yoavg / ioop2017

Assignment 5

Yoav Goldberg edited this page on 26 May 2017 · 7 revisions

Due date: June 4th, 2017. 23:59.

Block Removal, Lives, Scores and Multiple Levels

Introduction

In this assignment we are back to working on the Arkanoid game.

We will continue where we left of on Assignment 3, so please re-read that assignment (and your code for it) if you are not sure about the details.

As in the previous assignments, the size of the screen should be `800 x 600` .

We will be adding the following capabilities:

- Removing blocks when they are hit.
- "Killing" the player when all the balls fall of the screen.
- Keeping score.
- Keeping track of number of lives.
- Supporting multiple levels.
- a few additional features.

In terms of object-oriented techniques, the main new technique we will be using is the `Listener` pattern (also called `Observer`). You can also use inheritance if you find use for it.

Finally, in this assignment you will create an executable `jar` file, that will contain your compiled code in a single file instead of spread out over many `.class` files.

Part 0: Code Organization Into Packages

By this point, your project probably has a non-trivial number of classes. You may want to organize them into **packages**. Java packages provide a mechanism of grouping related pieces of code together, and separating less-related ones. For example, in your project you could have separate packages for classes that are related to the geometry primitives (Point, Circle, Rectangle), the collision detection part, the different sprites, and so on.

You can learn more about the Java packaging mechanism here.

Note that you are not required to use packages, but are strongly encouraged to. Packages are an important tool in organizing your code, and its a good idea to understand them and practice their use.

When using packages the use of `*.java` in the compilation command is no longer enough. To avoid listing all the java files manually, the compiler provides a way to get the list of files it needs to compile from a file. Combining this together with a handy **linux `find` command**, creates a way of compiling all files, ignoring the internal structure of our source folder. Here is an example of how the combination of these commands will look:

```
# write into sources.txt, the path to all java files inside src folder and any sub folder it
find src -name "*.java" > sources.txt
# compile all files that are listed in sources.txt file
javac @sources.txt
```

Remember that you still need to provide the other compiler flags.

Pages 21

- piazza
- Assignments

Clone this wiki locally

https://github.com/yoavg/io

Clone in Desktop

Part 1 -- Removing Blocks

An essnetial part of the game is to remove blocks that are hit by the ball.

We will begin with the `Game` class from assignment 3:

```
public class Game {
   private SpriteCollection sprites;
   private GameEnvironment environment;

   public void addCollidable(Collidable c);
   public void addSprite(Sprite s);

   // Initialize a new game: create the Blocks and Ball (and Paddle)
   // and add them to the game.
   public void initialize();

   // Run the game -- start the animation loop.
   public void run();
}
```

Currently, the method `initialize()` creates the board game (and populates the `SpriteCollection` and the `GameEnvironment`), and the `run()` method contains a loop that makes a Ball (or several Balls) bounce around the screen.

In order to remove a Block `b` from the game, the following things need to happen:

- `b` should be removed from the `SpriteCollection` .
- `b` should be removed from the `GameEnvironment` .

Add the methods `removeCollidable(Collidable c)` and `removeSprite(Sprite s)` to the `Game` class, and the method `public removeFromGame(Game game)` to the Block class.

We should now be able to remove a Block `b` from a Game `g` by calling `b.removeFromGame(g)` .

But who will call this method? The method needs to be be called when a ball hits a block, so a good candidate is to call the `removeFromGame(g)` from within the `hit(...)` method of `Block` . However, notice that we need to supply the remove method with a `Game` parameter -- which we do not have access to inside the `hit` method. While we could add Game as a member of Block, this performs a tight couplign between the `Block` and the `Game` classes, and it is also not very flexible.

We will use the `Listener` pattern to decouple the notification of hitting a block from the action to be performed when a block is hit.

The `HitNotifier` interface indicate that objects that implement it send notifications when they are being hit:

```
public interface HitNotifier {
   // Add hl as a listener to hit events.
   void addHitListener(HitListener hl);
   // Remove hl from the list of listeners to hit events.
   void removeHitListener(HitListener hl);
}
```

Objects that want to be notified of hit events, should implement the `HitListener` interface, and register themselves with a HitNotifier object using its `addHitListener` method.

```
public interface HitListener {
   // This method is called whenever the beingHit object is hit.
   // The hitter parameter is the Ball that's doing the hitting.
   void hitEvent(Block beingHit, Ball hitter);
}
```

