Data och Informationsteknik / Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Erland Holmström Göteborg 13 feb 2014

Examination in

PROGRAMMERINGSTEKNIK F1 TIN212

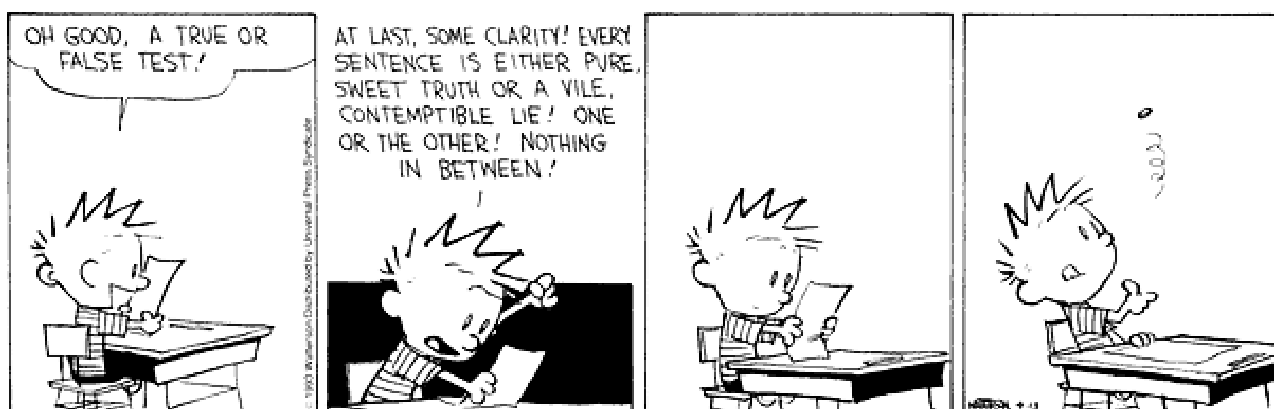
DAY: Saturday DATE: 2014-02-15 TIME: 8.30-13.30 (OBS 5 tim) ROOM: V

Responsible teacher:	Erland Holmström tel. 1007, home 0708-710 600
Results:	Are sent by mail from Ladok.
Solutions:	Are eventually posted on homepage.
Inspection of grading:	The exam can be found in our study expedition after posting of results.
	Time for complaints about grading are announced on homepage after the result are published or mail me and we find a time.
Grade limits:	CTH: 3=26p, 4=36p, 5= 46p, max 60p
Aids on the exam:	**Bravaco, Simonson: Java Programming From the Grounf Up**

Observe:

- Start by reading through all questions so you can ask questions when I come. I usually will come after appr. 2 hours.
- All answers must be motivated when applicable.
- Write legible! Draw figures. Solutions that are difficult to read are not evaluated!
- Answer concisely and to the point.
- The advice and directions given during course must be followed.
- Programs should be written in Java, indent properly, use comments and so on.
- Start every new problem on a new sheet of paper.

Good Luck!



Problem 1. Vilka av följande påståenden är korrekta? Inga motiveringar behövs men fel svar ger minuspoäng.

a) Om man tar bort alla tre uttrycken i en for loop så får man en oändlig loop.

b) Villkoret i for loopen är falskt när man går ur loopen.

c) En subklass kan överlagra en metod från sin superklass genom att skriva sin egen version och den kommer då att ersätta den version som ärvdes.

d) Klass variabler och klassmetoder deklareras genom att använda ordet "static".

e) Ett objekt av typen Choice är en "component".

f) Flera olika exceptions kan fångas antingen genom flera catch block eller genom att fånga en superclass' exception.

(3p)

Problem 2. super Primes - Testar: enkla metoder, val, snurror

Ett tal som 7331 är ett superprimtal av längd 4 eftersom

7331, 733, 73 och 7 alla är primtal.

Antag att det finns en metod isPrime(int p) som avgör om ett tal, p, är primtal eller inte som du kan använda. Du skall skriva 2 metoder:

`boolean superPrime(int p)` som avgör om p är ett superprimtal.

