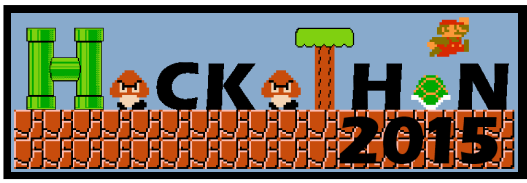


Welcome to the Université de Montréal's first video-game themed hackathon! This document will help guide you through the event, where you and your team will get to code a fully-interactive version of **Super Mario Bros.**'s iconic Level 1-1.

Development Structure. We've split the coding into 8 steps that'll bring you progressively closer to a finished game. Those of you lucky enough to get through these tasks will get to extend the game in some fun ways. You should split tasks up between team members, and we strongly suggest you follow the guidelines below:

	Core Functionality			Game Mechanics			Icing on the Cake		
Objective	1	2	3	4	5	6	7	8	Bonus
Depends on...	None	1	1	1 – 3	1, 2	1 –4	1 – 6	1 – 7	1 – 8

Expo. At the end of the event, each team will have **2 minutes** to show off its game to the crowd (and the judges)! Each game will be evaluated on various criteria, including how closely it reproduces the "Mario feel" and the coolness of any added features. Code structure and team balance will be used as tie-breaking criteria.

Online Example. You can play a free version of Super Mario Bros. online, to get an idea of what you'll be building: http://www.8bbit.com/play/super-mario-brothers/851 . Use this as a reference throughout the event.

Processing. You'll be coding in *Processing*, a Java-based language and IDE that we've installed on your hackathon laptop. You can find information and language reference documents here: https://processing.org/

Base Code We've provided a small, well-organized code base to help you get started. Open up `[dropbox]/hackathon2015/SuperMarioBros/SuperMarioBros.pde` in Processing and hit "play" to run the base code: it should bring up a window with an image in it. If not, say so and we'll help you get up and running.

The structure of the base code, and its data, is as follows (and you shouldn't have to change or add to it):

```
SuperMarioBros/
    code/                //Processing stuff, don't modify
    data/
        img/             //image data for the game
        level/           //level tile, enemy & image data
    SuperMarioBros.pde   //Processing glue code
    Game.pde             //Game class, controls everything
    Level.pde            //Level gridcells & state
    Player.pde           //Mario's position & state
    Body.pde             //parent class for game objects
    Enemy.pde            //enemy classes (goombas, etc.)
    Tile.pde             //tile classes (solid, etc.)
    Item.pde             //item classes (coins, etc.)
    Trigger.pde          //events (e.g. enemy spawning)
    Animation.pde        //special effects
    Utils.pde            //helper code, don't modify
```

Backups. After completing an objective, you will backup your work: save a timestamped copy of the entire project folder (including code and data) in your team's Dropbox. Keep track of which steps were completed for each backup, so you know which one to revert to if you need to check the behavior of a previous step. Backups serve as a safety net, as well as allowing us to look back at your progress and help solve problems.

Do not overlook backups: without them, random crashes and silly mistakes can lead to **hours of lost work**. Chances are you *will* run into a team-based coding conflict.

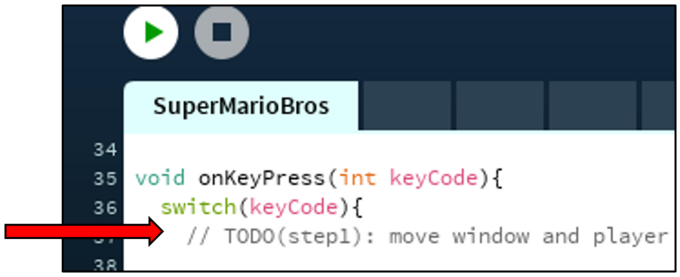
Split & Merge. When splitting into subteams, we suggest you:

1. put your common code on Dropbox (*push* the code)
2. on each computer, make a local copy of the shared code (*pull* the code)
3. each subteam modifies their local copy
4. when you want to combine added features (*merging* the code) :
   (a) make a local backup on each computer
   (b) each subteam merges their local code into the Dropbox version, one at a time
   (c) save a backup of the working merged code

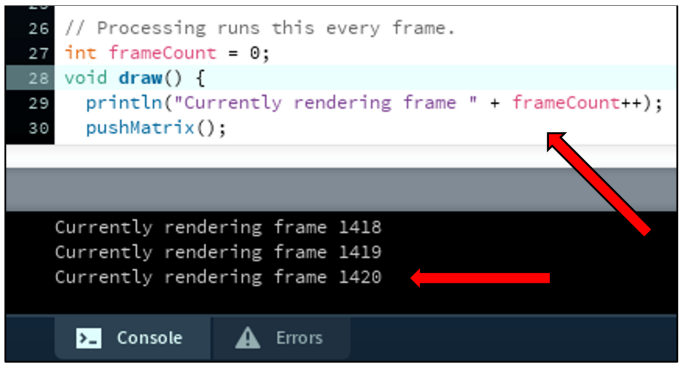
This procedure guarantees that two subteams won't edit the same files at the same time. Merging files can be tricky, so make sure to always communicate across subteams to ensure you don't *break the build*. Backups also help with merging problems.

