School of Engineering and Technology, UW Tacoma
TCSS 305 Programming Practicum
Assignment 4 – The Astonishing Race

Part A Value: 7% of the course grade
Part A Due: 26 February 2020
Part B Value: 7% of the course grade
Part B Due: 6 March 2020
Part C Value: 6% of the course grade
Part C Due: 13 March 2020

Program Description:

This assignment is designed to test both your understanding of writing graphical applications in Java and writing an efficient model. You will add event handlers to and edit a provided graphical user interface (GUI) for an application that displays and manipulates a race. You will also write a model that accepts a pre-made race file. The Oracle Swing tutorial provides explanations and examples which may be helpful while working on this assignment.

There are three main development goals for this assignment, a model of a race, a GUI controller for the model, and a GUI view of the race model. The race model will implement a provided interface called **`PropertyChangeEnabledRaceControls.java`** which extends the interface **`RaceControls.java`**(defined later). The interface defines the only allowed public methods to the model. All of the methods in the interface are mutators; there are no accessors. The only way for the model to communicate with interested parties is by implementing the PropertyChangeListener API. The application will read a race file (format defined later) that contains all of the information about a given race.

The GUI controller will be in a window and will contain a reference to the model using the **`PropertyChangeEnabledRaceControls.java`** interface. The GUI will include a timer that, when running, will ask the model to advance through the race. The provided GUI controls will call different methods found in the **`RaceControls`** interface, thereby manipulating the race.

The GUI view will be in a separate window than the GUI controller. It will not store a reference to the model and will only react to changes in the model through implementing the PropertyChangeListener API.

Race Model:

- The model must implement **`PropertyChangeEnabledRaceControls.java`**. The methods defined in the interface are the ***only*** public methods allowed in the class that implements the interface. However, this class can have a public constructor.

Method	Description
`advance()`	Advances the race's internal "clock" by 1 millisecond. All registered listeners will be notified of both the "time" change and any messages that occur during this advance. If the race is advanced beyond its total amount of time, it will not throw an exception but will notify all registered listeners no time remains.
`advance(`*`theMilliseconds`*`)`	Advances the race's internal "clock" by theMilliseconds milliseconds. All registered listeners will be notified of both the "time" change and any messages that occur during this advance. If the race is advanced beyond its total amount of time, it will not throw an exception but will notify all registered listeners no time remains.
`moveTo(`*`theMilliseconds`*`)`	Move the Race's internal "clock" to theMilliseconds milliseconds after the start of the race. All registered listeners will be notified of the "time" change. Starting at theMilliseconds and working backward in "time," all registered listeners will be notified of the new most recent leaderboard and telemetry message (for all racers).
`loadRace(`*`theRaceFile`*`)`	Load a file containing race information. All registered listeners will be notified of progress updates during the load. All registered listeners will be notified of information in the *header* message when the loading process completes. Throws IOException if theRaceFile is not in the correct format as defined in this document.

- All classes must reside in the **`model`** package (or sub-packages).
- The model will need to communicate changes to front-end modules. The model will need to implement the PropertyChangeListener API to achieve this communication. The main Race class in the model will implement PropertyChangeSupport as a member field. Since none of the allowed public methods in the model return values, you will need to fire property change events to all listening property change listeners. Carefully consider the property names used when calling **`firePropertyChange`**.
  - Use the provided PROPERTY_* constants defined in **`PropertyChangeEnabledRaceControls`** when firing property change events.
- The model is not allowed to store any explicit references to front end objects. All communication of the Race state to the GUI will be through the use of property change events. (See:PropertyChangeListener API)
- The following page contains the data definition specification (DDS) for applicable race files.
  - Race files generated from the race file generator application will generate files with a *.rce file extension. However, this file extension is not a requirement. Your application must be able to read a file with any file extension. If the file does not meet the required format as defined in the DDS, **`loadRace(file)`** should throw an **`IOException`**.

.rce Required Format

Race information header messages start with # and will be in the following order:
#RACE:<Race Name>
#TRACK:<Track GEOMETRY>
#WIDTH:<TRACK WIDTH RATIO>
#HEIGHT:<TRACK HEIGHT RATIO>
#DISTANCE:<SINGLE LAP DISTANCE>
#TIME:<TOTAL RACE TIME IN MILLISECONDS>
#PARTICIPANTS:<NUMBER (**N**) OF PARTICIPANTS>
#<RACER ID>:<RACER NAME>:<STARTING DISTANCE> (Repeated **N** times)

Immediately following the race information header messages are the race entries.

