yoavg / ioop2017

Assignment 6

Yoav Goldberg edited this page on 4 Jun 2017 · 5 revisions

Due data: Sunday, June 18, 2017 (midnight).

Java I/O: Reading and Writing Files

High-scores, level-specification files.

Introduction

In this assignment we will continue working on the Arkanoid game.

We will extend the game to support an opening Menu-screen, which will allow us to start a new game, see the current high-scores, or quit the game. When starting a new game, we could choose either an easy or advanced game, each game type will have different levels.

We will be saving the high-scores of players across different games. This means that we will be using `Java I/O` to read and write files. Another change from the previous assignment is that in this assignment, the different levels will not be hard-coded in your code, but instead will be read from level-description files. We will also be practicing the concept of `Factories` -- object that create other objects.

We will also practice a bit of **java generics**, as well as re-organize some of our code to remove code duplication using **object composition** and the **decorator** pattern.

Finally, in this assignment you will create an executable jar file, that will contain your compiled code as well as the resources needed for it to run.

As in the previous assignment, we provide you with an example jar file: ass6example.jar. In order to run the example, download it to your computer, and then type (at the console) `java -jar ass6example.jar` .

Part 1: Speed Control

We said that changes are going to happen, so here is one. Hopefully, your code is well organized, so this change will be relatively easy to accommodate.

A requirement change: currently, the game's speed is tied to the frame-rate (the number of frames shown per second) -- a higher frame-rate means a faster game. We want to decouple the two, so that we will have the same speed regardless of the computers we run on, or the frame-rate we choose. We want the game to stay at the same speed no matter which frame-rate value we pass to the AnimationRunner.

This requirement entails the following changes to the code:

- The `timePassed` method of the `Sprite` interface and the `doOneFrame` method of `Animation` will receive an additional parameter: `double dt` . It specifies the amount of seconds passed since the last call, as we will be dealing with speeds that show many frames per second, each invocation will result in a small value for `dt` . For example, in case we set `60` frames per second the `dt` value will be `1/60` .
- The pixel-change in the moving objects (ball and paddle) will now take into account both their speed and the amount of time change: `dt*paddleSpeed` , `dt*ballSpeed` . This means the speed is no longer and absolute number ("move x pixels each turn"), but is instead specified relative to the time ("move x pixels per 1 second").

Change the code in accordance to this requirement and set the `frame-rate` to **60**.

Pages 21

- piazza
- Assignments

Clone this wiki locally

https://github.com/yoavg/io

Clone in Desktop

Part 2: Keeping Scores

We would like to present players with the scores-history of previous games. For this, we need a table to store the historic high scores. This will be represented in a class called `HighScoresTable`, which will manage a table of `size` high-scores. The class should include at least the methods specified below. It will probably be beneficial to use some of Java's collection classes for the internal implementation of `HighScoresTable`.

Note the `load` and `save` method that will allow storing the table to file and loading it back again. This way, we could persist the score information between different invocations of the game. Think of a `file format` for the high-scores table, and implement the methods for reading and writing it. Another option is to use Java's `serialization mechanism` instead of coming up with your own format.

```
class HighScoresTable {

   // Create an empty high-scores table with the specified size.
   // The size means that the table holds up to size top scores.
   public HighScoresTable(int size) { }

   // Add a high-score.
   public void add(ScoreInfo score) { }

   // Return table size.
   public int size() { }

   // Return the current high scores.
   // The list is sorted such that the highest
   // scores come first.
   public List<ScoreInfo> getHighScores() { }

   // return the rank of the current score: where will it
   // be on the list if added?
   // Rank 1 means the score will be highest on the list.
   // Rank `size` means the score will be lowest.
   // Rank > `size` means the score is too low and will not
   //      be added to the list.
   public int getRank(int score) { }

   // Clears the table
   public void clear() { }

   // Load table data from file.
   // Current table data is cleared.
   public void load(File filename) throws IOException { }

   // Save table data to the specified file.
   public void save(File filename) throws IOException { }

   // Read a table from file and return it.
   // If the file does not exist, or there is a problem with
   // reading it, an empty table is returned.
   public static HighScoresTable loadFromFile(File filename) { }
}


public class ScoreInfo {
   public ScoreInfo(String name, int score) { }
   public String getName() { }
   public int getScore() { }
}
```

Test your code by creating a short program that will create a new table, add some scores, then print them to the console. See that only `size` scores are kept, and that these are indeed the highest scores entered to the table. Verify that saving and loading to files work as expected.

