{ CS121 } (index.php)

Spring 2016

Home (index.php) Schedule (schedule.php#content) Projects (projects.php#content) Piazza (https://piazza.com/boisestate/spring2016/cs121/home) Resources (resources.php#content) Syllabus (syllabus.pdf) Lab (http://cs.boisestate.edu/~cs121/lab)

Project 3: Jukebox

Due 3/9

Contents

- Project Overview
- Getting Started
- Specification
- Testing
- Submitting Your Project

A visualization of a playlist
(you are not actually creating a Graphical User Interface...yet)

Project Overview

In this assignment, you will develop a class to represent a song and another class to represent a playlist of songs. We will provide a driver class with a menu and some methods for reading Song data from files. You will be reusing both of the classes you write later in the semester.

Objectives

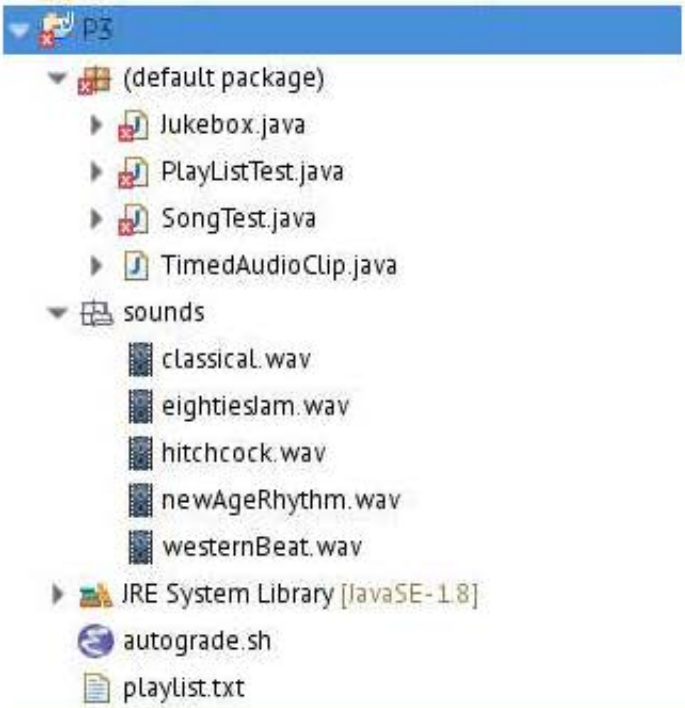
- Create an instantiable class.
- Write a method to test the constructors and methods of your instantiable class.
- Create a method for reading song data from the command-line.
- Write overloaded constructors.
- Override the toString method from the Object class

Getting Started

1. Create a new Eclipse project for this assignment.
2. The easiest way to import all the files into your project is to download and unzip the starter files directly into your project workspace directory (using the command-line or dolphin). The starter files are

available here: http://cs.boisestate.edu/~cs121/projects/p3/stubs (http://cs.boisestate.edu/~cs121/projects/p3/stubs) (You should download `p3-stubs.zip`)

3. After you unzip the files into your workspace directory outside of Eclipse, go back to Eclipse and refresh your project. It should look something like this.



> There will be several errors, but these are expected until you finish implementing your Song.java and PlayList.java classes.

4. Create a new Java class called `Song` and implement it according to the specifications below.

   When you have completed enough of your `Song` class, the errors in `SongTest` will go away and you can use it to test your `Song` class. You can still run it if there are errors in other classes. Just ignore the pop-up in Eclipse.

   Test often! Run your program after each task so you can find and fix problems early. It is really hard for anyone to track down problems if the code was not tested along the way.

   You should finish the entire Song class before moving on to PlayList.

5. When you are done implementing the `Song` class and all of the tests in `SongTest` pass, create a new Java class called `PlayList` and implement it according to the specifications below.

   When you have completed enough of your `PlayList` class, the errors in `PlayListTest` will go away and you can use it to test your `PlayList` class.

   Test often! Run your program after each task so you can find and fix problems early. It is really hard for anyone to track down problems if the code was not tested along the way.

   You should finish the entire PlayList class before moving on to Jukebox.

6. When you think you are done with `Song` and `PlayList`, run the `Jukebox` driver class to make sure everything works as expected.

7. Finally, run the final testing scripts as described in the testing section.

> This is super important!!

This is what we will be using to grade your program. **Most of your grade will depend on these tests passing, so make sure you take the time to get everything working. Start early so you have time to get help if you need it.**

Specification

- Existing classes that you will use: `SongTest.java, PlayListTest.java, TimedAudioClip.java`
- Existing classes that you will modify: `Jukebox.java`
- Classes that you will create: `Song.java, PlayList.java`

Task 1: Implement Song.java

You need to create a new class called `Song`.

- ## Instance Variables

  The data for a `Song` should include a *title*, an *artist*, a *playTime*, and a *fileName*. Your `Song` class should include instance variables for all of these plus one extra instance variable of your choice.
  - The *title*, *artist*, and *fileName* should be of type `String`.
  - The *playTime* should be an `int` representing the number of seconds the song takes to play.
  - Some possibilities for your extra instance variable are a string to represent the genre or an int to keep track of the number of times it has played.

> Following good object-oriented practice, your instance variables should be private.

- ## Constructors

  You should write two constructors for the class.
  1. The first constructor *must* take in values for and initialize all of the required instance variables.
  2. The second constructor should should initialize the minimum number of instance variables that you think are required for a valid song (perhaps title and artist).
  3. Any String instance variables that haven't been assigned values should be set to the empty String, `""`.

- ## Methods

  - Write *accessor* (getter) and *mutator* (setter) methods for all of your instance variables. If you added your own instance variables, you should provide a mutator method for any instance variables that aren't initialized in both of your constructors.
  - Write a method to play the song. Because we haven't covered how to do this in class, here is the method that you need to add.

```
public void play()
{
    TimedAudioClip clip = new TimedAudioClip(title, fileName, playTime);
    clip.playAndWait();
}
```

Make sure your volume is turned up when you test this! It will actually play the song

- Write a `toString` method for the Song class. The `toString` method is inherited from the Object class. In order to have it do something appropriate to the Song class, you need to override it. The signature of the `toString` method is

```
public String toString()
```

It can access the instance variables. This method should return a String containing a one line representation of the Song.

```
Classical           A Classical Artist       00:05                    sounds/cl
assical.wav
```

You must format the playtime as shown above (minutes:seconds). You will need to convert seconds to minutes and seconds (similar to what we did in one of your first labs) and use the `DecimalFormat` object to format the minutes and seconds to always print 2 digits.

To space the values correctly, use `String.format()` . (Hint: Look for `String.format()` in SongTest.java for an example of how we are expecting the songs to be formatted.)

- Your `Song` class may have a `main` method that tests all your constructors and methods as you write them.

SongTest.java

You do **not** need to modify this class. You will just use it to test your `Song` class.

When you think you are done with your Song class. Compile and run SongTest.java to make sure all the tests pass.

Task 2: Implement PlayList.java

You will write another instantiable class called `PlayList` . Your playlist will use an `ArrayList` of `Song` objects to keep track of a user's songs.

- ### Instance Variables

  The data for a PlayList should include a *name*, a *playCount*, and a *songList*.

  Your `PlayList` class should include instance variables for all of these.

  - The *name* should be a `String` .
  - The *playCount* should be an `int` representing the number of times the play list has been played.
  - The *songList* should be an `ArrayList<Song>` (e.g an ArrayList that holds Song objects).

Following good object-oriented practice, your instance variables should be private.

- ## Constructor

  You should write one constructor for this class.

  1. The constructor *must* take in a value for and set the *name* of the playlist, initialize the *playCount* to zero, and initialize the *songList* to an empty list.

