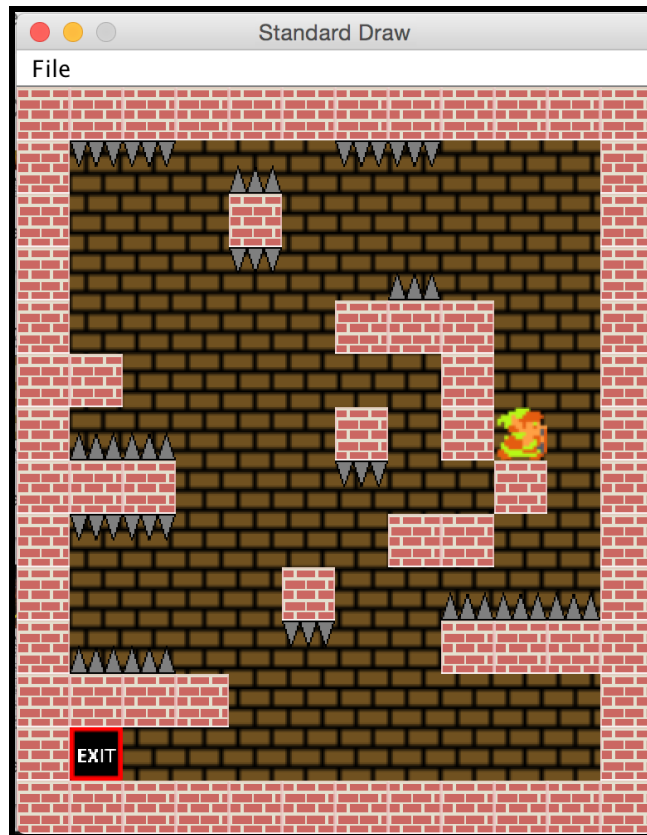


University of New Orleans - Department of Computer Science

PLATFORM JUMP

A Platformer Physics-based Gravity Action Game



Lab Summary

	Lab Overview	5 minutes
0	Planning: <i>What Game Objects are needed?</i>	10 minutes
1	Game class: <i>Architecture of Main Game Logic</i>	10 minutes
2	World class: <i>Manages all the level data, display it to console</i>	10 minutes
3	Scene class: <i>Manages a single Scene, display it to console</i>	10 minutes
4	GameObject class: <i>Manages all data common to game objects</i>	10 minutes
5	Block class: <i>Manages all data for a block, draw blocks</i>	10 minutes
6	Player class: <i>Manages all player data, draw player</i>	10 minutes
7	Physics class: <i>Manages all physics rules, add gravity</i>	10 minutes
8	Controller class: <i>Manages Player Controls, add jump</i>	10 minutes
9	Controller class: <i>Manages Player Controls, add left/right</i>	10 minutes
10	Physics class: <i>Check Floor Collisions</i>	10 minutes
11	Physics class: <i>Check Ceiling Collisions</i>	10 minutes
12	Physics class: <i>Check Left/Right Collisions</i>	10 minutes
13	Physics class: <i>Add friction</i>	10 minutes
14	FloorHazard, Ceiling Hazard classes: <i>Subclasses of Block</i>	10 minutes
15	Game class: <i>Check if Player is touching Hazard, game over</i>	10 minutes
16	Exit class: <i>Subclass of Block</i>	10 minutes
17	Game class: <i>Check if Player is touching exit, go to next level</i>	10 minutes
18	Animation class, Pose enum: <i>Add Player animations</i>	10 minutes

Introduction:


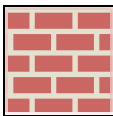
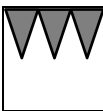
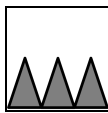

In this lab, you will build a platformer game. The player must jump on blocks to navigate to the exit. The player must avoid floor and ceiling hazards to prevent a game over. If the player gets to the exit another stage will load, until they exit the last stage and win. The player is affected by gravity and friction. This game is designed using inheritance and polymorphism to store common data for each type of related object.

Quick Explanation of Software Objects:

Inheritance and Polymorphism.

[image for conceptual reasons]	[image for conceptual reasons]
--------------------------------	--------------------------------

Game Objects (from Player Perspective):

	Player character The player can move left, right and jump up. The player is affected by physics i.e. gravity and friction.
	Brick Block The player collides with blocks and cannot move through them. These types of blocks build floor, walls, ceiling, and platforms
	Ceiling Hazard, (type of block) Player dies if they collide with the spike
	Floor Hazard, (type of block) Player dies if they collide with the spike
	Exit, (type of block) Player advances to next stage if they collide with this block

Version 0: Goal: Designing the Game classes.

Overview of Classes

Game	<i>Manages Game data and game methods</i>	GameObject	<i>Manages all common data for game objs</i>
World	<i>Manages All level setup data</i>	Player	<i>Extends GameObject to define Player</i>
Scene	<i>Manages Scene data and Scene methods</i>	Controller	<i>Manages Player inputs to move player</i>
Block	<i>Extends GameObject to define a Block</i>	Physics	<i>Manages physics: gravity, friction, velocity</i>
Exit	<i>Extends Block to define Exit</i>	Animation	<i>Manages which image to display</i>
Ceiling Hazard	<i>Extends Block to define Ceiling Hazard</i>	Pose	<i>Contains all possible animated poses</i>
Floor Hazard	<i>Extends Block to define Floor Hazard</i>	StdDraw	<i>Library Class for drawing graphics</i>

Config File (*worldData.txt*)

Contains data for all the levels in the game.

```

3
14 12
#####
#VV ! VV #
#   #   #
#   AA #
#   ### #
##    ## #
#AA   # # 
###   V # #
#VV   ## #
#   # AAA#
#AA  V ####
####      #
#@       #
#####
14 12
#####
#VV A VV #
#   #   #
#   V A  #
#   ### #
##     # #
#AA   #@ #
###   V # #
#VV   ## #
#   # AAA#
#AA  V ####
####      #
#!       #
#####
14 12
#####
#@       #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#        #
#AAAAAAA!#
#####
```