(blir väldigt enkel med rekursion men du får göra hur du vill)

`void allSuperPrimes(int l)` som skriver ut alla superprimtal med längd l dvs anropet `allSuperPrimes(4)` skall ge utskriften

2333 2339 2393 2399 2939 3119 3137 3733 3739 3793 3797 5939 7193 7331 7333 7393

och `allSuperPrimes(7)` ger

2339933 2399333 2939999 3733799 5939333 7393913 7393931 7393933

(9p)

Ibland klagar studenter på att det är så många olika problemställningar att sätta sig in i på en tenta och denna tentan är ett försök att undvika det. Jag har också lagt ner mycket jobb på att försöka formulera vad du skall göra tex exakt vilka metoder som skall/inte skall implementeras och på att dela upp uppgifterna i små delar. Kommentera gärna på kursenkäten (eller direkt i tentan) vad du tyckte så jag får lite respons

De följande frågorna handlar alla om samma grundproblem men kan till stora delar lösas individuellt och testar olika saker. Cell, BlobModel och findBlob uppgifterna är nästan helt oberoende av varandra och BlobMain2 har egentligen inget alls med dom andra att göra även om det handlar om samma grundproblem. Detta har både för och nackdelar. En nackdel är ju att om man inte förstår uppgiften så blir det svårt. Fråga i så fall när jag kommer.

När du löser en uppgift kan du anta att dom andra är lösta och använda dig av dom.

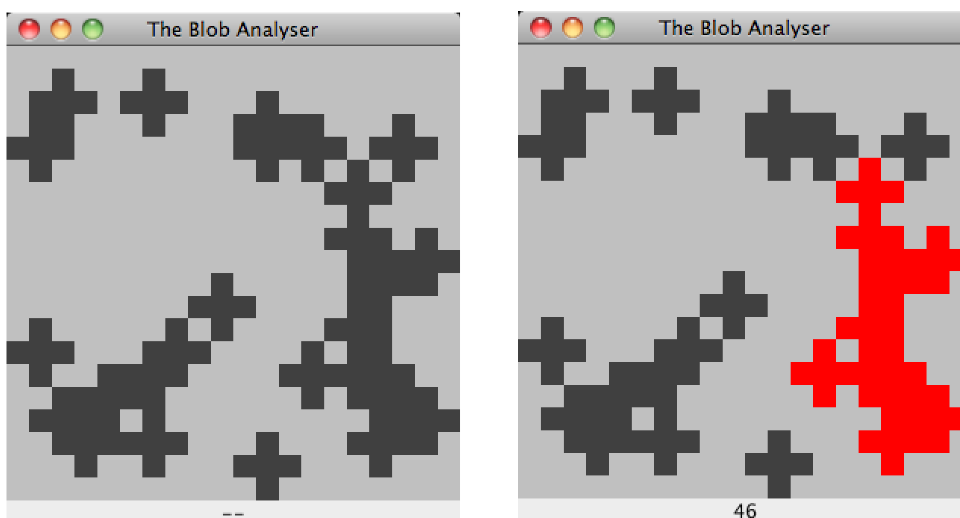
Du behöver inte skriva Javadoc, delar finns i slutet (läs), och inte skriva importsatser.

Introduktion:

Antag att vi har en 2-dimensionell bild med rektanglar eller ”celler” där varje cell innehåller en färg, antingen en normal färg (ljusgrått i figuren till vänster nedan) eller en annan färg som indikerar något avvikande (mörkgrått i figuren) . I en riktig tillämpning skulle man naturligtvis ha flera färger men vi nöjer oss med de viktigaste för principen här. Matrisen kan vara ett behandlat resultat från en röntgenbild, maligna celler i ett vävnadsprov, land på en karta, en satellitbild eller liknande.

En ”blob” är en samling celler som är sammanhängande dvs dom gränsar mot varandra.
Figuren till vänster visar en matris med celler, några är ”normala” eller tomma (ljusa i figuren), några ”abnormala” eller fyllda (mörka). De abnormala cellerna som är sammanbundna bildar ”blobs”. Vi vill kunna klicka på en cell och få reda på hur stor den blob är som cellen finns i, samtidigt som cellerna i bloben skall färgas röda. I figuren till höger har jag klickat på en av cellerna och då har den blobens storlek räknats ut samt färgats röd (ser ut som en grå ton mellan de bägge andra gråtonerna i sv/vitt tryck), bloben består av 46 celler och det står utskrivet under).

Vanligen läser man in matrisen från en fil (det är dock inte med på tentan) men man kan också i undervisningssyfte skapa en slumpmässig matris.



Vi kommer att behöva flera klasser för det här programmet. I slutet finns Javadoc för 2 av klasserna du skall skriva.

