|  |
| --- |
| **Exercise 15.4**  **The fields of the Time and Reading classes are all final and set via the class constructor. There is a getter method for each field and no setter methods. This makes them suitable for writing as record types.** |
| **Exercise 15.6**  **The times are printed as follows:**  **Time[hour=11, minute=5]**  **which is a less attractive format than in the previous version.** |
| **Exercise 15.7**  **This is the Reading record type in v1 of the project:**  **/\*\***  **\* A single reading within an experimental run.**  **\***  **\* @author David J. Barnes and Michael Kölling**  **\* @version 7.1**  **\* @param value The value of the reading.**  **\* @param time The time the reading was taken.**  **\*/**  **public record Reading(double value, Time time)**  **{**  **}** |
| **Exercise 15.8**  **The format is as follows:**  **Reading[value=12.72, time=Time[hour=12, minute=21]**  **As with the rewritten Time class, the format is less appropriate than in the version before rewriting, but this is the standard format provided by the default version of toString in a record.** |
| **Exercise 15.9**  **equals, hashCode and toString - versions overridden from the Object class.** |
| **Exercise 15.12**  **The immediate reason why Experiment cannot be rewritten as a record type is that its** runs **field is assigned to in its** readData **method. Related to this is that an Experiment object's state is not fixed at the point of its creation - there is a separate initialization step provided by the** readData **method. While it would be possible in** readData **to copy the data into the ArrayList created on construction of the Experiment object, that represents a mutation step, in effect, which conflicts with the immutable nature of record types.** |
| **Exercise 15.14**  **/\*\***  **\* A description of the particular type of unit used**  **\* in a measurement.**  **\* E.g., "cm", "Celsius", etc.**  **\***  **\* @author David J. Barnes and Michael Kölling**  **\* @version 7.3**  **\* @param theUnit The name of the unit.**  **\*/**  **public record Unit(String theUnit)**  **{**  **public Unit(String theUnit)**  **{**  **// Ensure consistency of representation.**  **this.theUnit = theUnit.toLowerCase().trim();**  **}**  **}** |
| **Exercise 15.15**  **/\*\***  **\* A record of a temperature reading.**  **\* The default unit for readings is Celsius**  **\* but alternatives may be used.**  **\***  **\* @author David J. Barnes and Michael Kölling**  **\* @version 7.3**  **\* @param temperature The temperature reading.**  **\* @param theUnit E.g., "Celsius" (the default).**  **\*/**  **public record Temperature(double temperature, Unit theUnit)**  **{**  **public static final double CELSIUS\_FP = 0;**  **public static final double FAHRENHEIT\_FP = 32;**  **public static final double ZERO\_CELSIUS\_IN\_KELVIN = 273.15;**    **public Temperature(double temperature)**  **{**  **this(temperature, new Unit("Celsius"));**  **}**  **}** |
| **Exercise 15.16**  **/\*\***  **\* Get the location in the form C3, etc.**  **\* @return the location.**  **\*/**  **public String toString()**  **{**  **return String.format("%c%d", row, column);**  **}** |
| **Exercise 15.17**  **/\*\***  **\* Get the list of locations in which the animal is found**  **\* with-in the given area.**  **\* @param animal The animal to find.**  **\* @param topLeft The top-left of a rectangular area.**  **\* @param bottomRight The bottom-right of a rectangular area.**  **\* @return the list of locations.**  **\*/**  **public List<Location> getSightingsInArea(String animal,**  **Location topLeft, Location bottomRight)**  **{**  **Set<Location> theLocations = new HashSet<>();**  **for(Location loc : sightingsGrid.keySet()) {**  **if(loc.row() >= topLeft.row() &&**  **loc.row() <= bottomRight.row() &&**  **loc.column() >= topLeft.column() &&**  **loc.column() <= bottomRight.column()) {**  **for(Sighting sighting : sightingsGrid.get(loc)) {**  **if(animal.equals(sighting.animal())) {**  **theLocations.add(loc);**  **}**  **}**  **}**  **}**  **return new ArrayList<>(theLocations);**  **}** |
| **Exercise 15.18**  **/\*\***  **\* Get the list of neighbors of the given location where the**  **\* neighbors contain at least one animal.**  **\* @param aLocation The location whose neighbors must**  **\*. be searched.**  **\* @return the list of neighbors containing animals.**  **\*/**  **public List<Location> getNonEmptyNeighbors(Location aLocation)**  **{**  **List<Location> theLocations = new ArrayList<>();**  **List<Location> neighbors = aLocation.getNeighbors();**  **for(Location loc : neighbors) {**  **if(sightingsGrid.containsKey(loc)) {**  **// Make sure there is at least one non-zero count.**  **if(sightingsGrid.get(loc)**  **.stream()**  **.filter(aSighting -> aSighting.count() > 0)**  **.findFirst()**  **.isPresent()) {**  **theLocations.add(loc);**  **}**  **}**  **}**  **return theLocations;**  **}** |
| **Exercise 15.19**  **/\*\***  **\* Find the closest distance between any pair**  **\* of locations that both contain the given animal.**  **\* @param The animal to find.**  **\* @return the closest distance, or -1 if there is no**  **\* pair of locations.**  **\*/**  **public int closestPair(String animal)**  **{**  **if(animalSightings.containsKey(animal)) {**  **List<Sighting> theSightings =**  **animalSightings.get(animal);**  **// Find the unique locations.**  **Set<Location> uniqueLocations = new HashSet<>();**  **for(Sighting aSighting : theSightings) {**  **if(aSighting.count() > 0) {**  **uniqueLocations.add(aSighting.location());**  **}**  **}**  **if(uniqueLocations.size() > 1) {**  **// Sort the locations by distance.**  **List<Location> theLocations =**  **new ArrayList<>(uniqueLocations);**  **theLocations.sort(**  **(loc1, loc2) -> loc1.distance(loc2));**  **return theLocations.get(0)**  **.distance(theLocations.get(1));**  **}**  **else {**  **return -1;**  **}**  **}**  **else {**  **return -1;**  **}**  **}** |
|  |
|  |