Photo by Marvin Meyer on Unsplash

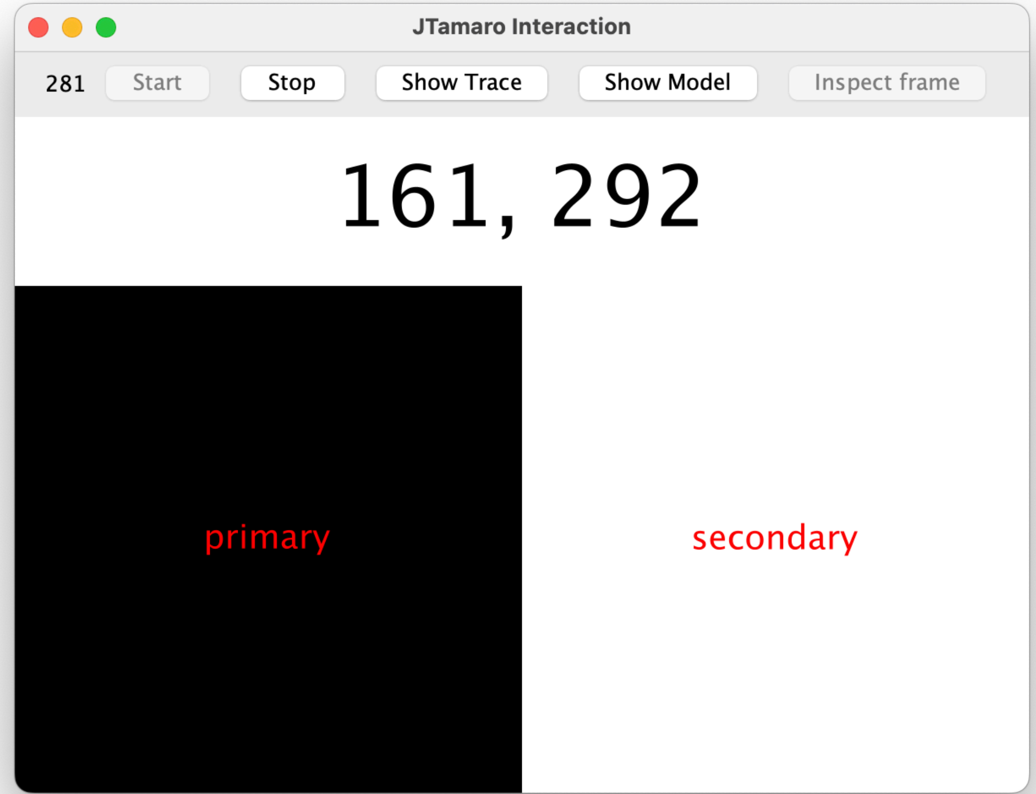
Lab 10

Mouse Visualizer • Dice Game

Copy methods from your Lab 9 Toolbelt

In earlier labs you added methods to your `Toolbelt` class. The `Toolbelt` class included in the Lab 10 starter repository is missing these methods. Please **add the methods of your Lab 9 `Toolbelt`** class to the Lab 10 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future). **DO NOT** remove the methods that are currently defined in the starter `Toolbelt` class, as they are needed across the project.

Mouse Visualizer



This exercise shows you how to create a simple GUI application.

Let's implement a “Mouse Visualizer” application that always shows the current state of the mouse, consisting of:

- the x and y coordinate of the mouse pointer
- the state (pressed or not) of the *primary* button (i.e., left mouse button / primary trackpad click)
- the state (pressed or not) of the *secondary* button (i.e., right mouse button / secondary trackpad click)

Let's create three classes: a main class (`Main`), a model class (`AppModel`), and a UI class (`AppUI`). The model keeps track of the position and the state of the two buttons. It also contains methods that create a new model whenever there are changes in the position or the status of a mouse button.
This architecture separates the UI code to produce graphics and the model.