- ## Methods

  - Write *accessor* (getter) methods for your *name* and *songList* instance variables. Write mutator (setter) methods for all instance variables. (Why don't we want a setter for the playCount?).
  - Add and complete these additional methods that will allow us to add/remove/play songs in the playlist.

```
public void addSong(Song s)
{
    // TODO: add the song to the songs list
}
```

```
public void removeSong(int id)
{
    // TODO: remove the song at position 'id' from the songs list
}
```

```
public int getNumSongs()
{
    // TODO: return the size of the songs list
}
```

```
public ArrayList<Song> getSongList()
{
    // TODO: return the songs list
}
```

```
public void playAll()
{
    // TODO: use a for-each loop to play all songs in the list.
    // TODO: increment the play count.
}
```

  - Write a `toString` method for the PlayList class. The `toString` method is inherited from the Object class. In order to have it do something appropriate to the PlayList class, you need to override it. The signature of the toString method is

```
public String toString()
```

    It can access the instance variables. This method should return a String containing the name, number of songs, and all the songs in the playlist in the following format

```
------------------
Sample Playlist (3 songs)
------------------
(0) Classical           A Classical Artist      00:05                   sound
s/classical.wav
(1) Eighties Jam        Some 80's band          00:03                   sound
s/eightiesJam.wav
(2) An awesome song     Coolio Jo               00:05                   sound
s/westernBeat.wav
------------------
```

Don't re-write the code to print each song. Just print the song using the toString you already implemented in Song.java.

- Your `PlayList` class may have a main method that tests all your constructors and instance methods as you write them.

PlayListTest.java

You do **not** need to modify this class. You will just use it to test your `PlayList` class.

When you think you are done with your `PlayList` class. Compile and run PlayListTest.java to make sure all the tests pass.

Task 3: Complete Jukebox.java

It is the driver class, meaning, it will contain the `main` method.

This class is mostly done for you. You must complete the `readSong` method defined in Jukebox.java. We have provided a "stub" of the method for you. Look for and complete the `TODO` comments.

1. The method should read a Song from the keyboard and return a Song. The data for each instance variable will be entered on a separate line.
   There are four lines of input for each song.

   A sample input is shown below.

   ```
   This is My Song
   Singing Artiste
   280
   sounds/hitchcock.wav
   ```

Testing

Once you have completed your program in Eclipse, copy all your .java and README files into a directory on the onyx server, with no other files (if you did the project on your computer).

Navigate to your project directory on the command-line and run the autograder using the following command.

```
./autograde.sh
```

When you pass all of the requirements, your output will look something like the output below. **Most of your grade will depend on these tests passing, so make sure you take the time to get everything working. Start early so you have time to get help if you need it.**

```
[you@onyx P3]$ ./autograde.sh
------------------
Testing Song.java
------------------
PASSED: song.getTitle()
PASSED: song.getArtist()
PASSED: song.getPlayTime()
PASSED: song.getFileName()
PASSED: song.setTitle()
PASSED: song.setArtist()
PASSED: song.setPlayTime()
PASSED: song.setFileName()
PASSED: song.toString()
----------------------
Testing PlayList.java
----------------------
PASSED: playList.getName()
PASSED: playList.getPlayCount()
PASSED: playList.addSong()
PASSED: playList.removeSong(1)
PASSED: playList.removeSong()
PASSED: playList.getSongList()
PASSED: song.toString()
PASSED: song.toString() on empty list
------------------
Testing Jukebox.java
------------------
============================
Welcome to the super jukebox!
============================
You must create a playlist to get started.
--------------------------
(f) load playlist from file
(n) create new playlist
--------------------------
Choose an option: Playlist added.
-----------------
Sample Playlist (3 songs)
-----------------
(0) Classical           A Classical Artist       00:05                    sounds/classical.
wav
(1) Eighties Jam        Some 80's band           00:03                    sounds/eightiesJa
m.wav
(2) New Age             Javya                    00:04                    sounds/newAgeRhyt
hm.wav
-----------------

What do you want to do (type 'm' to show menu)? Enter title: Enter artist: Enter play time (se
conds): Enter file: Added song: An awesome song
What do you want to do (type 'm' to show menu)? ----------------
```

```
Sample Playlist (4 songs)
------------------
(0) Classical           A Classical Artist       00:05                    sounds/classical.
wav
(1) Eighties Jam        Some 80's band           00:03                    sounds/eightiesJa
m.wav
(2) New Age             Javya                    00:04                    sounds/newAgeRhyt
hm.wav
(3) An awesome song     Coolio Jo                00:05                    sounds/westernBea
t.wav
------------------

Choose a valid song id: Removed song.
What do you want to do (type 'm' to show menu)? Playing songs.
Finished playing Classical.
Finished playing Eighties Jam.
Finished playing An awesome song.
What do you want to do (type 'm' to show menu)?
------------------
Sample Playlist (3 songs)
------------------
(0) Classical           A Classical Artist       00:05                    sounds/classical.
wav
(1) Eighties Jam        Some 80's band           00:03                    sounds/eightiesJa
m.wav
(2) An awesome song     Coolio Jo                00:05                    sounds/westernBea
t.wav
------------------

What do you want to do (type 'm' to show menu)? Goodbye!

PASS: Jukebox test passed.

Generated javadocs. Run the following command to view your documentation!

google-chrome doc/index.html

Make sure that each method has the correct documentation or you will lose points
When you are done, remove the entire doc directory using "rm -rf doc"

README found. Make sure it follows the correct format.

PASS: Looks good overall. Keep in mind that this doesn't test EVERYTHING.
      You should still make sure your indentation is correct and that you are submitting
      the correct files, you are using good coding practices, etc.
```

