Using JLEB

Andrew Davison

Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla, 90110, Thailand

E-mail: ad@fivedots.coe.psu.ac.th

JLEB is a Java API for programming the Dream Cheeky LED Message Board, an inexpensive desktop display which connects to your PC as a USB HID device. Figure 1 shows the board in the foreground, and Dream Cheeky's Windows software running at the back.

Figure 1. The Dream Cheeky LED Message Board.

The picture is a little misleading since only the top-half of the display contains LEDs, in a grid of 21 columns by 7 rows.

The board is sold online at http://dreamcheeky.com/led-message-board, but it's been out-of-stock for some time. However, there are plenty of other places to buy it from – Amazon sells it for around US $20.

The Windows software, and a brief technical specification, use to be available from the Dream Cheeky site, but they've recently gone offline. An alternative source for the manual is https://www.manualslib.com/manual/898427/Dream-Cheeky-Led-Message-Board.html. I couldn't find the software, but it's no great loss for my needs because it lacks an API. Instead, it's based around the creation of a list of messages, which can be sent to the board in a variety of ways, including:

- at eight speeds and three brightness levels;
- using five scrolling modes: left, right, up, down, freeze, and flash;
- with optional sound effects (in ".wav" format);
- the inclusion of up to 12 bitmap images.

Figure 2 is a screenshot of the list software, setup to deliver a bitmap, two text messages, and then another bitmap to the board.

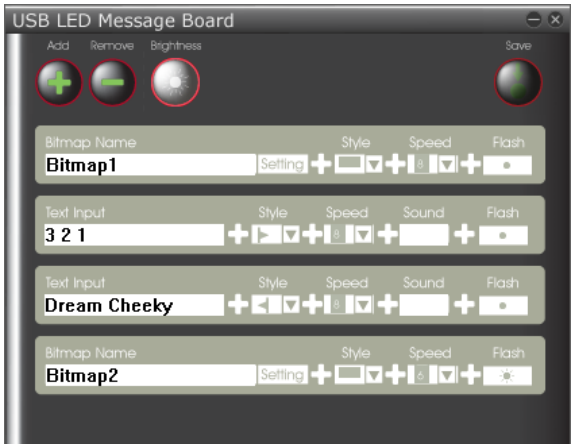

Figure 2. The Dream Cheeky Software.

Programmers are better served by choosing one of the many third-party drivers, such as:

- the Led Display Controller (https://sourceforge.net/projects/leddisplayctrl/), coded in C++ for Windows and Linux;
- dcled (http://www.last-outpost.com/~malakai/dcled/), which runs on Linux;
- LED Message Board (https://www.macupdate.com/app/mac/43250/led-message-board), which runs on OSX 10.7 or later;
- dream-cheeky-led (https://github.com/Aupajo/dream-cheeky-led), a driver coded in Ruby;
- cheekymsgboard (https://code.google.com/archive/p/cheekymsgboard/), a C# driver;
- the blog post, "Hacking the Dream Cheeky USB LED Message Board" (https://charliex2.wordpress.com/2009/11/24/hacking-the-dream-cheeky-usb-led-message-board/), by "Dr. Terrible" which explains how to use Jan Axelson's SimpleHIDWrite utility (http://janaxelson.com/hidpage.htm#tools) to communicate with the board.

Drivers coded in Java include:

- dcmsgboard4j (https://github.com/ullgren/dcmsgboard4j), which relies on libusbjava to access the board;
- jLedStripe (https://github.com/loreii/jLedStripe), which employs the javax-usb API.

Both these Java APIs utilize feature-rich USB libraries which aren't really needed for a simple device like the Dream Cheeky board. Axelson's SimpleHIDWrite utility shows that the board presents itself to the OS as a HID (Human Interface Device) which can only be sent a single kind of command string.

The big advantage of HID-based hardware is that all the major OSes (i.e. Windows, Linux, Mac) come with HID drivers. However, if the Dream Cheeky board is treated as a USB device, then a USB driver is required for it.

As a consequence, JLEB (my Dream Cheeky LED message board API) relies on javahidapi (https://github.com/torbjornvatn/javahidapi), a fork of the javahidapi library by Codeminders (https://code.google.com/p/javahidapi/). The API employs a JNI wrapper around a C/C++ HID API library, so no driver is nedded for the board.

