**Chapter 4**

Ex. 4.1 – Done

Ex 4.2 - If I try to remove a file before adding one no error occurs. After glancing at the code, it seems that I wouldn’t get an error and that is because of how the method it written. It merely checks to make sure it is a valid element and then clears its contents so even if there is nothing there it works as if there is.

Ex 4.3 – Depending how java works the deleted item works it will either re-assign subscripts to the following elements if I remove a item that comes before other so my item 1 will become item 0 when the original item 0 8 is removed. The other scenario is that it leaves a hole and we get null or an error. After testing it the first case appears true.

Ex. 4.4 – private ArrayList< Book > library = new ArrayList< String >();

Ex. 4.5 – private ArrayList< Student > cs101 = new ArrayList< Student >();

Ex. 4.6 – private ArrayList< MusicTrack > tracks = new ArrayList< MusicTrack >();

Ex. 4.7 – library.add(“1984”);

Cs101.add(“ Jacob Knott” );

Track.add(“ Stairway to Heaven” );

Ex. 4.8 – If a collection stores 10 items the returned value from the method size would be 9.

Ex. 4.9 – object.get( 5 )

Ex. 4.10 – The index attached to the last object in an Arraylist with 15 objects would be 14.

Ex. 4.11 – addFavorite(favoriteTrack, files)

Ex. 4.12 – dates.remove(3)

Ex. 4.13– If an object is stored with an index value of 6 and then objects with indexes 0 and 9 are removed the object that was 6th will become linked to the 5th index because everything will move to fill in to the empty space. However since 9 is greater than six its removal will have no effect on the object.

Ex. 4.14– Done and saved.

Ex. 4.15– Done and saved.

Ex. 4.16 – Done and saved.

Ex. 4.17 – Done

Ex. 4.18 – the header will likely look like “public void listAllFiles()”. It wouldn’t have a return type as I would print the information to the screen.

Ex. 4.19– We could write the listAll method like this but we would have to write the print command for every index. If there were 3 there would be 3 and if 10 then 10 etc. although doing this what we “Know” ridiculous. I’m assuming this is a Segway to the for loop, (it was invaluable in C++).

Ex. 4.20– Done and saved

Ex. 4.21– Done

Ex. 4.22– Done

Ex. 4.23– Done

Ex. 4.24– Done and saved

Ex. 4.25– Done and saved

Ex. 4.26– Done and saved

Ex. 4.27– REVISIT SOON

Ex. 4.28– public void tracks(){

for(Track track : tracks) {

}

Ex. 4.29– Boolean found = false;

While( found ){

If(the keys are in next place){

found = false;

}

}

Ex. 4.30– int index = 10;

While(index <= 95){

System.out.println(index);

Index = index \* 5;

}

Ex. 4.31– int total = 0;

Int x = 1

while (x < =10){

total += x;

}

Ex. 4.32– public void sum(int a, int b){

int sum = 0;

while (a <=b){

result += a;

a++;

}

}

Ex. 4.33– private static Boolean isPrime(int n){

if (n < 2) return false;

if (n == 2) return true;

if (n % 2 == 0) return false;

for (int x = 3; x \* x <= n; x += 2)

if (n % x == 0) return false;

return true;

}

Ex. 4.34– The value returned from size should not vary during the findFirst method.

Ex. 4.35– Done and saved

Ex. 4.36 – Done and saved

Ex. 4.37 – Done and saved

Ex. 4.38 – By calling the stopPlaying method first in any method that starts another track the playing track should be stopped and the next should begin. – can’t test figuring out Ex 4.17

Ex. 4.39 – Done and saved CHECK FUNCTIONALITY REDO WITH ITERATOR TOOL

Ex. 4.40 – Done and Saved

Ex. 4.41 – Done and Saved

Ex. 4.42 – Done and Saved

Ex. 4.43 – Done and Saved

Ex. 4.44 – My first thought would be to randomize the index values but this cannot be done as an index value is assigned to a specific track unless on is removed or added. If I want to hold to my initial though of shuffling it a list containing all tracks in the organizer could be created and randomized, then it could play them all. As to getting them to only play an equal amount a while or statement could be used with a counter to only play a track when its playCount is less than the counter and when no more tracks can be played it increments the counter. That would have it play music forever.

Ex. 4.45 – Done and Saved

Ex. 4.46 – Done

Ex. 4.47 – Done and Saved

Ex. 4.48 – Attempted not semi-functional Struggling to get it to print the actual value of called variables for the name it doesn’t assign it to the correct variable and for the highest bid it prints the address not the value because idk.

Ex. 4.49 – Attempted but non-functional add get unsold , iterates over lots fields storing insold lots in new ArrayList and returns the list of unsold lots

Ex. 4.50 – If lots can be removed it would make it all following indexes would change. If the lots update indexes like in previous exercises it won’t be a problem. If a person knew what a following lot was, it won’t be there. A message or list could be displayed with what I know now saying what happened and the affected lots.

Ex. 4.51 – I plan to use the search method built in Ex.4.25 - 4.27. It will definitely need tweaking. It searches for a string, not an int .Could add field

Ex. 4.52 – Use a variation of remove file from collection method built in Ex 4.25 – 4.27

Ex. 4.53 – Both are implementations of list interface. But there are differences. LinkedList allows for “constant-time” insertions and removals. One plus of there removals is that it wont mess up whatever follows in the list. But it can only go one step forward or backwards. So to travel to another spot in the list all in between items must be iterated over. Arraylist however can skip around but if an item is removed or added all following objects are affected

Ex. 4.54 – Attempted, having trouble getting value from external method

Ex. 4.55 – “ ”

Ex. 4.56 –Done and saved.

Ex. 4.57 – Done

Ex. 4.58 – Done

Ex. 4.59 – Done

Ex . 4.60 – Done and saved.

Ex . 4.84 – The do while loop is a control structure is similar to a while loop. But its conditions aren’t tested until after the statements have been executed.

Public void countToTen(){

int x = 1

do {

System.out.ln(x);

x++;

}

while ( x < 11);

}

Ex . 4.85 – Done and saved.

Ex . 4.86 –Done and saved.

Ex . 4.87 – A switch case statement is used to for “decision making”. It can be used to test multiple cases instead of one at a time.

int x = Integer.parseInt(args[0]);

switch{

case 1 :

System.out.println(“Case 1”);

Case 2 :

System.out,println(“Case 2”);

default :

System.out.println(“Default case”);

}