Den första är given och definierar ett antal konstanter för färger du kan (läs skall) använda

```
public class State {
   public static final Color EMPTY      = Color.white;
   public static final Color MARKED     = Color.red;
   public static final Color NORMAL     = Color.lightGray;
   public static final Color ABNORMAL   = Color.darkGray;
   public static final Color TEMPORARY  = Color.black;
}
```

Problem 3. *Cell - Testar: Enkla klasser, arv, kasta exception*

Den andra klassen, `Cell`, representerar en cell som vi skall använda vid utskrift senare och den ärver en `JButton` (enkelt att klicka på knappar) men lägger till några egenskaper: sin placering i matrisen (row, col) och en färg. `Cell`-klassen har en konstruktor enligt:

```
public Cell(String buttonText, int row, int col,
                                              Color cellColor) {...
```

Den lagrar sina data (obs row och col skall inte kunna ändras), sätter sin färg (sätt knappens bakgrundsfärg) och tillhandahåller getters för sina data och en setter för färgen (`setCellColor`). När man sätter färgen så tillåts bara färgerna i `State` ovan annars kastas en *lämplig* exception. Undvik att dubblera kod.

Du skriver klassen, konstruktorn och metoden setCellColor. Du behöver alltså inte skriva några getters men kan använda dom som om dom finns.

Man kan få knapparna att se ut som i figurerna istället för standardutseendet med satserna

```
setBorder(new LineBorder(Color.white, 0) );
setOpaque(true);
```

(6p)

Problem 4. *BlobModel - Testar: Matriser, lite svårare klasser, enkla kontrollstrukturer.*

Vår tredje klass, `BlobModel`, är en "modell" för matrisen och använder inte klassen Cell. Den innehåller en matris med tillståndet och alla data om matrisen (som antal rader/kolumner mm) och kan göra beräkningar mot matrisen men vet inget om hur matrisen skall presenteras tex på en skärm.
Matrisen lagras lämpligen som en matris av värden (dvs färger) från `State`. (Man skulle kunna använda heltal, eller helst en uppräkningstyp, men vi kan lika gärna använda en färg.) Vi låter konstruktorn skapa en slumpmässigt ifylld matris med värdena NORMAL och ABNORMAL:

```
   ... BlobModel(int nbrRows, int nbrCols, double probability)...
```

`Probability` är ett tal mellan 0.0 och 1.0 och den anger ungefär hur stor andel som skall vara fyllt med ABNORMAL så vi kan skapa slumpmässiga data. (Det bör såklart finnas fler konstruktorer men vi behöver inte dom)

En setter för att ändra tillstånd (dvs färg) på en cell:

```
void setState(int x, int y, Color newColor)...
```

samt en metod för att återställa matrisen dvs ändra tillbaka de celler man markerat med MARKED under räknandet av en blob till sin ursprungsfärg, ABNORMAL: (se även `findBlob`)

```
void reState()...
```

Vi behöver också getters `se Javadocen` men dom behöver du inte skriva (men du kan använda dom).

Låt konstruktorn skapa en slumpmässig matris på något lämpligt sätt genom att använda slumptal och parametern probability men det spelar inte så stor roll exakt hur du gör och ägna inte tid åt detaljer.

Du skriver alltså klassen, konstruktorn, setState och reState.

Här finns också metoden findBlob som beräknar storleken på en blob, men den får en egen uppgift.

(8p)

Problem 5. *findBlob – Testar: rekursion*

Skriv den *rekursiva* metoden som beräknar storleken på en blob som innehåller punkten (x, y) dvs givet (x, y) skall den räkna alla fyllda sammanhängande grannar och markera dem med tillståndet MARKED (dvs röd).

```
public int findBlob(int x, int y) {...
```

Klickar man utanför en blob skall noll returneras. Detsamma när man i rekursionen besöker "grannar" som ligger utanför matrisen. Men om färgen i cellen x, y är ABNORMAL skall man räkna den, ändra tillståndet till MARKED (så den inte besöks igen och så den syns när matrisen presenteras på skärmen) och besöka grannarna (det räcker om du tar med upp, ner, höger och vänster grannar, diagonalerna behöver inte vara med)

Du får inte besöka alla celler i matrisen om det inte är nödvändigt.
Det blir bara typ 15 rader inklusive "parentesrader" om man gör en sak per rad.

Den här metoden ligger i klassen `BlobModel` men allt sker (skall ske) via getters/setter så den skulle lika gärna kunna ligga utanför klassen.