Showing scores

We now move to create a graphical representation of the scores. Create an `HighScoresAnimation` class. It will display the scores in the high-scores table, until a specified key is pressed.

```
public class HighScoresAnimation implements Animation {
   public HighScoresAnimation(HighScoresTable scores, String endKey, ...) { }
   // ...
}
```

Game Integration

In order to keep track of game scores, we need to:

- **Create a high-scores table when the game starts.** If it is the first time we run the game, a high-scores file does not exist, and we create a new table and immediately save it to a file. In the next time we run the game on the same computer, a file will exist, and so the table will be read from that file.
- **Add a player-name and high-score to the table when the game ends.** When the game ends, check if the player's score entitles him to be listed on the high-scores table. If it does, ask the player for his name, add his name to the table, and save the table.
- **Show the high-scores table.** After the game ends, the player either added his name to the table or not, the high-scores table should be shown.

Specifics:

The high-scores file name should be `highscores`. It should be located in the folder the game was run from.

In order to read the player's name, you can use the DialogManager component from the `biuoop` package (see the documentation).

```
GUI gui = new GUI(...);
DialogManager dialog = gui.getDialogManager();
String name = dialog.showQuestionDialog("Name", "What is your name?", "");
System.out.println(name);
```

Part 2.5: Some re-organization

We now have several `Animation` implementations that wait for a key press: the `Game Over` screen, the `You Win` screen, the `Pause` screen and the `High Scores` table. This is a good opportunity to remove some duplicate code, as well as to practice **object composition** and the **decorator pattern**.

We will extract the "waiting-for-key-press" behavior away from the different screens, and into a `KeyPressStoppableAnimation` decorator-class that will wrap an existing animation and add a "waiting-for-key" behavior to it.

```
public class KeyPressStoppableAnimation implements Animation {
   public KeyPressStoppableAnimation(KeyboardSensor sensor, String key, Animation animation)
   // ...
   // think about the implementations of doOneFrame and shouldStop.
}
```

Change the Game Over, You Win, Pause and High Score screens to run forever and not wait for a key-press. Wrap them with the `KeyPressStoppableAnimation` decorator to gain back the reaction to key-press behavior.

A Bug and a Fix

Consider the following piece of code:

```
// ....
AnimationRunner runner = ...;
Animation a1 = YouWinAnimation(...);
Animation a2 = HighScoresAnimation(...);
Animation a1k = new KeypressStoppableAnimation("m", a1);
Animation a2k = new KeypressStoppableAnimation("m", a2);
```

```
runner.run(a1k);
runner.run(a2k);
```

Here we create two key-stoppable animations, both stoppable using the `m` key, and run them one after the other. We expect to see the first animation, press `m`, see the second animation, press `m` again, and then exit. However, when we press the `m` key in the first animation, we immediately exit also the second animation.

Why does this bug happen? The `isKeyPressed(...)` method of the `KeyboardSensor` asks if the key is down, not if it went down **now**. When asking `isKeyPressed` inside the `a2k.doOneFrame()`, we get a `true` response based on the key press that exited us from `a1k`. This behavior will happen whenever we have two animations in a row that check for the same key press -- the same key press is likely to trigger both the events.

This behavior was already present in our previous code design in which each animation was in charge of checking its own key presses. But now that the key-press checking is centralized in the `KeypressStoppableAnimation` class, we have a chance of conveniently fixing it in one place.

