Chapter 7

7.1: hour 19, 15, 11 were the busiest hours (when thinking about index values from 1-24 instead of 0-23)

7.2: Person[] people;

7.3: boolean[] vacant;

7.4: bracket uses noted;

7.5: int[] counts;
Boolean[] occupied = new boolean[5000;]

7.6: a) readings = new double[60];
b) urls = new String[90];
c) machines = new TicketMachine[5];

7.7: No objects are created from that declaration but rather an array to hold 20 string objects

7.8: needs to be square brackets
double[] prices = new double**[**50**]**;

7.9: You get the error message
java.lang.ArrayIndexOutOfBoundsException: 24

7.10:
```
System.out.println("Hr: Count");
int hour = 0;
while(hour < hourCounts.length)
{
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

7.11:
```
public void printGreater(double[] marks, double mean)
{
  for(int index = 0; index < marks.length; index++) {
     if(marks[index] > mean) {
          System.out.println(marks[index]);
     }
  }
}
```

7.12:
```
public LogAnalyzer(String filename)
{
      hourCounts = new int[24];
      reader = new LogfileReader(filename);
}
```

7.13:
```
public int numberOfAccesses()
{
    int total = 0;
    for(int hour : hourCounts)
    {
        total += hour;
    }
    return total;
}
```

7.14

7.15/7.16:
```
public int quitestHour()
{
    int quietHour = 0;
    for(int hour = 1; hour < hourCounts.length; hour++)
    {
        if(hourCounts[quietHour] > hourCounts[hour])
        {
            quietHour = hour;
        }
    }
    return quietHour;
}
public int busiestHour()
{
    int busyHour = 0;
    for(int hour = 1; hour < hourCounts.length; hour++)
    {
        if(hourCounts[busyHour] < hourCounts[hour])
        {
            busyHour = hour;
        }
    }
    return busyHour;
}
```

For each loop might be better because we had to iterate over the whole collection and it would look nicer but i used a for loop for practice

7.17: When using a for each loop the first value that is found is returned in the case of matching values

7.18:

7.19:

7.20:

7.21:
7.22: fixed size arrays have an advantage because they perform better and the number of students in a class is always known but it is harder to add and drop students so a list that could change easily might be better
7.23:
```
public void listAllFiles()
{
    for(int i=0; i<files.size(); i++) {
        System.out.println(i);
    }
}
```

7.24: automaton tested
7.25: the exact same patterns emerge
7.26: There are two different fill methods with int[] as a parameter type and they are used for filling arrays with a value. One methods fills the whole array the other fills a specified range of index values.

Arrays.fill(state, 0);

This is used in the reset() to change the arraylist to all have a value of 0
7.27: The patterns are different but the reset method would also need to be changed for same new pattern to emerge on further tests.
7.28:
```
int left = i == 0 ? 0 : state[i -1];
int center = state[i];
int right = i + 1 < state.length ? state[i+1] : 0;
```
Rewritten as conditional operators
7.29: a new array allows for it to be updated without altering the array in the loop
7.30: only one cells values need to be retained because the placement is based off the cells neighbors
7.31: code updated
7.32:
```
` public static int calculateNextStep(int left, int center, int right)
  {
     int nextStep = (left + right + right) % 2;
     return  nextStep;
  }
```

7.33: according to wolfram codes there are 256 ways
7.34: coder implemented
7.35:
7.36:
7.37:
7.38:

7.39:
7.40: