Block Removal, Lives and Scores

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.

We will be adding the following capabilities:

- Removing blocks that are hit by a ball of different color.
- When the ball hits a block it changes it's color to the blocks color.
- When all the balls fall of the screen the player loses.
- Keeping track of the player's score.
- A few more small features.

In terms of object-oriented techniques, the main new technique we will be using is the `Listener` pattern (also called `Observer`). Any use of inheritance in your assignment is also welcome.

Part 0: Code Organization Into Packages

By this point, your project probably has a non-trivial number of classes. we 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. You should also take a look at our guide for organizing your code with packages for understanding what should be the full structure of your project.

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 may contain
find src -name "*.java" > sources.txt
# compile all files that are listed in sources.txt file
javac @sources.txt
```

This solution is suitable for linux. But even if you're developing on Windows, don't be worry; the `build.xml` we provide you for this assignment will know how to handle compilation and execution even if you use packages and the command `ant compile` will compile the entire project.

Part 1 – Removing Blocks

An essential part of the game is removing blocks that are hit by the ball. In this game, the block will only be removed if the ball that hit him has a different color than said block. After contact, the color of the block will change to the color of the block hit. For example, if a blue block is hit by a green ball, the block will be removed and the ball's color will change to blue. If the ball then hits a blue block, he will continue to move.

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 ballColorMatch(Ball`

`ball)` and `public removeFromGame(Game game)` to the Block class.

We should now be able to know if the block `b` needs to be removed after he hits by calling `b.ballColorMatch(ball)`. We also should be able to remove a Block `b` from a Game `g` by calling `b.removeFromGame(g)`.

But who will call the `removeFromGame` method? The method needs to be called when a ball hits a block, and the `ballColorMatch` method returns false. So a good candidate is to call the `removeFromGame(g)` from within the `hit(...)` method of `Block`, after we check that the color does not match. 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. Some classes would be notifiers and will be able to send messages to a list of listeners, and we will add listener classes that can receive these messages and preform what's required.

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 will notify 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 the
   // Collidable interface accordingly.
   Velocity hit(Ball hitter, Point collisionPoint, Velocity currentVelocity) {
      // ... as before.
      if (!ballColorMatch(hitter){
          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)
```

Additionally, we need to change the ball color after every contact.

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. (not need submit)

```
public class PrintingHitListener implements HitListener {
   public void hitEvent(Block beingHit, Ball hitter) {
      System.out.println("A Block 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 remainingBlocks) { ... }

   // Blocks that are hit 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 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. 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 displayed and updated correctly. Moreover, check that blocks are removed when the ball's color is different from the block's color. Finally, check that the ball changes color after contact (when the ball & block color are different) and ignores every block of the same color.

What to submit

For this assignment we will use this build.xml.

The `run` target will run the main in the `Ass5Game` class (this class corresponds to the `Ass3Game` class from assignment 3). `Ass5Game` should reside in the default package, namely, directly under `src`.

You need to submit a file called `ass5.zip` containing:

- a src folder (the classes, packages and interfaces described above).
- the build.xml file.

To remind you, the structure of the zip file should be:

- ass5.zip
  - ass5
    - build.xml
    - src
      - Ass5Game.java
      - ...

Once you are in the ass5 folder,the compilation command is

```
ant complie
```

and to run:

```
ant run
```

In this assignment we do not have the `ant check` command in `build.xml`. This is because checkstyle needs to know where your source files are. In this assignment we assume you work with packages, so each of the students can have different paths to classes. We will, however, have a script that run checkstyle on all

your source files, so as always, you must verify you comply to the checkstyle demands.

Please follow the general Submission Instructions for the course.

Clarifications

These clarifications are for this exercise and all future exercises in Java:

- When the paddle hits the edges, it should continue the movement to the other side of the screen in a circular motion.
- Using graphic packages like java.awt.Graphics is not allowed.
- Using `instanceof`, `isInstance (Class, obj)` and `getClass()` is not allowed.
- The scale of the gui window should be 600X800.
- We supply declarations for methods in classes. You are allowed to add methods to a given class, even with an existing name, but with different input parameters.
- It is allowed (and perhaps recommended) to add new classes.
- It is not allowed to change signatures of given methods.