1. Lite exceptions

a) Vad kommer hända?

```
try {
     int[] a;
     a[0] = 1;
} catch (IndexOutOfBoundsException e) {
     System.out.println(“Index out of bounds!”);
} catch (StringIndexOutOfBoundsException e) {
     System.out.println(“StringIndex out of bounds!”);
} catch (ArrayIndexOutOfBoundsException e) {
     System.out.println(“ArrayIndex out of bounds!”);
} catch (Exception e) {
     System.out.println(“Exception!”);
} finally {
     System.out.println(“Finally!“);
}
System.out.println(“Moving on!“);
```

b) Vad kommer hända?

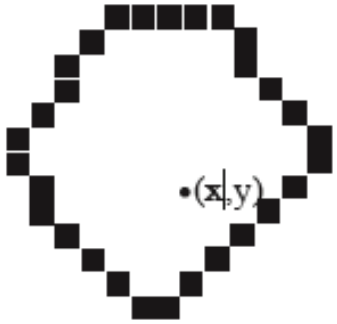
```
try {
     int[] a = new int[5];
     a[5] = 1;
} finally {
     System.out.println(“Finally!“);
}
System.out.println(“Moving on!“);
```

c) Vad kommer hända?

```
try {
     int[] a = new int[5];
     a[5] = 1;
} catch (Exception e) {
     System.out.println(“Exception!”);
} catch (ArrayIndexOutOfBoundsException e) {
     System.out.println(“ArrayIndex out of bounds!”);
} finally {
     System.out.println(“Finally!“);
}
System.out.println(“Moving on!“);
```

2. Lite mera Rekursion

Att fylla ett slutet område. Bilden till höger är en sk. rasterbild som schematiskt kan beskrivas som ett rutmönster i vilket en ruta (pixel) kan vara tänd eller släckt. De fyllda kvadraterna markerar svarta punkter (i verkligheten är de cirklar), alla övriga punkter är vita. (Vita punkter syns ej.)



Skriv en rekursiv metod, som givet en punkt (x, y) innanför den svarta randen fyller hela området innanför randen med svarta pixlar.

```
public void fill(char[][] m, int x, int y)
```

Det kan förutsättas att det omslutna området är sådant att man från en godtycklig punkt inom det kan komma till varje annan genom att förflytta sig vertikalt och horisontellt (men ej diagonalt). Det kan också förutsättas att området verkligen är slutet så det går inte att "smita ut" om man bara förflyttar sig en pixel vertikalt eller horisontellt.
Parametern `m` är en representation av bilden i form en matris med ettor (svarta pixlar) och nollor (vita pixlar)

Exempel på en matris, `m`, hittar du till vänster nedan. (Observera att matrisens utseende inte stämmer riktigt med figuren ovan).

Exempel på utdata efter anrop av fill(m, 2,4); till höger.

000111000000	000111000000
001000100000	001111100000
001000100000	001111100000
001000111100	001111111100
010000001000	011111111000
010000010000	011111110000
001000100000	001111100000
000111000000	000111000000

Tips. Du skall INTE testa om dina grannar är tända/släckta. Det måste dom göra själva. Du ansvarar bara för dig själv.
Du kan förutsätta att matrisen existerar och att det inte finns tomma rader.

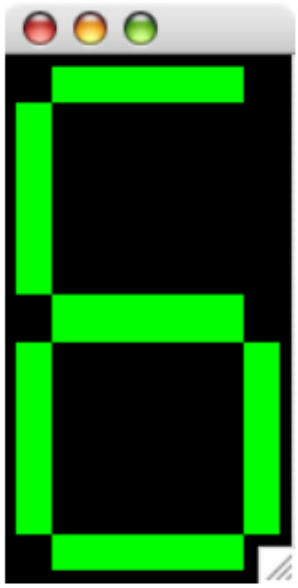
(6p)

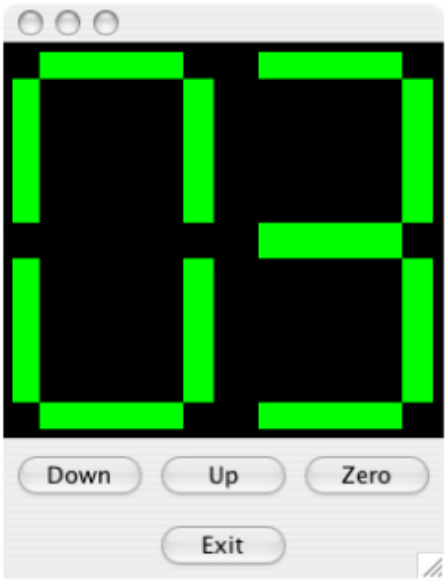
3. Lite mera Swing

En klass med följande gränssnitt finns tillgänglig, klassen är en digital siffra enligt vänstra figuren nedan där `setNumber(6)` anropats.

```
public class Digit extends JPanel {
     // prefered, but not mandatory, colors
     // color is set in Digit by calling getForeground()
     // so users of the class can set color by calling setForeground
     public static final Color greenDigit      = Color.green;
     public static final Color redDigit        = Color.red;
     public static final Color backgroundColor = Color.black;

     public Digit();      // sets the digit to 0
     public Digit(int n); // sets the digit to n
     public void setNumber(int n); // sets the digit to n
     public int getNumber();
     public void paintComponent(Graphics g);
}
```





Du skall nu skriva en klass, `DigitalCounter`, som ser ut som figuren till höger ovan och som använder sig av klassen Digit. Den skall, förutom konstruktor och actionPerformed, innehålla följande metod:

```
// outputs a number in the range 0..99.
// when nbr<0 outputs 99, when nbr>99 outputs 0
public void outputNumber(int nbr) {
```

När man klickar på "down" skall talet som visas minskas med ett och när man klickar på "up" ökas med ett, bägge med "wraparound" dvs `up(99)` blir 0 osv. Knappen "zero" nollställer räknaren och "exit" avslutar programmet.

(14p)