(8p)

Problem 6. *BlobView - Testar: Swing, användande av klasserna vi skrivit tidigare, händersehantering*

Fjärde klassen, `BlobView,` är en "vy" klass som hanterar hur modellen skall presenteras visuellt. Naturligtvis är detta en JPanel. Hur man ritar matrisen kan variera, man kan tex rita små rutor för varje värde men vi skall använda vår klass Cell, som ju är en knapp, när vi ritar upp matrisen. Lägger man knappar i en JPanel så sköts ritandet automatiskt. Lämpligen har vi även här en matris men nu gör vi alltså en matris av objekt av typen Cell. När man klickar på en cell skall en händelse skickas till `BlobControl`-klassen i nästa uppgift som i sin tur anropar `findBlob` i modellen. `BlobView` måste alltså registrera lyssnare för dessa knapptryck hos `BlobControl`.
Konstruktorn behöver en modell och en controler som parameter

```
public BlobView(BlobModel bm, BlobControl bc) {...
```

Modellen behövs för att kunna ställa frågor om matrisen och controllern för att kunna registrera lyssnare.

Det behövs också en metod `reDraw` som överför modellens färger till våra "celler".

Storleken på panelen skall vara antal celler gånger en konstant (tex 10)

Du skriver klassen, konstruktorn och reDraw.

(8p)

Problem 7. *BlobControl - Testar: Swing, händelsehantering*

Femte klassen, `BlobControl`, har hand om "kontrollen" dvs användarens interaktion med programmet. `BlobControl` är en Jframe. Den skapar det slutliga fönstret med en `BlobView` och en `Jlabel.`

Konstruktorn får en `BlobModel` som parameter och skapar en `BlobView.`

```
public BlobControl(BlobModel aBlob) {...
```

Du behöver inte centrera din Jlabel.

Du skriver klassen, konstruktorn och händelsehanteraren.

(8p)

Klasserna ovan kan köras med det korta huvudprogrammet som finns längst ner på nästa sida eller det du skriver i nästa uppgift.

Problem 8. BlobMain2 - Testar: Argument från kommandoraden, inläsning från kommandoraden, läsning med scanner, fånga exception.

Den sista klassen är ett mer avancerat huvudprogram som läser in storlek på matrisen mm. Den här klassen är motorn för det hela men egentligen har den ingenting med de andra klasserna att göra som den ser ut nu och handlar mest om att läsa argument från användaren, sköta felhantering och sedan läsa in från en fil med en scanner.

Man skall kunna starta programmet på tre sätt:

```
java BlobMain2
java BlobMain2 x y
java BlobMain2 filnamn
```

I det första fallet skall programmet fråga efter antalet rader och antalet kolumner med hjälp av en (eller två) JOptionPane. Sedan skapas en model och en control enligt

```
BlobModel bm = new BlobModel(nRows, nCols, 0.35);
new BlobControl(bm);
```

I det andra fallet används x och y givna på kommandoraden som värden för nRows och nCols och sedan sker samma anrop som ovan. (skilj på x y fallet och filnamnsfallet genom att testa om första tecknet i första argumentet är en siffra)

I det tredje fallet skall programmet läsa in en matris från en fil. I filen ligger först två rader med *maximala* antalet rader respektive kolumner i matrisen (det behöver alltså inte vara så många rader/kolumner i filen utan kan vara färre). Sedan kommer ett antal rader med ettor och nollor som representerar ABNORMAL respektive NORMAL. Du behöver dock bara läsa in raderna med nollor/ettor till ett fält med strängar här, inte uppdatera modellen med dom (då hade vi behövt fler metoder i modellen). Istället för att uppdatera modellen skall du bara skriva ut fältet i kommandofönstret på skärmen. Exempel på indata i filen och på utdata på skärmen:

```
20
21
101011101110
101010010010
101010010010
110010001000
```

Du kan anta att innehållet i filen är korrekt skriven tex samma längd på raderna och att raderna inte är längre eller fler än angivits (20/21 i det här fallet).
Och du kan anta att det finns minst 2 rader (20 och 21 ovan).

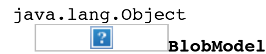
Du skall dock hantera 3 andra fel med exceptions. Du skall hantera att användaren inte skriver ett tal och att filen inte finns samt den exception som klassen `Cell` eventuellt kastar. Skriv en kort förklaring på varje och på en av dom en ”stacktrace” som åtgärd och avsluta programmet.

(10p)

Det korta huvudprogrammet:

```
public class BlobMain1 {
  public static void main(String[] args) {
    BlobModel bm = new BlobModel(30, 30, 0.35);
    new BlobControl(bm);
  }
}
```

Class BlobModel

```
java.lang.Object
```

BlobModel

```
public class BlobModel
extends java.lang.Object
```

Constructor Summary

BlobModel(int nbrRows, int nbrCols, double probability) Construct a Two Dim Grid of the specified size filled with values NORMAL and ABNORMAL.