- All fields found in an entry are separated by a `:` (colon - ascii #58) character
- There is no whitespace inside an entry
- Entries are delimited by a newline character “\n”
- Timestamps are milliseconds, increasing from 0 (race start)
- Telemetry messages occur for all drivers approximately every 5ms.
- Leaderboard messages occur whenever one racer passes another racer.
- Finish Line Crossing messages occur whenever a racer crosses the start/finish line.
- The entries provide a telemetry message for all racers at timestamp 0.
- The entries provide a leaderboard message at timestamp 0.

Telemetry

`$T:timestamp:racerId:distance:lap`

Racer ID	Non-negative integer less than 100
Distance	Distance travelled this lap. Double rounded to two digits
Lap #	0 indexed integer

Leaderboard

`$L:timestamp:racerInFirst:racerInSecond : … :`

Racer ID’s	Everything after the timestamp is a racer id, sorted by place

Finish Line Crossing

`$C:timestamp:racerId:newLap:isFinish`

Racer ID	The racer who just went over the finish line
New Lap #	The lap the racer is just started
Is Finished	“true” if the racer is now finished with the race, “false” otherwise

Model - Implementation Guidelines and Hints:

Messages

The race files contain many messages, some as many as two million. While this may seem like a lot, you can still load all of them into a data structure in memory. The model should read each file only once and not refer back to it.

A message consists of the parsed data of one line in the race file. Each line in the file past the header is a message. The messages will need to be parsed into information and turned into objects. You will need to decide how to store the information in each message. With three different types of messages, can you create an inheritance hierarchy? Are there behaviors/state that all message types perform the same (abstract class)? Should they be mutable or immutable?

Please use the interfaces found in the **`model.messages`** package to define your concrete message classes. Pay attention to the inheritance hierarchy within these interfaces. Do not add any public methods beyond those defined in these interfaces.

Race Information

Race information encompasses all fields in the beginning of the file that start with "#". Use the interfaces defined in the **`model.info`** package to define your concrete race information classes. Race information should be broadcasted (fire a property change event with `PROPERTY_RACE_INFORMATION`) when the model completes loading the entire race file. Your GUI may use this broadcast as a signal that the file load is complete.

Race Logic

Please use the interfaces found in the **`model`** package to define your concrete race logic class(es).

Once you have established how you will store individual messages, you will need a data structure to store all of the messages. The model needs to keep track of its current location in regard to the time stamps. Races start at time stamp 0 and move forward in time from there. You may want to consider a two-dimensional structure. The first dimension's index values can represent the time stamps while the second dimension will represent any messages at the time stamp. Just note that a given millisecond may have 0 to many messages.

Use the following set of messages for the following examples.

```
$T:1:10:8.32:0
$T:4:10:20.79:0
$T:4:30:-9975.61:0
$T:4:50:-19973.27:0
$T:5:40:-14971.31:0
$T:8:20:-4954.02:0
$T:9:50:-19946.54:0
$L:9:10:20:30:40:50
$T:10:10:45.75:0
$T:10:20:-4943.80:0
$T:12:20:-4933.58:0
$T:12:30:-9936.58:0
```

`advance()` - Assume race's current time is 4 and **`advance()`** –or– **`advance(1)`** is called. The race's current time should change to 5 (fire a property change event with `PROPERTY_TIME`) and all 3 messages with a timestamp of 4 should individually be broadcast (fire a property change event with `PROPERTY_MESSAGE` for EACH message).

`advance(n)` - Assume race's current time is 4 and **`advance(5)`** is called. The race's current time should change to 9 (fire a property change event with `PROPERTY_TIME`) and all 5 messages with a timestamp of 4-8 should individually be broadcast (fire a property change event with `PROPERTY_MESSAGE` for EACH message).

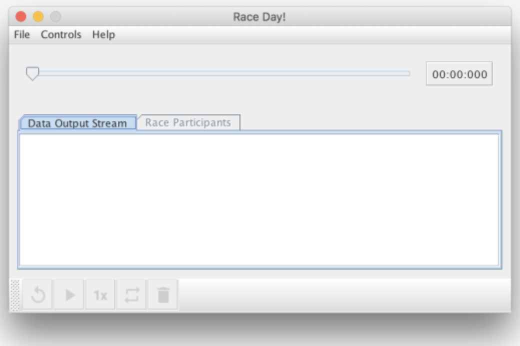
Notice in both cases, the messages ***at*** the current time are not broadcasted.

`moveTo(n)` - Assume race's current time is 4 and **`moveTo(12)`** is called. The race's current time should change to 12 (fire a property change event with `PROPERTY_TIME`) and the following 5 messages should individually be broadcast (fire a property change event with `PROPERTY_MESSAGE` for EACH message). NOTE: If **`advance()`** is called immediately following this **`moveTo(12)`**, the messages with a timestamp of 12 will be re-broadcasted.