Extra Credit (5 points)

To be determined...

Submitting Your Project

Documentation

Javadoc Comments

If you haven't already, add **javadoc comments** to your program. They should be located immediately before the class header and before each method. If you forgot how to do this, go look at the Documenting Your Program (http://cs.boisestate.edu/~cs121/lab/labs.php?lab=01#docs) section from lab.

- Have a class javadoc comment before the class.
  Your class comment must include the `@author` tag at the end of the comment. This will list you as the author of your software when you create your documentation.
- Have javadoc comments before *every* method that you wrote. Comments must include `@param` and `@return` tags as appropriate.
- To build and view your comments, run the following commands.

```
javadoc -author -d doc *.java
google-chrome doc/index.html
```

README

Include a plain-text file called **README** that describes your program and how to use it. Expected formatting and content are described in README_TEMPLATE (https://raw.githubusercontent.com/BoiseState/CS121-resources/master/projects/README_TEMPLATE.md). See README_EXAMPLE (https://raw.githubusercontent.com/BoiseState/CS121-resources/master/projects/README_EXAMPLE.md) for an example.

Submission

You will follow the same process for submitting each project.

1. Open a console and navigate to the project directory containing your source files,
2. Remove all the `.class` files using the command, `rm *.class`.
3. In the same directory, execute the submit command for your section as shown in the following table.
4. Look for the success message and timestamp. If you don't see a success message and timestamp, make sure the submit command you used is EXACTLY as shown.

Required Source Files

Required files (be sure the names match what is here exactly):

- `Song.java`
- `PlayList.java`
- `Jukebox.java` (main class)
- `SongTest.java` (provided class)
- `PlayListTest.java` (provided class)

- `TimedAudioClip.java` (provided class)
- `playlist.txt` (provided test file)
- `autograde.sh` (provided test file)
- `README`

Section	Instructor	Submit Command
1	Nathan Schmidt (TuTh 1:30 - 2:45)	`submit nschmidt cs121-1 p3`
2	Marissa Schmidt (TuTh 9:00 - 10:15)	`submit marissa cs121-2 p3`
3	Elena Sherman (TuTh 10:30 - 11:45)	`submit esherman cs121-3 p3`
4	Marissa Schmidt (TuTh 4:30 - 5:45)	`submit marissa cs121-4 p3`

After submitting, you may check your submission using the "check" command. In the example below, replace <instructor> with your instructor and <x> with your section.

```
submit -check instructor cs121-x p2
```

CS121 (https://cs.boisestate.edu/~cs121) is maintained by Marissa Schmidt (https://cs.boisestate.edu/~marissa).