Number of Levels, THREE
Number of Rows: 14, Number of Columns: 12
Text representation of level 1, each character represents a tile type

- # WALL
- @ PLAYER
- ! EXIT
- V CEILING HAZARD
- A FLOOR HAZARD

Number of Rows: 14, Number of Columns: 12
Text representation of level 2, each character represents a tile type

- # WALL
- @ PLAYER
- ! EXIT
- V CEILING HAZARD
- A FLOOR HAZARD

Number of Rows: 14, Number of Columns: 12
Text representation of level 3, each character represents a tile type

- # WALL
- @ PLAYER
- ! EXIT
- V CEILING HAZARD
- A FLOOR HAZARD

GitLab Repo: *Clone this project to get necessary Assets for this Lab*

<https://gitlab.com/scalemaited/platformer-lab.git>

Goal 1: Create a Game class that manages all of the game data and game rules.

Planning:

Design the base architecture of Game System

Implementation:

Create a Game class & stub out the necessary methods and attributes to get a game loop to start executing.

Game.java → instance variables

```
public class Game {  
    private boolean isOver;  
}
```

Game.java → constructor

```
/*Create a new Platform Game*/  
public Game() {  
    this.isOver = false;  
}
```

Game.java → update()

```
public void update() {  
    //game update code  
}
```

Game.java → render()

```
public void render() {  
    //game draw code  
}
```

Game.java → main()

```
/* The main game loop*/  
public static void main(String[] args) {  
    Game game = new Game();  
    while (game.isOver == false) {  
        game.update();  
        game.render();  
    }  
}
```

Testing:

Try to compile your Game.java code. There is no reason to execute yet, as there is no logic in the program to evaluate. It will infinitely loop.

Goal 2: Save World data into a Multidimensional Array

Read the configuration file, display world data to console & save into array.

Planning:

Create a World class responsible for holding all the level data in the game. The world class gets its data from the configuration file and stores it into an array.

Implementation:

World.java → instance variables

```
import java.util.Scanner;

public class World {
    private String[][][] levels;
}
```

World.java → constructor

```
public World() {
    //get all map data and save it for later
    Scanner input = new Scanner(System.in);
    int count = input.nextInt();
    levels = new String[count][][];

    for (int lvl=0; lvl<count; lvl++) {
        int rows = input.nextInt();
        int cols = input.nextInt();
        input.nextLine();

        setLevel(lvl, rows, cols, input);
    }
}
```

World.java → setLevel()

```
private void setLevel(int lvl, int rows, int cols, Scanner input) {
    levels[lvl] = new String[rows][cols];

    input.useDelimiter("");
    for (int y=0; y < rows; y++) {
        for (int x=0; x < cols; x++) {
            String tile = input.next();
            levels[lvl][y][x] = tile;
            System.out.print(tile);
        }
        input.nextLine();
        System.out.print("\n");
    }
    input.reset();
}
```


Goal 3: Create a Scene class to display a level

Planning:

Scene → row, col data, that initially prints the scene to console

World → save all level data into 3 dim array, a collection of 2d maps (rows, columns)

Game → add level variable, start a Scene by passing it a 2d text representation of the level map

Implementation:

World.java → getLevel()

```
public String[][] getLevel(int level) {  
    return levels[level];  
}
```

Scene.java → instance variables

```
public class Scene {  
    private int rows;  
    private int cols;  
}
```

Scene.java → constructor

```
public Scene( String[][] map ) {  
    this.rows = map.length;  
    this.cols = map[0].length;  
  
    for (int y=0; y<rows; y++) {  
        for (int x=0; x<cols; x++) {  
            String tile = map[y][x];  
            System.out.print(tile);  
        }  
        System.out.print("\n");  
    }  
}
```

Game.java → instance variables

```
/*Platformer Game Attributes*/  
private World world;  
private boolean isOver;  
private int level;  
private Scene scene;
```


Game.java → constructor

```

/*Create a new Platform Game*/
public Game() {
    this.isOver = false;
    this.level = 0;
    this.world = new World();
    String[][] map = world.getLevel(level);
    this.scene = new Scene(map);
}

```

Testing:

```

#####
#V ! V  #
#  #    #
#    AA  #
#   ###  #
##   ##  #
#AA  #  # #
###  V  # #
#VV   ##  #
#   #  AAA#
#AA  V  ####
####    #
#@      #
#####

```

To Execute your Program:

```
java PlatformGame < worldData.txt
```

Expected Output:

The console should print the first level map.

Clean Up:

After testing, remove the `System.out.print` statements.

Goal 4: Define a model for all Game Objects**Planning:**

Determine the common data that all game objects share, build a class to maintain that data

Implementation:

Define a Game object that represents any type of object in the game. Then use it to build background object in scene

GameObject.java → instance variables

```
public class GameObject {  
    private double x;  
    private double y;  
    private int width;  
    private int height;  
    private String image;  
}
```

GameObject.java → constructor

```
public GameObject(double x, double y, int width, int height, String image) {  
    this.x = x;  
    this.y = y;  
    this.width = width;  
    this.height = height;  
    this.image = image;  
}
```

GameObject.java → draw() method

```
public void draw() {  
    double screenX = x + width/2;  
    double screenY = y + height/2;  
    StdDraw.picture(screenX, screenY, image, width, height);  
}
```

Scene.java → instance variables

```
/*Attributes: Instance Variables*/  
private int rows;  
private int cols;  
private GameObject background;
```

Scene.java → constructor

```

public Scene( String[][] map) {
    this.rows = map.length;
    this.cols = map[0].length;

    int width = cols * 32;
    int height = rows * 32;
    this.background = new GameObject(0, 0, width, height, "../Assets/background.png");

    for(int y=0; y<rows; y++) {
        for(int x=0; x<cols; x++) {
            String tile = map[y][x];
        }
    }

    StdDraw.setCanvasSize(width, height);
    StdDraw.setXscale(0.0,width);
    StdDraw.setYscale(height,0.0);
}