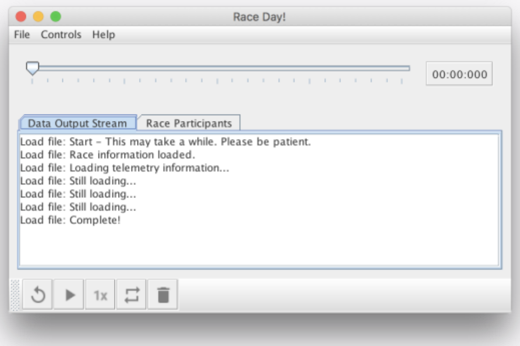
```
$T:5:40:-14971.31:0
$T:9:50:-19946.54:0
$L:9:10:20:30:40:50
$T:10:10:45.75:0
$T:12:20:-4933.58:0
$T:12:30:-9936.58:0
```

Race Controller:

Controller GUI initial state:

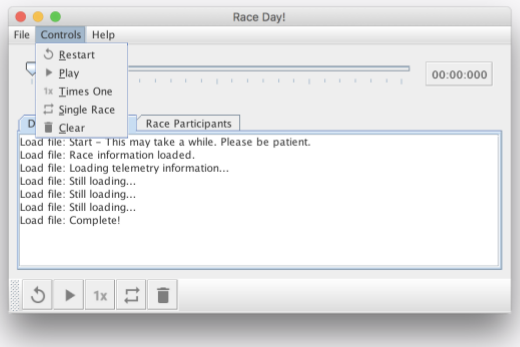


Race loaded:



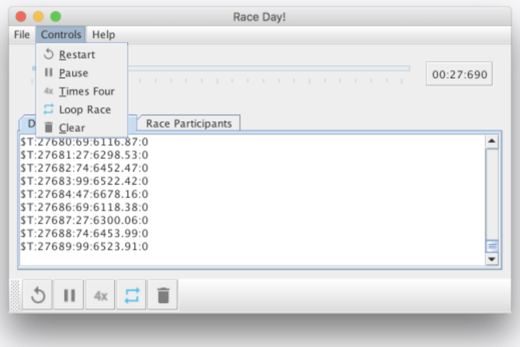
```
Load file: Start – This may take a while. Please be patient.
Load file: Race information loaded.
Load file: Loading telemetry information...
Load file: Still loading...
Load file: Still loading...
Load file: Still loading...
Load file: Complete!
```

Controls in initial state:



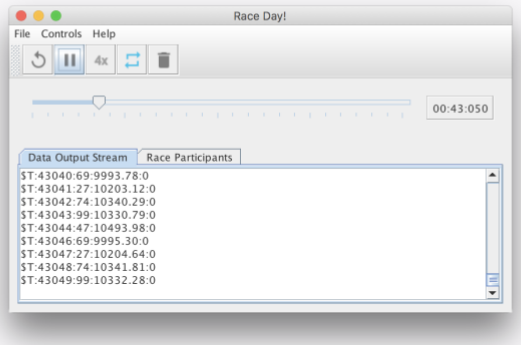
```
Load file: Start – This may take a while. Please be patient.
Load file: Race information loaded.
Load file: Loading telemetry information...
Load file: Still loading...
Load file: Still loading...
Load file: Still loading...
Load file: Complete!
```

Controls in switched state:



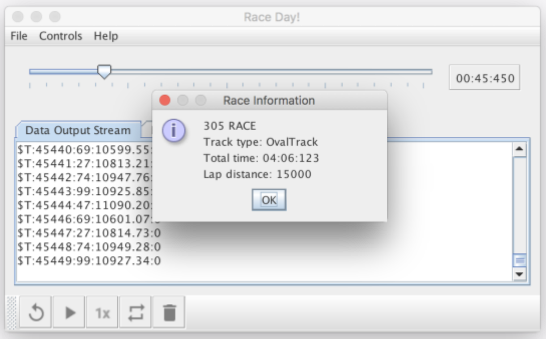
```
$T:27680:69:6116.87:0
$T:27681:27:6298.53:0
$T:27682:74:6452.47:0
$T:27683:99:6522.42:0
$T:27684:47:6678.16:0
$T:27686:69:6118.38:0
$T:27687:27:6300.06:0
$T:27688:74:6453.99:0
$T:27689:99:6523.91:0
```

Move the toolbar and race running:



```
$T:43040:69:9993.78:0
$T:43041:27:10203.12:0
$T:43042:74:10340.29:0
$T:43043:99:10330.79:0
$T:43044:47:10493.98:0
$T:43046:69:9995.30:0
$T:43047:27:10204.64:0
$T:43048:74:10341.81:0
$T:43049:99:10332.28:0
```

