|  |
| --- |
| **Exercise 4.4**  private ArrayList<Book> library; |
| **Exercise 4.5**  ArrayList<Student> cs101; |
| **Exercise 4.6**  private ArrayList<MusicTrack> tracks; |
| **Exercise 4.7**  **using diamond notation, these can be written as:**  library = new ArrayList<>();  cs101 = new ArrayList<>();  tracks = new ArrayList<>();  using older style notation these would be written as:  library = new ArrayList<Book>();  cs101 = new ArrayList<Student>();  tracks = new ArrayList<MusicTrack>(); |
| **Exercise 4.8**  10 |
| **Exercise 4.9**  items.get(4); |
| **Exercise 4.10**  14 |
| **Exercise 4.11**  files.add(favoriteTrack); |
| **Exercise 4.12**  dates.remove(2); |
| **Exercise 4.13**  5 |
| **Exercise 4.14**  public void checkIndex(int index)  {  if(index < 0 || index >= files.size()) {  System.out.println("The valid range is 0 to " +  (files.size() – 1));  }  }  **If the collection is empty, there is no valid index and the error message will still appear. This special case could be catered for my a separate if statement that checks for an empty collection and prints an error message specific to that case.** |
| **Exercise 4.15**  **The following are all valid. The first is in the spirit of the previous exercise. The second rewrites to the opposite form and the third gives the most concise representation.**  public boolean validIndex(int index)  {  if(index < 0 || index >= files.size()) {  return false;  }  else {  return true;  }  }  public boolean validIndex(int index)  {  if(index >= 0 && index < files.size()) {  return true;  }  else {  return false;  }  }  public boolean validIndex(int index)  {  return index >= 0 && index < files.size();  } |
| **Exercise 4.16**  /\*\*  \* List a file from the collection.  \* @param index The index of the file to be listed.  \*/  public void listFile(int index)  {  if(validIndex(index)) {  String filename = files.get(index);  System.out.println(filename);  }  }    /\*\*  \* Remove a file from the collection.  \* @param index The index of the file to be removed.  \*/  public void removeFile(int index)  {  if(validIndex(index))) {  files.remove(index);  }  } |
| **Exercise 4.18**  public void listAllFiles() |
| **Exercise 4.19**  No. We have no idea how many lines we would need. |
| **Exercise 4.24**  /\*\*  \* Show a list of all the files in the collection,  \* along with their index values.  \*/  public void listAllFiles()  {  // A variable to keep track of the index position.  int position = 0;  for(String filename : files) {  System.out.println(position + ": " + filename);  position++;  }  } |
| **Exercise 4.26**  /\*\*  \* List the names of files matching the given search string.  \* If none are found, print a message.  \* @param searchString The string to match.  \*/  public void listMatching(String searchString)  {  // Assume that there will be no matches.  boolean noMatch = true;  for(String filename : files) {  if(filename.contains(searchString)) {  System.out.println(filename);  // We found at least one match.  noMatch = false;  }  }  if(noMatch) {  System.out.println("No files matched: " +  searchString);  }  } |
| **Exercise 4.27**  /\*\*  \* Play a sample from all files matching the given search string.  \* @param artist The string to match.  \*/  public void playSamplesBy(String artist)  {  int position = 0;    for(String filename : files) {  if(filename.contains(artist)) {  player.playSample(position);  }  position++;  }  } |
| **Exercise 4.28**  for(Track track : tracks) |
| **Exercise 4.32**  **Add:**  stopPlaying();  **to** playTrack() **once the index has been validated.** |
| **Exercise 4.33**  boolean found = false; while(!found) {  if(*the keys are in the next place*) {  found = true;  } } |
| **Exercise 4.34**  public void multiplesOfFive()  {  int multiple = 10;  while(multiple <= 95) {  System.out.println(multiple);  multiple = multiple + 5;  }  } |
| **Exercise 4.35**  int sum = 0; int num = 1;  while(num <= 10) {  sum += num;  num++; } System.out.println("The sum of the values 1 to 10 is " + sum); |
| **Exercise 4.36**  /\*\*  \* Sum the numbers from a to b, inclusive.  \*/  public int sum(int a, int b)  {  int sum = 0;  int number = a;  while(number <= b) {  sum = sum + number;  number = number + 1;  }  return sum; } |
| **Exercise 4.37**  \* A crude determination of whether n is prime or not.  \* More efficient methods are possible.  \*/  public boolean isPrime(int n)  {  int divisor = 2;  boolean ok = true;  while(divisor < n && ok) {  if(n % divisor == 0) {  ok = false;  }  else {  divisor = divisor + 1;  }   }  return ok; } |

|  |
| --- |
| **Exercise 4.38**  The value does not vary.  public int findFirst(String searchString)  {  int index = 0;  // Record that we will be searching until a match is found.  boolean searching = true;  int size = files.size();  while(searching && index < size) {  String filename = files.get(index);  if(filename.contains(searchString)) {  // A match. We can stop searching.  searching = false;  }  else {  // Move on.  index++;  }  }  if(searching) {  // We didn't find it.  return -1;  }  else {  // Return where it was found.  return index;  } } |