```

Scene.java → draw() method

```

public void draw() {
    background.draw();
}

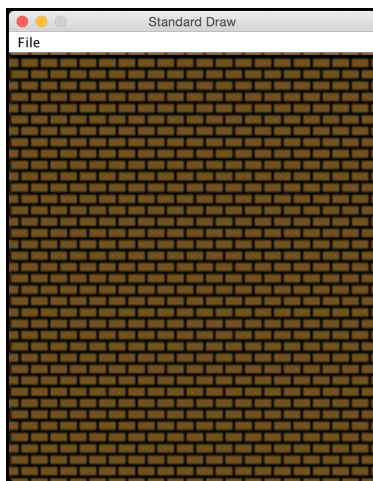
```

Game.java → render()

```

public void render() {
    scene.draw();
    StdDraw.show(10);
}

```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

GUI window display the background image

Goal 5: Model a Block class by extending the GameObject class

Planning:

Design the basic type of brick block in Game System

Implementation:

Create Block class by extending GameObject and draws blocks in scene

Block.java → instance variables

```
public class Block extends GameObject {
    public static final int SIZE = 32;
}
```

Block.java → constructor

```
public Block(double x, double y) {
    super( x*SIZE, y*SIZE, SIZE, SIZE, "../Assets/tile-brick.png");
}
```

Scene.java

```
import java.util.ArrayList;

public class Scene {
    ...
}
```

Scene.java → instance variables

```
/*Attributes: Instance Variables*/
private int rows;
private int cols;
private GameObject background;
private ArrayList<Block> blocks;
```

Scene.java → constructor

```
public Scene( String[][] map) {
    this.rows = map.length;
    this.cols = map[0].length;
    int width = cols * 32;
    int height = rows * 32;
    this.background = new GameObject(0, 0, width, height, "../Assets/background.png");
    this.blocks = new ArrayList<Block>();

    for(int y=0; y<rows; y++) {
        for(int x=0; x<cols; x++) {
            String tile = map[y][x];
            setTile( x, y, tile);
        }
    }

    StdDraw.setCanvasSize(width, height);
    StdDraw.setXscale(0.0,width);
    StdDraw.setYscale(height,0.0);
}
```

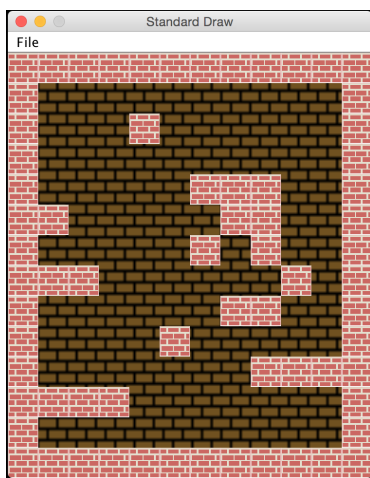
Scene.java → setTile() method

```
private void setTile(int x, int y, String tile) {  
    if (tile.equals("#") )  
    {  
        Block block = new Block(x, y);  
        this.blocks.add(block);  
    }  
}
```

Scene.java → draw() method

```
public void draw() {  
    background.draw();  
    for ( Block block : this.blocks) {  
        block.draw();  
    }  
}
```

Testing:



To Execute your Program:

```
java PlatformGame < worldData.txt
```

Expected Output:

GUI window display with:

- *Background image*
- **Brick blocks**

Goal 6: Model a Player class by extending the GameObject class

Planning:

Design the player class for use in the platformer game.

Implementation:

Create Player class by extending GameObject class and draw player in scene

Player.java

```
public class Player extends GameObject {  
}
```

Player.java → constructor

```
public Player(double x, double y) {  
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "../Assets/link-down.png");  
}
```

Scene.java → instance variables

```
/*Attributes: Instance Variables*/  
private int rows;  
private int cols;  
private GameObject background;  
private ArrayList<Block> blocks;  
private Player player;
```

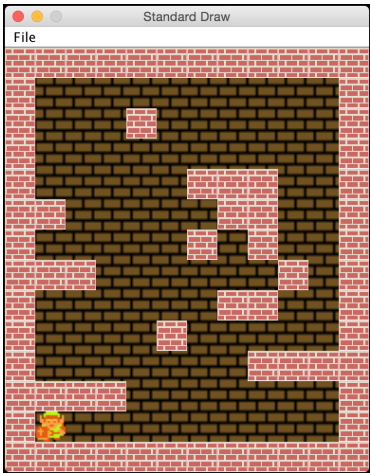
Scene.java → setTile() method

```
private void setTile(int x, int y, String tile) {  
    if (tile.equals("#") ) {  
        Block block = new Block(x, y);  
        this.blocks.add(block);  
    }  
    else if (tile.equals("@") ) {  
        this.player = new Player(x, y);  
    }  
}
```

Scene.java → draw() method

```
public void draw() {  
    background.draw();  
    for ( Block block : this.blocks) {  
        block.draw();  
    }  
    player.draw();  
}
```

Testing:

	<p>To Execute your Program:</p> <pre>java PlatformGame < worldData.txt</pre> <p>Expected Output:</p> <p><i>GUI window display with:</i></p> <ul style="list-style-type: none">- <i>Background image</i>- <i>Brick blocks</i>- <i>Player character</i>
---	---

Goal 7: Create a Physics class and add to Player as an attribute

Planning:

Design the physics system of a platform game. Gameplay relies on pushing the character with velocities in X-axis & Y-axis. Different forces influence the players velocity such as gravity, constantly pulling the player downward. Velocity affects the players position in the world, it represents the momentum they have for movement.