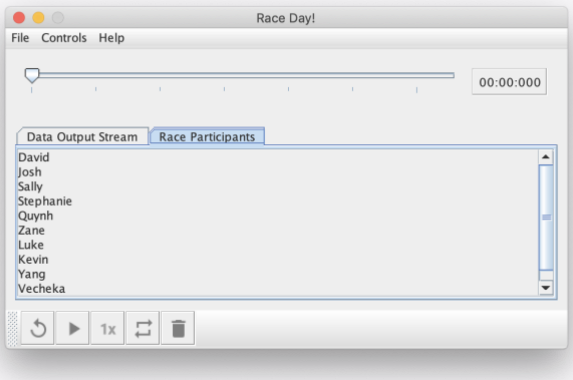
Race Information dialog:



```
305 RACE
Track type: OvalTrack
Total time: 04:06:123
Lap distance: 15000
```

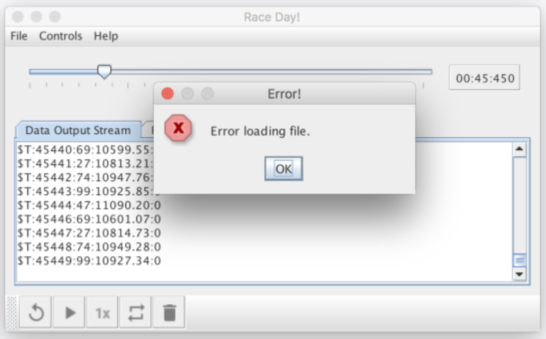
```
$T:45440:69:10599.55:
$T:45441:27:10813.21:
$T:45442:74:10947.76:
$T:45443:99:10925.85:
$T:45444:47:11090.20:
$T:45446:69:10601.07:
$T:45447:27:10814.73:0
$T:45448:74:10949.28:0
$T:45449:99:10927.34:0
```

Race Participants tab:



```
David
Josh
Sally
Stephanie
Quynh
Zane
Luke
Kevin
Yang
Vecheka
```

Error Message for incorrectly formatted file:



```
Error loading file.
```

```
$T:45440:69:10599.55:
$T:45441:27:10813.21:
$T:45442:74:10947.76:
$T:45443:99:10925.85:
$T:45444:47:11090.20:0
$T:45446:69:10601.07:0
$T:45447:27:10814.73:0
$T:45448:74:10949.28:0
$T:45449:99:10927.34:0
```

Notes:

- When the window is closed, the program should terminate.
- All classes must reside in the `controller` package (or sub-packages).

The controller will manipulate the model with the help of a timer. The timer is responsible for advancing the race in "time" by calling the `advance()` method in the model. The controller will have several different modes of operation.

Race Modes:

- **Play mode**: the timer is running and currently advancing the model.
    - Slider is disabled
    - Pause icon and text is visible
- **Pause mode**: the timer is not running and not advancing the model.
    - Slider is enabled
    - Play icon and text is visible
- **Times one mode**: may be set independently of Play/Pause mode but only has a visible effect when Play mode is enabled. When the race is in Times One mode and the race is in Play mode, the time will advance the race model one second per one real world second.
    - Times one icon and text is visible
- **Times four mode**: may be set independently of Play/Pause mode but only has a visible effect when Play mode is enabled. When the race is in Times Four mode and the race is in Play mode, the time will advance the race model four seconds per one real world second.
    - Times four icon and text is visible
- **Single Race mode**: may be set independently of Play/Pause mode but only has a visible effect when Play mode is enabled. When the model is advanced to the end of the race, the timer will stop.
    - Single Race icon and text is visible
- **Loop Race mode**: may be set independently of Play/Pause mode but only has a visible effect when Play mode is enabled. When the model is advanced to the end of the race, the timer continues and the model's race time is reset to the beginning of the race.
    - Loop race icon and text is visible

NOTE: `javax.swing.Timer` is not guaranteed to fire an event at the exact expected millisecond. Some "time slippage" is acceptable in your simulation. You are not required to write code that adjusts for this affect. The more often your timer fires events, the more the time slippage will become evident. If your timer does not fire an event often enough, your racers may appear to jitter or jump upon render. Start with your timer firing an event every 30ms and adjust from there as needed.

The window should have a menu bar with the following items.