Fixing the bug: The bad behavior happens when you just entered the animation, checked for a key-press, and caught a key that was already pressed before the animation started running. A solution would be to verify that the key press started only after the animation started. Here is one way of doing this (assuming we are waiting for the key `m`):

1. Add an `private boolean isAlreadyPressed` to the `KeypressStoppableAnimation` class, and have it initialized to `true`.
2. In `doOneFrame`, when you check if the key `m` is pressed, don't do anything in case `isAlreadyPressed == true`. This means that if the key was pressed before the animation started, we ignore the key press.
3. In `doOneFrame`, if the key `m` is not pressed, set `isAlreadyPressed=false`. Now, we know that there was a time point after the animation started in which `m` was not pressed.

Fix `KeypressStoppableAnimation` to remove the bug described above.

Part 3: Opening Menu

Currently, we need to wait until the game is over just to see the high-scores. What will we do if we just want to brag to our friends with our highest score achievement? Should we play an entire game?

We will add an **opening menu** to our game. When the game starts, the user will see a screen stating the game name (Arkanoid), maybe some graphics, and a list of several options of what to do next. Currently, the options will include:

- Press "s" to start a new game.
- Press "h" to see the high scores.
- Press "q" to quit.

However, we will be creating a general menu viewer, so we could easily extend it later with more options, and maybe re-use it in other situations.

This is how the Menu will be used:

```
Menu<String> menu = new MenuAnimation<String>("Menu Title", ...);
// the parameters to addSelection are:
// key to wait for, line to print, what to return
menu.addSelection("a", "First Choice", "option a");
menu.addSelection("b", "Second Choice", "option b");
menu.addSelection("c", "Third Choice", "option c");

AnimationRunner runner = ...;
runner.run(menu);
// A menu with the selections is displayed.
// Runs until user presses "a", "b"  or "c"

String result = menu.getStatus();
```

```
// result will contain "option a", "option b"
// or "option c"
System.out.println("You chose:" + result);
```

Notice the use of **generics** in the menu definition:

```
Menu<String> menu = new MenuAnimation<String>("Menu Title", ...);
```

This is used to specify the return type expected from the menu. By using generics, we allow the menu to be used with different return types. For example, if we want the selection to result in a `java.awt.Color` object instead of a String, we could do:

```
Menu<java.awt.Color> menu = new MenuAnimation<java.awt.Color>("Menu Title", ...);
// the parameters to addSelection are:
// key to wait for, line to print, what to return
menu.addSelection("a", "First Choice", java.awt.Color.RED);
menu.addSelection("b", "Second Choice", java.awt.Color.BLUE);
menu.addSelection("c", "Third Choice", java.awt.Color.GREEN);

AnimationRunner runner = ...;
runner.run(menu);

java.awt.Color selectedColor = menu.getStatus();
```

Our Menu will need to be displayed on screen, so it will be an `Animation`. Unlike the other animation loops we had, this one will need to return a value when it is done. We may want to add a nice background to our menu. For this, we will provide it with a method that will accept a background sprite and display it.

Below is the `Menu` interface. Implement a `MenuAnimation` class implementing this interface.

```
public interface Menu<T> extends Animation {
   void addSelection(String key, String message, T returnVal);
   T getStatus();
}
```

Menu Actions

Our menu will be run in a loop like the following:

```
while (true) {
   animationRunner.run(menu);
   status = menu.getStatus();
   // do something according to status:
   // play a game, show high score, or quit
}
```

What will the type of status be, and how will we use it? One option would be to make the returned status a String:

```
// BAD SOLUTION
menu.addSelection("s", "Play", "game");
menu.addSelection("h", "High Scores", "scores");
// ...
while (true) {
   animationRunner.run(menu);
   String status = menu.getStatus();
   if (status.equals("game")) {
      // play game
   } else if (status.equals("scores")) {
      // show scores
   }
   // ...
}
```

This will work, but it is a bad design, for two reasons:

1. We are using strings to communicate between objects, so there are no compile time checks for correctness. If we wrote "score" in one place and "scores" in another, this will result in an annoying hard-to-catch bug.

