Exercise 7.1.1

We have decided to create an instance of DownloadWorker for each url. For each DownloadWorker we pass an url it should look up and download content from. This way we ensure that each of the urls will be downloaded concurrently:

```
List<DownloadWorker> downloadWorkers = new ArrayList<DownloadWorker>();
for (String url : urls) {
    downloadWorkers.add(new DownloadWorker(url, textArea));
}
```

Later on in the code we have iterate through list of downloadWorkers and add the execute() function to the button listeners, for executing the workers on fetch button click:

```
for (DownloadWorker worker : downloadWorkers) {
      fetchButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          worker.execute();
        }});
}
```

Exercise 7.1.2

For cancellation of the fetching on all of the workers, we have used the downloadWorkers list mentioned in the previous exercise. Again we iterate through the list of downloadWorkers where we (again) set on button click listener to perform a SwingWorker's base function called cancel, to cancel the execution of the worker:

```
for (DownloadWorker worker : downloadWorkers) {
       cancelButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          worker.cancel(false);
      }});
}
```

Exercise 7.1.3
For the progress bar we have implemented an AtomicInteger private field:

```
private static AtomicInteger atomicProgress = new AtomicInteger();
```

We increment the atomicProgress and set the progress bar accordingly with setProgress function:

```
setProgress((100 * atomicProgress.incrementAndGet()) / urls.length);
```

Exercise 7.2.1
Since multi-threaded GUIs are very hard to use and prone to deadlocks the Swing GUI only uses a single thread for all GUI related things, which is the event thread. It then has a separate main thread that should be used for computational work (heavy lifting) whilst the event thread should be used for things like listening for the user input and drawing stuff on the screen.

In the Lift exercise we manage the concurrency by making sure to follow those guidelines and only use the event thread for the GUI (paint, listeners) but NOT for any heavy lifting (sleep, move to, etc). Furthermore, we make sure that all operations on the main thread that can run concurrently are correctly synchronized.

Exercise 7.2.2

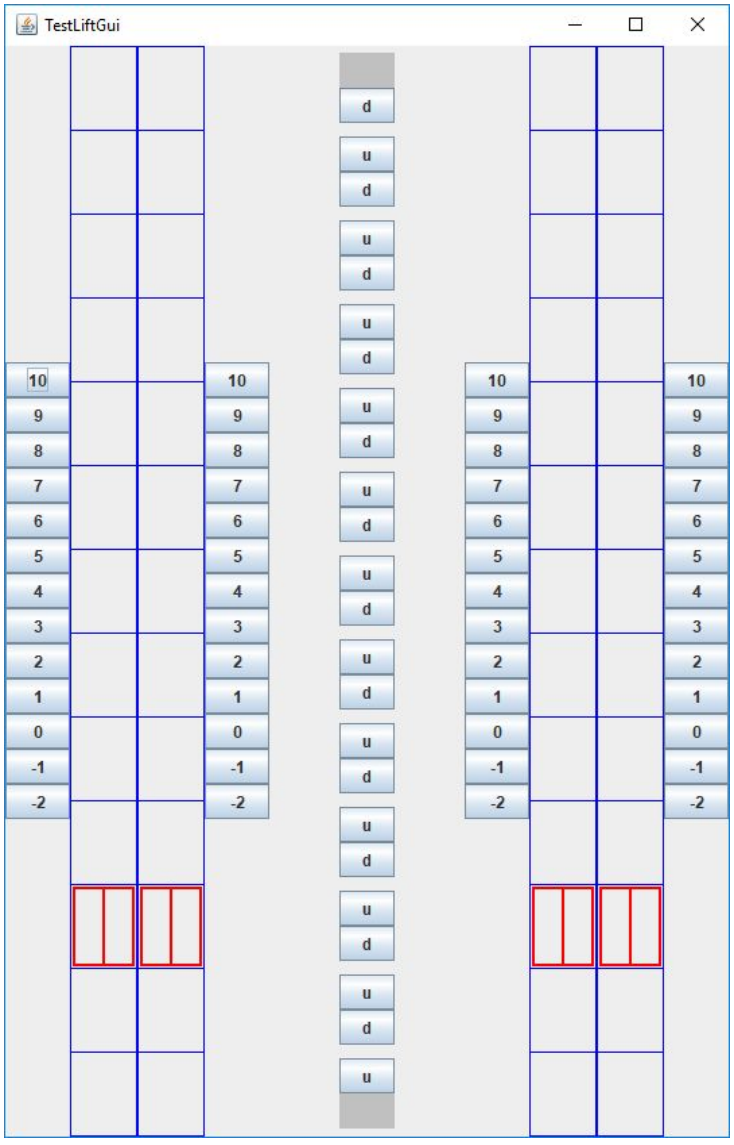
Changes in main method:

```
final LiftShaft
       shaft1 = new LiftShaft(),
       shaft2 = new LiftShaft(),
       shaft3 = new LiftShaft(),
       shaft4 = new LiftShaft();
final Lift
       lift1 = new Lift("Lift1", shaft1),
       lift2 = new Lift("Lift2", shaft2),
       lift3 = new Lift("Lift3", shaft3),
       lift4 = new Lift("Lift4", shaft4);
final LiftDisplay
       lift1Display = new LiftDisplay(lift1, true),
       lift2Display = new LiftDisplay(lift2, true),
       lift3Display = new LiftDisplay(lift3, false),
       lift4Display = new LiftDisplay(lift4, false);

LiftController controller = new LiftController(lift1, lift2, lift3, lift4);
Thread
       t1 = new Thread(lift1),
       t2 = new Thread(lift2),
       t3 = new Thread(lift3),
       t4 = new Thread(lift4);
t1.start(); t2.start(); t3.start(); t4.start();

// The graphical presentation
final JFrame frame = new JFrame("TestLiftGui");
final JPanel panel = new JPanel();
frame.add(panel);

panel.setLayout(new BorderLayout());
panel.add(new PairLiftDisplay(lift1Display, lift3Display),
BorderLayout.WEST);
panel.add(new PairLiftDisplay(lift2Display, lift4Display),
BorderLayout.EAST);

panel.add(new OutsideLiftButtons(controller), BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);
```

