|  |
| --- |
| **Exercise 5.4**  The method removes items from the collection, so a for-each loop cannot be used. A while loop is the most general type of loop and it can be used in place of any of the other types. |
| **Exercise 5.9**  sightingList.filter(name equals elephant)  .filter(spotter == spotterID)  .filter(period == dayID)  .reduce(count elements); |
| **Exercise 5.10**  sightingList.filter(count > 0) |
| **Exercise 5.11**  One possible ordering is:  trackList.filter(artist equals Lisa Davies)  .map(played count)  .reduce(sum) |
| **Exercise 5.13**  /\*\*  \* Print details of the sightings on the given day.  \* @param dayID The period of interest.  \*/  public void printSightingsOnDay(int dayID)  {  sightingList.stream()  .filter(aSighting -> aSighting.getPeriod() == dayID)  .forEach(aSighting -> System.out.println(aSighting.getDetails()));  } |
| **Exercise 5.14**  /\*\*  \* Print details of the sightings on the given day.  \* @param animal The animal of interest.  \* @param dayID The period of interest.  \*/  public void printSightingsOfAnimalOnDay(String animal, int dayID)  {  sightingList.stream()  .filter(aSighting -> animal.equals(aSighting.getAnimal()))  .filter(aSighting -> aSighting.getPeriod() == dayID)  .forEach(aSighting -> System.out.println(aSighting.getDetails()));  } |
| **Exercise 5.15**  The order does not matter since all details printed must have the property that both the animal and the period match the parameter values. Each filter operation removes all those sightings from the stream that do not match one of the parameters. |
| **Exercise 5.17**  /\*\*  \* Print the counts of the given animal.  \* @param animal The animal of interest.  \*/  public void printCountsOfAnimal(String animal)  {  sightingList.stream()  .filter(aSighting -> animal.equals(aSighting.getAnimal()))  .forEach(aSighting -> System.out.println(aSighting.getCount()));  } |
| **Exercise 5.18**  The order does matter in many cases. Since the output stream of a map operation typically is of a different type from its input stream, the appropriate position of the filter operation will depend on the type of its input stream. |
| **Exercise 5.19**  /\*\*  \* Print a list of the types of animal considered to be endangered.  \* @param animalNames A list of animals names.  \* @param dangerThreshold Counts less-than or equal-to to this level  \* are considered to be dangerous.  \*/  public void printEndangered(ArrayList<String> animalNames,  int dangerThreshold)  {  animalNames.stream()  .filter(animal -> getCount(animal) <= dangerThreshold)  .forEach(animal -> System.out.println(animal + " is endangered."));  } |
| **Exercise 5.20**  Use of the :: syntax in the example is equivalent to the original lambda expression. See Chapter 6 for the various ways in which it can be used. |
| **Exercise 5.22**  /\*\*  \* Return a count of the number of sightings of the given animal  \* by a particular spotter on a particular day.  \* @param animal The type of animal.  \* @param spotterID The spotter.  \* @param dayID When.  \* @return The count of sightings of the given animal.  \*/  public int getCount(String animal, int spotterID, int dayID)  {  return sightingList.stream()  .filter(sighting -> animal.equals(sighting.getAnimal()))  .filter(sighting -> spotterID == sighting.getSpotter())  .filter(sighting -> dayID == sighting.getPeriod())  .map(sighting -> sighting.getCount())  .reduce(0, (runningSum, count) -> runningSum + count);  } |
| **Exercise 5.23**  /\*\*  \* Return a String giving the names of the animals seen by  \* a particular spotter on a particular day.  \* @param spotterID The spotter.  \* @param dayID When.  \* @return The animal names.  \*/  public String seenBySpotterOnDay(int spotterID, int dayID)  {  return sightingList.stream()  .filter(sighting -> sighting.getSpotter() == spotterID)  .filter(sighting -> sighting.getPeriod() == dayID)  .filter(sighting -> sighting.getCount() > 0)  .map(sighting -> sighting.getAnimal())  .reduce("", (list, animal) -> list + " " + animal).trim();  } |