2. We have an "if-else" condition on the return type -- if we want to add or change a menu item, we now need to change two different places: where the item is added to the menu, and when the item is used after getStatus().

Instead, we prefer to tell the menu in advances what we would like to happen when the selection is made, and then, when a user makes a selection, the menu will return to us the action we wanted to happen. To do this, we will define a `Task` interface. A task is something that needs to happen, or something that we can `run()` and return a value.

```
public interface Task<T> {
   T run();
}
```

For example, we can have a `ShowHiScoresTask`:

```
public class ShowHiScoresTask implements Task<Void> {
   public ShowHiScoresTask(AnimationRunner runner, Animation highScoresAnimation) {
      this.runner = runner;
      this.highScoresAnimation = highScoresAnimation;
   }
   public Void run() {
      this.runner.run(this.highScoresAnimation);
      return null;
   }
}
```

Here, we do not need to return anything from the `run()` method, so we pass `Void` as the return type, and return null from the method. In other situations, we may want to return something from the task, in which case we will specify the appropriate return type.

```
A note about the Void type:
===========================
Java uses the `void` primitive in order to signify "nothing", or "no return type".
However, "void" cannot be used in a generics definition, and for this reason we
have the "Void" type (upper case) which is a class, and thus can be used with generics.
The `Void` class is a class from which no objects can be created (it has a private
constructor), so methods that return Void are forced to return null, which is a way
to ensure they don't have any "real" return value.
```

We can then have a menu holding tasks, and after a selection is made, we just run the selected task:

```
Menu<Task<Void>> menu = new MenuAnimation<Task<Void>>();
menu.addSelection("h", "Hi scores", new ShowHiScoresTask(runner, scores));
// ...
while (true) {
   runner.run(menu);
   // wait for user selection
   Task<Void> task = menu.getStatus();
   task.run();

   ...
}
```

You need to implement your menu selection using the `Task` interface.

Anonymous Classes it may seem cumbersome to create a class and give it a name just to define a task we will run later. We are only going to have one object of this class, and we are going to use it only in one place... While it is perfectly OK to create such a class and give it a name, java (and other languages) provides us with a shortcut called Anonymous Classes. Anonymous classes are classes without a name. This is how it works:

```java
public class Example {

   public void foo() {

      final String a = "this";

      Task<Integer> t = new Task<Integer>() {

         String b = "that";
         Integer number = 20;

         public Integer run() {
            System.out.println("The message is:" + a + " " + this.b);
            this.number++;
            return this.number;
         }
      };

      int result = 0;
      for (int i = 0; i < 5; i++) {
         result += t.run();
      }
      System.out.println("result:" + result);
   }

   public static void main(String[] args) {
      Example example = new Exapmle();
      example.foo();
   }
}
```

Here, we define a variable called `t` of type `Task<Integer>`, create an anonymous class that implements the `Task<Integer>` interface, that has two members, `b` and `number`. It also accesses the variable 'a' from the containing method. We then call the `run` method using `t.run()`.

Anonymous classes have the benefit of being able to access members of the class they are defined in (without the `this` prefix as it will points to members inside the anonymous class), or final variables in the method they are defined in (such as `a` in our example).