1. The JLEB Design

JLEB lets the programmer think of the board as a low-resolution screen containing animated sprites. Only the sprites (or parts of sprites) located between 'pixels' (0,0) and (20,6) are rendered by the board. This region corresponds to the 21 columns and 7 rows of the board's LEDs, with the x-axis running to the right, and the y-axis down, as in Figure 3.

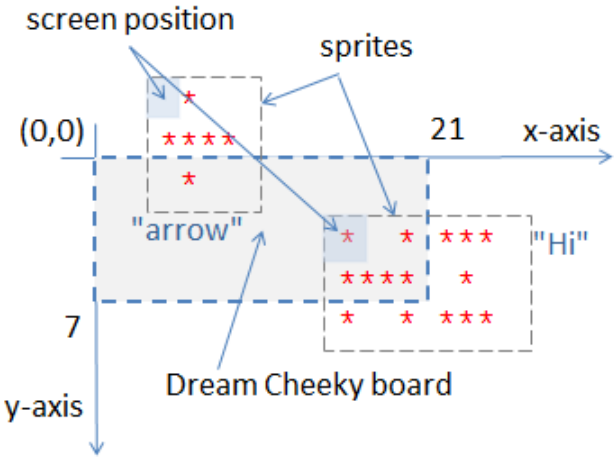

Figure 3. JLEB Screen and Sprites.

A sprite contains a 2D character array made up of '*' and ' 's, and stores a name and a position for the array's top-left corner on the screen. A '*' is drawn by turning on the LED at that character's screen position, while ' ' means that the LED is off.

It's possible to create a 'multi-picture' sprite consisting of multiple character arrays which allows the appearance of a sprite to be changed by switching between the arrays.

A sprite can be animated in three main ways: it can be moved by modifying its position, its 'picture' can be altered by switching to another array, and an array's contents can be changed.

Common animation types are available as static methods in a Utils class. These include sprite flashing, looping through the arrays in a sprite, drawing each in turn, and moving a sprite across the screen to create a scrolling effect.

There's a separate ArrUtils class for character array manipulation methods. These include functions for appending arrays, rotating an array's contents in steps of 90 degrees, and trimming blank columns from an array's left and right edges.

The JLEB examples in the next section use many of these Utils and ArrUtils methods.

JLEB employs a SpriteStore object for storing sprites. The store always includes a set of character sprites representing a font (e.g. "A", "a", "?", and all the other printable ASCII characters). Three different sized fonts are included with JLEB, and there are tools for creating more. Character sprites can be combined into sprite strings, such as the "Hi" sprite on the right of Figure 3. More sprites, such as the arrow on the left of Figure 3, can be loaded into the store as necessary.

JLEB's Screen class utilizes a lower-level HIDScreen class to control the Dream Cheeky board but, if the board isn't detected, falls back to a Swing version of the board (the LedScreen class) instead. Figure 4 shows the LedScreen display.

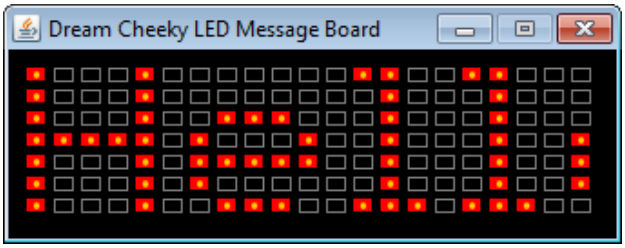

Figure 4. The LedScreen Display.

The class diagram in Figure 5 illustrates how Screen utilizes HIDScreen or LedScreen via a Board interface.