Method Summary

int	**findBlob**(int x, int y) Finds the number of cells in the blob at (x,y).
int	**getNbrCols**() Obvoius functions for the getters.
int	**getNbrRows**()
java.awt.Color	**getState**(int x, int y) Get the state at a given coordinate
void	**reState**() Change the state of all cells that are marked with MARKED to ABNORMAL.
void	**setState**(int x, int y, java.awt.Color newColor) Change the state at a given coordinate

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

BlobModel

```
public BlobModel(int nbrRows,
                 int nbrCols,
                 double probability)
```

Construct a Two Dim Grid of the specified size filled with values NORMAL and ABNORMAL.

Exactly how the filling is done is not so important.

Parameters:
`nbrRows` - - Number of rows
`nbrCols` - - Number of columns
`probability` - - nbr between 0.0..1.0 how big part that is filled with ABNORMAL or used somehow to create an interesting matrix

Method Detail

getNbrCols

```
public int getNbrCols()
```

Obvoius functions for the getters.

Returns:
nbrCols

---

getNbrRows

```
public int getNbrRows()
```

Returns:
nbrRows

---

getState

```
public java.awt.Color getState(int x,
                               int y)
```

Get the state at a given coordinate

Parameters:
`x` - - The column number
`y` - - The row number
Returns:
The state at the given coordinate

---

setState

```
public void setState(int x,
                     int y,
                     java.awt.Color newColor)
```

Change the state at a given coordinate

Parameters:
`x` - - The column number
`y` - - The row number
`newColor` - - The state to set the cell to

reState

```
public void reState()
```

Change the state of all cells that are marked with MARKED to ABNORMAL.

findBlob

```
public int findBlob(int x,
                    int y)
```

Finds the number of cells in the blob at (x,y).

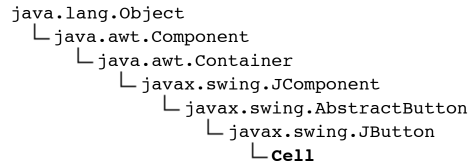
Parameters:

`x` - The x-coordinate of a blob cell
`y` - The y-coordinate of a blob cell

Returns:

The number of cells in the blob that contains (x, y) if the cell at (x,y) are not in the state ABNORMAL zero is returned

Class Cell



```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └javax.swing.AbstractButton
                  └javax.swing.JButton
                      └Cell
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.ItemSelectable, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.SwingConstants

```
public class Cell
extends javax.swing.JButton
```

See Also:

Serialized Form

.. nerkortat...

Constructor Summary

Cell(java.lang.String buttonText, int row, int col, java.awt.Color cellColor) Construct a cell.

Method Summary

java.awt.Color	**getCellColor**()
int	**getCol**()
int	**getRow**() Obvoius functions for the getters.
void	**setCellColor**(java.awt.Color cellColor) Sets the background color for the cell to one of the colors from State.

Methods inherited from class javax.swing.JButton

getAccessibleContext, getUIClassID, isDefaultButton, isDefaultCapable, paramString, removeNotify, setDefaultCapable, updateUI

Methods inherited from class javax.swing.AbstractButton

actionPropertyChanged, addActionListener, addChangeListener, addImpl, addItemListener, checkHorizontalKey, checkVerticalKey, configurePropertiesFromAction, createActionListener, createActionPropertyChangeListener, createChangeListener, createItemListener, doClick, doClick, fireActionPerformed, fireItemStateChanged, fireStateChanged, getAction, getActionCommand, getActionListeners, getChangeListeners, getDisabledIcon, getDisabledSelectedIcon, getDisplayedMnemonicIndex, getHideActionText, getHorizontalAlignment, getHorizontalTextPosition, getIcon, getIconTextGap, getItemListeners, getLabel, getMargin, getMnemonic, getModel, getMultiClickThreshhold, getPressedIcon, getRolloverIcon, getRolloverSelectedIcon, getSelectedIcon, getSelectedObjects, getText, getUI, getVerticalAlignment, getVerticalTextPosition, imageUpdate, init, isBorderPainted, isContentAreaFilled, isFocusPainted, isRolloverEnabled, isSelected, paintBorder, removeActionListener, removeChangeListener, removeItemListener, setAction, setActionCommand, setBorderPainted, setContentAreaFilled, setDisabledIcon, setDisabledSelectedIcon, setDisplayedMnemonicIndex, setEnabled, setFocusPainted, setHideActionText, setHorizontalAlignment, setHorizontalTextPosition, setIcon, setIconTextGap, setLabel, setLayout, setMargin, setMnemonic, setMnemonic, setModel, setMultiClickThreshhold, setPressedIcon, setRolloverEnabled, setRolloverIcon, setRolloverSelectedIcon, setSelected, setSelectedIcon, setText, setUI, setVerticalAlignment, setVerticalTextPosition

Methods inherited from class javax.swing.JComponent

addAncestorListener, addNotify, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBaseline, getBaselineResizeBehavior, getBorder, getBounds, getClientProperty, getComponentGraphics, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap, getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPopupLocation, getPreferredSize, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getWidth, getVisibleRect, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isOpaque, isOptimizedDrawingEnabled, isPaintingForPrint, isPaintingTile, isRequestFocusEnabled, isValidateRoot, paint, paintChildren, paintComponent, paintImmediately, paintImmediately, print, printAll, printBorder, printChildren, printComponent, processComponentKeyEvent, processKeyBinding, processKeyEvent, processMouseEvent, processMouseMotionEvent, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setComponentPopupMenu, setDebugGraphicsOptions, setDefaultLocale, setDoubleBuffered, setFocusTraversalKeys, setFont, setForeground, setInheritsPopupMenu, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setUI, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, getMousePosition, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove,

removeAll, removeContainerListener, setComponentZOrder, setFocusCycleRoot, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component
action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, inside, isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Cell