Implementation:

Create Physics class and add to player using composition

Physics.java → instance variables

```
public class Physics {  
    private int speed;  
    private double gravity;  
    private double terminal;  
    private double velocityX;  
    private double velocityY;  
}
```

Physics.java → Constructor

```
public Physics(int speed) {  
    this.speed = speed;  
    this.gravity = 0.3;  
    this.terminal = 8;  
    this.velocityX = 0.0;  
    this.velocityY = 0.0;  
}
```

Physics.java → applyGravity() method

```
public void applyGravity() {  
    if (velocityY < terminal) {  
        velocityY += gravity;  
    }  
}
```

Physics.java → update() method

```
public void update() {  
    applyGravity();  
}
```

Physics.java → getter methods

```
public double getVelocityX() {  
    return this.velocityX;  
}  
  
public double getVelocityY() {  
    return this.velocityY;  
}
```


GameObject.java → getter methods

```
public double getX() {  
    return this.x;  
}  
  
public double getY() {  
    return this.y;  
}
```

GameObject.java → move() method

```
public void move(double x, double y) {  
    this.x = x;  
    this.y = y;  
}
```

Player.java → instance variables

```
/*Attributes: Instance Variables*/  
private Physics physics;
```

Player.java → constructor

```
public Player(double x, double y) {  
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "../Assets/link-down.png");  
    this.physics = new Physics(4);  
}
```

Player.java → move() method

```
public void move() {  
    double dx = this.getX() + physics.getVelocityX();  
    double dy = this.getY() + physics.getVelocityY();  
    super.move(dx, dy);  
}
```

Player.java → update() method

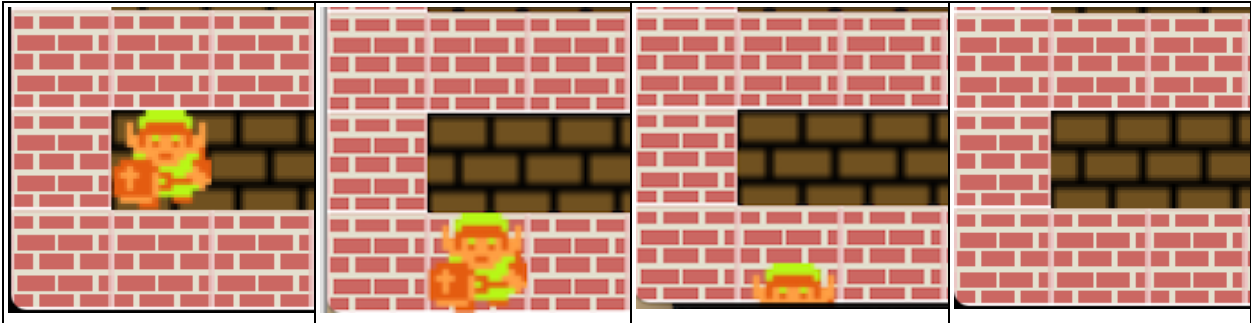
```
public void update() {  
    physics.update();  
    this.move();  
}
```

Scene.java → update() method

```
public void update() {  
    player.update();  
}
```

Game.java → update() method

```
public void update() {  
    scene.update();  
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

Character falls down, through the floor and off the screen.

Goal 8: Model a Controller class to trigger the Player to jump

Planning:

Design the controller of the game. We should separate the logic that handles user input from the logic that moves the character. Thus we may have different control schemes for the game without refactoring the code base. This is called a model-view-controller design pattern

Implementation:

Create a controller class that listens for player input and responds by issuing commands to the player instance. The controller invokes methods in the player to move it, and the player will relay those calls to its internal physics system which defines the rules for movement.

Physics.java → jump() method

```
public void jump() {  
    velocityY = -speed*2;  
}
```

Player.java → instance variables

```
/*Attributes: Instance Variables*/  
private Physics physics;  
private boolean isJumping;
```

Player.java → Constructor

```
public Player(double x, double y) {  
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "../Assets/link-down.png");  
    this.physics = new Physics(4);  
    this.isJumping = false;  
}
```

Player.java → jump() method

```
public void jump() {  
    if (this.isJumping == false) {  
        physics.jump();  
        this.isJumping = true;  
    }  
}
```

Controller.java → define class and data

```
public class Controller {  
    private Player player;  
}
```

Controller.java → Constructor

```
public Controller(Player player) {  
    this.player = player;  
}
```

Controller.java → keyboard() method

```
public void keyboard() {  
    //Jumping  
    if ( (StdDraw.isKeyPressed(38) || StdDraw.isKeyPressed(32) ) ) {  
        player.jump();  
    }  
}
```

Controller.java → update() method

```
public void update() {  
    keyboard();  
}
```

Scene.java → getPlayer() method

```
public Player getPlayer() {  
    return this.player;  
}
```

Game.java → instance variables

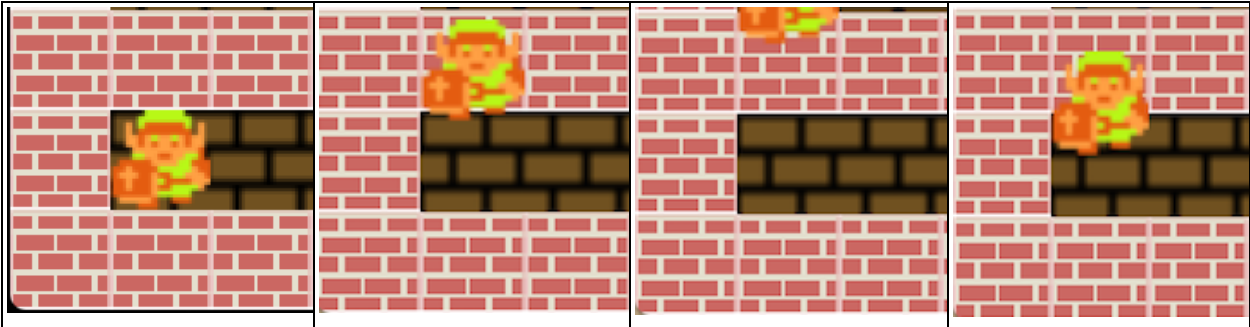
```
/*Platformer Game Attributes*/  
private World world;  
private boolean isOver;  
private int level;  
private Scene scene;  
private Controller controller;
```

Game.java → constructor

```
/*Create a new Platform Game*/  
public PlatformGame() {  
    this.isOver = false;  
    this.level = 0;  
    this.world = new World();  
    String[][] map = world.getLevel(level);  
    this.scene = new Scene(map);  
    Player player = scene.getPlayer();  
    this.controller = new Controller(player);  
}
```

Game.java → update() method

```
public void update() {  
    controller.update();  
    scene.update();  
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

When player presses up/spacebar then character jumps up, then falls back down, through the floor and off the screen.