Modify `Block` so that it implements the `HitNotifier` interface. Make sure to also add a `notifyHit(Ball hitter)` method to `Block` , which will be called whenever a `hit()` occurs, and notifiers all of the registered HitListener objects by calling their `hitEvent` method.

```
public class Block implements Collidable, Sprite, HitNotifier {
   List<HitListener> hitListeners;

   // ... implementation

   private notifyHit(Ball hitter) {
      // Make a copy of the hitListeners before iterating over them.
      List<HitListener> listeners = new ArrayList<HitListener>(this.hitListeners);
      // Notify all listeners about a hit event:
      for (HitListener hl : listeners) {
         hl.hitEvent(this, hitter);
      }
   }

   // Notice that we changed the hit method to include a "Ball hitter" parameter -- update t
   // Collidable interface accordingly.
   Velocity hit(Ball hitter, Point collisionPoint, Velocity currentVelocity) {
      // ... as before.
      this.notifyHit(hitter);
   }

}
```

```
NOTE: why did we copy all the elements before iterating in the notifyAll method?
--------------------------------------------------------------------------------
This is needed because we may be calling the removeHitListener or the addHitListener
method inside the notifyHit method. This means that the hitListeners list will be
changed while it is being iterated -- which will cause an exception. For this reason,
we chose to perform the iteration on a copy of the list instead.

(You probably want to do the same when iterating over the elements of the
SpriteCollection and GameEnvironment classes)
```

Simple test

In order to see if our listener is working, lets implement a simple HitListener that prints a message to the screen whenever a block is hit.

```
public class PrintingHitListener implements HitListener {
   public void hitEvent(Block beingHit, Ball hitter) {
      System.out.println("A Block with " + beingHit.getHitPoints() + " points was hit.");
   }
}
```

Change the `initialize()` method of `Game` to create a `PrintingHitListener`, and add it to all the blocks that are being created. Run a game and verify that you indeed see the message being printed to the console whenever a block is hit.

Block removal

Now that the infrastructure is in place, we move on to implementing the actual block removal. We need a HitListener that will remove blocks that are being hit. This notifier needs to hold a reference to the Game object, in order to be able to remove blocks from it. We will also use the notifier to keep track of the remaining number of blocks, so that we could recognize when no more blocks are available.

```
// a BlockRemover is in charge of removing blocks from the game, as well as keeping count
// of the number of blocks that remain.
public class BlockRemover implements HitListener {
   private Game game;
   private Counter remainingBlocks;

   public BlockRemover(Game game, Counter removedBlocks) { ... }

   // Blocks that are hit and reach 0 hit-points should be removed
   // from the game. Remember to remove this listener from the block
   // that is being removed from the game.
```

```
    public void hitEvent(Block beingHit, Ball hitter) { ... }
}
```

`Counter` is a simple class that is used for counting things:

```
public class Counter {
   // add number to current count.
   void increase(int number);
   // subtract number from current count.
   void decrease(int number);
   // get current count.
   int getValue();
}
```

Notice that the Counter is passed to the BlockRemover on its constructor, and so it can also be accessed from outside of the BlockRemover.

Modify the Game class to include a member of type `Counter` , keeping track of the number of remained (or removed) blocks. Implement the BlockRemover, create a BlockRemover object that holds a reference to the counter, and register the block remover object as a listener to all the blocks. Run a game, and verify that blocks that reach 0 hit-points are indeed being removed. Now, change the `run()` method of Game so that it will exit (`return`) when no more blocks are available. Note that once you created a GUI window, your program will not terminate until you release/close the window. When you are done with your program and want to terminate, you should call the `gui.close()` method that will clear the GUI resources and close the window.

Run a game and verify that it works.

Part 2 -- "Killing" the player when all the balls fall from the screen

Now that we can remove blocks, it is time to make the game more "fair" and allow the player to "die" as well. We will need to identify when a ball reaches the bottom of the screen, and remove it from the game. When all the balls reach the bottom of the screen, we need to end the game (return from the `run` method).

We will again use the Listener pattern. We will create a HitListener called `BallRemover` that will be in charge of removing balls, and updating an availabe-balls counter. Create a special block that will sit at (or slightly below) the bottom of the screen, and will function as a "death region". Register the BallRemover as a listener of the death-region block, so that BallRemover will be notified whenever a ball hits the death-region. Whenever this happens, the BallRemover will remove the ball from the game (you will need to add a `removeFromGame(Game g)` method to Ball) and update the balls counter.

What to do

Implement the strategy described above:

- Implement the `BallRemover` .
- Put a death-region block at (or below) the bottom of the screen, and make sure to register the BallRemover class as a listener of the death-region.
- Add a `Counter` member to Game to keep track of the number of available balls.
- Update the `run()` method of `Game` so that it will exit (`return`) when there are either no more blocks or no more balls.

Start a Game with 3 balls, and verify that the balls are indeed removed from the game when they hit the death-region at the bottom of the screen, and that the game exits when all the balls fall of the bottom of the screen.

```
An Opportunity
--------------
If you are adventurous and want your game to be even harder than it currently is,
you could create a special "killing block" that will sit among the regular blocks
and will remove balls that hit it.
```

```
With some more effort, you could also add a special kind of block that will introduce
a new ball whenever it is being hit.

You can do these things quite easily without changing the implementation of Block.
How?

(you are not required to actually implement this, but think of how you could implement
 it if we asked you to)
```

Part 3 -- Keeping track of scores

We would like to be able to keep a score - the player should receive some points whenever the ball hits a block. We will implement the following scoring rule: hitting a block is worth 5 points, and destroying a block is worth and additional 10 points. Clearning an entire level (destroying all blocks) is worth another 100 points.

Keeping track of scores

We will add a Counter called score as a member of Game.