Task A1

Class:	`lab.mouse.AppModel`
Task:	We always should start with the model. For this simple application, the model's logic is trivial. Define an `AppModel` record with four components: `x` and `y` (integers), and two booleans that indicate whether the primary and the secondary mouse button is pressed, respectively. Add a static, parameterless `create` method that returns an `AppModel` instance with default values (coordinates at 0 and buttons not pressed). Add a method `updatePosition(int x, int y)` that returns a new model with an updated position based on the parameters. Similarly, add two methods `updatePrimaryButton(boolean pressed)` and `updateSecondaryButton(boolean pressed)` that return a new model updating respectively the primary and the secondary button (with `true` or `false`, depending on the parameter value).
Run in JShell:	`AppModel.create()`

Result:	`AppModel[primaryButtonPressed=false,`
	`secondaryButtonPressed=false,`
	`x=0, y=0]`

Run in JShell:	`AppModel.create()`
	`.updatePosition(10, 20)`
	`.updateSecondaryButton(true)`

Result:	`AppModel[primaryButtonPressed=false,`
	`secondaryButtonPressed=true,`
	`x=10, y=20]`

Task A2

Class:	`lab.mouse.AppUI`
Task:	Implement a method that renders an `x` and `y` position: `public static Graphic position(int x, int y)` Do this by overlaying a text on top of a rectangle. The rectangle should have width twice as big as `INDICATOR_SIZE` and the height should be 1/6 of the width.
Run in JShell:	`show(AppUI.position(100, 200));`
Output:	100, 200

Task A3

Class:	`lab.mouse.AppUI`
Task:	Implement a method that produces a graphical indicator for a mouse button: `public static Graphic mouseButtonIndicator(String name, boolean pressed)` Do this by overlaying a red text over a square. The square on the background should be black if the button is pressed, and white otherwise.
Run in JShell:	`show(AppUI.mouseButtonIndicator("pressed", true));` `show(AppUI.mouseButtonIndicator("not pressed", false));`
Output:	pressed and not pressed

Task A4

Class:	`lab.mouse.AppUI`
Task:	Implement a method that renders the complete app: **`public static`** `Graphic render(AppModel model)` The two mouse button indicators are below the position.

Run in JShell:	`show(`
	`  AppUI.render(`
	`    AppModel.create()`
	`    .updatePosition(10, 25)`
	`    .updateSecondaryButton(true)`
	`  )`
	`);`

Output:	10, 25 primary secondary

Task A5

<table>
<tr><td>Class:</td><td><code>lab.mouse.Main</code></td></tr>
<tr><td>Task:</td><td>We can finally write mouse handlers to react to mouse events, updating our model.

Define a method
<code>public static AppModel mouseMoveHandler(AppModel m, Coordinate c)</code>
which extracts <code>x</code> and <code>y</code> from a <code>Coordinate</code> and calls <code>updatePosition</code> on the model.

Define then two methods
<code>public static AppModel mousePressHandler(AppModel m, Coordinate c, MouseButton b)</code>
and
<code>public static AppModel mouseReleaseHandler(AppModel m, Coordinate c, MouseButton b)</code>
which both first update the coordinates in the model.
They should then check which mouse button has been pressed. You can call <code>getButton</code> on a <code>MouseButton</code> instance and check which one has been pressed. Use <code>MouseButton.PRIMARY</code> and <code>MouseButton.SECONDARY</code> as constants for the two mouse buttons of our interest.

Call <code>updatePrimaryButton</code> and <code>updateSecondaryButton</code> on the model, depending on which button is involved in the event. Pass as an argument a boolean, indicating whether the button has been pressed (true) or released (false).

Lastly, implement a <code>play</code> method to configure the <code>Interaction</code> so it responds to mouse events.