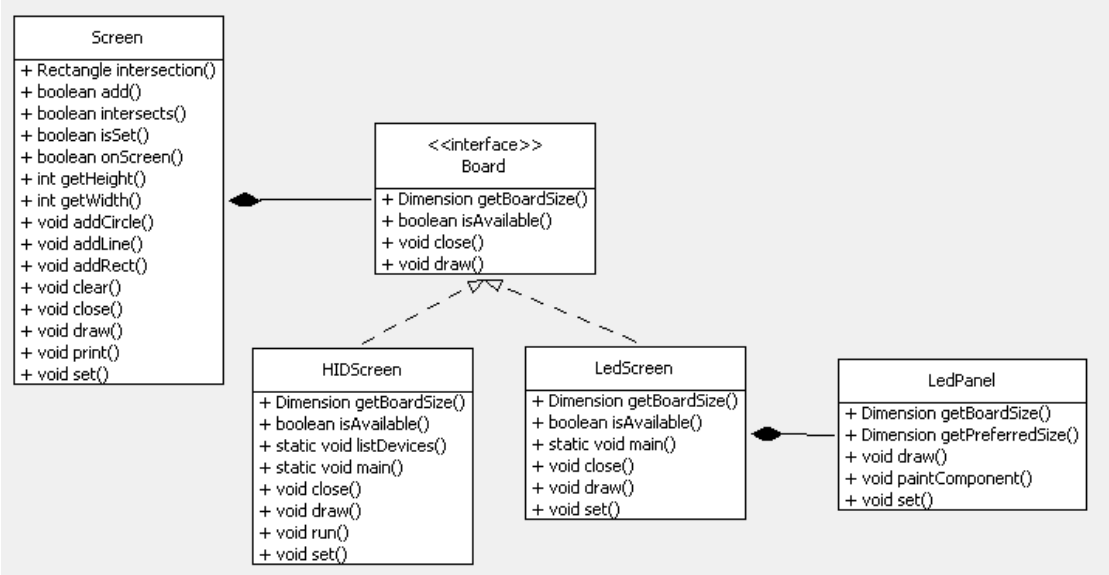

Figure 5. The Screen Class Diagrams.

The Board interface means that it should be possible to have JLEB use other types of LED matrix displays, which will be tested out in future work.

The next section explains JLEB features through examples, and is followed by a summary of the public methods in JLEB's Screen, Sprite, SpriteStore, Utils, and ArrUtils classes.

2. JLEB Examples

The ten programs in this section fall into three groups: notification examples (Hello.java, Alarm.java, Alarm2.java, and Message.java), time-related (Clock.java, Date.java, BarsClock.java, and TimeLeft.java), and animations (Walker.java and Cycle.java).

2.1. Hello.java

In this example, the message "Hello World!" scrolls across the board from right to left while a fanfare sound clip plays. Figure 6 shows the program running on a Dream Cheeky board over the course of a few seconds.

Figure 6. Hello.java Executing.

The code is:

```
public class Hello
{
  public static void main(String[] args)
  {
    Screen scr = new Screen();
    SpriteStore ss = new SpriteStore(SpriteStore.BIG);
                         // SMALL, MEDIUM, BIG, or no argument
    // ss.printNames();

    Sprite hw = ss.buildMsg("Hello World!");
    // hw.print();

    Utils.play("fanfare", 3);  // play 3 times
    Utils.scrollLeft(scr, hw);

    scr.close();
  }  // end of main()
```



```
}  // end of Hello class
```

The first two lines initialize the Screen and SpriteStore objects, and specifies that the 'BIG' font will be used. SpriteStore.buildMsg() uses this font to construct a sprite string, which is scrolled right to left across the screen by Utils.scrollLeft(). At the same time, the "fanfare" sound clip is played three times.

When the sprite has disappeared off the left edge of the screen, Utils.scrollLeft() returns and the screen is closed and the program ends.

The commented-out lines include calls to SpriteStore.printNames() and Sprite.print() which print details about the store and sprite, and are useful during debugging.

If SpriteStore is initialized with the 'SMALL' font:

```
SpriteStore ss = new SpriteStore(SpriteStore.SMALL);
```

then the sprite string appears as in Figure 7.

Figure 7. "Hello World!" in the Small Font.

Implementing Scrolling

The scrolling effect is achieved by updating the position of the sprite string inside a loop, waiting a short time between each change so the user can see the movement.

The sprite starts just off the right edge of the screen, and stops moving when it has disappeared off the left edge. These starting and ending positions are illustrated using the "Hi" sprite string in Figure 8.