```
public Cell(java.lang.String buttonText,
            int row,
            int col,
            java.awt.Color cellColor)
```

Construct a cell.

Parameters:

`buttonText` - - text for the button

`row`, - col - basically two numbers that the cell only save

`cellColor` - - background color for the cell

Method Detail

getRow

```
public int getRow()
```

Obvoius functions for the getters.

---

getCol

```
public int getCol()
```

---

getCellColor

```
public java.awt.Color getCellColor()
```

---

setCellColor

```
public void setCellColor(java.awt.Color cellColor)
```

Sets the background color for the cell to one of the colors from State.

Parameters:

`cellColor` - - the color to set

Throws:

`hmmmmhmmmmException` - - if color not in State (hmmmmmhmmmm is to be chosen by you)

```
1  Tenta  140215
2  
3  Uppg 1
4  ================================================================
5  alla sanna
6  
7  Uppg 2
8  ================================================================
9  static public boolean superPrimeR(int p) {
10     // undvik att anropa isPrime i onödan tex med negativa tal
11     if (p<=1) {
12         return false;
13     } else if (p<9) {
14         return isPrime(p);
15     } else {
16         return isPrime(p) && superPrimeR(p/10);
17     }
18 }
19 
20 static public void allSuperPrimes(int l) {
21     if (l>=1) {
22         // sätt upp start och slutvärden. I princip kan man börja
23         // på 2 istället och undvika tal som börjar på
24         // 4, 6 och 9 men det är overkill här
25         int start = 1; // skall bli 10000...., l st
26         int slut  = 9; // skall bli 99999...., l st
27         for (int i = 1; i<l; i++) {
28             start = start*10;
29             slut = slut*10+9;
30         }
31         int tab = 0;
32         for (int i = start; i <= slut; i++) {
33             if (superPrimeR(i)) {
34                 System.out.print(i + "\t");
35                 // tabellering behövdes inte
36                 tab++;
37                 if (tab%10==0) {System.out.println();}
38             }
39         }
40     }
41     System.out.println();
42 }
43 
44 
```

```
45 Uppg 3
46 ================================================================
47 public class Cell extends JButton {
48     private final int row;
49     private final int col;
50     private String buttonText;
51     private Color cellColor = null;
52 
53     public Cell(String buttonText, int row, int col, Color cellColor) {
54         // row/col behöver inte felhanteras här eftersom en Cell inte
55         // vet vad dom används till (och inte använder dom)
56         //cellColor felhanteras i setcellColor
57         super(buttonText);
58         this.buttonText = buttonText;
59         this.row = row;
60         this.col = col;
61         setCellColor(cellColor);
62         // de 2 följande behövdes ej på tentan
63         setBorder(new LineBorder(Color.white, 0) );
64         setOpaque(true);
65     }
66 
67     public void setCellColor(Color cellColor) {
68 
69         if ( cellColor.equals(State.EMPTY)
70           || cellColor.equals(State.MARKED)
71           || cellColor.equals(State.NORMAL)
72           || cellColor.equals(State.ABNORMAL)
73           || cellColor.equals(State.TEMPORARY) ) {
74             this.cellColor = cellColor;
75             setBackground(cellColor);
76         } else {
77             throw new IllegalArgumentException("color not allowed");
78         }
79     }
80 }
81 
82 
```