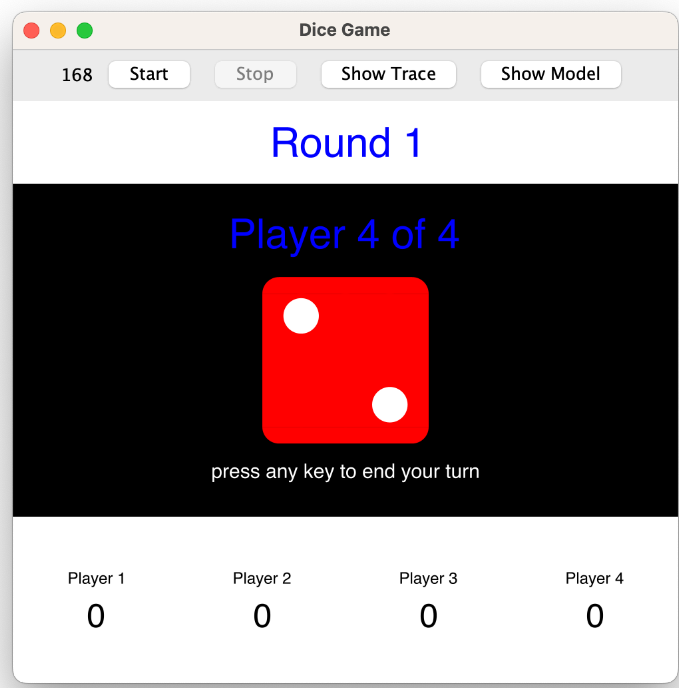
<code>public static void play()</code>

• Create a new <code>Interaction</code>, passing an <code>AppModel</code> to the constructor (use the <code>AppModel.create()</code> method you defined to create one).
• Call <code>withRenderer</code> on the <code>Interaction</code> object to pass a method reference to your <code>render</code> method.
• Call <code>withMouseMoveHandler</code>, <code>withMousePressHandler</code> and <code>withMouseReleaseHandler</code> on the resulting object to pass respectively a method reference to your mouse move, press and release handler methods.
• Call <code>run()</code> on the resulting object to run the interaction</td></tr>
<tr><td>Run in JShell:</td><td><code>lab.mouse.Main.play();</code></td></tr>
</table>

B. Dice Game

Now that you have an idea for the architecture of a complete GUI application, we can raise our ambitions and develop a slightly more sophisticated game.

We will develop a dice game, supporting a variable number of players. Every round, each player rolls a die; when everyone has rolled, all the players with the highest rolled number get one point.



The current round is always visible at the top, while the scores of all players are always visible at the bottom.
The part in the middle shows different information, depending on the phase of the game: when the round is about to begin, it asks to click with the mouse to start taking turns. Then, it will ask each player to click with the mouse to roll, show the result and click to proceed to the next player. When everyone has rolled in a round, it shows the results with the round winners.
At any point, pressing R on the keyboard resets the game to its original state.

See the next page for an example of a game with four players progressing for a full round.