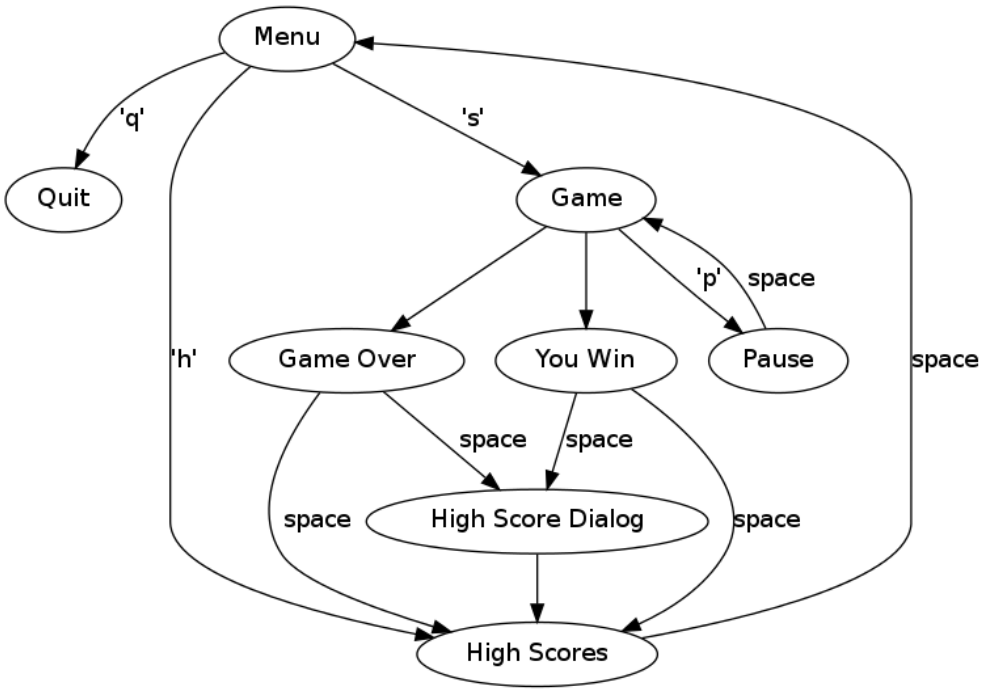
Notice that this is the only way we can instantiate an Interface directly: by providing an inline implementation of the methods it requires.

For more information on anonymous classes, see here

You may want to use anonymous classes for the tasks in your menu.

Outcome of this part

Change your code so that when the program is run, it behaves according to the following diagram:



The player is presented with an opening menu, from which he can choose to play a game or view the high scores. When he is done playing or viewing scores, he sees the opening screen again and so on, until he decides to quit the game by pressing `q`. Your menu needs to hold `Task` objects, as described above.

To easily quit the game, you will want to read about: System.exit().

Part 4: Levels IO

In the previous assignment, we introduced the `LevelInformation` interface for defining the layout and behavior of each level, and the given levels were implemented in code: adding a new level to the game meant writing the code for the new level and re-compiling the game.

An alternative approach, which we will use in this assignment, is to extract the level specification out from the code and into a configuration file. The program will read the levels information from the file, and produce matching LevelInformation objects. For this to work, we need, of course, to define a **file format**: a specification detailing the structure of the configuration file, the meaning of each element in the file, and a mapping between the information in the file to the level information.

Our file format is detailed below. You will need to write code that can read level specification files and produce `LevelInformation` objects.

Why separate the level definitions from the code?

An important benefit of separating the level definitions from the game code and placing them in human-readable files with a known format, is that now we can design new levels without doing any programming. We could delegate the job of level design to "Level Designers", who are people that have a different set of skills than that of programmers (for example they know about graphic design and the psychology of what makes games fun to play), and will do a better job at designing levels than a programmer. When the level definitions is in external file, the level designers can do their work and design levels without ever interacting with a programmer.

(In our project, the file format is human readable, and you are expected to edit it using a text-editor. In a larger project, with a more complex file format and level structure, it may make sense to create a specialized program that the game designers will use to design levels. This program will then produce the level-specifications files as its output. These files can then be read by the game program. The key point is, again, to separate the design process from the actual code, and put the level definitions and graphics in a place which is external to the code.)

Format Overview

The level specification file format will be based on human-readable text files. This means the information is written to the file in textual form (as opposed to binary form), and the files are editable in a text editor such as `notepad++` or `eclipse`.

The level specification format consists of two different file types `levels definitions` and `block definitions`.