Objective 1: Start by creating a basic tile world

Get comfortable by working as a team of 4 to complete this first task: create a checkerboard world and a rectangular "player" that moves in it.



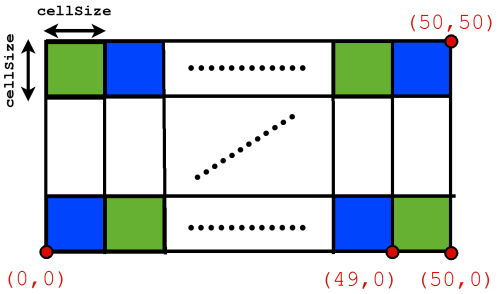
Tip 1: hunt for our TODO code blocks.



Tip 2: Output to the console with `println(...)`.

1. We will build Mario's 2D world from *gridcells*: 1 meter × 1 meter squares, aligned on a grid. Modify `Game.draw()` to create a 50×50 array of cells, in a blue-and-green checkerboard. Be sure to draw those cells without borders, using `strokeWeight(0)`.

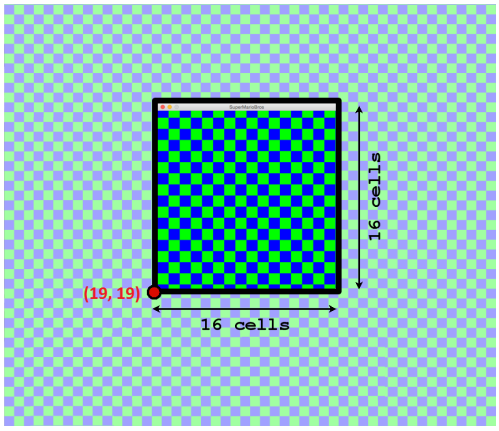
   A gridcell is represented by `cellSize` × `cellSize` screen pixels.



2. At any time, your game **window** will only display a 16×16 cell area of the entire 50×50 cell game **world**. Eventually, when Mario moves in the **world**, the **window** will have to move along with him. Start by sizing and placing the window at (19,19).

   Next, we'll create a very simple player in our world: first, start by placing your player at (25,25) in the world in `Game.init()`.

   For now, our player will simply be a boring 2×1 rectangle: initialize his size in `Game.init()` and call `player.draw()` in `Game.draw()`. Finally, draw the player in `Body.draw()`.

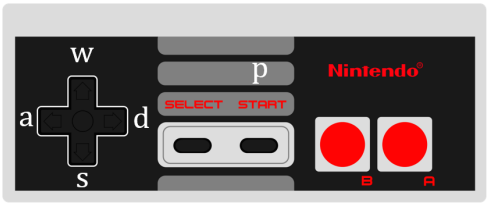


3. Let's get our player moving with some keyboard interaction!

   Modify `onKeyPress` in `SuperMarioBros.pde` to move the player by 1/2 gridcell in all four directions, using the keys `W`, `A`, `S`, and `D`. You can do this by adding displacement vectors to the player's position, with `game.player.pos.add(new Vec2(...))`

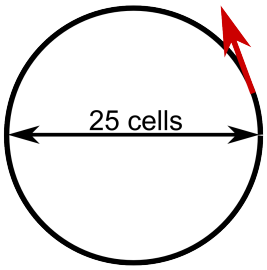
   Similarly, add code to move the window through the world using the arrow keys (keycodes `UP`, `DOWN`, `LEFT`, and `RIGHT`), again by 1/2 gridcell per key press. Here, you add displacements using `game.window.translateBy(new Vec2(...))`

   Lastly, use `P` to toggle `game.play`, and short-circuit the `Game.step()` function to do nothing if `game.play == false`.



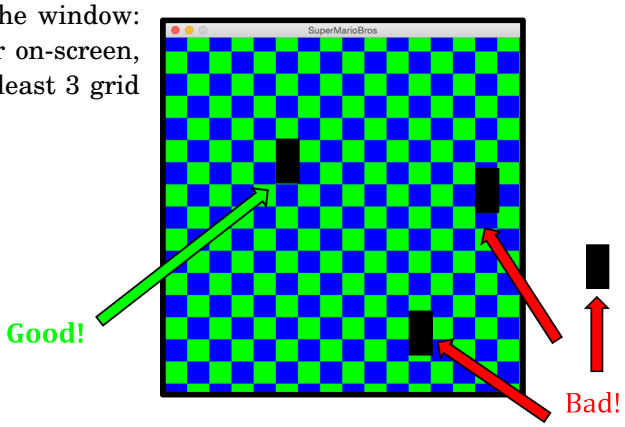
4. It'll be handy for debugging some of the upcoming features if we can get the player to automatically move in, say, a circle.