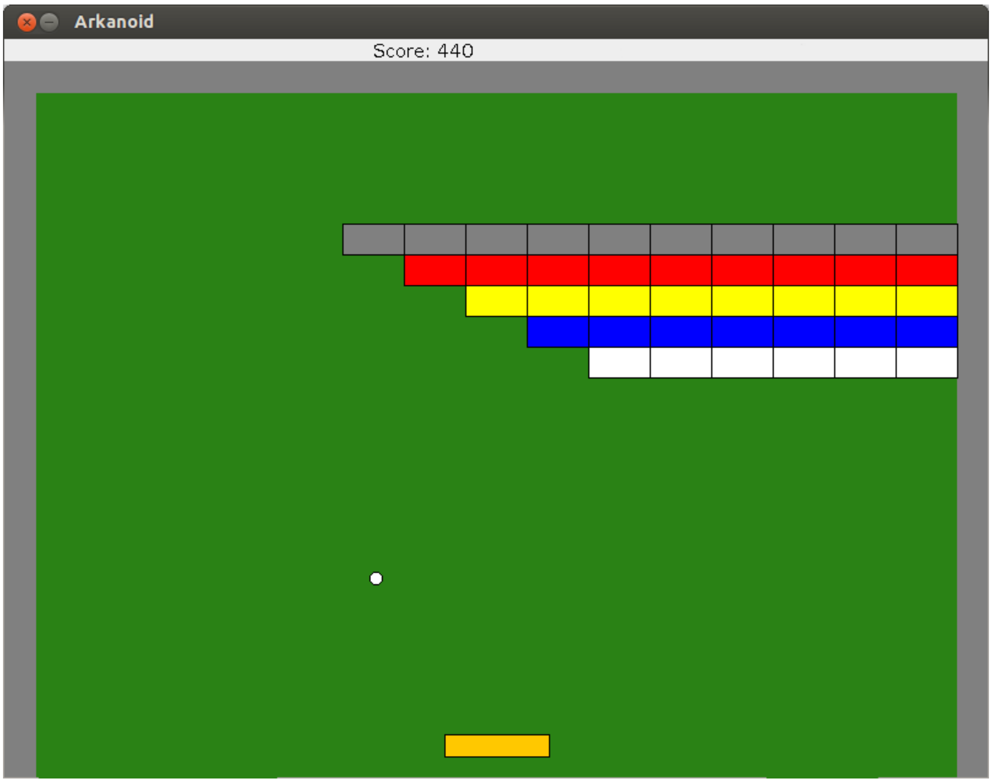
We will implement a HitListener called `ScoreTrackingListener` to update this counter when blocks are being hit and removed.

```
public class ScoreTrackingListener implements HitListener {
   private Counter currentScore;

   public ScoreTrackingListener(Counter scoreCounter) {
      this.currentScore = scoreCounter;
   }

   public void hitEvent(Block beingHit, Ball hitter) {
       ...
   }
}
```

Remember to also add 100 points when all the blocks are removed (this will happen outside of the ScoreTrackingListener).

Displaying the score

We would like to display the scores at the top of the screen, similar to the following screenshot:



Displaying a score is achieved by creating a sprite called `ScoreIndicator` which will be in charge of displaying the current score. The ScoreIndicator will hold a reference to the scores counter, and will be added to the game as a sprite positioned at the top of the screen.

Notice how value of the scores counter is updated by the ScoreTrackingListener and displayed by the ScoreIndicator, and that the ScoreIndicator doesn't even know that the ScoreTrackingListener exists, and vice versa.

Run a game and verify that the score is being displayed and updated as blocks are being hit and removed.

Part 4 -- Multpile Lives

At this stage we will add support for "lives". Currently, the game ends whenever all the balls fall from the screen. We want to change the behavior to the following:

- The game starts with k lives.
- When all the balls fall from the screen:
  - one life is lost
  - the paddle is created (if it doesn't exist yet) and moved to the center of the screen
  - a new ball (or several balls) are created
  - the game continues
- When the player has 0 lives, or when there are no more blocks, the game ends

This behavior will be implemented as follows:

- Add a number-of-lives counter to the Game.
- Create a `LivesIndicator` sprite that will sit at the top of the screen and indicate the number of lives.
- Rename the `run()` method of Game to `playOneTurn()` .
  - The `playOneTurn()` method should return when either there are no more balls or no more blocks (as it is currently doing).
  - Make sure that `playOneTurn()` starts by creating balls and putting the paddle at the bottom of the screen.
  - Note that `playOneTurn()` should not create new blocks, this should still happen in the initialize method.
- Create a new `run()` method that will:

  - call `playOneTurn()` .
  - Update the number of lives.
  - If there are more lives left, call `playOneTurn` again, else exit (`return`).

Run a game with 4 lives and 2 balls in each life. Verify that lives decrease when all the balls fall from the screen, and that the game ends when all the lives are lost.

Currently, there is no way to distinguish between the two end-game scenarios (winning by clearing all the blocks, or loosing by dropping all the balls). This will be addressed in part 7.

Part 5 -- Reorganization and a small rest.