- The `block definitions` format specifies a list of block types (each block has properties such as width, hit-points, appearance, and so on) and associates each type with a name.
- The `levels definitions` format specifies a list of levels. Each level contains information such as paddle size, level name, and blocks layout. The specification of the block layout refers to the block types defined in a `block definitions` file.

For the complete specification read: Levels Specification Files Format

You can also look at the Level Specification examples.

Reading the files:

You need to write code that will read the levels specification and produce LevelInformation objects. This section provides some hints about how to do this.

General
Use the Java I/O classes that you saw in the practical sessions to read the files line by line. Use methods of the String objects to interpret each line you read, and produce an appropriate output.

You probably want to have at least one class dedicated to reading each file format.

Note about Block Definitions File

Notice that the blocks can have a different appearance for each number of remaining hit points, and that some blocks are drawn using an external image file. You will need to update the implementation of class `Block` to support these new features.

Level Specification

The job is simple, you will need to implement an object with a method that will get a file name and returns a list of `LevelInformation` objects:

```
public class LevelSpecificationReader {

   public List<LevelInformation> fromReader(java.io.Reader reader) {
      // ...
   }
}
```

In order to do that, you will probably need some helper objects and methods. Try thinking of the different parts of the problem that need to be solved:

- Splitting the file into levels, reading a single level specification block from file (from `START_LEVEL` to `END_LEVEL`). This task will get a `java.io.Reader` and return a list of strings.
- Understanding the content of the level specification of a single level: this will go over the strings, split and parse them, and map them to java objects, resulting in a `LevelInformation` object.
- Perhaps the most challenging part is creating the `List<Block>` list of blocks based on the information in the block-definitions file and the `BLOCK` part of the level specification (between `START_BLOCKS` and `END_BLOCKS`. The next section outlines the basis of this part.
- For parsing individual lines into their components, you will need to use various methods of the java's String classs

Block Definitions

The block definition files define a mapping from symbols to spaces and blocks. These symbols are then used in the level specification files to define the blocks that need to be created. You will thus need a mechanism (object) with a method that will get a symbol and create the desired block. You will implement a `BlocksFromSymbolsFactory` class, with at least the following methods:

```
public class BlocksFromSymbolsFactory {
   // returns true if 's' is a valid space symbol.
   public boolean isSpaceSymbol(String s) {...}
   // returns true if 's' is a valid block symbol.
   public boolean isBlockSymbol(String s) {...}

   // Return a block according to the definitions associated
   // with symbol s. The block will be located at position (xpos, ypos).
   public Block getBlock(String s, int xpos, int ypos) {...}

   // Returns the width in pixels associated with the given spacer-symbol.
   public int getSpaceWidth(String s){...}
}
```

You could then use this class to convert the information in the `BLOCKS` sections of the levels specification files to a list of blocks.

You will also need a `BlocksDefinitionReader` class, that will be in charge of reading a block-definitions file and returning a `BlocksFromSymbolsFactory` object.

```
public class BlocksDefinitionReader {

   public static BlocksFromSymbolsFactory fromReader(java.io.Reader reader) {
      // ...
   }
}
```

How will the `BlocksFromSymbolsFactory` be implemented? We suggest it will hold the following members:

```
private Map<String, Integer> spacerWidths;
private Map<String, BlockCreator> blockCreators;
```

and have method implementations similar to the following:

```
public int getSpaceWidth(String s) {
   return this.spacerWidths.get(s);
}

public Block getBlock(String s, int x, int y) {
   return this.blockCreators.get(s).create(x, y);
}
```

`BlockCreator` is an interface of a factory-object that is used for creating blocks:

```
public interface BlockCreator {
   // Create a block at the specified location.
   Block create(int xpos, int ypos);
}
```

When reading the file, the code in `BlocksDefinitionReader` will create the appropriate `BlockCreator` implementations according to the definitions in the `bdef` lines, and populate the `BlocksFromSymbolsFactory` with the `BlockCreator` s and their associated symbols.