File

Load Race…	A menu item which invokes a file chooser dialog
----	(separator)
Exit	A menu item which closes the application

Controls

Restart	a menu item; sets the race time to 0.
Play/Pause	a menu item that switches between the text Play/Pause; switches the controller between Play mode and Pause mode.
Times One/Times Four	a menu item that switches between the text Times One/Times Four; switches the controller between Time one mode and Times four mode.
Single Race/Loop Race	a menu item that switches between the text Single Race/Loop Race; switches the controller between Single Race mode and Loop Race mode.
Clear	a menu item; clears all text from the Output area.

NOTE: all controls menus item should be disabled when the model does not have a race loaded
NOTE: the menu items with a "first/second" will either display first or second, but never both.
NOTE: the controls menu items should display a small icon and text. The icon should match the toolbar icons, but smaller. The menu icons should be 12x12 pixels. The icons are provided and found in the images folder in the Eclipse project for this assignment.

Help

Race Info... (pops up an option pane with the following message:

```
<Current Race Name>
Track type: <Current Track Type>
Total time: <Total Time of Current Race in MM:SS:mmm>
Lap distance: <Current race lap distance>
```

The Icon shown on this message should be the system information icon)
NOTE: this menu item should be disabled when the model does not have a race loaded

About... (pops up an option pane with the following message:

```
<Your Name>
Winter 2020
TCSS 305
```

You may add other information to this message if you wish.
The Icon shown on this message should match the one on the JFrame.)

NOTE: The three dots shown in the Load Race…, Race Info…, and About… menu items are called an ellipsis, which is a standard convention to indicate that selecting that item will open a pop up dialog.

When the user clicks `" Load Race..."`, a file selection dialog appears to let the user load a race file. Have the file selection dialog open in the current directory (use relative addressing). When the user selects a race file, the model will load the file and inform any interested parties of its progress and completion. When the model informs the Controller GUI of file load completion, all initially disabled components should enable.

Use a **`JFileChooser`** for your file selection dialog. An open dialog can be shown by calling **`showOpenDialog`**. You can ask for the file the user selected by calling the **`getSelectedFile`** method on the **`JFileChooser`**. If race file loading fails, catch the **`IOException`** and use a **`JOptionPane`** to display an error message dialog like the one shown on the page 6 of these instructions. Your program should create only one **`JFileChooser`** object and reuse it repeatedly.

The window should contain the following components:

- The main window may be broken into three portions, top, bottom, and a tool bar.
  - The top portion should contain a slider control and a label. The top portion's width should match the width of the bottom portion.
    - The slider is initially disabled and does not contain any tick marks. After a race file is loaded into the model, the slider enables, has a maximum value set to the length of the race, displays major tick marks every minute and displays minor tick marks every 10 seconds. When the user adjusts the slider, a call to the **`moveTo(int)`** method in the model is made. The slider should adjust its value when a time change has occurred in the model. If a different race file is loaded into the model, the slider should adjust its maximum value accordingly. You may either wait to call **`moveTo`** when the user stops moving the slider or make multiple calls to **`moveTo`** while the slider is moving.
    - The label should have a border surrounding the text. The text reflects the current time of the race in MM:SS:mmm (M = minute, S = second, m = millisecond). The label should not update from any action in the Controller GUI. It will only update when a time change has occurred in the model.
  - The bottom portion should contain a tabbed pane.
    - The first tab's label is "**Data Output Stream**." It contains non-editable text area with a vertical scroll bar. The text area should be sized with 10 rows and 50 columns (this will dictate the width of the bottom portion). The text area should not update from any action in the Controller GUI. It will only update when a change that requires output display has occurred in the model. When the model notifies via PropertyChangeEvent of any race message update, display the update in this text area. Note: The model will send messages during file loading. See Part B extra credit.
    - The first tab's label is "**Race Participants**." This tab is initially disabled. After a race file is loaded into the model, the tab enables and contains a label component for every race participant delivered from the model. The text for each of the participant checkboxes should include the name of the racer.
  - A **toolbar** (`JToolBar`) with five buttons in horizontal orientation in the south region of the window. The first four buttons represent controls that command the model. The fifth button clears the text from text area. All buttons are initially disabled. When a race file is loaded into the model, the buttons enable. Notice that the toolbar and the "Controls" menu contain the same commands; they should behave identically when invoked from either place. For example, selecting a control button (such as the "Play/Pause" button) should change the icon and text and cause the corresponding menu option to change, and vice-versa. For full credit, the behavior of each control must be defined in your code only once, and you must be able to add controls to both the toolbar and the "Controls" menu without writing redundant code (without redundantly specifying the display names, or event handlers for the controls). The control buttons in the toolbar should display only icons, no text. The icons are provided and found in the images folder in the Eclipse project for this assignment. The toolbar icons should be 24x24 pixels.
    - **Restart button**: Sends a command to the model requesting the race time to 0.
    - **Play/Pause button**: This button will change between the Play icon and the Pause icon. When the button is pressed while displaying the Play icon, the controller will switch to Play mode. The slider should disable when the controller is in Play mode. When the button is pressed while displaying the Pause icon, controller will switch to Pause mode. The slider should enable when the controller is in Pause mode.
    - **Times one/Times four button:** This button will change between the Times one icon and the Times four icon. When the button is pressed while displaying the Times one icon, the controller will switch

to Time four mode. When the button is pressed while displaying the Times four icon, the controller will switch to Time one mode.