We will now perform some code reorganization which will be useful soon.

Extracting the Animation Loop code from Game to its own class

A central component of the Game class is the main loop in the `playOneTurn()` method.

Recall, the loop probably looks something like this:

```
public void playOneTurn() {
   //...
   while (true) {
      long startTime = System.currentTimeMillis(); // timing
      DrawSurface d = gui.getDrawSurface();

      // game-specific logic
      this.sprites.drawAllOn(d);
      this.sprites.notifyAllTimePassed();

      // stopping condition
      if (this.ballsCounter.getValue() == 0) {
         break;
      }
      if (this.remainingBlocks.getValue() == 0) {
         break;
      }

      gui.show(d);
      long usedTime = System.currentTimeMillis() - startTime;
      long milliSecondLeftToSleep = millisecondsPerFrame - usedTime;
      if (milliSecondLeftToSleep > 0) {
          sleeper.sleepFor(milliSecondLeftToSleep);
      }
   }
}
```

The loop combines game-specific logic (like displaying the sprites and notifying them time has passed, and keeping track of the number of balls and blocks) with frame-rate and time-tracking code. In addition, this loop in Game requires for game to know the GUI object, and we would like to avoid this if possible (the Game should not care about the GUI which is used to display it, all it should care about is the DrawSurface it is drawing on).

We would like to separate the GUI and frame-rate management code (which is general, and can be used in other places) from the actual loop body (which is specific to the Game class). By separating the loop management from the game logic in the loop body, we will be able to re-use the loop in many different places.

One way to achieve this kind of separation is by using *template methods*:

```
public void playOneTurn() {
   //...
   while (!this.shouldStop()) { // shouldStop() is in charge of stopping condition.
      long startTime = System.currentTimeMillis(); // timing
      DrawSurface d = gui.getDrawSurface();

      this.doOneFrame(d); // doOneFrame(DrawSurface) is in charge of the logic.

      gui.show(d);
      long usedTime = System.currentTimeMillis() - startTime;
      long milliSecondLeftToSleep = millisecondsPerFrame - usedTime;
      if (milliSecondLeftToSleep > 0) {
          sleeper.sleepFor(milliSecondLeftToSleep);
      }
   }
}
```

Here the main loop includes only the gui and frame-management code, while the game-specific logic and stopping conditions are handled in the `void doOneFrame(DrawSurface d)` and `boolean shouldStop()` methods (which could be abstract, and differ from class to class).

One shortcoming of the template-method pattern is that it forces us to use inheritance. We will instead choose a different solution, based on *composition*.

First, we will take the template-methods and put them in an interface called `Animation`.

```
public interface Animation {
   void doOneFrame(DrawSurface d);
   boolean shouldStop();
}
```

Next, we will put the looping code in its own class, which we will call `AnimationRunner`.

```
public class AnimationRunner {
   private GUI gui;
   private int framesPerSecond;
   // ...
   public void run(Animation animation) {
      int millisecondsPerFrame = ...;
      while (!animation.shouldStop()) {
         long startTime = System.currentTimeMillis(); // timing
         DrawSurface d = gui.getDrawSurface();

         animation.doOneFrame(d);

         gui.show(d);
         long usedTime = System.currentTimeMillis() - startTime;
         long milliSecondLeftToSleep = millisecondsPerFrame - usedTime;
         if (milliSecondLeftToSleep > 0) {
             this.sleeper.sleepFor(milliSecondLeftToSleep);
         }
      }
   }
}
```

The AnimationRunner takes an Animation object and runs it. Now, we implement the task-specific information in the Animation object, and run it using the loop in the `AnimationRunner` class. (Note that now the `AnimationRunner` has `framesPerSecond` as a member, which should be set in the constructor. You should use a frame rate of 60 frames per second.)

We can now change the code of the Game class work with the AnimationRunner. We will make Game implement the `Animation` interface, and change `playOneTurn` to use the `AnimationRunner` to drive the Game.

```
public class Game implements Animation {
   private AnimationRunner runner;
   private boolean running;
   // ...
   public boolean shouldStop() { return !this.running; }
   public void doOneFrame(DrawSurface d) {
      // the logic from the previous playOneTurn method goes here.
      // the `return` or `break` statements should be replaced with
      // this.running = false;
   }

   public void playOneTurn() {
      this.createBallsOnTopOfPaddle(); // or a similar method
      this.running = true;
      // use our runner to run the current animation -- which is one turn of
      // the game.
      this.runner.run(this);
    }
    // ...
  }
```

Complete this change, and verify that your game still works as it did before.