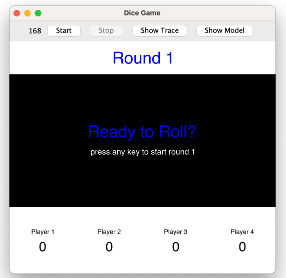


Figure 1: The first round begins

Figure 2: The first player is about to roll

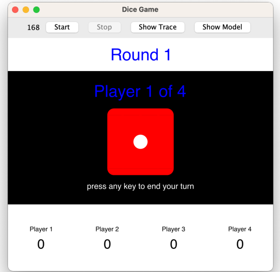


Figure 3: The first player has rolled 1

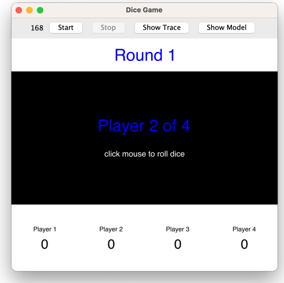


Figure 4: The second player is about to roll

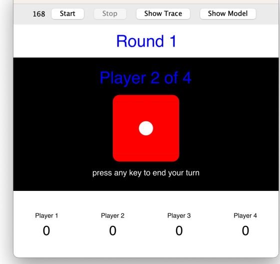


Figure 5: The second player has rolled 1 too

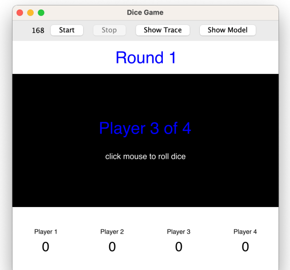


Figure 6: The third player is about to roll

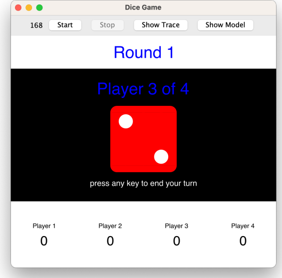


Figure 7: The third player has rolled 2

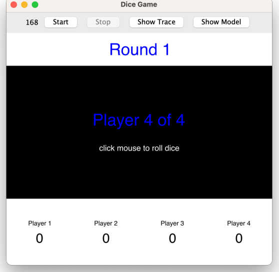


Figure 8: The last player is about to roll

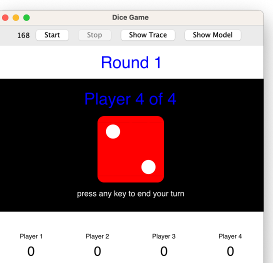


Figure 9: The last player has rolled 2

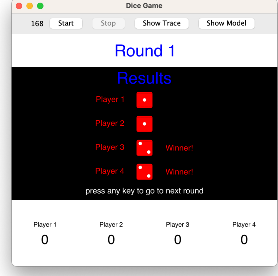


Figure 10: Both player 3 and 4 rolled a 2, so they get a point for the round

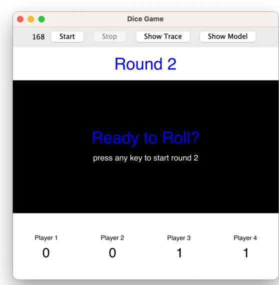


Figure 11: A new round begins

...

Task B1

First, we must do some leg work; we need to be able to get and set elements at a given index in a sequence. This is a good exercise to practice your recursion skills (if you are in good shape, implementing the following methods should be quick).

<table>
<tr><td>Class:</td><td><code>lab.dice.game.IndexedAccess</code></td></tr>
<tr><td>Task:</td><td>Implement the following methods:

<code>public static <T> T get(int index, Sequence<T> seq)</code>

<code>public static <T> Sequence<T> set(int index, T value,
Sequence<T> seq)</code>

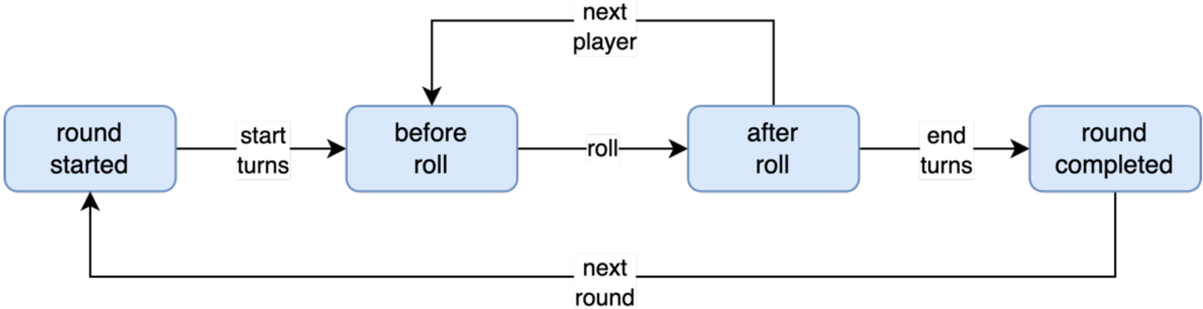
The <code>get</code> method should return the element at the given index (the first element is at index 0) in the given sequence.

The <code>set</code> method should return a new sequence, which looks exactly like the given sequence, but with the element at the given index replaced by the given value.</td></tr>
<tr><td>Run in JShell:</td><td><code>IndexedAccess.get(0, of("A", "B"))</code></td></tr>
<tr><td>Output:</td><td><code>==> A</code></td></tr>
<tr><td>Run in JShell:</td><td><code>IndexedAccess.get(1, of("A", "B"))</code></td></tr>
<tr><td>Output:</td><td><code>==> B</code></td></tr>
<tr><td>Run in JShell:</td><td><code>print(IndexedAccess.set(1, "X", of("A", "B")))</code></td></tr>
<tr><td>Output:</td><td>AX</td></tr>
</table>

Task B2

Class:	`lab.Toolbelt`
Task:	For this exercise, we provided you with tests. As you progress through the tasks, more of them will succeed. Most of them require the ability to compare two sequences (for example, we could want to compare two sequences of players). **Implement** in your `Toolbelt` a generic method `equalTo` that is used by the tests to compare two generic sequences. Two sequences are considered equal if they have the same number of elements, and each element of the first sequence is equal to the corresponding element of the second sequence. To compare two individual elements `el1` and `el2`, use `el1.equals(el2)`.
Run in JShell:	`Toolbelt.equalTo(of(0, 1, 2, 3), range(4))`
Output:	`==> true`
Run in JShell:	`Toolbelt.equalTo(of("A", "B"), of("B", "A"))`
Output:	`==> false`
Run in JShell:	`Toolbelt.equalTo(of("A", "B", "C"), of("A", "B"))`
Output:	`==> false`

We model the phases of our dice game with the State Pattern, an object-oriented design pattern which can elegantly model the phases of our game.
During each round, a phase (blue rectangle in the diagram) can initiate a transition (an arrow in the diagram) of the game to another phase (another blue rectangle).



Task B3

<table>
<tr><td>Class:</td><td><code>lab.dice.game.Game, lab.dice.game.Player</code></td></tr>
<tr><td>Task:</td><td>We start by considering the <code>Game</code> class and the related <code>Player</code> class, which are the backbone of the model for our game.

At any given point in time, we would like to know:
<code>int round</code> – Which round we are in
<code>Phase roundPhase</code> – Which of the four phases within the round we are in
<code>Sequence<Player> players</code> – A <code>Sequence</code> of participating <code>Player</code>s, in the order they play. Each <code>Player</code> has a <code>Die</code> (initially yet to roll, and then rolled) and a score (the number of rounds they won).
<code>int currentPlayerIndex</code> – A 0-based index of the player whose turn it is.
First, implement the factory method <code>Player.create</code> that builds a new <code>Player</code> with a yet-to-roll die (see the <code>Die</code> class) and a score of 0.

Then, implement the factory method <code>Game.create(int playerCount)</code> so it builds a new <code>Game</code> with the given player count, starting in round 1, in the first phase (<code>RoundStarted</code>), with current player index set to 0.</td></tr>
<tr><td>Run in JShell:</td><td><pre>Game game = Game.create(4);

game</pre></td></tr>
<tr><td>Output:</td><td><pre>==> Game[round=1, roundPhase=RoundStarted[],
         currentPlayerIndex=0, players=Sequence[
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0]]]</pre></td></tr>
</table>

Task B4

Class: `lab.dice.game.phase.RoundStarted`

Task:

Inside the `phase` folder you find an interface named `Phase` which says that each concrete phase implements two methods: one to handle a mouse click and another to render itself. Let us postpone our rendering concerns and focus on the functionalities for now.

We start from the first phase, `RoundStarted`. This is the phase that happens at the beginning of each round, just before the players start taking turns.

next player
round started
start turns
before roll
roll
after roll
end turns
round completed
next round

Implement the `startTurns(Game game)` method.
Call the method `changePhase` on the `game` to ask it to transition to the next phase (`BeforeRoll`). Return the updated game.

When a user clicks in this phase, the action is always to start taking turns. That's indeed the only possible outgoing transition shown in the diagram.

Implement `handleMouse(Game game)`, which for this phase makes no decision and delegates the work to the `startTurns` method you just implemented.

Run in JShell:

```
Game beforeFirst = new RoundStarted().startTurns(game);

beforeFirst
```

Output:

```
==> Game[round=1, roundPhase=BeforeRoll[],
         currentPlayerIndex=0, players=Sequence[
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0]]]
```

Task B5

<table>
<tr><td>Class:</td><td><code>lab.dice.game.Die, lab.dice.game.Game</code></td></tr>
<tr><td>Task:</td><td>Before proceeding with the implementation of the other phases of the game, we need to implement some functionalities for our <code>Die</code>.

First and foremost, a die can be rolled. Implement <code>Die.roll</code> to return a rolled die. Use the generator <code>RND_GEN</code> and its instance method <code>nextInt(a, b)</code> that returns an integer between <code>a</code> (included) and <code>b</code> (excluded).

We also need to compare dice (to determine winners).

Implement <code>isHigherOrEqual(Die other)</code> to determine whether the die it is called on (<code>this</code>) has a number higher than or equal to the <code>other</code> die received in the parameter.

Implement <code>isHigherOrEqualAll(Sequence<Die> dice)</code> to determine whether the die it is called on (<code>this</code>) has a number higher than or equal to all the dice received in the parameter.
Hint: use the comparison method between two dice you just implemented!