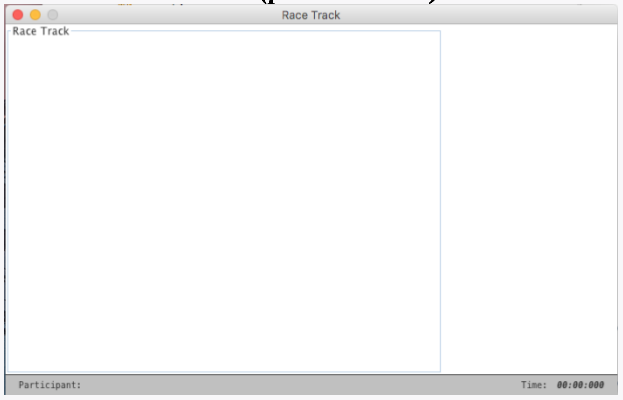
- **Single Race/Loop Race:** This button will change between the Single Race icon and the Loop Race icon. When the button is pressed while displaying the Single Race icon, the controller will switch to Loop Race mode. When the button is pressed while displaying the Loop Race icon, the controller will switch to Single Race mode.
- **Clear button**: Removes all text from the text area.

Note: To synchronize the control buttons and the control menu items, use the `Action` interface.

Race View:

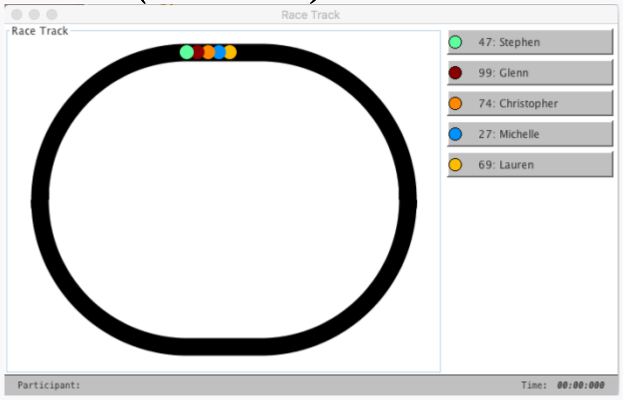
- The window title must be **`"Race Track"`** and the title bar must contain an icon other than the default Java icon. Be creative; choose an icon that seems appropriate to you. (The screenshots do not show an icon because in Mac OS X, title bars do not have icons.)
- When the application is initially started, and no race is loaded, the Race View will appear but will be largely empty.
- The window must fit the preferred size of the components in it (using `pack`) and should not be resizable.
- The GUI should use the 'Metal' look and feel as shown in the second JFrame code example this quarter.
- When the Race View window is closed, the program should not terminate. The Controller window should remain open.
- All classes must reside in the **`view`** package (or sub-packages).
- Styling:
  - The only component that is explicitly sized in this UI is the panel that contains the **`track.VisibleRaceTrack`** (more on usage later). The panel is required to be sized **500x400**.
  - Once a race is loaded, the leader board must display the list of racer names in the order of first to last. You have complete freedom on how to style and where to display the leaderboard.
  - A status bar at the bottom of the frame must display the current time of the race simulation.
    - EXTRA CERDIT: When the user selects a race participant, the status bar must also display the participant's name, number, and current distance around the track.
    - Note: There is NO **`JStatusBar`** component. My solution involves a styled JPanel placed at the bottom of the frame.
- Below are some screen shots of the window in various states:

View GUI initial state (pre-race load):



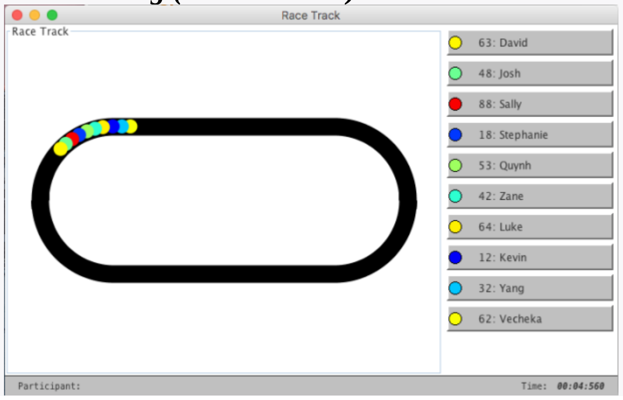
```
Participant:                                   Time:  00:00:000
```

