CIS 201 – Computer Science I
Laboratory Assignment 9

Introduction

In this lab you will create a game that reads its configuration from a file.

Reading a file

Consider the process of writing a pinball game. Given a screen and a ball that bounces, different levels or pinball games frequently differ in how their bumpers (and other things) are laid out.

What we really want, then, is to build a pinball *engine*, a program that can simulate any pinball machine described in an appropriate way. You will begin with a simulator that has ball physics but no balls or bumpers. Your job is to extend the program so that it prompts the user for the name of a file (format described below) and then creates the appropriate sprites in the machine before setting the ball(s) loose.

Copy the files `Pinball.java` and `PinballGame.java` into a directory `Lab10`. You can find the files to copy in the directory

```
/home/student/Classes/201/Labs/10
```

Also in that directory is a file named `first.pnd`. We will be using the extension `.pnd` to stand for a "Pinball Description" file, an extension we coined. This file will have a format described later.

Download the Java files and the `first.pnd` file into your lab directory. Compile all of your Java source files and run the pinball game (in the class `PinballGame`). The game should run but it should not do anything because the code to process the pinball description file has not been completed.

Look at the `PinballGame` class. Find all of the stub functions (they are the ones with little or no code in them). In this lab, you will replace the stubs with working code.

Modify `getFileName` so that it displays the prompt that is passed into it followed by a single space and then reads a string from the user, returning that value. Read on to see how to do this.

Reading information from the user's keyboard is somewhat like printing information to the user's screen. Code such as `System.out.print(...)` will display the expression (whatever is in the "`...`" part) to your screen based on the type of the expression – such as a `String`, an `int`, or

whatever. Reading information from your keyboard uses `System.in`, which should not be surprising. But how will the program know the type of the value to be read? Here is where a `Scanner` comes in.

As you have seen in class, a `Scanner` is a Java object that takes raw input from the keyboard (or from a file, as we shall see) and interprets it as a string or integer or whatever, depending on what the program asks it to do.

How does one read a string from the keyboard, for example? Here is a snippet of code that would do so:

```
Scanner kbd = new Scanner(System.in); // tokenize raw input from the keyboard
String str = kbd.next(); // interpret the next token from the keyboard
```

For a `Scanner` object, the term "token" means a string of input characters that does not have any whitespace (such as spaces, tabs, or newlines). The `next()` method skips any whitespace until it encounters a non-whitespace character and then collects all of the following non-whitespace characters into a single string that it returns to the caller.

A `Scanner` can also return `int`s using `nextInt()` and `double`s using `nextDouble()`, for example. You should look at the `Scanner` class documentation to see all of the things that a `Scanner` can process.

Use a `Scanner` as described above to implement the `getFileName` method.

Checkpoint 1

Compile and run your program and show us your results.

The `ensureExtension` method

The `ensureExtension` method is passed two strings, a file name and an "extension", and returns that file name with a given extension. The term "extension" refers to the last part of a full filename that starts with a dot: for example, the extension of the file name `Pinball.java` is `java`, and the extension of the file name `TermPaper.doc` is `doc`. So if `ensureExtension` were passed the parameters `something` and `txt`, it would return the string `something.txt`. The method is called *ensure*Extension because it should only add the extension if the file name does not already end with it. Thus calling the method with parameters `something.txt` and `txt` should also return `something.txt`.

Before you implement the `ensureExtension` method, look at the documentation for the `String` class and study the following methods:

```
equals
endsWith
```

As indicated above, the `ensureExtension` method takes two `String` parameters. The first parameter is the file name (which you would ordinarily get using the `getFileName` method), and the second is the desired extension. If the file name does not end with a dot and the given extension, it needs fixing; otherwise return it unchanged. To fix a file name that ends with a dot, just return file name plus the extension; otherwise return file name plus dot plus extension. Here are possible inputs and their corresponding return values:

fname	**ext**	**return value**
`abcde`	`pnd`	`abcde.pnd`
`abcde.`	`pnd`	`abcde.pnd`
`abcde.x`	`pnd`	`abcde.x.pnd`
`abcde.pnd`	`pnd`	`abcde.pnd`

You should use the `ensureExtension` method in the `readPNDFile` method, after you get the file name (from `getFileName` above) but before passing it to `connectScannerToFile`:

```
fname = ensureExtension(fname, "pnd");
```

Checkpoint 2

Show us your working code.