Goal 9. Left & Right movement to player**Planning:**

Design horizontal movement of player. Requires updating the velocity X of character. We can add a positive number to velocity to go right or a negative number to go left. The more you press in a direction, the more the velocity should increase, moving the character faster in that direction.

Implementation:

Add methods that define leftward and rightward movement of character in physics class, then make calls to them in player so that the controller can invoke them.

Physics.java → moveLeft() method

```
public void moveLeft() {  
    if (velocityX > -speed) {  
        velocityX--;  
    }  
}
```

Physics.java → moveRight() method

```
public void moveRight() {  
    if (velocityX < speed) {  
        velocityX++;  
    }  
}
```

Player.java → moveLeft() method

```
public void moveLeft() {  
    physics.moveLeft();  
}
```

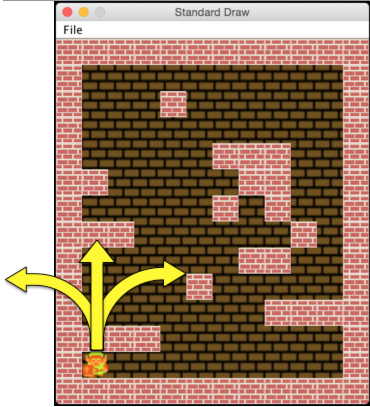
Player.java → moveRight() method

```
public void moveRight() {  
    physics.moveRight();  
}
```

Controller.java → keyboard() method

```
public void keyboard() {  
    //Jumping  
    if ( (StdDraw.isKeyPressed(38) || StdDraw.isKeyPressed(32) ) ) {  
        player.jump();  
    }  
    // player holding left  
    if (StdDraw.isKeyPressed(37) ) {  
        player.moveLeft();  
    }  
    // player holding right  
    if (StdDraw.isKeyPressed(39) ) {  
        player.moveRight();  
    }  
}
```

Testing:

	<p>To Execute your Program:</p> <pre>java PlatformGame < worldData.txt</pre> <p>Expected Output:</p> <p><i>When the player presses up/spacebar then character jumps up,</i> <i>When the player presses left/right they arc in that direction.</i> <i>Then falls back down, through the floor and off the screen.</i></p>
---	--

Goal 10: Floor collisions between Player & Blocks**Planning:**

Design the collisions for the game. There are two concepts: Are two game objects touching? Yes or no. Then if they are, which sides are they touching? Top, Left, Right, or Bottom. There will be four discrete points of collision, it's possible to have multiple sides colliding between two objects, such as when corners touch, that counts as both a vertical and horizontal touch.

Implementation:

Physics class should detect collisions between player and blocks before moving player down

GameObject.java → getter methods

```
public int getWidth() {  
    return this.width;  
}  
  
public int getHeight() {  
    return this.height;  
}
```

Block.java → isTouchingX() method

```
public boolean isTouchingX(GameObject gameObject, double ratio) {  
    double overlap = this.getWidth() * ratio;  
    return ( Math.abs( this.getX()-gameObject.getX() ) < overlap );  
}
```

Block.java → isTouchingY() method

```
public boolean isTouchingY(GameObject gameObject, double ratio) {  
    double overlap = this.getHeight() * ratio;  
    return ( Math.abs( this.getY()-gameObject.getY() ) < overlap );  
}
```

Block.java → isTouching() method

```
public boolean isTouching(GameObject gameObject) {  
    return isTouchingY(gameObject, 1.0) && isTouchingX(gameObject, 0.75);  
}
```

Physics.java

```
import java.util.ArrayList;  
  
public class Physics {  
    ...  
}
```


Physics.java → checkCollisions() method

```
public void checkCollisions(ArrayList<Block> blocks, Player player) {
    for (Block block : blocks) {
        if (block.isTouching(player) ) {
            checkCollisionFloor(block, player);
        }
    }
}
```

Physics.java → checkCollisionFloor method

```
public void checkCollisionFloor(Block block, Player player) {
    if (player.getY() < block.getY() && velocityY > 0 ) { //player higher than block & falling
        if ( block.isTouchingX(player, 0.5) ) { //ensure player and block on same column
            this.velocityY = 0;
            player.isJumping(false);
        }
    }
}
```

Physics.java → update() method

```
public void update(ArrayList<Block> blocks, Player player) {
    applyGravity();
    checkCollisions(blocks, player);
}
```

Player.java

```
import java.util.ArrayList;

public class Player {
    ...
}
```

Player.java → update() method

```
public void update(ArrayList<Block> blocks) {
    physics.update(blocks, this);
    this.move();
}
```

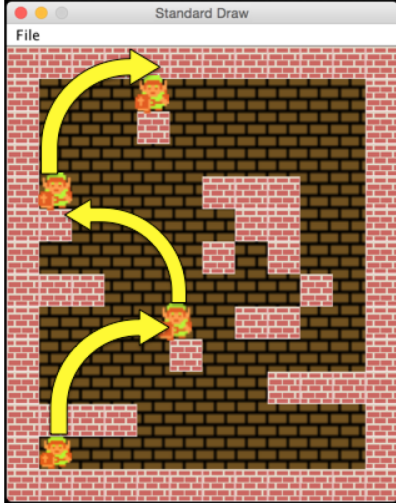
Player.java → isJumping() method

```
public void isJumping(boolean isJumping) {
    this.isJumping = isJumping;
}
```

Scene.java → update() method

```
public void update() {
    player.update(blocks);
}
```

Testing:

	<p>To Execute your Program:</p> <pre>java PlatformGame < worldData.txt</pre> <p>Expected Output:</p> <p><i>When player presses up/spacebar character jumps up, When the player presses left/right they move in that direction.</i></p> <p>Player lands on top of blocks.</p> <p>NOTE: No collisions on blocks' right, left or undersides. NOTE: No friction is applied so player continues moving, never stopping</p>
---	---

Goal 11: Ceiling collisions between Player & Blocks**Planning:**

A ceiling collision is when the player and block are touching, and the player is lower than the block (i.e. the player Y axis is larger than the block Y axis) and when the player's Y velocity is moving the player upward (i.e. the velocity Y is a negative number). In response, let's reduce the velocity and push the player in the opposite direction by multiplying the current velocity by a fractional negative number.