Give me a break

Now that we have the `Animation` interface and `AnimationRunner` class, lets put them to good use. We will begin by adding an option to pause the game when pressing the `p` key. We will do this by creating a new kind of `Animation` , called `PauseScreen` . It is a very simple animation, that will display a screen with the message `paused -- press space to continue` until a key is pressed.

```
  public class PauseScreen implements Animation {
    private KeyboardSensor keyboard;
    private boolean stop;
    public PauseScreen(KeyboardSensor k) {
      this.keyboard = k;
      this.stop = false;
    }
    public void doOneFrame(DrawSurface d) {
      d.drawText(10, d.getHeight() / 2, "paused -- press space to continue", 32);
      if (this.keyboard.isPressed(KeyboardSensor.SPACE_KEY)) { this.stop = true; }
    }
    public boolean shouldStop() { return this.stop; }
  }
```

We will now add the following lines to the `doOneFrame` method of `Game` :

```
  public class Game ... {
    // ...
    public void doOneFrame(DrawSurface d) {
      // ...
      if (this.keyboard.isPressed("p")) {
        this.runner.run(new PauseScreen(this.keyboard));
      }
    }
    // ...
  }
```

In the game, when we identify the key `p` being pressed, we start running the PauseScreen animation instead of the Game one. The Game animation will resume as soon as we will return from the PauseScreen animation.

Make sure your game works and supports pausing and resuming.

3... 2... 1... GO

At the beginning of a level, and after a player loses a life, we would like to have a few seconds of wait before the game starts. The feature we will add now is an on-screen countdown from 3 to 1, which will show up at the beginning of each turn. Only after the countdown reaches zero, things will start moving and we will start with the game play.

To do this, we will implement a `CountdownAnimation` and run it before the game at each turn.

```
    public void playOneTurn() {
      this.createBallsOnTopOfPaddle(); // or a similar method
      this.runner.run(new CountdownAnimation(...)); // countdown before turn starts.
      // use our runner to run the current animation -- which is one turn of
      // the game.
      this.running = true;
      this.runner.run(this);
    }
```

Unlike the PauseScreen animation, The CountdownAnimation should display the counting **on top** of the game screen itself, so that the player will know what to expect when the game starts. For this reason, we pass the SpriteCollection to the CountdownAnimation constructor.

```
// The CountdownAnimation will display the given gameScreen,
// for numOfSeconds seconds, and on top of them it will show
// a countdown from countFrom back to 1, where each number will
// appear on the screen for (numOfSeconds / countFrom) secods, before
// it is replaced with the next one.
public class CountdownAnimation implements Animation {
   public CountdownAnimation(double numOfSeconds,
                             int countFrom,
                             SpriteCollection gameScreen) { ... }
   public void doOneFrame(DrawSurface d) { ... }
   public boolean shouldStop() { ... }
}
```

Implement the CountdownAnimation and add it to the game. Use a countdown from 3 that lasts 2 seconds. Make sure everything works.

Part 6 -- Multiple Levels

Our game only has one level. We will now add support for multiple levels: whenever we clear all the blocks on one level, we move to another, more challenging one. The lives and the scores should carry on from level to level. If we finish the last level, we win the game.

Levels can differ by:

- The background color.
- The number of balls at each turn.
- The initial angle and speed of the balls.
- The paddle size and speed.
- and, of course, the number of blocks and their layout.

In addition, each level will have a name (such as "Level 1" or "My Cool Level") which will be displayed at the top of the screen.

Rename Game to GameLevel.

At this point, we realize that the Game class is actually just playing a single level, not an entire game. Therefor, now will be a good point to rename the `Game` class to `GameLevel` . In general, whenever we realize that we chose a wrong name for a class or an interface, it is a good idea to think of a better name, and just rename it. (renaming a class is a bit tedious, but not very hard. In addition, IDEs such as Eclipse have tools to make this task even easier).