Race loaded (track size 5x4):



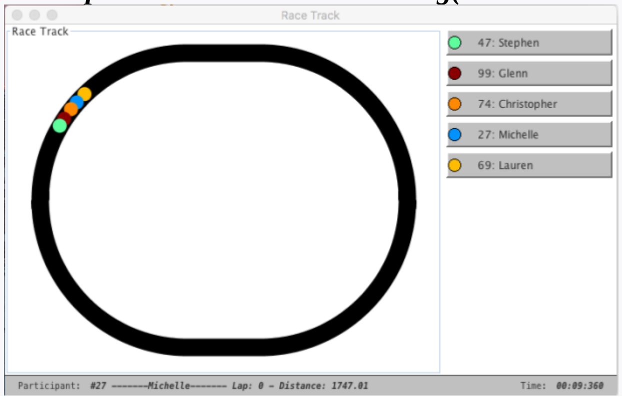
```
Participant:                                   Time:  00:00:000
```

Race Running (track size 5x2):



```
Participant:                                   Time:  00:04:560
```

Participant Selected – Race Running(track size 5x4):



```
Participant:  #27 -------Michelle------- Lap: 0 - Distance: 1747.01      Time:  00:09:360
```

Race View: Displaying the Race Track

The Eclipse starter archive includes a library called track.jar. You can test that it is added to your project by right-clicking on the top-level project -> Build Path -> Configure Build Path. In the dialog, click the Libraries tab. Verify track.jar is included.

This library includes a single class called **`VisibleRaceTrack`** inside of the **`track`** package. **`VisibleRaceTrack`** is a direct child of **`Ellipse2D.Double`** and may be used just like you would use an **`Ellipse2D.Double.`** Use this class to display the track in the panel. To display the current location of each racer, **`VisibleRaceTrack`** includes a public method: **`getPointAtDistance()`**.

An example using this class is included in the Lecture materials: *Swing GUI Observer Example (Time)*.

To use this class, you will need to import it. In the using class, add the statement:

```
import track.VisibleRaceTrack;
```

Documentation of the class is posted here: http://faculty.washington.edu/cfb3/raceday

Requirements: the visible race track must:

- be centered both vertically and horizontally inside of the containing panel and include a small amount of spacing from the sides of the panel (20-40 pixels)
  - Note: tracks may have a minimum ratio of 5x2 and a maximum ratio of 5x4
  - ensure that tracks of different ratios are centered
- include a line boarder around the containing panel that includes the text Race Track

Race View: Displaying the Racers

Requirements: the visible race track, each race participant must:

- be visually displayed in the correct location on the race track (You may use any shape/form to display the racers)
- have a unique color when displayed – this color must be visible both on the race track and the leaderboard
- move around the track as the model sends telemetry updates

Requirements: the visible leader board must:

- display the race participants in order
- display the color, number, and name of each race participant
- change order of race participants as the model sends leader board updates

Race View: Status Bar

Requirements: the Status Bar must:

- always display the current race time
- EXTRA CREDIT:
  - when an individual race participant is selected, display the racers name, number, and current distance around the track
    - the user may select a race participant by clicking on the race participant in the leaderboard or the visible track

Race View: Implementation Guidelines and Hints

- Most of the classes in the view package will need to implement **`PropertyChangeListener`**. View classes are informed of model changes through the **`propertyChanged()`** method.
- Classes in the view package must not include any references to classes in the controller package.
- Classes in the view package must not include any references to **`PropertyChangeEnabledRaceControls`** or its subclasses (other than the Property constants). References to Message (Telemetry, LeaderBoard, LineCrossing) objects are acceptable as they are passed to the view via **`PropertyChangeEvents`**. It is recommended to keep the scope of these references to local variables.
- Attempt to decouple your solution as best as possible. You should be able to implement the leaderboard, track, and status bar as separate classes. You may implement the Observer patter to inform the status bar of which racer is selected!

Submission and Grading:

It is a good idea to test your code on lab computers in SCI 106/108 or DOU 110 before your final submission.