Implementation:

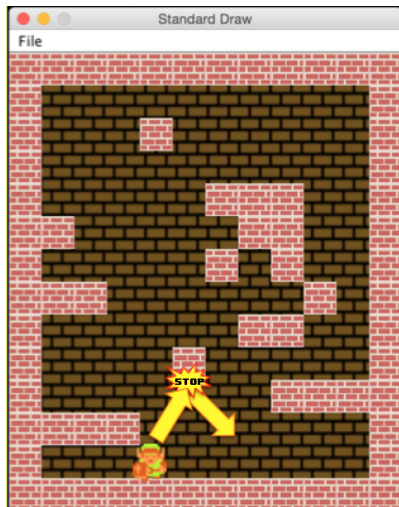
Physics class should detect collisions between player and blocks before moving player up

Physics.java → checkCollisionCeiling method

```
public void checkCollisionCeiling(Block block, Player player) {
    if (player.getY() > block.getY() && velocityY < 0) { //player lower than block & jumping
        this.velocityY *= -0.5;
    }
}
```

Physics.java → checkCollisions() method

```
public void checkCollisions(ArrayList<Block> blocks, Player player) {
    for (Block block : blocks) {
        if (block.isTouching(player) ) {
            checkCollisionFloor(block, player);
            checkCollisionCeiling(block, player);
        }
    }
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

When player presses up/spacebar character jumps up,
When the player presses left/right they move in that direction.
Player lands on top of blocks.

Player bounces back down when jumping under block

NOTE: No collisions on blocks' right, left.

NOTE: No friction is applied so player continues moving,
never stopping

Goal 12: Left/Right collisions between Player & Blocks**Planning:**

Define collisions on left/right sides. Three checks are necessary, first are two blocks touching, if so, then if the player X is left of block X and velocity is positive (i.e. moving rightward) then a right collision is occurring, however we want to avoid corner touches (i.e. touching a block on another row than the one the player is on) so check the Y position of both blocks to make sure they are adjacent horizontally.

Implementation:

Physics class should detect collisions between the player and blocks before moving the player left/right. Implement a horizontal check by decreasing the touch zone between two objects, at 50% on Y only objects on the same row will trigger a touch detection.

Physics.java → checkCollisionRight method

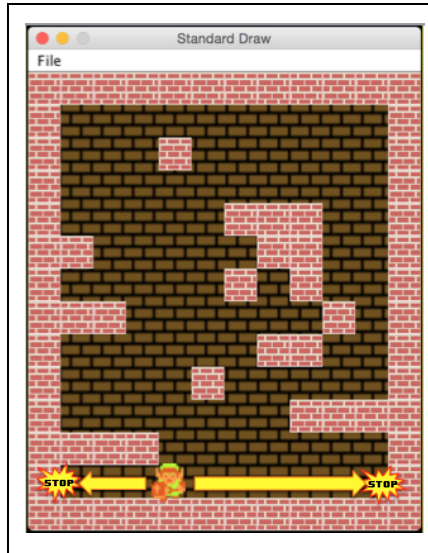
```
public void checkCollisionRight(Block block, Player player) {  
    if (player.getX() < block.getX() && velocityX > 0 ) { //player left of block & moving right  
        if ( block.isTouchingY(player, 0.5) ) { //ensure player and block on same row  
            this.velocityX *= -1;  
        }  
    }  
}
```

Physics.java → checkCollisionLeft method

```
public void checkCollisionLeft(Block block, Player player) {  
    if (player.getX() > block.getX() && velocityX < 0 ) { //player right of block & moving left  
        if ( block.isTouchingY(player, 0.5) ) { //ensure player and block on same row  
            this.velocityX *= -1;  
        }  
    }  
}
```

Physics.java → checkCollisions() method

```
public void checkCollisions(ArrayList<Block> blocks, Player player) {  
    for (Block block : blocks) {  
        if (block.isTouching(player) ) {  
            checkCollisionFloor(block, player);  
            checkCollisionCeiling(block, player);  
            checkCollisionRight(block, player);  
            checkCollisionLeft(block, player);  
        }  
    }  
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

*When player presses up/spacebar character jumps up,
When the player presses left/right they move in that direction.
Player lands on top of blocks.*

Player bounces back down when jumping under block

Player bounces off the left/right sides of blocks

NOTE: *No friction is applied so player continues moving,
never stopping*

Goal 13: Add friction to Physics**Planning:**

Define friction in Physics. Without friction, the player continues in horizontal motion indefinitely. Friction is the force that should be applied to the player's horizontal movement to slow them down to a stop. This can be accomplished by multiplying the velocity X by a value smaller than 1. Each time the friction is applied to the velocity X, it reduces it to a smaller value, meaning the player is moving slower than the previous update, until they finally stop.