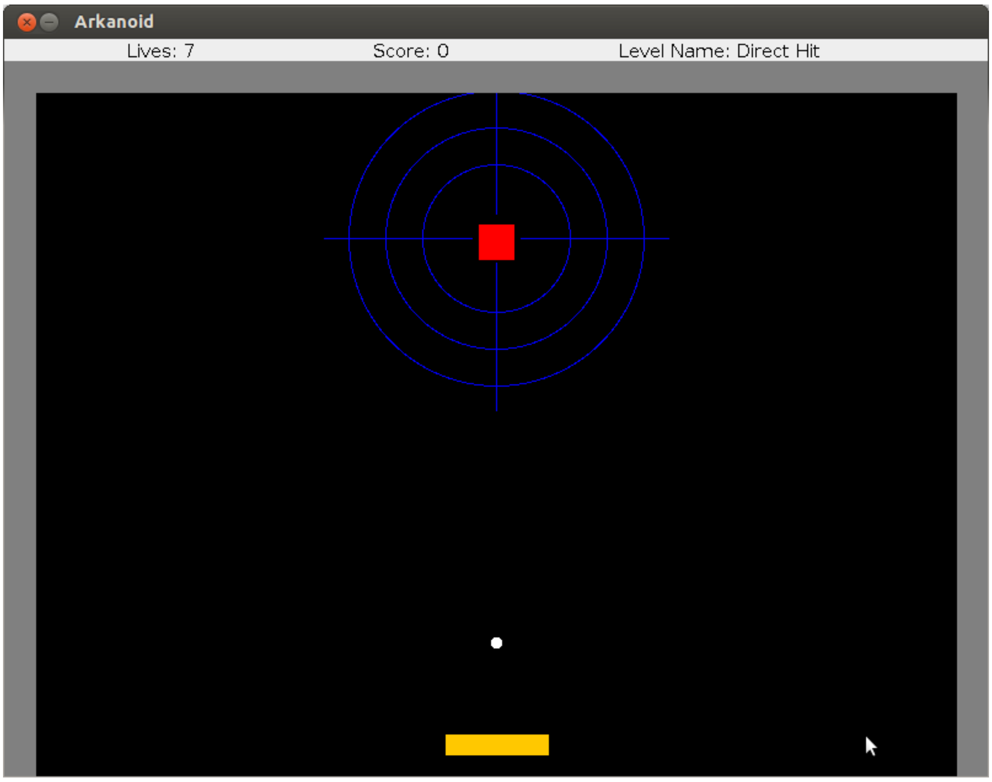
The LevelInformation interface

The `LevelInformation` interface specifies the information required to fully describe a level:

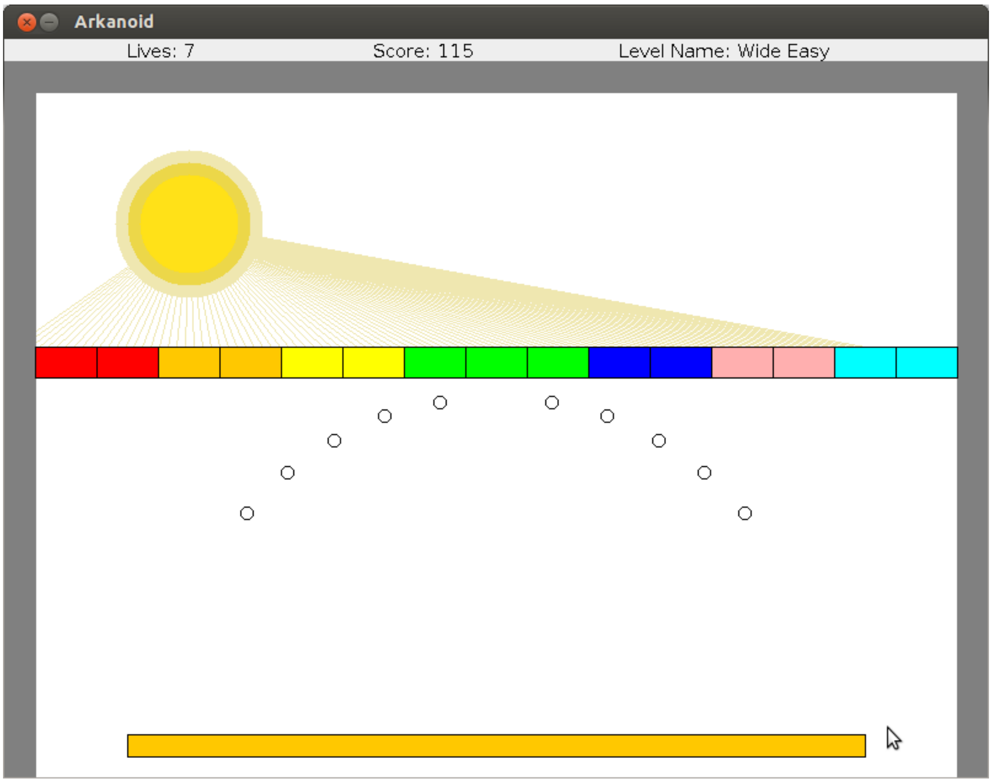
```
public interface LevelInformation {
   int numberOfBalls();
   // The initial velocity of each ball
   // Note that initialBallVelocities().size() == numberOfBalls()
   List<Velocity> initialBallVelocities();
   int paddleSpeed();
   int paddleWidth();
   // the level name will be displayed at the top of the screen.
   String levelName();
   // Returns a sprite with the background of the level
   Sprite getBackground();
   // The Blocks that make up this level, each block contains
   // its size, color and location.
   List<Block> blocks();
   // Number of levels that should be removed
   // before the level is considered to be "cleared".
   // This number should be <= blocks.size();
   int numberOfBlocksToRemove();
}
```

The Four Levels

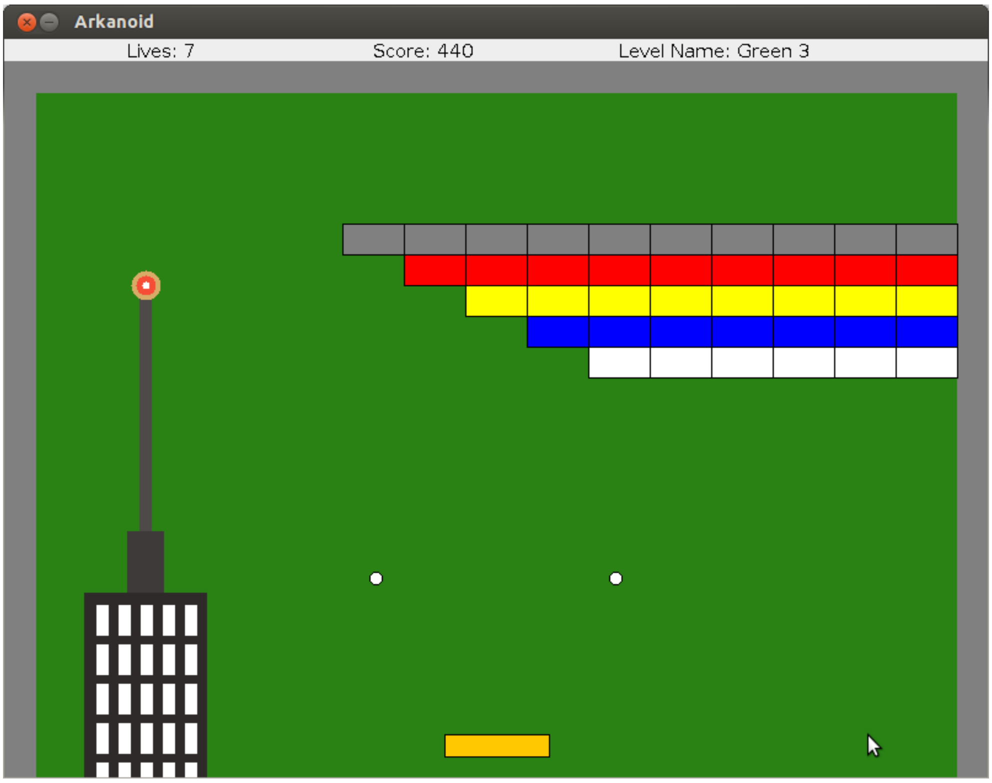
Create 4 classes implementing the LevelInformation interface. The classes should correspond to the following layouts (notice also the level names displayed at the top):

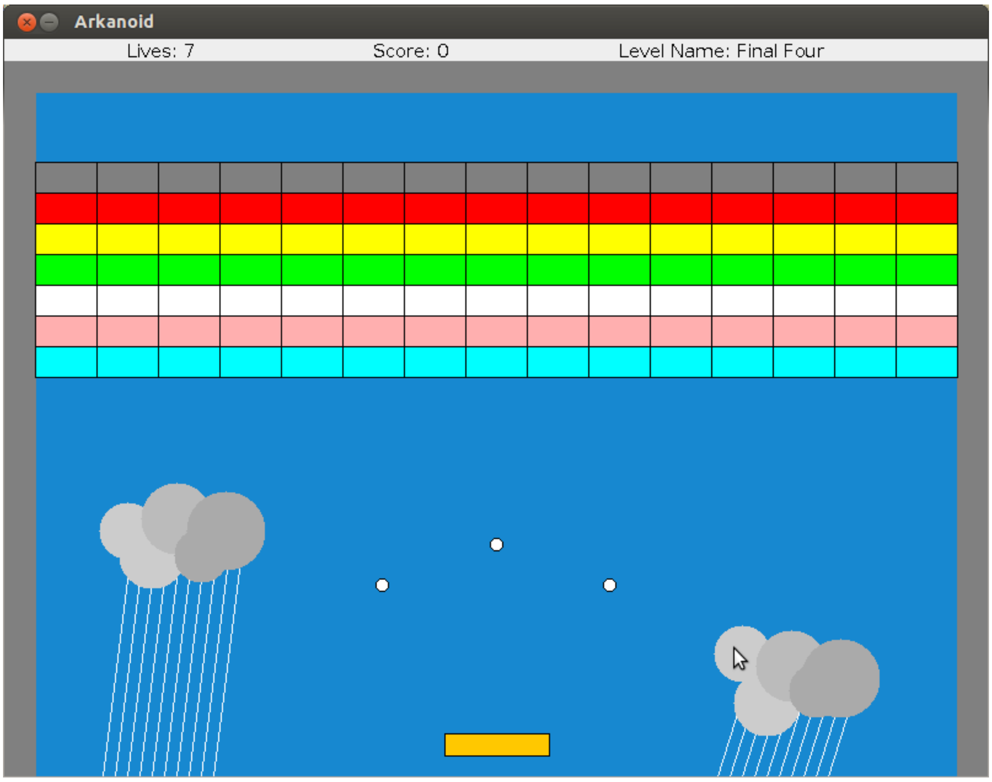


Lives: 7 Score: 0 Level Name: Direct Hit



Lives: 7 Score: 115 Level Name: Wide Easy





Note that in the first layout ("Direct Hit") we expect the ball to fly directly to the single block and destroy it.

We don't expect the backgrounds of the levels to be exactly as our but they should be similar in complexity. You are free to create your own abstract art with the basic draw commands. *Note:* you are not allowed to use the `drawImage()` method for drawing the backgrounds.

Initialize the GameLevel based on LevelInformation

Change the constructor of the `GameLevel` class to accept a LevelInformation as a parameter.

Change the `initialize()` (and other) methods of the `GameLevel` class to create the blocks, balls, paddle, etc based on the supplied LevelInformation.

Check that things work: start by playing a single level each time. Modify you `main()` method to create a LevelInformation, pass it to GameLevel, and then run the GameLevel. Verify that you get different behaviors from the game if you supply a different LevelInformation objects to the GameLevel.