Finally, in class <code>Game</code> implement <code>isWinner</code>. The method checks whether a given <code>Player</code> of the game is a winner at the end of a round. A <code>Player</code> is considered a winner if their die is higher than or equal to all the dice of all the players of the game.
Hint: “being higher or equal” is a reflexive relation; every number is “higher or equal” to itself. Thus, you do not need to exclude the very player passed as a parameter from the comparison. A player always wins against themselves.</td></tr>
<tr><td>Run in JShell:</td><td><code>Die.roll()</code></td></tr>
<tr><td>Output:</td><td><code>==> Die[number=5]</code>

Note: Given that rolling a die uses a random number generator, the values of rolled dice above and in all the subsequent examples will likely be different for you.</td></tr>
<tr><td>Run in JShell:</td><td><code>new Die(5).isHigherOrEqual(new Die(5))</code></td></tr>
<tr><td>Output:</td><td><code>==> true</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new Die(5).isHigherOrEqual(new Die(6))</code></td></tr>
<tr><td>Output:</td><td><code>==> false</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new Die(5).isHigherOrEqualAll(of(new Die(5), new Die(6)))</code></td></tr>
<tr><td>Output:</td><td><code>==> false</code></td></tr>
<tr><td>Run in JShell:</td><td><code>new Die(5).isHigherOrEqualAll(of(new Die(5), new Die(4)))</code></td></tr>
</table>

Output:	`==> true`

Run in JShell:	`Player mark = new Player(new Die(6), 0);`
	`Player elon = new Player(new Die(1), 0);`
	`Game aiGame = Game.create(2).changePlayers(of(mark, elon));`
	`aiGame.isWinner(mark)`

Output:	`==> true`
Run in JShell:	`aiGame.isWinner(elon)`
Output:	`==> false`

Task B6

Class: `lab.dice.game.phase.BeforeRoll`

Task:

We can focus again on the phases.
In the `BeforeRoll` phase, a click causes the current player to roll a die.

round started
start turns
before roll
roll
after roll
end turns
round completed
next player
next round

Implement the method `roll`.
It should ask the game for the current player and use its `Player.rollDie` method to get an updated `Player` with a rolled die.

Then, build a new sequence where the player at the current index is replaced with the player with the rolled die you just created above.

Finally, update the game so that it uses the new sequence of players, and change the phase to `AfterRoll`.

Use `Game.currentPlayer()` to get the current player.
Use `IndexedAccess.set` to create an updated `Sequence<Player>`.

Implement `handleMouse(Game game)`, which also for this phase makes no decision and delegates the work to the `roll` method you just implemented.

Run in JShell:

```
Game afterFirst = new BeforeRoll().roll(beforeFirst);

afterFirst
```

Output:

```
==> Game[round=1, roundPhase=AfterRoll[],
         currentPlayerIndex=0, players=Sequence[
           Player[die=Die[number=1], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0],
           Player[die=Die[number=0], score=0]]]
```

Task B7

Class:	`lab.dice.game.phase.AfterRoll`
Task:	In the `AfterRoll` phase, a user click can lead to two different actions: • If the current player is the last one, everybody has rolled a die and we can thus conclude the round by checking who are the winners and updating the scores. • If instead there are still more players that need to roll their die, we need to give the turn to the next one. round started → start turns → before roll → roll → after roll → end turns → round completed; after roll → next player → before roll; round completed → next round → round started **Implement** the method `nextPlayer` to execute the second scenario described above. The game should transition to the `BeforeRoll` state for the immediate next player. Use `Game.advancePlayer` to increment the current player index. **Implement** the method `endTurns` to execute the first scenario. This involves determining for each player whether they are a winner (note that multiple players can win a round, if they all have a score higher than or equal to everybody else). Use `Game.isWinner` to figure out if a `Player` is a winner, and `Player.updateScore` to possibly increase a player's score. The game should then be updated with the players updated as described and transition to the `RoundCompleted` phase. Finally, **implement** the `handleMouse` method. Either call `nextPlayer` or `endTurns`, depending on whether the current player is the last one. The game conveniently offers the `isLastPlayer` method, which checks the current player index for you.

Run in JShell:	`Game beforeSecond = new AfterRoll().nextPlayer(afterFirst);`
	
	`beforeSecond`

Output:	`==> Game[round=1, roundPhase=BeforeRoll[],`
	`currentPlayerIndex=1, players=Sequence[`
	`Player[die=Die[number=1], score=0],`
	`Player[die=Die[number=0], score=0],`
	`Player[die=Die[number=0], score=0],`
	`Player[die=Die[number=0], score=0]]]`

Run in JShell:	`Game solitaryGame = Game.create(1);`
	`Game solitaryBefore = new RoundStarted().startTurns(solitaryGame);`
	`Game solitaryAfter = new BeforeRoll().roll(solitaryBefore);`
	`Game solitaryCompleted = new AfterRoll().endTurns(solitaryAfter);`
	`solitaryCompleted`