Implementation:

Physics class should apply friction to player when moving horizontally

Physics.java → instance variables

```
private int speed;
private double gravity;
private double terminal;
private double velocityX;
private double velocityY;
private double friction;
```

Physics.java → Constructor

```
public Physics(int speed) {
    this.speed = speed;
    this.gravity = 0.3;
    this.terminal = 8;
    this.velocityX = 0.0;
    this.velocityY = 0.0;
    this.friction = 0.8;
}
```

Physics.java → applyFriction() method

```
public void applyFriction() {
    velocityX *= friction;
}
```

Physics.java → update() method

```
public void update(ArrayList<Block> blocks, Player player) {
    applyGravity();
    applyFriction();
    checkCollisions(blocks, player);
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

Friction is applied to the player's horizontal movement, slowing the player to a resting position.

Goal 14: Add Floor Hazards & Ceiling Hazards into Scene**Planning:**

Design the hazards in the game, hazards are a special type of block that can kill the player. They are half sized blocks so their collision detection behavior will be different than a standard block.

Implementation:

Inheritance to extend block to have Floor Hazard and Ceiling Hazard

Block.java → *Overloaded Constructor (i.e. keep original)*

```
public Block(double x, double y, String image) {
    super( x*SIZE, y*SIZE, SIZE, SIZE, image);
}
```

FloorHazard.java

```
public class FloorHazard extends Block {
    public FloorHazard(double x, double y) {
        super(x,y, "../Assets/tile-spikes-floor.png");
    }

    public boolean isTouching( GameObject player ) {
        return super.isTouchingX(player, 0.75) && this.isTouchingY(player);
    }

    public boolean isTouchingY( GameObject player ) {
        return this.getY()+this.getHeight()/2 <= player.getY() + player.getHeight()
            && player.getY() <= this.getY() + this.getHeight();
    }
}
```

CeilingHazard.java

```
public class CeilingHazard extends Block {
    public CeilingHazard(double x, double y) {
        super(x,y, "../Assets/tile-spikes-ceiling.png");
    }

    public boolean isTouching( GameObject player ) {
        return super.isTouchingX(player, 0.75) && this.isTouchingY(player);
    }

    public boolean isTouchingY( GameObject player ) {
        return this.getY() <= player.getY() + player.getHeight()
            && player.getY() <= this.getY() + this.getHeight()/2 ;
    }
}
```

Scene.java → *instance variables*

```
private int rows;
private int cols;
private GameObject background;
private ArrayList<Block> blocks;
private ArrayList<Block> monsters;
private Player player;
```

Scene.java → Constructor

```
public Scene( String[][] map) {
    this.rows = map.length;
    this.cols = map[0].length;
    int width = cols * 32;
    int height = rows * 32;
    this.background = new GameObject(0, 0, width, height, "../Assets/background.png");
    this.blocks = new ArrayList<Block>();
    this.monsters = new ArrayList<Block>();

    for(int y=0; y<rows; y++) {
        for(int x=0; x<cols; x++) {
            String tile = map[y][x];
            setTile( x, y, tile);
        }
    }

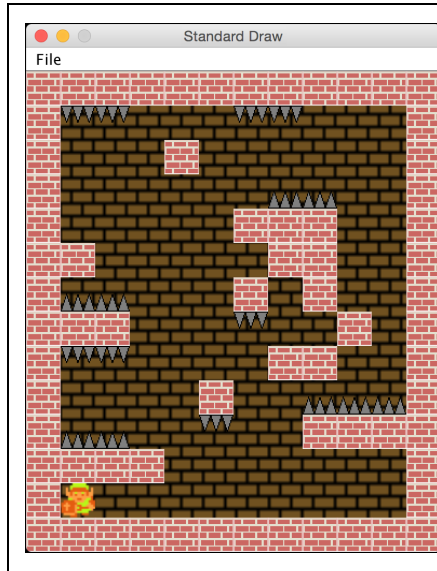
    StdDraw.setCanvasSize(width, height);
    StdDraw.setXscale(0.0,width);
    StdDraw.setYscale(height,0.0);
}
```

Scene.java → setTile() method

```
private void setTile(int x, int y, String tile) {
    if (tile.equals("#") ) {
        Block block = new Block(x, y);
        this.blocks.add(block);
    }
    else if (tile.equals("@") ) {
        this.player = new Player(x, y);
    }
    else if (tile.equals("A") ) {
        FloorHazard spike = new FloorHazard(x,y);
        this.monsters.add(spike);
    }
    else if (tile.equals("V") ) {
        CeilingHazard spike = new CeilingHazard(x,y);
        this.monsters.add(spike);
    }
}
```

Scene.java → draw() method

```
public void draw() {
    background.draw();
    for ( Block block : this.blocks) {
        block.draw();
    }
    for ( Block spike : this.monsters) {
        spike.draw();
    }
    player.draw();
}
```


Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

GUI window display with:

- Background image
- Brick blocks
- Player character
- Ceiling Hazards
- Floor Hazards

NOTE: No collisions with hazards yet

Goal 15: Game Over when Player touches a Hazard**Planning:**

Define the Lose condition of the game. When the player touches any hazard tile, then the game should end.

Implementation:

For each Game update, the game asks the scene if the player is dead or not, and sets the game over based on that. The scene iterates through each hazard & determines if player is touching, if so then player is dead, if not then player is alive.

Scene.java → isPlayerDead() method

```
public boolean isPlayerDead() {  
    for ( Block hazard : monsters ) {  
        if ( hazard.isTouching(player) ) {  
            return true;  
        }  
    }  
    return false;  
}
```

PlatformerGame.java → update() method

```
public void update() {  
    controller.update();  
    scene.update();  
    this.isOver = scene.isPlayerDead();  
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

If player touches a hazard, the game should stop, i.e. Game Over.

Goal 16: Model Exit class & add it to the Scene**Planning:**

Exits are a special type of block that advances the player to the next stage. They are doorway blocks so their collision detection behavior will be different than a standard block.