| **Exercise 4.39**  /\*\*  \* Remove tracks whose titles match the search string.  \* A partial match is used.  \* @param title The string to be matched.  \*/  public void removeTitles(String title)  {  Iterator<Track> it = tracks.iterator();  while(it.hasNext()) {  Track t = it.next();  if(t.getTitle().contains(title)) {  it.remove();  }  }  } |
| **Exercise 4.40-4.42**  import java.util.ArrayList;  /\*\*  \* Store details of club memberships.  \*   \* @author (your name)   \* @version (a version number or a date)  \*/ public class Club {  private ArrayList<Membership> members;  /\*\*  \* Constructor for objects of class Club  \*/  public Club()  {  members = new ArrayList<>();  }  /\*\*  \* Add a new member to the club's list of members.  \* @param member The member object to be added.  \*/  public void join(Membership member)  {  members.add(member);  }  /\*\*  \* @return The number of members (Membership objects) in  \* the club.  \*/  public int numberOfMembers()  {  return members.size();  } } |
| **Exercise 4.43**  /\*\*  \* Select and play a single random track.  \*/  public void randomPlay()  {  if(tracks.size() > 0) {  Random rand = new Random();  int index = rand.nextInt(tracks.size());  playTrack(index);  }  } |
| **Exercise 4.45**  **The** Collections.shuffle() **method offers a way to randomly order a collection. This is used in the first version, which does not require a separate random number generator:**  /\*\*  \* Play all tracks once in a random order.  \*/  public void randomPlayAll()  {  ArrayList<Track> leftToPlay = new ArrayList<>(tracks);  Collections.shuffle(leftToPlay);  for(Track t : leftToPlay) {  player.playSample(t.getFilename());  }    }  **The second version makes a copy of the list and repeatedly removes a random track to be played until the list is empty.**  /\*\*  \* Play all tracks once in a random order.  \*/  public void randomPlayAll()  {  Random rand = new Random();  ArrayList<Track> leftToPlay = new ArrayList< >(tracks);  while(leftToPlay.size() > 0) {  int index = rand.nextInt(leftToPlay.size());  Track t = leftToPlay.remove(index);  player.playSample(t.getFilename());  }  } |
| **Exercise 4.47**  boolean successful = selectedLot.bidFor(new Bid(bidder, value)); |
| **Exercise 4.48**  public void close()  {  for(Lot lot : lots) {  System.out.println(lot.getNumber() + ": " +  lot.getDescription());  // Include any details of a highest bid.  Bid highestBid = lot.getHighestBid();  if(highestBid != null) {  System.out.println(" Highest bidder: " +   highestBid.getBidder().getName());  System.out.println(" Bid: " +   highestBid.getValue());  }  else {  System.out.println(" Not sold");  }  }  } |
| **Exercise 4.49**  /\*\*  \* Returns a list of unsold lots  \*/  public ArrayList<Lot> getUnsold()  {  ArrayList<Lot> unsoldLots = new ArrayList<>();  for(Lot lot : lots) {  Bid highestBid = lot.getHighestBid();  if(highestBid == null) {  unsoldLots.add(lot);  }   }  return unsoldLots;  } |
| **Exercise 4.50**  The getLot method assumes that a Lot is stored at location getLotNumber()-1 in its ArrayList. If lots can be removed then index numbers may be changed. The getLot method always checks for consistency so if there is an inconsistency the an error message is printed in the terminal window. |
| **Exercise 4.51**  /\*\*  \* Return the lot with the given number. Return null  \* if a lot with this number does not exist.  \* @param number The number of the lot to return.  \*/  public Lot getLot(int number)  {  int nextIndex = 0;  while(nextIndex < lots.size() && lots.get(nextIndex).getNumber() != number) {  nextIndex++;  }  if (nextIndex == lots.size()) {  System.out.println("Lot number: " + number +  " does not exist.");  return null;  }   else {  return lots.get(nextIndex);  }  }  The fact that the lot numbers in the list are strictly increasing makes it possible to terminate an unsuccessful search before the end in many circumstances. For instance, once lots.get(nextIndex).getNumber() is larger than the sought lot number. Furthermore, a binary search rather than a linear search provides an alternative. Both could be explored as extension exercises. |
| **Exercise 4.52**  /\*\*  \* Remove the lot with the given lot number.  \* @param number The number of the lot to be removed  \* @return The Lot with the given number, or null if  \* there is no such lot.  \*/  public Lot removeLot(int number)  {  //First we find the lot with the given number  Lot lot = getLot(number);  if(lot != null) {  //Then we can use the method remove with lot as argument  lots.remove(lot);  }   return lot;  } |
| **Exercise 4.53**  LinkedList has these methods that ArrayList does not have:  void addFirst(E o) void addLast(E o) E getFirst() E getLast()  E removeFirst() E removeLast()  Where ‘E’ refers to the type of item stored in the list.  ArrayList has these methods that LinkedList does not have:  void ensureCapacity(int minCapacity)  void trimToSize() |