   In `Player.step()`, use `pos.add(new Vec2(vx,vy))` to displace the player along the path of a circle, where `(vx,vy)` is a small vector tangent to a circle that depends on `game.time`. Play with the size of this vector and speed of change of this vector to make the player move in a circle of diameter roughly equal to 25 gridcells.



---

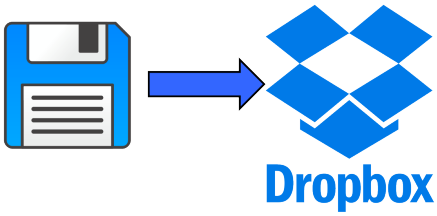
5. Let's make sure our Mario rectangle doesn't run outside the window: instead of manually moving the window to keep the player on-screen, automatically shift the window to maintain a border of at least 3 grid cells between the player and any window edge.



---

6. Take a deep breath, save your progress, and upload to your team's Dropbox folder. Remember to backup the entire directory structure, including code and data, using a time stamped naming scheme.

   For example, if you finish this objective at 10:30am on November 7th, then use a name like `[DROPBOX FOLDER]/backups/2015.11.07_10.30/SuperMarioBros/`



Objective 2: Collisions

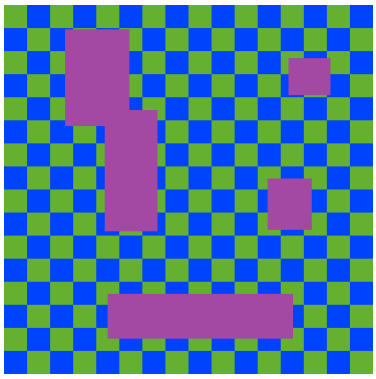
Our player's ability to collide with obstacles in the game world will form the basis for all in-game interactions.

1. Add the following **five** mouse and keyboard controls :
   - use `onMouseMove(...)` in `SuperMarioBrox.pde` to move the player with the mouse cursor
   - use `M` to toggle this mouse control for the player
   - use `L` to toggle automatic window movement
   - use `T` to toggle automatic circular player movement
   - use `1` to reset the game (by calling `game.init()`)

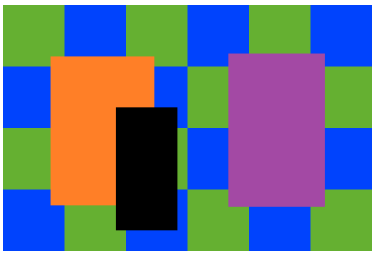
2. Before implementing collisions, let's set up some test objects that we'll use to collide our player with: add a couple of `Body` objects to `game.obstacles` using `obstacles.add(new Body(...))` in `Game.init()`.

   Add some variety to your objects: randomize the position and size of your obstacles using `random(a, b)`, which returns a random float between `a` and `b`. Be sure to maintain a minimum object size of 1.

   Color the obstacles purple, and draw all of them with a loop in `Game.draw()`, like `for(Body o : obstacles){o.draw();}`.

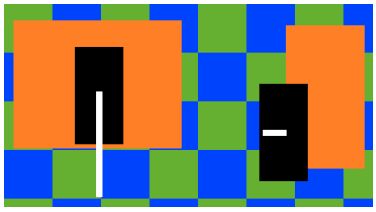


We've divided the collision process into **three** simpler problems, below.



3. Implement `Body.intersects(...)`, returning a `boolean` denoting whether one body intersects another.

   Draw objects that intersect the player in a different color.



4. Implement `Body.computePushOut(Body arg)`, returning a vector that corresponds to the **smallest displacement** necessary to move `arg` by in order to eliminate its collision with `this`.

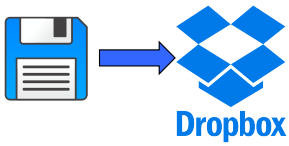
   If a body intersects the player, draw the `pushOut` vector on-screen, starting from the center of the player, using `line(x1, y1, x2, y2)`.

5. Bind key `O` to a new boolean, `solveObstacles`, in `Game`: when true, translate the player's position by `pushOut` to solve the collision; if false, only visualize the `pushOut` vectors, as in step 4 above. Watch out for the order in which you draw the objects and player.

   For your convenience, we've already created various `handle<X>` and `interactWith<X>` functions in `Body`.