```
83  Uppg 4
84  ================================================================
85  public class BlobModel {
86  
87      private Color[][] theGrid;
88      private int nbrRows;
89      private int nbrCols;
90  
91      public BlobModel(int nbrRows, int nbrCols, double probability) {
92          if (nbrRows<1 || nbrCols<1) {
93              throw new IllegalArgumentException("Rows and Cols must be >0");
94          }
95          // similar test for probability
96          this.nbrRows = nbrRows;
97          this.nbrCols = nbrCols;
98          theGrid = new Color[nbrRows][nbrCols];
99          // initialise with NORMAL
100         for (int i = 0; i < nbrRows; ++i) {
101             for (int j = 0; j < nbrCols; ++j) {
102                 theGrid[i][j] = State.NORMAL;
103             }
104         }
105         // exactly how the filling is done is not so important
106         // details are not important
107         // Fill the grid of squares randomly.
108         int filledCells = 0;
109         while (filledCells < probability*nbrRows*nbrCols) {
110             int i = (int) (1+Math.random()*(nbrRows-2));
111             int j = (int) (1+Math.random()*(nbrCols-2));
112             // System.out.println("i=" + i + " j=" + j); // debug
113             // här gjorde jag en stjärna men det räcker med
114             // en cell på tentan
115             theGrid[i][j]   = State.ABNORMAL;
116             filledCells = filledCells+5;
117         }
118     } // end Constructor
119 
120     public void setState(int x, int y, Color newColor) {
121         //take the automatic ArrayIndexOutOfBounds here for x,y
122         // newColor: see test in Cell.setCellColor
123         theGrid[x][y] = newColor;
124     }
125 
126     public void reState() {
127         for ( int i = 0; i < getNbrRows(); ++i ) {
128             for ( int j = 0; j < getNbrCols(); ++j ) {
129                 if ( getState(i, j).equals(State.MARKED) ) {
130                     theGrid[i][j] = State.ABNORMAL;
131                 }
132             }
133         }
134     } // end reState
135 }
136 
137 
```

```
138 Uppg 5
139 ================================================================
140 public int findBlob(int x, int y) { // 8
141     if (x < 0 || x >= getNbrRows()  // outside the grid
142         || y < 0 || y >= getNbrCols()) {
143         return 0;
144     } else if (getState(x, y).equals(State.ABNORMAL)) {
145         setState(x, y, State.MARKED);
146         // alla riktningar behövs ej på tentan
147         return 1                             // medsols
148             //+ findBlob(x - 1, y + 1)  // övre vänstra diag
149             + findBlob(x, y + 1)        // upp
150             //+ findBlob(x + 1, y + 1)  // övre högra diag
151             + findBlob(x + 1, y)        // höger
152             //+ findBlob(x + 1, y - 1)  // nedre högra diag
153             + findBlob(x, y - 1)        // ner
154             //+ findBlob(x - 1, y - 1)  // nedre vänstra diag
155             + findBlob(x - 1, y);       // vänster
156     } else {
157         return 0;
158     }
159 } // end
160
161
```

```
162 Uppg 6
163 ==============================================================
164 public class BlobView extends JPanel {
165 
166     /** Prefered button size */
167     private static final int PREFERED_BUTTON_SIZE = 15;
168     private BlobModel bm;
169     private BlobControl bc;
170     private Cell[][] theButtonGrid;
171     private int nRows;
172     private int nCols;
173 
174     public BlobView(BlobModel bm, BlobControl bc) {
175         if (bm==null || bc==null) {
176             throw new IllegalArgumentException("bm==null || bc==null");
177         }
178         this.bm = bm;
179         this.bc = bc;
180         nRows = bm.getNbrRows();
181         nCols = bm.getNbrCols();
182         // a panel for the cells
183         //JPanel cellPanel = new JPanel();
184         Dimension d = new Dimension(nCols * PREFERED_BUTTON_SIZE,
185                                     nRows * PREFERED_BUTTON_SIZE);
186         setPreferredSize(d);
187         setLayout(new GridLayout(nRows, nCols));
188         theButtonGrid = new Cell[nRows][nCols];
189         for (int i = 0; i < nRows; ++i) {
190             for (int j = 0; j < nCols; ++j) {
191                 theButtonGrid[i][j] = new Cell( "", i, j, bm.getState(i, j) );
192                 theButtonGrid[i][j].addActionListener(bc);
193                 add(theButtonGrid[i][j]);
194             }
195         }
196     } // end constructor
197 
198     public void reDraw() {
199         for (int i = 0; i < nRows; ++i) {
200             for (int j = 0; j < nCols; ++j) {
201                 // överför färgen från modellen till knapparna
202                 theButtonGrid[i][j].setCellColor( bm.getState(i, j) );
203             }
204         }
205     } // end reDraw
206 }
207 
```