Reading the configuration file

A PND file is a text file containing any number of lines. Each line in a PND file has one of the following formats:

```
Ball <color> <x> <y> <w> <h> <dX> <dY>
Oval <color> <x> <y> <w> <h>
Rectangle <color> <x> <y> <w> <h>
```

Here `<color>` refers to a color name (such as `red` or `yellow`, for example) that is recognized by the `getColor` method. The other terms (`<x>`, `<y>`, etc.) refer to decimal numbers (such as `0.2`) that can be interpreted as `doubles` in Java.

Now you are prepared to read the whole configuration file by implementing the rest of the `readPNDFile` method. You will write a loop that reads the next token from the file (a token is, in this case, a word) and processes it according to the structure of a Pinball Description file as

described above. The `connectScannerToFile` method, which is provided to you, will associate the `Scanner` object with the filename you give when you run the program – the one with the `pnd` extension.

Reading from a `Scanner` associated with a file uses a loop that normally looks something like this, where `in` is assumed to be a `Scanner` object (your mileage may vary):

```
while (in.hasNext()) {
    String obj = in.next(); // get the next token string
    // handle the obj string
}
```

This loop will continue to read tokens from the file until there are no more tokens to read – in other words, until all the contents of the file have been read and processed. Here the term "token" refers to character strings that consist entirely of non-whitespace characters.

If the token that the `Scanner` sees is one of `Ball`, `Oval`, or `Rectangle`, you should call the appropriate routine that reads the rest of the information for that type of sprite and creates and adds the sprite to the game canvas. For a `Ball`, you will call `handlePinball`; for a `Oval`, you will call `handleOval`; and for a `Rectangle`, you will call `handleRectangle`. If the token is none of these, your program should terminate gracefully with an informative message indicating that the PND file has the wrong format.

When using a `Scanner`, if you know the type of the next token to be read, you can simply call the appropriate `Scanner` method such as `nextInt()` (if you know the next token is an `int`), `nextDouble()` (for a `double` token), or just `next()` (for a `String` token). For example, if you have a PND file line that begins with `Oval`, you will call the `handleOval` method that will then get a `String` (the color) and four `double`s. The `handleOval` method will then create an `OvalSprite` object with the appropriate width and height, set its color to the given color value, locate it at the appropriate x and y values, and add it to the game canvas. (Note that you got to the `handleOval` method because you already saw the token `Oval` in the `readPNDFile`.) Similar remarks apply to PND lines that begin with `Rectangle`. For `Ball` lines, you need to create a `Pinball` object (look at `Pinball.java` to see what the constructor requires for this class!), adjust its color and location, add it to the game canvas, *and* add it to the `pinballs` list.

In the PND file, the string `<color>` can be passed to `getColor` to return a color. All the rest of the entries for a particular game object are double values. The variables `x` and `y` are the location of the bumper (or ball); `w` and `h` are width and height. The `dX` and `dY` values determine the velocity of the pinball in the *x*- and *y*-directions, respectively.

After you have finished the `readPNDFile` method and implemented the three `handle...` methods, your program should work perfectly. Use the `first.pnd` configuration file to test your program's behavior.

Checkpoint 3

Show us your working code at this point.

Write your own PND file

Create a PND file of your own choosing that describes a pinball "level". It's best to start with a sketch of what you want first, so that you can get a good idea of how your objects (bumpers and pinballs) will appear and so you can come up with reasonable values for sprite dimensions and locations. Your PND file should end with a `.pnd` extension.

Checkpoint 4

Run your `PinballGame` file and enter your newly created PND file as the filename. Show us your results.

Are you done?

If you have finished with the checkpoints, start working on your assignment to re-do your exam.