   Place your code that moves the player away from obstacles in `player.interactWith(Body)`. Review the structure of these `handle<X>` and `interactWith<X>` functions to understand how your displacement code gets called for each obstacle. This structure will become important later on when there are different types of collisions to deal with in the game.

6. Save your progress and upload to your team's Dropbox folder.



Objective 3: Loading the Level

Let's spice up our game world, replacing the boring checkerboard with tiles from World 1-1 of Super Mario Bros. The level is composed of a grid of cells, as usual, but now each cell can contain a combination of **up to three elements** (see steps 3 to 5, below): a background image, a tile (e.g., a brick), and a static item (e.g., a coin).

All the data you'll need for the level is in a directory with the following file structure and content:

```
level1-1/
    lvl.txt //locations of the tile images for the level, and the level's background color
    map.txt //a simple character-formatted description of the map's tiles
    cellProperties.txt //explains what each character in map.txt represents
    triggers.txt //the location of enemies and other dynamic game elements (see Objective 6)
```

We provide you with the code to read and parse `lvl.txt` in `Level.load()`, you just need to uncomment it: easy! This code reads the three other files, and their information is used to populate the `backgroundImages`, `tiles` and `items` arrays. Take a bit of time to get a rough idea of what this data corresponds to.

---

1. Complete the implementation of `Level.loadMap(...)` to populate `map` appropriately. You can access the j^{th} character of the i^{th} row using `lines[i].charAt(j)`. **Be careful to load the level in the right order!**

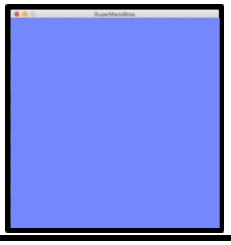
---

2. Complete `loadTileProperties` to parse the properties of each grid cell symbol. The code already tokenizes `cellProperties.txt`, so you can access line attributes with `properties.get("<`**attribute name**`>")`.

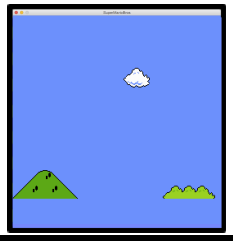
   When parsing a line with only a single character, we create a new `CellContent` and populate it with data from the following lines. We repeat this process, appending the `CellContent` objects to `tileProperties` as we go.

   Your job is to parse the arguments of a line and assign the correct data to the current `CellContent` object: for backgrounds, load the image in the `image` attribute with `currentCellContent.background = resources.getImage(<`**image filename**`>)`; for tiles, only load `solid` tiles, for now. This requires creating a `SolidTile` and assigning it the right image. We will deal with other tile types later in Objective 6.

We've divided level display into **four** steps, depending on each tile's content, below.



3. Implement and call `level.drawBackgroundImages()` in `Game.draw()` to replace the checkerboard with the level tiles: start by drawing a rectangle (with color `level.backgroundColor`, loaded from `Level.load(...)`) that covers the entire map **plus** an additional strip of height 2 at the top that we'll use later to display game info.

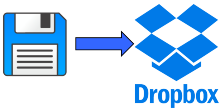


4. Next, still in `level.drawBackgroundImages()`, loop over every cell and draw its background image (if it has one) in the correct location with `drawer.draw(<image>, x, y)`

   Note: we're not showing it in our pictures here, but you should still be able to see your player's rectangle... in fact, you can even start *moving around in the world* using the mouse and keyboard features you coded earlier.

5. Implement and call `level.drawTiles()` in `Game.draw()`: this function loops over every gridcell, drawing its `Tile` (if it has one).

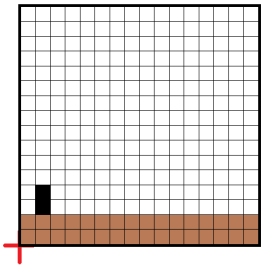
6. In `Body.draw()`, implement two types of drawings: if `Body.img` exists, draw this image, otherwise draw the body as a rectangle like we did earlier.



7. Save your progress and upload to your team's Dropbox folder.

Objective 4: Advanced Level Management

1. Set the bottom-left window corner at the origin and the player on the ground.

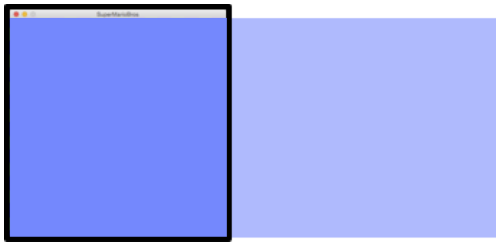


2. Before you get tired of mashing the `WASD` keys to move your player around the map, bind the keys `1`, `2`, ..., `0` to reset the game and transport the player at different positions equally spaced horizontally in the level.