Output:	`==> Game[round=1, roundPhase=RoundCompleted[],`
	`currentPlayerIndex=0, players=Sequence[`
	`Player[die=Die[number=1], score=1]]]`

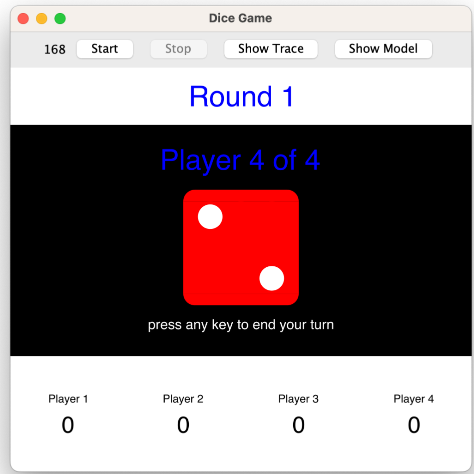
Task B8

Class:	`lab.dice.game.phase.RoundCompleted`
Task:	The `RoundCompleted` phase simply shows to the players the result of the current round, with the winners and the updated scores. The only possible action is moving to a fresh new round. round started → start turns → before roll → roll → after roll → end turns → round completed; after roll → next player → before roll; round completed → next round → round started **Implement** the method `nextRound`. We should prepare for the next round: all the players should reset their die to a yet-to-roll state (see `Player.resetDie`). The game should advance to the new round by incrementing the round number and resetting the current player index to 0 (use `Game.advanceRound` to accomplish both). And, finally, the phase should also change to `RoundStarted`. **Hint**: `map` is very convenient to update all the `Player`s in a sequence. **Implement** `handleMouse(Game game)`, which for this phase too makes no decision and delegates the work to the `nextRound` method you just implemented.

Run in JShell:	`Game twoPlayersGame = Game.create(2);`
	`Game twoBeforeFirst = new RoundStarted().startTurns(twoPlayersGame);`
	`Game twoAfterFirst = new BeforeRoll().roll(twoBeforeFirst);`
	`Game twoBeforeSecond = new AfterRoll().nextPlayer(twoAfterFirst);`
	`Game twoAfterSecond = new BeforeRoll().roll(twoBeforeSecond);`
	`Game twoCompleted = new AfterRoll().endTurns(twoAfterSecond);`
	`Game twoSecondRound = new RoundCompleted().nextRound(twoCompleted);`
	`twoSecondRound`

Output:	`==> Game[round=2, roundPhase=RoundStarted[],`
	`currentPlayerIndex=0, players=Sequence[`
	`Player[die=Die[number=0], score=0],`
	`Player[die=Die[number=0], score=1]]]`

We have now implemented the logic for our game (the "model"). How do we render it as a graphic? The header and the footer are independent of each phase, but the main body content varies depending on which round phase we are in.



If we were to completely separate the model entities (the game, the phase, the die…) from the rendering (as we did for the "Mouse Visualizer", and as it would be good practice), we would not be able to know which phase to render without some ugly code that switches across phases.
We want to avoid this: it is not a clean object-oriented approach.
(Quite advanced note: there is a way to solve this problem with a rather elaborate design pattern, the visitor pattern.)

Instead, let's practice polymorphism and exploit dynamic dispatch. Each class that needs to be rendered will have an instance method `render(Game)`. This method (and possibly, smaller helper methods defined in the same class) will be responsible for rendering the model.

To avoid polluting our classes with lots of `Graphic`-related methods, we will package our "utility methods" to draw certain UI components inside the `ui` folder. These methods will *not* depend on our model classes.

Let's give this a try for rendering a die.