For Part A

- The *model* should have a concrete class implements the **`PropertyChangeEnabledRaceControls`** interface but with full functionality. You will be provided with a JUnit test case to check your process and correctness. You must implement the provided interfaces to allow the test cases to work.
    - The model should be free of all PMD/Checkstyle warnings
- The appearance of **all** GUI *controller* elements must be exactly as described except as noted in this paragraph. The only component that should have functionality is the "Load Race…" menu item. When the "Load Race…" menu item is clicked, the file chooser dialog should appear exactly as described in the document. When a file is chosen, it should be sent to the model through the loadRace() method. If an IOExcpetion occurs, the error message dialog must appear. Nothing should happen with the controller when an IOExeption does not occur.
    - Note: You need to disable all components that are required to be disabled upon application start.
- No implementation of the GUI *view* is required.

For Part B

- The *model* implementation should be complete from part A.
- The appearance and functionality of all GUI *controller* elements should be complete. This includes implementing the timer and implementation of the PropertyChangeListener API to listen for and react to notifications from the *model*.
- No implementation of the GUI *view* is required.

Extra Credit (Part B)

Up to 5% extra credit can be earned by completing the following challenge. File I/O causes blocking operations. For the GUI window to display the updates while the race file is loading, the file load must be performed in a background thread. Without using a background thread, you can use visual clues for your user that the application is performing work such as changing the cursor from the standard arrow to the wait icon.

You will receive up to 1% extra credit if the cursor changes to the wait icon before the file starts to load and returns to the default icon when the file load completes.

You will receive up to 1% extra credit if the model sends updates via **`PropertyChangeEvent`** while the file is loading and a **`PropertyChangeListener`** displays the update messages to the Console. NOTE: the **`System.out.println`** calls must be in a PCL defined in the controller package, not the model package.

-OR-

You will receive up to 4% extra credit if you implement a background thread that allows the JTextArea to display updates while a file is loaded. You will need to research working with a background thread. Try: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html

For Part C

- All features must be finished, and you will be graded on both external and internal correctness. It is possible to lose points for the same issues on Part A, Part B, and Part C; if you have external correctness problems with Part A or B, and they remain in your Part C submission, you will lose points for them twice.
- The appearance and functionality of all GUI *view* elements should be complete. This includes implementation of the property change listener API to listen for and react to notifications from the *model.*

Create your Eclipse project by downloading the `hw4-project.zip` file from Canvas, importing it into your workspace, and using "Refactor" to change "username" to your UWNetID. Remember to make this change ***before*** you first commit the project to Subversion. When you have checked in the revision of your code you wish to submit, make a note of its Subversion revision number. To get the revision number, *perform an update* on the top level of your project; the revision number will then be displayed next to the project name. Your revision number will pick up where you left off on Assignment 4; if you submitted revision 48 of Assignment 4, the first commit of Assignment 4 will have a number greater than 48. To submit Part A, commit your project to Subversion and submit an executive summary on Canvas as was done for previous assignments. Then, continue working *on the same Eclipse project* for Part B. When you are finished with Part B, commit your changes to Subversion and submit another executive summary on Canvas. The required filename for the Part A executive summary is "**`username-raceday-a.txt`**". The required filename for the Part B executive summary is "**`username-raceday-b.txt`**". The required filename for the Part C executive summary is "**`username-raceday-c.txt`**", where `username` is your UWNetID. As on previous assignments, executive summaries will *only* be accepted in plain text format.

External correctness of Part A is based on the GUI's appearance and on the ability to load a race file. Part A code *will not be examined* except under extraordinary circumstances (odd GUI appearance); Part A is graded *entirely* on external correctness.

The external correctness of your Part B submission will be graded on the GUI controller's behavior, which is observed by running the GUI, clicking various buttons, attempting to "play" a race, and examining the result. Your GUI should match the expected layout and should be positioned, sized, and resize identically to expectations. Exceptions should not occur under normal usage. Your program should not produce any console output.

Internal correctness of Part B will be based on following the program specification, inclusion of reasonable comments, the use of meaningful identifier names, encapsulation, and the avoidance of redundancy. In addition, the output of the plugin tools will be used for Part B. The internal correctness grade will focus on the code inside of your model package but the code in the GUI controller will also be considered.

The external correctness of your Part C submission will be graded on the GUI view's behavior, which is observed by running the GUI controller, clicking various buttons, attempting to "play" a race, and examining the result in the view. Your GUI should match the expected layout and should be positioned, sized, and resize identically to expectations. Exceptions should not occur under normal usage. Your program should not produce any console output.

The Part A percentage breakdown is 10% executive summary and 60% external correctness and 30% internal correctness.
(the grade for external correctness will be deduced mainly -but not entirely- from the ability of your model to pass the provided JUnit tests)
For Part B, the percentage breakdown is 10% executive summary, 60% external correctness, and 30% internal correctness.
For Part C, the percentage breakdown is 10% executive summary, 60% external correctness, and 30% internal correctness.