---

3. Restrict the window size so it doesn't show the outside of the level.

   Restrict the player so it remains in the level horizontally (i.e., it can only leave the level from the top or bottom). In the real game, the window is never allowed to move to the left, but we suggest you avoid this constraint for now and impose it later if you really want to fully mimic the original game's feel.

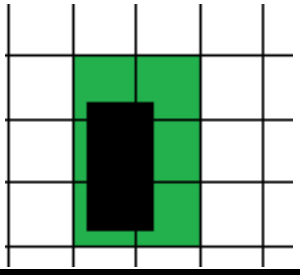


---

4. In `Player.interactWith(Tile)`, reuse the collision code from `player.interactWith(Body)` to make the player collide with `Tiles`.

5. Testing for collisions with all the level tiles is inefficient, as tiles that are very far away have no chance of intersecting the player.

   In `Body.handleTiles()`, use the fact that tiles live on a grid to only test collisions with nearby tiles: `floor`, `ceil`, `min` and `max` will come in handy.

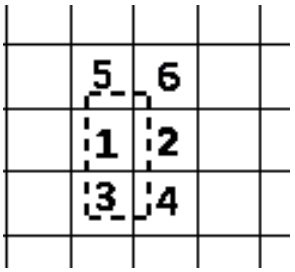


---

6. You may notice some weird behavior when moving, like having the player "snap" to the grid when moving during the automatic circular debug motion. It's **essential** to fix this, since it can cause more serious problems later on.

   To do so, solve collisions with tiles closer to the player first: order the tiles in `Body.handleTiles()` in increasing order of the distance between the player's center and the tile's center. Use a simple sorting algorithm to sort the tile subset array (e.g., selection, bubble, or insertion sort).

   Your player should now glide smoothly on the ground!



---

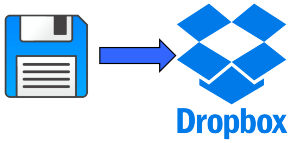
7. Draw the game's status text in the two extra rows of background space at the top of the screen: we already coded a helper function for this, `drawer.draw(String, x, y)`, where `String` is in UPPERCASE.

   The `nf(int, length)` function is also useful: it converts a number into a string of a given length (padded with zeros). A special character, lowercase `x`, can also be used to draw the × symbol next to the number of coins.

   - Don't draw the flashing coin, for now.
   - Store the coint count in `Game` and use it in the display. We'll discuss how to update the coin count, later.
   - Start the timer at 400 and have it decrease slowly as the game progresses.
   - You can use the empty space above the coin count to output debug data, if desired.

---

8. Save your progress and upload to your team's Dropbox folder.



Objective 5: Make Mario Move!

Let's get our player to move realistically according to the following laws of *physics*:

$$d\mathbf{x}(t)\Big/dt = \mathbf{v}(t) \qquad d\mathbf{v}(t)\Big/dt = \mathbf{a}(t) - b\,\mathbf{v}(t)$$

where $\mathbf{x}$ is our player's position, $\mathbf{v}$ his velocity, $\mathbf{a}$ his acceleration, and b is a damping constant that simulates the effects of friction.

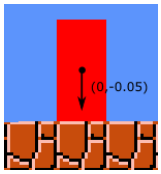
We can discretize and solve these equations with the following algorithm:

```
v += dt*a;
v *= b*dt;
x += dt*v;
```

where the *time step* `dt` dictates how much "simulation time" elapses between frames (defined in `Game`). Note: this algorithm is just a more complex version of the motion in Objective 1, where we updated the position by adding small vectors to it at each frame.

---

1. In `Game`, add a `gravity` vector and initialize it to point downwards with magnitude 0.05. Also initialize the damping constant to a value slightly less than 1; we suggest using different values for vertical and horizontal damping, which you can tweak later to adjust the player's behavior.
2. In `Player.step()`, set `player.accel` to the game's gravity and update `player.vel` and `.pos`.



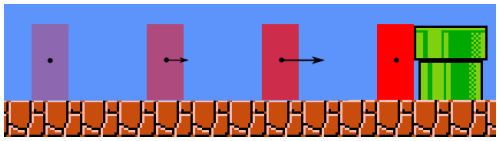
3. Remove `WASD` controls from `onKeyPress`. In `Player.step`, use `Keyboard.isPressed` and `Keyboard.isReleased` to query keyboard state: this way, we can hold the keys down instead of pressing them repeatedly.

---

4. Change the `A` & `D` keys to modify player acceleration, instead of directly changing position. This will result in smoother lateral motion. Adjust the damping so the player slows down nicely when the keys are released.

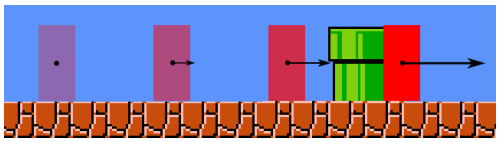
---

5. When the player collides with an obstacle, set the `x` or `y` components of his velocity to zero, depending on the situation.