Task B9

Class:	`lab.dice.game.Die, lab.dice.game.ui.DieUI`
Task:	**Implement** the `render` method to render a die. Only a rolled die can be rendered, thus assert that it has been actually rolled. Delegate all the UI part to the `DieUI.die`, which takes in just the number of the rolled die (between 1 and 6). In `ui.DieUI` there is already some useful code to draw a die. **Implement** the method `die` to overlay the dots (at 80% of the side) over a square with rounded corners. The radius of the corner should be 10% of the side. **Implement** the method `dots` to draw a 3-by-3 grid of dots. The `CONFIGURATIONS` constant provides all 6 possible dice configurations, each represented as a 3-by-3 matrix of booleans (where `true` means a dot, `false` means no dot). Don’t forget `Toolbelt.aboves` and `besides` to get the job done. To access the correct configuration, use `IndexedAccess.get`. Note that the first element of `CONFIGURATIONS` (at index 0) corresponds to the number 1 for the die.
Run in JShell:	`show(new Die(5).render());`
Output:	

Task B10

Class:	`lab.dice.game.phase.AfterRoll`, `lab.dice.game.phase.RoundCompleted`
Task:	The `render` method for the first two phases (`RoundStarted` and `BeforeRoll`) is already implemented. They just show some instructions to the user. **Complete the implementation** of `render` in `AfterRoll` to render the die the player has just rolled. You can ask the game who is the current player, and then render their die. **Complete the implementation** of the methods called by `render` in `RoundCompleted`. The method `renderScoreTable` should render one row per player using the `renderScoreTableRow` method. Use `Sequences.zipWithIndex` to process the players alongside their index in the sequence. For each player, call `renderScoreTableRow` with their index, their die, and whether they are winners (ask the `Game`!). `Sequences.map` is particularly useful to do this for all players.
Run in JShell:	`show(new AfterRoll().render(afterFirst))`
Output:	Player 1 of 4 click mouse to end your turn
Run in JShell:	`show(new RoundCompleted().render(twoCompleted))`
Output:	Results Player 0 Winner! Player 1 Winner! click mouse to go to next round

Task B11

Class:	`lab.dice.game.Game`
Task:	**Implement** the `render` method to render the game. Place one above the other the header, the body and the footer. Call the three related methods. **Implement** `renderHeader` calling `GameUI.headerWithTitle`. The title should be "Round 1", "Round 2", ... depending on the current round number. **Implement** `renderFooter`. The footer consists of all the score indicators for each player of the game, one next to the other. You can call `GameUI.scoreIndicator` to draw one score indicator, passing in the player index in the sequence, their score, and the total number of players. Use again `Sequences.zipWithIndex` to process the players alongside their index in the sequence. **Implement** `renderBody`. The content of the body depends on the phase: each phase knows how to draw itself (remember `Phase.render(Game)` in the interface?). We can thus produce the body content by calling `render` on the current game phase (this is the polymorphic call!), passing in the current game as an argument. Use `this` to get a reference to the game. Then, pass to `GameUI.body` as an argument the `Graphic` for the body content rendered by the phase. The method will place it on top of a black background.
Run in JShell:	`show(beforeFirst.render())`
Output:	Round 1 Player 1 of 4 click mouse to roll your die Player 1 0 · Player 2 0 · Player 3 0 · Player 4 0
Run in JShell:	`show(afterFirst.render())`
Output:	Round 1 Player 1 of 4 click mouse to end your turn Player 1 0 · Player 2 0 · Player 3 0 · Player 4 0

Task B12

<table>
<tr><td>Class:</td><td><code>lab.dice.Main</code></td></tr>
<tr><td>Task:</td><td>Implement the <code>interaction</code> in the body of the <code>play</code> method.

Call <code>IO.interact</code> with one argument: the initial <code>Game</code> object, which you can obtain with <code>Game.create</code>. As an argument, choose the number of players you prefer.

Then, using the fluent API we have already seen multiple times, configure the interaction using the following <code>with…</code> methods:
<code>withRenderer(…)</code> – pass a lambda or method reference that, given a game instance, renders it using the <code>Game.render</code> method.
<code>withKeyTypeHandler(…)</code> – pass a method reference to the <code>Main.onKeyType</code> method.
<code>withMousePressHandler(…)</code> – pass a method reference to the <code>Main.onMousePress</code> method.
Finally, conclude the chain of method invocations with <code>.run()</code> to execute your interaction.

Notice a difference compared to the Pacman game of previous labs: there is no tick handler! In this case, our game only changes in response to keyboard keys or mouse button presses.

Now, call the <code>Main.play</code> method from JShell to run the dice game!

If everything has been implemented correctly, you should be able to roll a die every round for each player. At the end of each round, one point is assigned to all players who rolled the highest number.</td></tr>
<tr><td>Run in JShell:</td><td><code>lab.dice.Main.play()</code></td></tr>
</table>