Colors

You may find a class such as the following useful:

```
public class ColorsParser {
   // parse color definition and return the specified color.
```

```
    public java.awt.Color colorFromString(String s);
}
```

Integrating the levels into the game

Change your main program so that it reads the levels information from a file whose name is specified as a command-line parameter (The file should also be relative to the `class-path` as described in the Levels Specification Files Format section).

Part 5: Different Level-Sets

Now that we can read the levels information from file, we can easily provide the player with the option to select a **level-set** when the game starts. After selecting the "Play Game" option in the main menu, the user will be shown another menu, in which he will be asked to select a level-set (for example "Easy" and "Hard"). After this selection, he will then proceed to play a game based on the levels defined for the set.

Behind the scenes, level-sets are implemented as different level specification files. Each level-set corresponds to one level-specification file. The level-sets will be specified in a `levelset` file with the following simple format:

Level-Sets file format

```
a:level 1 name
path-to-level-1-file
b:level 2 name
path-to-level-2-file
c:level 3 name
path-to-level-3-file
...
```

Odd-numbered lines are level names. The level name lines have the format `k:description` where `k` is a single character (the key for this item) and `description` is a string containing the level-set's description.

Even-numbered lines are the corresponding filenames, containing the level specifications. As before, *all filenames should be relative to the classpath*.

You might find the use of `LineNumberReader` useful, it extends `BufferedReader` and provides an additional method: `int getLineNumber()` .

Sub-menus

You will need to add sub-menus support to the `Menu<T>` interface and its implementation:

```
public interface Menu<T> extends Animation {
   void addSelection(String key, String message, T returnVal);
   T getStatus();
   void addSubMenu(String key, String message, Menu<T> subMenu);
}
```

After making these changes, you will need to write code for reading the level-sets file, creating a menu out of it, and registering this menu as a sub-menu of the main-menu.

Outcome of this part

You should have a class named `Ass6Game` , containing a `main` method. The `main` method should be able to accept a single commandline parameter. This parameter is a relative path to a file containing level-sets information. This path will be *relative to the class-path* (don't forget to change the class-path accordingly when executing the `java` command). You should read the information in from this file and start the game accordingly.

When run without parameters, your program should read a default level-sets file, containing at least three level-sets, and run the game accordingly. The first two level sets will be the ones we provide in this file. You should add at least one additional level-set of your own.

The location of the default level-sets file should be hard-coded in your code, and be *relative to the classpath*.

Part 6: 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 ass6game.jar
```

You should create a new Makefile target called `jar` that will produce a jar file with the name `ass6game.jar`, containing all binaries and resources needed for the game to run.

You already saw how to create an executable `.jar` file in Assignment 5.

This time, though, there is an extra catch, as your code needs to read files such as the level definitions and background images, and these files should also be bundled inside the `.jar` file. How do we read a file that is not on disk but inside the `.jar` file? Luckily, Java provides a mechanism for accessing such files (called "resources") as streams. This is explained in the end of the Levels Specification Files Format Page. Make sure your game works as expected also when run with a directory containing only your created `.jar` file and the `biuoop` jar file.

What to submit

You need to submit a file called `ass6.zip` containing all the classes and interfaces described above, as well as the resources (level-sets, level specifications, block-definitions, images and so on) needed to run a complete game.

Your `compile` target should compile all the classes.

Your single `run` target should run the main in the `Ass6Game` class (without parameters).

Your `jar` target should produce a file called `ass6game.jar` as specified in **Part 6**. We should be able to run your `.jar` file using the command:

```
java -jar ass6game.jar
```

We should also be able to call the main class directly, providing our own level-sets file (remember that all our paths are relative to the class-path so you will need to add another path to the class-path so that the JVM will be able to find your file and the other files it points to):

```
java -cp biuoop-1.4.jar:ass6game.jar:resources Ass6Game
our_level_sets_file_relative_to_resources_folder
```

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 ass6.zip
make compile
make jar
java -jar ass6game.jar
```

Like in the previous assignment, it is OK to have the `hashCode` checkstyle error.