| **Exercise 4.54**  /\*\*  \* Determine the number of members who joined in the  \* given month  \* @param month The month we are interested in.  \* @return The number of members.  \*/  public int joinedInMonth(int month)  {  int count = 0;  if(month < 1 || month > 12) {  System.out.println("Month " + month +  " out of range. " +  "It must be in the range 1 ... 12");  }  else {  for(Membership member : members) {  if(member.getMonth() == month) {  count++;  }  }  }  return count;  } |
| **Exercise 4.55**  /\*\*   \* Remove from the club’s collection all members who   \* joined in the given month,   \* and return them stored in a separate collection object.   \* @param month The month of the membership.   \* @param year The year of the membership.   \* @return The members who joined in the given month and year.  \*/  public ArrayList purge(int month, int year)  {  ArrayList<Membership> purged = new ArrayList<>();  if(month < 1 || month > 12) {  System.out.println("Month " + month +  " out of range. " +  "It must be in the range 1 ... 12");  }  else {  Iterator<Membership> it = members.iterator();  while(it.hasNext()) {  Membership member = it.next();  if(member.getMonth() == month &&  member.getYear() == year) {  // Must use the remove method from the iterator.  // Check the documentation for the Iterator  // for more info.  it.remove();  purged.add(member);   }  }  }  return purged;  } |
| **Exercise 4.56**  public void printProductDetails()  {  for(Product product : stock) {  System.out.println(product.toString());  }  } |
| **Exercise 4.57**  public Product findProduct(int id)  {  boolean found = false;  int index = 0;  while(index < stock.size() && !found) {  if(stock.get(index).getID() == id) {  found = true;  }  else {  index++;  }  }  if(found) {  return stock.get(index);  }  else {  return null;  }  } |
| **Exercise 4.58**  public int numberInStock(int id)  {  Product product = findProduct(id);  if(product != null) {  return product.getQuantity();  }   else {  return 0;  }  } |
| **Exercise 4.59**  public void delivery(int id, int amount)  {  Product product = findProduct(id);  if(product != null) {  product.increaseQuantity(amount);  }   } |
| **Exercise 4.60**  /\*\*  \* Print details of all the products which has stock  \* levels below the given amount  \*/  public void printLowStockProducts(int upperLimit)  {  for(Product product : stock) {  if(product.getQuantity() < upperLimit) {  System.out.println(product.toString());  }  }  }  /\*\*  \* Add a product to the list if there is not already  \* one with the same ID.  \* @param item The item to be added.  \*/  public void addProduct(Product item)  {  if(findProduct(item.getID() == null) {  stock.add(item);  }  }  /\*\*  \* Try to find a product in the stock with the given name.  \* @return The identified product, or null if there is none  \* with a matching name.  \*/  public Product findProduct(String name)  {  boolean found = false;  int index = 0;  while(index < stock.size() && !found) {  if(stock.get(index).getName().equals(name)) {  found = true;  }  else {  index++;  }  }  if(found) {  return stock.get(index);  }  else {  return null;  }  } |
| **Exercise 4.61**  **A do-while loop only tests it condition after executing the statements in the loop’s body at least once. The loop continues to execute while the condition is true, as with the while loop.**  int i = 1; do {  System.out.print(i + " ");  i++; } while(i <= 10); System.out.println(); |
| **Exercise 4.62**  /\*\*  \* Show a list of all the tracks in the collection.  \*/  public void listAllTracks()  {  System.out.println("Track listing:");  if(! trackList.isEmpty()) {  int index = 0;  do {  Track aTrack = trackList.get(index);  System.out.println(aTrack.getDetails());  index++;  } while(index < trackList.size());  }  System.out.println();  } |
| **Exercise 4.63**  /\*\*  \* Find the index of the first file matching the given  \* search string.  \* @param searchString The string to match.  \* @return The index of the first occurrence, or -1 if  \* no match is found.  \*/  public int findFirst(String searchString)  {  // Record that we will be searching until a match is found.  boolean searching = true;  int index = 0;  if(! trackList.isEmpty()) {  do {  Track aTrack = trackList.get(index);  String filename = aTrack.getFilename();  if(filename.contains(searchString)) {  // A match. We can stop searching.  searching = false;  }  else {  // Move on.  index++;  }  }  while(searching && index < trackList.size());  }  if(searching) {  // We didn't find it.  return -1;  }  else {  // Return where it was found.  return index;  }  } |
| **Exercise 4.64**  switch(day) {      case 1: dayString = "Monday";               break;      case 2: dayString = "Tuesday";               break;      case 3: dayString = "Wednesday";               break;      case 4: dayString = "Thursday";               break;      case 5: dayString = "Friday";               break;      case 6: dayString = "Saturday";               break;      case 7: dayString = "Sunday";               break;      default: dayString = "invalid day";       break;  } |