| **Exercise 5.25**  /\*\*  \* Count the number of sighting records by the given spotter.  \* @param spotter The ID of the spotter.  \* @return The number of sighting records.  \*/  public long countBySpotter(int spotter)  {  return sightingList.stream()  .filter(aSighting -> aSighting.getSpotter() == spotter)  .count();  } |
| **Exercise 5.26**  This is slightly tricky because max returns an OptionalInt. In other words, there might be no matching animal in the stream, so a default return value is required in that case.  /\*\*  \* Return the largest count in a sighting of the given animal.  \* @param animal The animal of interest.  \* @return The largest count.  \*/  public int getLargestCount(String animal)  {  return sightingList.stream()  .filter(aSighting -> animal.equals(aSighting.getAnimal()))  .map(aSighting -> aSighting.getCount())  .max((x, y) -> x - y)  .orElse(0);  } |
| **Exercise 5.27**  /\*\*  \* Find the first sighting record for the given  \* spotter and animal.  \* @param spotter The spotter.  \* @param animal The animal of interest.  \* @return The sighting, or null if there are none.  \*/  public Sighting findFirstSightingBy(int spotter, String animal)  {  return sightingList.stream()  .filter(aSighting -> aSighting.getSpotter() == spotter)  .filter(aSighting -> animal.equals(aSighting.getAnimal()))  .findFirst()  .orElse(null);  } |
| **Exercise 5.29**  /\*\*  \* Show a list of all the tracks in the collection.  \*/  public void listAllTracks()  {  System.out.println("Track listing: ");  trackList.stream().map(aTrack -> aTrack.getDetails())  .forEach(details -> System.out.println(details));  System.out.println();  }  /\*\*  \* List all tracks by the given artist.  \* The match may be partial.  \* @param artist The artist's name.  \*/  public void listByArtist(String artist)  {  trackList.stream().filter(aTrack -> aTrack.getArtist().contains(artist))  .map(aTrack -> aTrack.getDetails())  .forEach(details -> System.out.println(details));  } |
| **Exercise 5.30**  /\*\*  \* List all tracks containing the given phrase in the title.  \* The match may be partial.  \* @param phrase The phrase to be matched.  \*/  public void listByTitle(String phrase)  {  trackList.stream().filter(  aTrack -> aTrack.getTitle()  .contains(phrase))  .map(aTrack -> aTrack.getDetails())  .forEach(details ->  System.out.println(details));  } |
| **Exercise 5.31**  This version would not work as it will call the toString method of the Track object rather than the getDetails method. The fix is given in the following exercise. |
| **Exercise 5.32**  When an object is printed directly via the print or println methods of System.out, a method called toString is automatically called on it. |
| **Exercise 5.33**  /\*\*  \* Output the tracks sorted by track title.  \*/  public void listSortedByTitle()  {  trackList.stream()  .sorted((t1, t2) -> t1.getTitle().compareTo(t2.getTitle()))  .map(track -> track.getDetails())  .forEach(details -> System.out.println(details));  }  /\*\*  \* Output the tracks sorted by play count: low to high.  \*/  public void listSortedByPlayCount()  {  trackList.stream()  .sorted((t1, t2) -> t1.getPlayCount() - t2.getPlayCount())  .map(track -> track.getDetails())  .forEach(details -> System.out.println(details));  } |
| **Exercise 5.35**  /\*\*  \* Output the tracks sorted by track title.  \*/  public void listSortedByTitle()  {  listSortedBy((t1, t2) ->  t1.getTitle().compareTo(t2.getTitle()));  }  /\*\*  \* Output the tracks sorted by play count: low to high.  \*/  public void listSortedByPlayCount()  {  listSortedBy((t1, t2) ->  t1.getPlayCount() - t2.getPlayCount());  } |