---

6. Collisions may sometimes fail when the player hits small obstacles. This can happen if the player displacement is too large between time steps, since then the position update can place it past an obstacle. To prevent this, limit the player's displacement per step to 0.49 tiles. You can use the `clamp` function given in `Utils.pde`.



---

7. Implement realistic jumping by increasing the player's `vel.y` and letting the physics do the rest. Remember, the player can't jump in mid-air... only when he's "grounded". Also, jump higher if the jump key is held longer.

---

8. Make the player crouch when `'S'` is pressed. When crouching, change his size to $(1, 1)$ instead of $(1, 2)$.

---

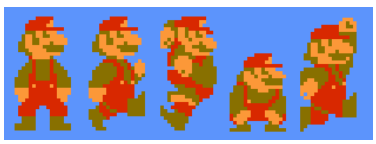
9. **Time to spice things up**: let's replace our rectangle with an **actual Mario**!

   In `Player.draw`, load Mario's `ImageSet` with `imgSet = bigMarioSet`. Note: this `ImageSet` groups images in `/data/img/players/`, and we'll be able to change Mario's appearance later on my simply changing `ImageSets`.

   Use the correct image in the `ImageSet` depending on Mario's state (running, dead, etc.): e.g., when dead, set `img = imgSet.get("dead")`.

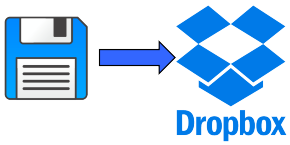
   Images can be animated (like `run` in `bigMarioSet`); set the animation speed with `img.setSpeed` so that it plays at a speed relative to Mario's velocity.

   Draw images with `drawer.draw` (defined in `Utils.pde`), using the function that allows you to flip and orient images according to the way Mario is facing.



---

10. Save your progress and upload to your team's Dropbox folder.



Objective 6: Get Bad Guys and Items

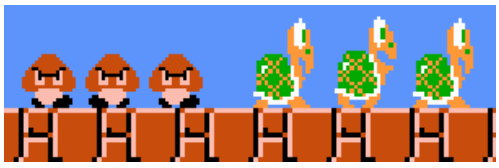
Let's get `Enemies` and `Items` moving and interacting in the world, but not with the player, just yet...

1. Some `Enemy` and `Item` classes are already created for you. Start by placing fake (i.e., generated by you) `Goombas`, `Koopas`, `Mushrooms`, `Flowers`, `OneUps` and `Stars` in the map. Draw them in `Game.draw()`.

2. At start-up, load the actual images for `Mushrooms`, `Flowers`, `OneUps` and `Stars`. Note that `Flowers` and `Stars` have animated images and can be loaded using a wildcard character `%d`: our image loading system will handle any image that matches the wildcarded string, creating an animated image. For example, you can load the animated image for `Flower` using `img = resources.getImage("data/img/items/flower/%d.png")`.

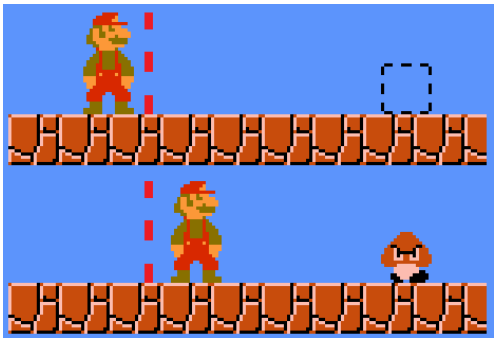
3. Implement the correct movements for every item. This will be similar to your player motion implementation, and you can (carefully) reuse code if you structure all common movement code in `Body.step()` and then call `super.step()` from the objects' `step` method. Pay attention to variations in the items' motion: `Goombas` and `Koopas` bounce off walls and have constant speed, whereas `Stars` constantly bounce, etc.

4. For `Goombas` & `Koopas`, assign an appropriate `ImageSet` of your choice, like what was done for the player. Look in `/data/img/enemies/`. Then, draw the correct image based on their motion (again, similarly to what you did for the player).

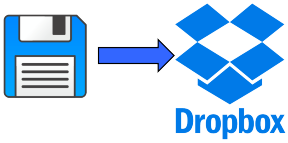


5. Now that enemies & items are "alive", let's get the *real* enemies where they belong in the level (*real* items will be loaded later). Enemies appear in the world according to `Triggers`, which are events that only happen after a certain action. In the case of enemies, their *spawning event* is triggered by Mario's position.