The GameFlow class

Now that we can support different level layouts, we will add support for moving from one level to the next. For this, we will create a new class called `GameFlow` . This class will be in charge of creating the differnet levels, and moving from one level to the next.

```
public class GameFlow {

   public GameFlow(AnimationRunner ar, KeyboardSensor ks, ...) { ... }

   public void runLevels(List<LevelInformation> levels) {
      // ...
      for (LevelInformation levelInfo : levels) {

         GameLevel level = new GameLevel(levelInfo,
               this.keyboardSensor,
               this.animationRunner,
               ...);

         level.initialize();

         while (level has more blocks and player has more lives) {
            level.playOneTurn();
         }

         if (no more lives) {
            break;
         }

      }
   }
}
```

Complete the code in the GameFlow class, and create a class with a `main` method that will create a GameFlow and run it with the list of four LevelInformation created above.

Loose-ends

The *score* and *lives* should be kept across levels, throughout the entire game. They should be created and kept at the GameFlow, not the GameLevel, and only passed to the game level as parameters so it could update them. Make sure that this is indeed the case, and the score and lives are kept across different levels.

Part 7 -- End Screen

As a last touch, we will add an "end screen" to the game. Once the game is over (either the player run out of lives or managed to clear all the levels), we will display the final score. If the game ended with the player losing all his lives, the end screen should display the message "Game Over. Your score is X" (X being the final score). If the game ended by clearing all the levels, the screen should display "You Win! Your score is X".

The "end screen" should persist until the space key is pressed. After the space key is pressed, your program should terminate.

Add the needed code in order to support this feature.

Finally

Put all the pieces together and create a fully-functioning game!

Create a class called `Ass5Game` with a main method that starts a game with four levels, that supports all of the features described above.

When run without arguments, you should start a game with four levels that run one after the other. When run with additional arguments, the arguments should be treated as a list of level numbers to run, in the specified order. Discard (ignore) any argument which is not a number, or not in your levels range.

For example, running the game like this:

```
java Ass5Game 3
```

Will run a game with level 3.

```
java Ass5Game 1 3 2 1 9 1 bla 3 4 3
```

will run a game with the levels 1, 3, 2, 1, 1, 3, 4, 3.

Your game should behave similar to the following example: ass5example.jar. In order to run the example, download it to your computer, and then type (at the console) `java -jar ass5example.jar`.

The size of the screen should be `800 x 600`.

Start the game with 7 lives.

The countdown before each turn should start from 3 and last for 2 seconds.

You should support pausing with the `p` key and resuming the game with the space key.

If you want to make the pause and end-game screens nicer then our examples, feel free to do so!

Part 8: Deployment -- creating an executable .jar file.

We want the ability to distribute the game as a single self containing unit, so your goal is to create a *self-executing jar* containing everything you need for the game (except the `biuoop-1.4.jar`).

This means that we should be able to run your game using the command:

```
java -jar ass5game.jar
```

Jar files are created using the `jar` command. You can read about it here:

Creating a JAR File

Modifying a Manifest File

You should create a new Makefile target called `jar` that will produce a jar file with the name `ass5game.jar`, containing all binaries and resources needed for the game to run.

We suggest that you begin by figuring out the command for producing a "regaular" jar file (not self executing). That is, producing a jar file that can be run like this:

```
java -cp biuoop-1.4.jar:ass5game.jar Ass6Game
```

(notice how all the classes are inside the .jar files, but we tell the java command which class to run.)

At the next step you should make the jar self-executable (so that we don't need to specify which class to run, and that it will know by it self to load the `biuoop-1.4.jar`). we would like to just run:

```
java -jar ass5game.jar
```

To make the jar self-executable you will need to change the `META-INF/MANIFEST.MF` file generated inside the jar. You will need it to include the property `Main-Class` . This property will specify to the `JVM` what *Main Class* to execute when using the `java -jar` command. We will also include a `Class-Path` property so that it would automatically know to load `biuoop-1.4.jar` (which will be located in the same directory).

To add properties to the Manifest you will need to create a file `Mainfest.mf` with two lines:

```
Main-Class: Ass5Game
Class-Path: biuoop-1.4.jar
```

The `jar` command can be run with the `m` flag, to add properties to the Manifest from the another file, in our case we will use the `Mainfest.mf` we just created.

If you get stuck, remember you can open the generated `.jar` file as if it was a `.zip` file. Seeing what was generated may help you understand better what you should change in your command.

What to submit

You need to submit a file called `ass5.zip` containing all the classes and interfaces described above.

Your `compile` target should compile all the classes.

Your single `run` target should run the main in the `Ass5Game` class (with parameters for running the 4 levels in order).

Your `jar` target should produce a file called `ass5game.jar` as specified in **Part 8**. We should be able to run your `.jar` file using the command:

```
java -jar ass5game.jar
```

The jar file should be created in the same directory as the makefile. To be clear, we should be able to use the following commands to run your code:

```
unzip ass5.zip
make compile
make jar
java -jar ass5game.jar
```