```
208 Uppg 7
209 ==============================================================
210 public class BlobControl extends JFrame implements ActionListener {
211
212     private BlobModel bm;
213     private BlobView cellPanel;
214     private JLabel blobResult;
215
216     public BlobControl(BlobModel bm) {
217         // null test see BlobView
218         this.bm = bm;
219         setTitle("The Blob Analyser");
220         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
221         setLayout(new BorderLayout());
222
223         // a panel for the cells
224         cellPanel = new BlobView(bm, this);
225         add(cellPanel, BorderLayout.NORTH);
226
227         blobResult = new JLabel("--");
228         blobResult.setHorizontalAlignment(SwingConstants.CENTER); // behövs ej
229         add(blobResult, BorderLayout.CENTER);
230         // New random blob and choise list here
231
232         setVisible(true);
233         pack();
234     } // end constructor
235
236     public void actionPerformed(ActionEvent e) {
237         if (e.getSource() instanceof Cell) {
238             Cell b = (Cell) e.getSource();
239             bm.reState();
240             blobResult.setText( "" + bm.findBlob(b.getRow(), b.getCol()) );
241             cellPanel.reDraw();
242
243         }
244     }
245 }
246
247
```

```
248 Uppg 8
249 ==============================================================
250 public class BlobMain2 {
251
252     public static void main(String[] args) {
253         try {
254             if (args.length < 1) {
255                 // no args given
256                 String reply =
257                     JOptionPane.showInputDialog("Enter number of rows");
258                 int nRows = Integer.parseInt(reply);
259                 reply =
260                     JOptionPane.showInputDialog("Enter number of columns");
261                 int nCols = Integer.parseInt(reply);
262                 BlobModel bm = new BlobModel(nRows, nCols, 0.35);
263                 new BlobControl(bm);
264
265             } else if ( Character.isDigit(args[0].charAt(0)) ) {
266                 // no filename given
267                 BlobModel aGrid = new BlobModel(Integer.parseInt(args[0]),
268                                             Integer.parseInt(args[1]), 0.35 );
269                 new BlobControl(aGrid);
270
271             } else { // filename given
272                 // Create grid from a data file, 1=abnormal, 0=normal
273                 Scanner sc = new Scanner(new File(args[0]));
274                 // assuming at least 2 lines
275                 int nRows = Integer.parseInt(sc.nextLine().trim());
276                 int nCols = Integer.parseInt(sc.nextLine().trim());
277                 String line;
278                 String[] lines = new String[nRows];
279                 int i = -1;
280                 while ( sc.hasNext() ) {
281                     line = sc.nextLine();
282                     i++;
283                     lines[i] = line;
284                 }
285                 // utskrifter
286                 System.out.println("nbr of rows= " + nRows);
287                 System.out.println("nbr of cols= " + nCols);
288                 for (int j=0; j<=i; j++) {
289                     System.out.println(lines[i]);
290                 }
291             }
292         // skall vara si rätt ordning dvs Number format först
293     } catch (NumberFormatException ex) {
294             System.err.println("*****BlobControl: fel format på indata
295                                                             (parseInt)");
296             ex.printStackTrace();
297             System.exit(1);
298         } catch (IllegalArgumentException ex) {
299             System.err.println("*****BlobControl: felaktigt argument");
300             ex.printStackTrace();
301             System.exit(1);
302         } catch (FileNotFoundException ex) {
303             System.err.println("*****BlobControl: filen existerar inte");
304             ex.printStackTrace();
305             System.exit(1);
306         } catch (ArrayIndexOutOfBoundsException ex) {
307             System.err.println("*****BlobControl: AIOB, perhaps in
308                                                      parseInt(args[1] ");
309             ex.printStackTrace();
310             System.exit(1);
311         }
312
313     } // end main
314
315 }
316
```