The new JPanel wrapper that wraps the two lifts:

```
class PairLiftDisplay extends JPanel {
 public PairLiftDisplay(LiftDisplay lift1, LiftDisplay lift2)
 {
   setLayout(new BorderLayout());
   add(lift1, BorderLayout.WEST);
   add(lift2, BorderLayout.EAST);
 }
}
```

The new look:



Exercise 7.2.3

For this one we added two new fields to the Lift model:

```
public boolean doorOpening;
public boolean doorClosing;
```

Then added an extra if/else to the run method:

```
public void run() {
   final double steps = 16.0;
   if (!doorOpening && !doorClosing) {
       switch (direction)
   {
           .........
   }
   else
   {
     if (doorOpening)
     {
         shaft.moveTo(floor, doorOpen);
         doorOpen += 1/steps;
         if (doorOpen > 1.0)
         {
             doorOpening = false;
             doorClosing = true;
         }
     }
     else if (doorClosing)
     {
         shaft.moveTo(floor, doorOpen);
         doorOpen -= 1/steps;
         if (doorOpen < 0.0)
         {
             doorClosing = false;
             doorOpen = 0.0;
         }
     }
```

To make that work we replaced both calls to “OpenAndCloseDoors” method with line:

```
doorOpening = true;
```

Then finally we used an executorservice to run the threads at a fixed interval rate:

```
Thread
       t1 = new Thread(lift1),
       t2 = new Thread(lift2),
       t3 = new Thread(lift3),
       t4 = new Thread(lift4);

executorService.scheduleAtFixedRate(t1, 0, 63, TimeUnit.MILLISECONDS);
executorService.scheduleAtFixedRate(t2, 0, 63, TimeUnit.MILLISECONDS);
executorService.scheduleAtFixedRate(t3, 0, 63, TimeUnit.MILLISECONDS);
executorService.scheduleAtFixedRate(t4, 0, 63, TimeUnit.MILLISECONDS);
```

Exercise 7.2.4

Added the buttons to the Lift model:

```
public final ArrayList<JButton> liftButtons;
```

Changes within the for loop on the InsideButtonsClass:

```
for (int floor = lift.highFloor; lift.lowFloor <= floor; floor--) {
   final int myFloor = floor;
   JButton button = new JButton(Integer.toString(floor));
   lift.liftButtons.add(button);
   panel.add(button);
   button.addActionListener(
           e -> {
               button.setForeground(Color.BLUE);
               lift.goTo(myFloor);
           }
   );
}
Collections.reverse(lift.liftButtons);
```

And a refresh button method on the Lift:

```
public synchronized void refreshButton(int floor) {
   int index = floor + 2;
   liftButtons.get(index).setForeground(Color.BLACK);
}
```

And finally we needed an appropriate place to put the call to refresh button which was right before we open/close the doors:

```
if (afterStop != null && (afterStop != Direction.Down || (int) floor ==
highestStop())) {
   refreshButton((int) floor);
   doorOpening = true;
   subtractFromStop((int) floor, direction);
}

.....

if (afterStop != null && (afterStop != Direction.Up || (int) floor ==
lowestStop())) {
   refreshButton((int) floor);
   doorOpening = true;
   subtractFromStop((int) floor, direction);
}
```