the marbles position in relation to the `Item`; if the marble passes it, activate the `Item`
            - each subclass has a specific `activate()` method
        - decreases `timeUntilNextItem` based on `Game.FPS`
    - `public static Item generateNextItem(int randNum);`
        - gets called when `timeUntilNextItem` is 0 from `updateItem()` in `World`
        - gets passed a random Number, which determines which item will spawn
        - takes the topmost visible screen in `visibleScreens`, and if, at time of generation, the item would be in an acceptable x/y range (aka on the path), an item gets initialized and returned at that location
            - 0 = Bumpers, 1 = Bomb, 2 = changeSpeed, 3 = changeSize, 4 = Alien, 5 = Coin, 6 = Ammo
    - `public void pickup();`

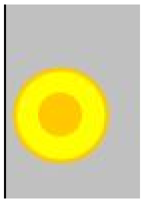
        - makes `drawn = false` (so the item is not drawn to screen); gets called for certain subclasses when the marble passes over the item
    - `public void draw(Graphics g);`
        - empty method, which gets overridden in subclasses
    - `public void activate();`
        - empty method, which gets overridden in subclasses

`class Bomb extends Item`



- A class which, when activated, kills you instantly and ends the game. It cannot be detonated, but it can be avoided.
- Inherits methods and variables from Item class, but overrides a bunch.
- New/overridden member variables:
    - `boolean exploded;`
        - `true` if activated, `false` otherwise
        - used to tell `draw()` what version to draw (bomb shape if `!exploded`, else explosion shape BOOM)
- Constructor:
    - takes in x, y values to send to `super()`
    - sets `exploded = false`
- Overridden methods:
    - `public void draw(Graphics g);`
        - draws a Bomb shape if `!exploded`
        - draws an explosion shape if `exploded`
    - `public void activate();`
        - if the bomb is activated (which is set and called in `super.update()`), then `Game.alive = false, exploded = true`

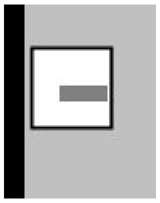
`class Coin extends Item`



- Item that increases your points by a random amount when you pass over it
- New member variables:
    - `int increase;`
        - set to a random int in constructor

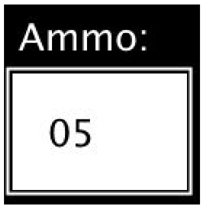
- Constructor:
  - takes in x, y values to send to `super()`
  - sets `increase` to a random integer (3 < x < 6)
- Overridden methods:
  - `public void draw(Graphics g);`
    - draws a coin shape at specified location
  - `public void activate();`
    - increases your `World.points` by `increase` value
    - deactivates item (`this.activated = false`)
    - calls `pickUp()` so the item disappears once you pass over it
    - calls `deactivate()`
  - `public void deactivate()`
    - sets increase to 0
      - This happens because otherwise, your points increase for as many frames as you're over the coin item

`class Ammo extends Item`



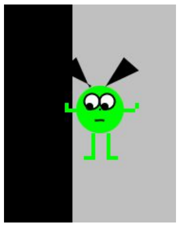
- An item that increases your ammo count when picked up, and draws the `ammoCounter` (a little box at the top of the screen with `ammoCount`)
- New member variables:
  - `int increase;`
    - how much you increase ammoCount by; set randomly in constructor from 1-3
  - `static final int counterHeight, counterWidth;`
    - height and width of the `ammoCounter` box that is drawn to the screen
- Constructor:
  - takes in x, y values to send to `super()`
  - sets increase to random number 1 < x < 3
- Overridden methods:
  - `public void draw(Graphics g);`
    - draws a little ammo box shape at specified location
  - `public void activate();`
    - increases your `World.ammoCount` by `increase` value
    - deactivates item (`this.activated = false`)
    - calls `pickUp()` so the item disappears once you pass over it
    - calls `deactivate()`
  - `public void deactivate()`
    - sets increase to 0

            - This happens because otherwise, your ammoCount increases for as many frames as you're over the ammo item
- New methods
    - `public static void drawAmmoCounter(Graphics g);`



        - draws a box in the upper left corner which displays your current `ammoCount`
        - uses `StringBuilder` library to append a "0" to the beginning of the `ammoCount` value

`class Alien extends Item`

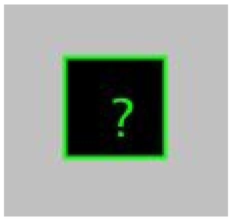


- An enemy which kills you if you pass over it; however, you can use the ammo to shoot it, which kills it and renders it harmless
- New member variables:
    - `boolean deadly;`
        - `true` when the alien is alive, `false` when dead
- Constructor:
    - takes in x, y values to send to `super()`
    - sets `deadly` to true (meaning Alien is alive)
- Overridden methods:
    - `public void draw(Graphics g);`
        - draws an alien that is the cutest thing we've ever seen
            - if `!deadly` (the alien is dead), draw the eyes and mouth differently so the alien looks dead

    - `public void activate();`
        - kills you if you pass over the Item and the Item is deadly, otherwise does nothing
    - `public void deactivate(int ammoActiveIndex)`
        - sets `deadly` to be `false`

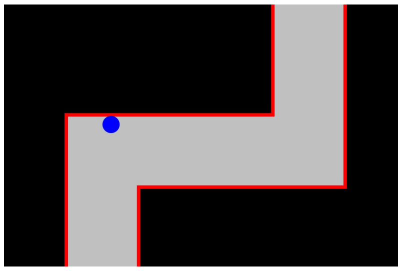
        - deactivates the `ammoReleased` at the given index (so it knows which one killed the alien)
        - increases `World.points` by 10
    - `public void update()`
        - calls super.update() (to see if the marble is passing over the `Alien`)
        - checks if any individual `ammoReleased` is going over the `Alien` by comparing x and y values; if so, sets `deadly` to `false`

`class Booster extends Item`



- This class, which has its own subclasses, is special because all subclasses of Booster are time-sensitive (they deactivate after a given amount of time).
- Constructor:
    - takes in x, y values to send to `super()`
    - sets `activated to false`
    - sets `deactivateTime to 5`
- Overridden Methods:
    - `public void draw(Graphics g);`
        - draws box with a question mark in it
    - `public void activate();`
        - sets `this.activated` to `true`, so the subclass's `activate()` method runs
    - `public void deactivate();`
        - sets `this.activated` to `false`
    - `public void update();`
        - calls `super.update()` to check if marble is over the `Item`
        - decreases `deactivateTime` while the item is activated, and deactivates once `deactivateTime` is 0

`class Bumpers extends Booster`



- This booster adds `Bumpers` to the edges of the path, so that you are unable to die by going off the path. It does this by setting a static variable in `World` (`bumpersOn`) to be true when activated and false when time runs out. This then affects the `draw()` methods for each `Path` type, so that red rectangles show up on the edges.

- It also affects which method is called in Marble's `update()`. If (`bumpersOn`), `checkForBumpers()` is called, and if bumpers are off, `checkDead()` is called.
- Constructor:
    - takes in x, y values to send to `super()`
    - `deactivateTime` is set to 10
- Overridden Methods:
    - `public void activate();`
        - sets `World.bumpersOn = true`
    - `public void deactivate();`
        - sets `World.bumpersOn = false`
    - `public void update();`
        - calls `super.update()`
        - calls `checkTopEdge()`
- New methods:
    - `public void checkTopEdge();`
        - This method checks if the marble is hitting up against a horizontal bumper (where path is a `Horizontal` or a any type of `Corner`). If it is, it brings the marble down by one pixel (so it moves down at the same rate the map is moving down)

`class ChangeSpeed extends Booster`

- When this booster is activated, it randomly increases or decreases the speed of the marble. The speed returns to normal in 5 seconds.
- New member variables:
    - `boolean increase;`
        - set randomly in the Constructor; if true, marble speed is increased, otherwise it's decreased.
    - `double xincrement; double yincrement;`
        - the amount by which the value of how much the marble moves changes
    - `double originalXIncrement; double originalYIncrement;`
        - the amounts by which the marble moves at the moment that the marble activates the booster; used to return the marble speed to its previous value in `deactivate()`
- Constructor:
    - takes in x, y values to send to `super()`
    - `deactivateTime` is set to 7
    - sets `increase` to a random `true/false` value
    - sets `xincrement/yincrement` to be proportional to the current incrementation
    - gets current incrementation values
- Overridden methods:
    - `public void activate();`

        - if `increase` is `true`, add to the x and y increments to increase speed of marble (if `false`, subtract to decrease speed)
        - then immediately set `xincrement/yincrement` to 0, since `activate()` runs as long as the marble is over the Booster (so it's speed will either speed too much up or slow to a stop)
    - `public void deactivate();`
        - reset the marble's x and y increments to the values you saved in the constructor

`class ChangeSize extends Booster`

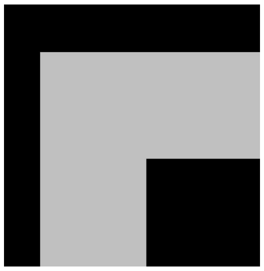
- When this booster is activated, it randomly increases or decreases the size of the marble. The size returns to normal in 5 seconds.
- New member variables:
    - `boolean increase;`
        - set randomly in the Constructor; if true, marble size is increased, otherwise it's decreased.
    - `double proportion;`
        - the amount by which the value of marble's diameter changes
    - `double originalSize;`
        - the diameter of the marble at the moment that the marble activates the booster; used to return the marble size to its previous value in `deactivate()`
- Constructor:
    - takes in x, y values to send to `super()`
    - `deactivateTime` is set to 7
    - sets `increase` to a random `true/false` value
    - sets `proportion` to be proportional to the current size
    - gets current diameter value
- Overridden methods:
    - `public void activate();`
        - if `increase` is `true`, add to the diameter to increase size of marble (if `false`, subtract to decrease size)
        - then immediately set `proportion` to 0, since `activate()` runs as long as the marble is over the Booster (so it's size will either get massive and go off the edge of the path or get so small the marble disappears)
    - `public void deactivate();`
        - reset the marble's diameter to the value you saved in the constructor

`class Path`

- There are many instances of Path subclasses, but no instances of Path
    - new instances of Path get created in the `getScreen1()-getScreen4()` methods, which are called in the `generateNext()` method in the `Map` class.
- Path passes a number of variables and methods to its children:
- Member variables:
    - `final int bumperWidth`

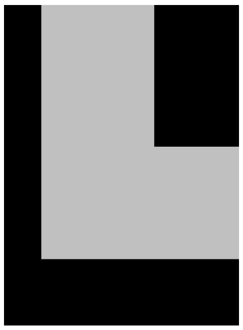
        - This variable represents the width of the bumpers (when they are turned on). The variable is final because it stays the same for the entire game, and should never be changed.
    - `static final int WIDTH`
        - This variable represents the width of a path. The variable is final because it is the same for the entire game and for all of the subclasses of path. It is static because it needs to be accessed in classes outside of Path.
    - `static final int HEIGHT`
        - This variable represents the height of a path. The variable is final because it is the same for the entire game and for all of the subclasses of path. It is static because it needs to be accessed in classes outside of Path.
    - `int x;`
        - This variable corresponds to the x value of the path's leftmost side of the path's exit
    - `int y;`
        - This variable corresponds to the y value of the path's upper-left corner
    - `int enterX;`
        - This variable corresponds to the x value of the bottom-left corner (i.e. the x value that the marble enters the path on)
    - `String name;`
        - This variable corresponds to the path's name; will be overridden in Path's subclasses
    - `final Color color;`
        - This variable corresponds to the color of the Path. It is final because it remains the same for the entire game.
- Constructors: (there are 2)
    - `public Path (int enterX);`
        - Initializes all of Path's member variables
        - Relies on enterX to specify aspects of the Path
    - `public Path (Path previous);`
        - Initializes all of Path's member variables
        - Relies on previous Path's member variables to specify aspects of current Path
    - There are two constructors because sometimes we need to create a Path depending on enterX (for example, the first Path in a Screen), while other times we need to create a Path depending on another Path (for example, how we move from path to path in generateNext()).
- Methods:
    - `public void draw (Graphics g)`
        - This method is not specified in Path; it is overridden in Path's subclasses

`class TopLeftCorner extends Path`



- The TopLeftCorner class is a subclass of Path. It represents a corner in which the marble is traveling up the screen in a straight direction, and must turn right (i.e. if you were looking at a square, it would be the top left corner of the square). It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a TopLeftCorner Path on the Jpanel.
- Constructors:
  - `public TopLeftCorner(int enterX)`
    - uses super(enterX) to retrieve member variables from Path
    - overrides Name variable
    - overrides x variable
  - `public TopLeftCorner(Path previous)`
    - uses super(previous) to retrieve member variables from Path
    - overrides Name variable
    - overrides enterX variable
    - overrides x variable
    - overrides y variable
- Overriden methods:
  - `public void draw (Graphics g0ri)`
    - draws the TopLeftCorner Path based on x and y member variables
    - if boolean bumpersOn is true, then draws bumpers

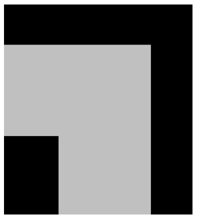
`class BottomLeftCorner extends Path`



- The BottomLeftCorner class is a subclass of Path. It represents a corner in which the marble is traveling across the screen horizontally, and must turn right(i.e. if you were looking at a square, it would be the bottom left corner of the square). It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a BottomLeftCorner Path on the Jpanel.
- Constructors:
  - `public BottomLeftCorner(int enterX)`

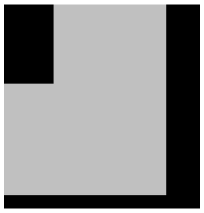
        - uses super(enterX) to retrieve member variables from Path
        - overrides Name variable
        - overrides x variable
    - `public BottomLeftCorner(Path previous)`
        - uses super(previous) to retrieve member variables from Path
        - overrides Name variable
        - overrides enterX variable
        - overrides x variable
        - overrides y variable
- Overriden methods:
    - `public void draw (Graphics gOri)`
        - draws the BottomLeftCorner Path based on x and y member variables
        - if boolean bumpersOn is true, then draws bumpers

`class TopRightCorner extends Path`



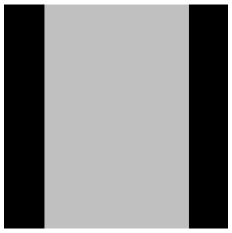
- The TopRightCorner class is a subclass of Path. It represents a corner in which the marble is traveling up the screen in a straight direction, and must turn left (i.e. if you were looking at a square, it would be the top right corner of the square). It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a TopRightCorner Path on the Jpanel.
- Constructors:
    - `public TopRightCorner(int enterX)`
        - uses super(enterX) to retrieve member variables from Path
        - overrides Name variable
        - overrides x variable
    - `public TopRightCorner(Path previous)`
        - uses super(previous) to retrieve member variables from Path
        - overrides Name variable
        - overrides enterX variable
        - overrides x variable
        - overrides y variable
- Overriden methods:
    - `public void draw (Graphics gOri)`
        - draws the TopRightCorner Path based on x and y member variables
        - if boolean bumpersOn is true, then draws bumpers

`class BottomRightCorner extends Path`



- The BottomRightCorner class is a subclass of Path. It represents a corner in which the marble is traveling across the screen horizontally, and must turn right(i.e. if you were looking at a square, it would be the bottom right corner of the square). It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a BottomRightCorner Path on the Jpanel.
- Constructors:
  - `public BottomRightCorner(int enterX)`
    - uses super(enterX) to retrieve member variables from Path
    - overrides Name variable
    - overrides x variable
  - `public BottomRightCorner(Path previous)`
    - uses super(previous) to retrieve member variables from Path
    - overrides Name variable
    - overrides enterX variable
    - overrides x variable
    - overrides y variable
- Overriden methods:
  - `public void draw (Graphics g0ri)`
    - draws the BottomRightCorner Path based on x and y member variables
    - if boolean bumpersOn is true, then draws bumpers

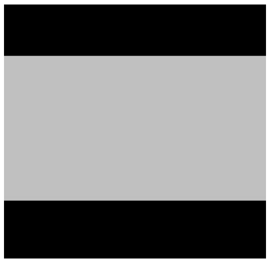
`class Straight extends Path`



- The Straight class is a subclass of Path. It represents a straight, vertical path. It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a Straight Path on the Jpanel.
- Constructors:
  - `public Straight(int enterX)`
    - uses super(enterX) to retrieve member variables from Path
    - overrides Name variable
    - overrides x variable
  - `public Straight(Path previous)`
    - uses super(previous) to retrieve member variables from Path

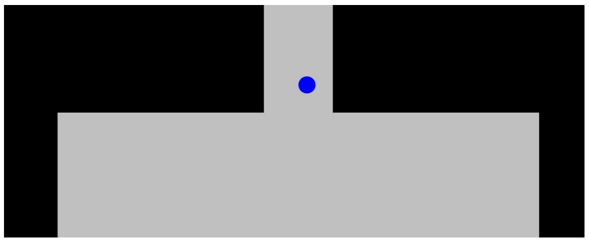
        - overrides Name variable
        - overrides enterX variable
        - overrides x variable
        - overrides y variable
- Overriden methods:
    - `public void draw (Graphics g0ri)`
        - draws the Straight Path based on x and y member variables
        - if boolean bumpersOn is true, then draws bumpers

`class Horizontal extends Path`



- The Horizontal class is a subclass of Path. It represents a straight, horizontal path. It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a Horizontal Path on the Jpanel.
- Constructors:
    - `public Horizontal(int enterX)`
        - uses super(enterX) to retrieve member variables from Path
        - overrides Name variable
        - overrides x variable
    - `public Horizontal(Path previous)`
        - uses super(previous) to retrieve member variables from Path
        - overrides Name variable
        - overrides enterX variable
        - overrides x variable
        - overrides y variable
- Overriden methods:
    - `public void draw (Graphics g0ri)`
        - draws the Horizontal Path based on x and y member variables
        - if boolean bumpersOn is true, then draws bumpers

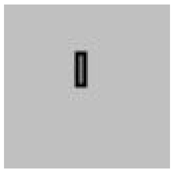
`class BigRect extends Path`



- The BigRect class is a subclass of Path. It represents a large rectangle that it is only drawn in the beginning of the game. It's purpose is to give the user some space to get used to the controls. It's main (and only) method is a draw method, that overrides Path's draw method, and draws the a large rectangular Path on the Jpanel.

- Constructors:
  - `public BigRect(int y)`
    - uses super(0) to retrieve member variables from Path
    - overrides Name variable
    - overrides x variable
    - overrides y variable
  - Only one constructor because BigRect is only created once at the beginning of the game (i.e. it never needs to follow another path)
- Overridden methods:
  - `public void draw (Graphics g)`
    - Draws a large rectangle at the bottom/center of the screen
    - Used exclusively to begin the game

`public class AmmoReleased`

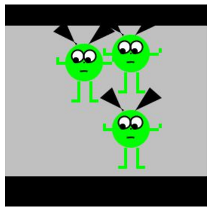


- There are many instances of AmmoReleased; a new instance is created each time 'a' is pressed
- Member Variables:
  - `int x;`
    - the 'x' value of the particular instance of `AmmoReleased`
  - `int y;`
    - the 'y' value of the particular instance of `AmmoReleased`
  - `int direction;`
    - designates the direction that the particular instance of `AmmoReleased` goes ([1] up; [2] right; [3] down; [4] left)
  - `int width;`
    - the width of the particular instance of `AmmoReleased`
  - `int length;`
    - the length of the particular instance of `AmmoReleased`
- Constructors:
  - takes in int x, int y, and int direction and sets each of those inputs to their corresponding member variables
  - determines the `width` and `length` member variables based on the `direction`; i.e. if the `direction` is up or down, then `length` > `width` or if the `direction` is left or right, then `length` < `width`

- Methods:
  - `public void draw(Graphics g)`
    - loops through `World.ammoActive` and draws each instance of `AmmoReleased` individually
  - `public void update()`
    - sets `World.ammoReleased` to false if `World.ammoActive` doesn't contain anything
    - increments or decrements the instance's `x` or `y` values, depending on its `direction`
    - if the instance has exceeded the limits of the screen, it calls `deactivate()`
  - `public static void activate()`
    - adds a new instance of `AmmoReleased` to `World.ammoActive`
    - decrements `World.ammoCount` by 1
  - `public static void deactivate()`
    - removes the first element of `World.ammoActive`
  - `public static void deactivate(int index)`
    - removes the `index` element from `World.ammoActive`

`public class TriggerEvent`

- When called, 1 of 3 random things could happen:
  - The path could speed up
  - A conglomeration of aliens could appear on the path to block your way



  - A mass of coins could appear on the path
- The class is called every 25 points, and is intended to make the game harder/more interesting as the game goes on
- Constructor
  - `public TriggerEvent(int randNum)`
    - Randomly picks one of the three trigger events to happen

`class Logger`

- File input/output class, which reads and writes high scores and displays them so the user can have a goal and a reason to keep playing.
- Both of its methods are static so you can access them from anywhere (and you don't have to create an instance of `Logger`).
- `readHighScore` and `writeHighScore` are called on the same file ("`highscore.txt`")
- Methods:

- `public static void writeHighScore(double points, String outFileName);`
    - calls the Logger's `readHighScore` method on `outFileName` to read in the current high score
    - compares the given points to the current high score; if points are more, use a `PrintWriter` to write the points to the file. Otherwise, write the current high score again.
- `public static double readHighScore(String inFileName);`
    - uses a `Scanner` to see if there's a double in `inFileName`, and if there is, return it
    - otherwise, return 0.0
    - Called to see what the current high score is