   - Implement `Level.loadTriggers`: the code already reads and tokenizes each line of the file. For each line related to an enemy (i.e. not the flag triggers, these will be used later), create an `EnemyTrigger` that places the right enemy at the given position, assigning the correct `ImageSet` and storing this enemy in the `EnemyTrigger`. Finally, add this `EnemyTrigger` to the list of triggers, `level.triggers`.
   - Implement the `EnemyTrigger` class: when Mario gets close to the stored enemy's starting position, the enemy should be added to `Game.enemies`. Look at the three functions in the `Trigger` class and how they are used in `Game`. This will give you an idea of how they should be overloaded in `EnemyTrigger`.



6. Save your progress and upload to your team's Dropbox folder.



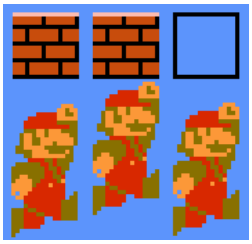
Objective 7: Get Mario to Smash Things!

Let's start implementing the remaining tile types, and some more complex interactions. Don't worry about getting the animations working yet (soon, soon...)

---

1. In `Level.loadTileProperties`, write code to load `BreakableTiles`. These tiles use an `ImageSet`, instead of a single image, to store two images: one for before the tiles is broken, and one for after.

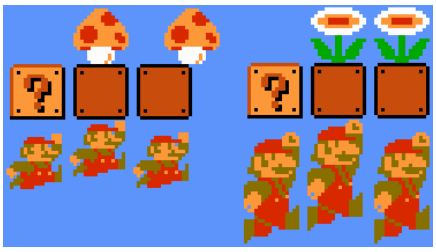
   Use the `ImageSet` listed in the `cellProperties.txt` and display the appropriate image depending on the state of the tile.



---

2. Also load `ContainerTiles`. These store `n Items` (e.g., a `ContainerTile` can contain 10 `Coins`), and also use an `ImageSet` to display different images based on whether they still contain an item or not.

   Note the special `Grow` item: this can either be a `Mushroom` or a `Flower` depending on Mario's state when he hits the `ContainerTile`.



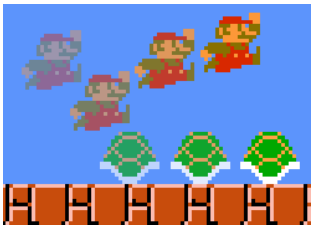
---

3. Implement player-enemy interactions:
   - When Mario hits an enemy **from above**, he kills it (i.e., the enemy should simply disappear, for now).
   - Enemies that hit Mario **from the side** change his state, his `ImageSet`, and sometimes his size: he can either be small, big or flowered.
   - When Mario gets hit, he becomes invincible and his image *flickers* for a short period of time.
4. Implement `Player.interactWith(Tile)`: `BreakableTiles` break when hit from below (but not if Mario is small), and `ContainerTiles` dispense their item. Dispense the correct item for tiles with a `Grow` item.
5. Make enemies bounce off each other.

---

6. Koopas don't die directly when Mario jumps on them: they turn into a shell that can be kicked around, killing other enemies it hits.

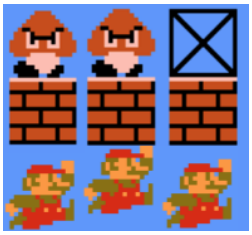
   Implement this tricky behavior.



---

7. If an enemy is over a tile that gets hit from below by Mario, the enemy dies.

   Implement this behavior, too.

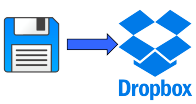


---

8. Implement item interactions: `Mushrooms` & `Flowers` change Mario's state, `Stars` make him temporarily invincible, `OneUps` add a life, and coins affect the coin count.
   - To create the *flashing* effect when Mario is invincible, change his `ImageSet` using the images in `smallStarMarioSet` or `bigStarMarioSet`.

---

9. Save your progress and upload to your team's Dropbox folder.

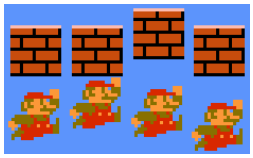


Objective 8: Special Effects

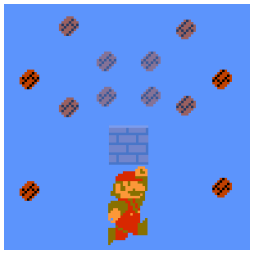
Let's spice up our game with some `Animations`. These images will appear on screen but have no physical interactions, so they're easier to code but add a lot of "production value". The code for stepping and deleting an `Animation` is already provided in `Game` but, unlike earlier, you'll have to write the code to draw them to `Game`.

For each of the different effects below, you'll need to create a class that implements the `Animation` interface.

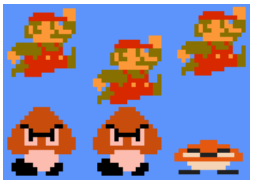
When small Mario hits a `ContainerTile` or `BreakableTile` from below, draw the tile as invisible and replace it with an `Animation` sequence, instead. Once the `Animation` sequence is done, make the tile visible again and make sure that the `completed()` function returns `true` so that the `Animation` object will be deleted by `Game`.