Implementation:

Create Exit class by extending Block class and then draw the exit into scene

Exit.java

```
public class Exit extends Block {
    public Exit(double x, double y) {
        super(x,y,"../Assets/tile-exit.png");
    }

    public boolean isTouching( GameObject player ) {
        return super.isTouchingX(player,0.25) && super.isTouchingY(player, 0.25);
    }
}
```

Scene.java → instance variables

```
/*Attributes: Instance Variables*/
private int rows;
private int cols;
private GameObject background;
private ArrayList<Block> blocks;
private ArrayList<Block> monsters;
private Player player;
private Exit exit;
```

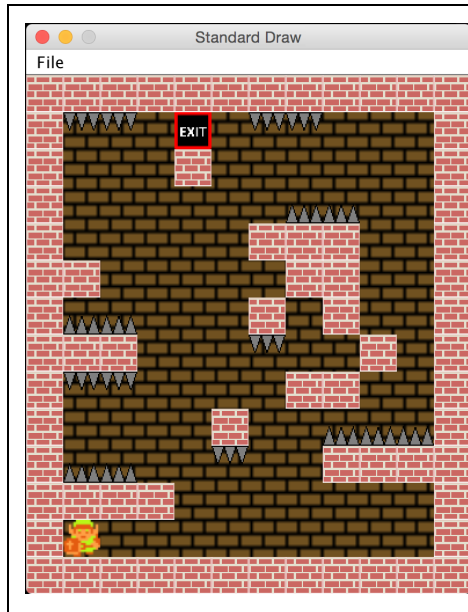
Scene.java → setTile() method

```
private void setTile(int x, int y, String tile) {
    if (tile.equals("#") ) {
        Block block = new Block(x, y);
        this.blocks.add(block);
    }
    else if (tile.equals("@") ) {
        this.player = new Player(x, y);
    }
    else if (tile.equals("A") ) {
        FloorHazard spike = new FloorHazard(x,y);
        this.monsters.add(spike);
    }
    else if (tile.equals("V") ) {
        CeilingHazard spike = new CeilingHazard(x,y);
        this.monsters.add(spike);
    }
    else if (tile.equals("!") ) {
        this.exit = new Exit(x,y);
    }
}
```

Scene.java → draw() method

```
public void draw() {  
    background.draw();  
    for ( Block block : this.blocks) {  
        block.draw();  
    }  
    for ( Block spike : this.monsters) {  
        spike.draw();  
    }  
    exit.draw();  
    player.draw();  
}
```

Testing:



To Execute your Program:

```
java PlatformGame < worldData.txt
```

Expected Output:

GUI window display with:

- Background image
- Brick blocks
- Player character
- Ceiling Hazards
- Floor Hazards
- **Exit**

NOTE: No collisions with Exit yet

Goal 17: Goto Next Level when Exit is touched**Planning:**

Define the Win condition. When a player touches the exit tile advance to the next scene. The player wins if they exit the last level.

Implementation:

Check if the player is touching exit during each update. If so, increment level, and check to see if it's the last level. If it isn't then update the scene to that level by getting the level map from world and constructing a new scene in the game, then register the new player object to the controller. If it is the last level, then the game is over.

Scene.java → getExit() method

```
public Exit getExit() {  
    return this.exit;  
}
```

World.java → getLength() method

```
public int getLength() {  
    return this.levels.length;  
}
```

Game.java → update() method

```
public void update() {  
    controller.update();  
    scene.update();  
    this.isOver = scene.isPlayerDead();  
    if ( scene.getExit().isTouching( scene.getPlayer() ) ) {  
        this.level++;  
        if (this.level < world.getLength() ) {  
            String[][] map = world.getLevel(this.level);  
            this.scene = new Scene(map);  
            this.controller = new Controller(this.scene.getPlayer() );  
        }  
        else {  
            this.isOver = true;  
        }  
    }  
}
```

Testing:**To Execute your Program:**

```
java PlatformGame < worldData.txt
```

Expected Output:

When the player touches exit, game should load the next level. If last level, then game over.

Goal 18: Player Animations**Planning:**

Design an approach that can animate the player by switching the images in a certain order after a certain duration. Then organize all the animated poses in a way to easily access them.

Implementation:

Animation class manages the collection of images for an animation, which is the current image, and how to switch to the next one. The Pose enum, defines all the character animations. Player has a current pose which is updated whenever the player moves left/right. Override the draw method in player to update the image before drawing.

Animaton.java

```
public class Animation {
    long startTime;
    int frameIndex;
    String[] sequence;

    public Animation(String... images) {
        this.startTime = System.currentTimeMillis();
        this.frameIndex = 0;
        this.sequence = images;
    }

    public String getImage() {
        int index = this.frameIndex;

        long now = System.currentTimeMillis();
        if(now - this.startTime > 75) {
            this.frameIndex = (index + 1) % this.sequence.length;
            this.startTime = now;
        }
        return this.sequence[index];
    }
}
```

Pose.java

```
public enum Pose {
    RIGHT("../Assets/link-right.png", "../Assets/link-right2.png"),
    LEFT("../Assets/link-left.png", "../Assets/link-left2.png");

    private final Animation animation;

    private Pose(String... images) {
        this.animation = new Animation(images);
    }

    public String getImage() {
        return this.animation.getImage();
    }
}
```

GameObject.java → setImage method

```
public void setImage(String image) {
    this.image = image;
}
```

Player.java → instance variables

```
/*Attributes: Instance Variables*/
private Physics physics;
private boolean isJumping;
private Pose currentPose;
```

Player.java → Constructor

```
public Player(double x, double y) {
    super( x*Block.SIZE, y*Block.SIZE, Block.SIZE, Block.SIZE, "../Assets/link-down.png");
    this.physics = new Physics(4);
    this.isJumping = false;
    this.currentPose = Pose.RIGHT; //ANIMATION POSES
    super.setImage( currentPose.getImage() );
}
```

Player.java → moveLeft() method

```
public void moveLeft() {
    physics.moveLeft();
    this.currentPose = Pose.LEFT;
}
```





Player.java → moveRight() method

```
public void moveRight() {
    physics.moveRight();
    this.currentPose = Pose.RIGHT;
}
```

Player.java → draw() method

```
public void draw() {
    super.setImage( currentPose.getImage() );
    super.draw();
}
```

Testing:

		Right animation, consists of two frames
		Left animation, consist of two frames