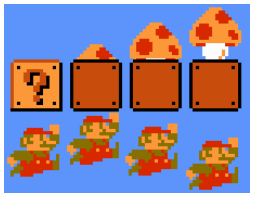
Animate the brick pieces when a `BreakableTile` is smashed: we provide you with the animated image sequence for the pieces in the `BreakableTile ImageSet`.



Implement a squish animation for `Goombas` that get stomped on by Mario.

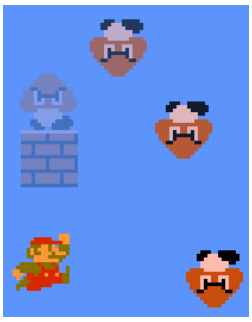


Have items slowly emerge from the `ContainerTiles` when dispensed.

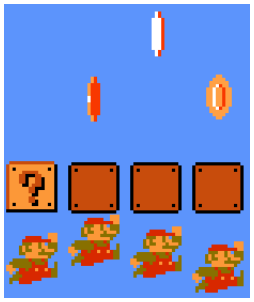


When Mario hits a block with an enemy on top, animate the enemies death. Also animate enemies killed by Mario touching them after he's become invincible from picking up a `Star`.

Enemy `ImageSets` have a "dead" image for these animations.

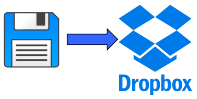


Animate coins jumping out of `ContainerTiles`.



Animate the flashing coin in the overhead display: the appropriate images are given in `/data/img/misc/hudCoin`.

Save your progress and upload to your team's Dropbox folder.



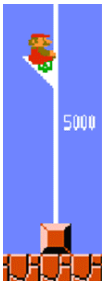
Bonus: Hit us with your best shot!

Congratulations: if you've made it this far, consider yourself a Mario coding Kamek! But why stop there? It's time for **you** to take the controls... no more instructions... no more hand-holding. Draw outside the lines! Don't look both ways before crossing the street! Do crazy things! Surprise us!

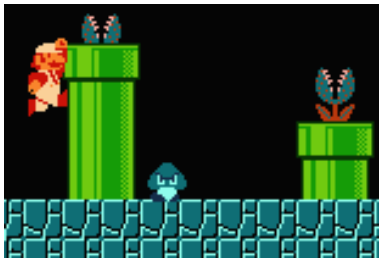
We're giving you free reign to implement any crazy new additions to the game, but just be careful to not bite off more than you can chew: carefully plan out small, manageable extra features. Feel free to create new images, if necessary; take advantage of the file structure and our data loading system, too, if you like. Make sure to save backups after you've implemented a significant feature.

Here are some suggestions, to give you an idea of what we'd do, if we were in your shoes.

Don't you find it unsatisfying when Mario reaches the end of 1-1, but the flag doesn't work.... maybe you can fix that?



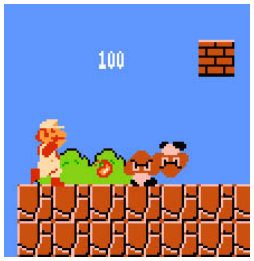
If you ask really nicely, we *might* be able to dig up the data for **Level 1-2**. Getting the level to work, including implementing the new `PiranhaPlants` enemy, would be cool!



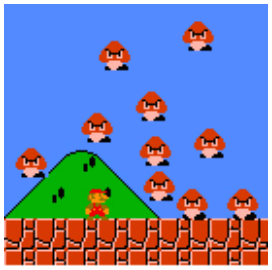
Nothing's stopping you from staying stuck in the first Mario game. Why not introduce some familiar faces from other Mario games?

No game is complete until there's an epic boss battle... how would you craft your own boss showdown?

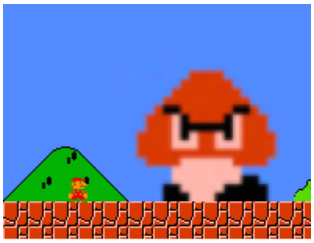
Poor flower-powered Mario still can't throw fireballs at his enemies....



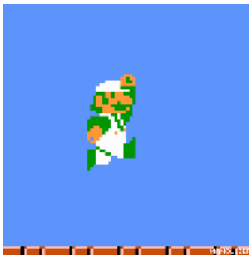
Create a ridiculously difficult version of level 1-1. Challenge your friends to beat it!



Make Mario giant. Or, better yet, make giant enemies! How would you kill them?



Why not invite Luigi back to the party?



Have a powerup that lets Mario shoot lazers!

Make a special tile flips the direction of gravity.

Give Mario a jetpack that shoots fireballs to fly.

What happens when the timer runs out?