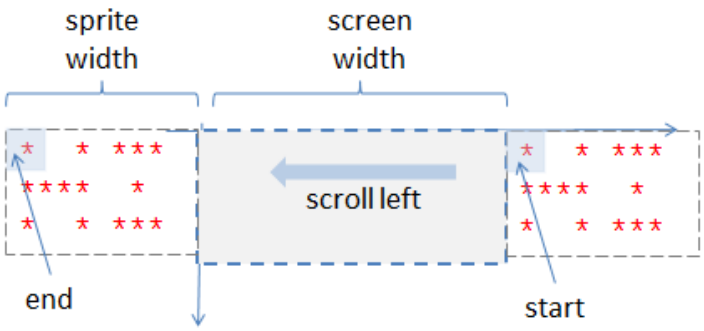

Figure 8. The Starting and Ending Positions for Scrolling Left.

These positions can be calculated from the widths of the screen and the sprite, which is done at the start of Utils.scrollLeft():

```
// in the Utils class
public static void scrollLeft(Screen scr, Sprite s)
{
  int x0 = scr.getWidth();
  int x1 = -s.getWidth();
  moveBetween(scr, s, x0, 0, x1, 0);  // move from (x0,0) to (x1,0)
}  // end of scrollLeft()
```

Utils.moveBetween() is passed references to the screen, sprite, and the (x, y) coordinates of the start and end points. It uses calcLine() to generate a list of points in a line between the two points, which are used to incrementally update the position of the sprite inside a loop:

```
// in the Utils class
public static void moveBetween(Screen scr, Sprite s,
                        int x0, int y0, int x1, int y1)
{
  ArrayList<Point> path = calcLine(x0, y0, x1, y1);

  if (path.size() == 0)
    System.out.println("No path generated");
  else {
    // move along path
    for (int i=0; i < path.size(); i++) {
      scr.clear(s);   // wipe the sprite from the screen
      s.setPos(path.get(i));
      scr.add(s);
      scr.draw();
      delay(200);
    }
  }
}  // end of moveBetween()
```

moveBetween() implements an **update-draw-delay** loop: the sprite's position is updated by calling Sprite.setPos(), draw() is called, and delay() pauses execution for a number of milliseconds.

The screen is updated in two steps: first the sprite is 'cleared' from the screen with Screen.clear(), then 'added' back to the screen at its new position with Screen.add(). However, these changes only become visible to the user when the screen is redrawn by Screen.draw(). This separation allows the screen to be updated in multiple ways before it is redrawn.

Utils.calcLine() uses the Bresenham line algorithm to generate points in a line between the two supplied points.

It is quite easy to implement other types of scrolling by calling Utils.moveBetween() with different start and end points. For example, Utils.scrollUp() moves a sprite upwards, starting below the bottom edge of the screen and finishing above the top edge. The sprite is positioned so it is in the center of the horizontal axis, as in Figure 9.

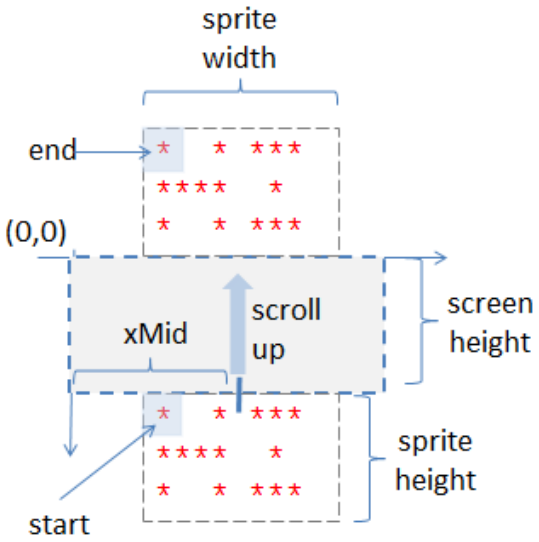

Figure 9. The Starting and Ending Positions for Scrolling Up.

The code for Utils.scrollUp():

```
// in the Utils class
public static void scrollUp(Screen scr, Sprite s)
{
  int xMid = (int) Math.round((scr.getWidth() - s.getWidth())/2);
  int y0 = scr.getHeight();
  int y1 = -s.getHeight();
  moveBetween(scr, s, xMid, y0, xMid, y1);
              // move from (xMid, y0) to (xMid,y1)
}  // end of scrollUp()
```

Sound Clips

JLEB includes the ability to play WAV sound clips, with the option of waiting for the clip to finish. Utils.play() was called in Hello.java:

```
    Utils.play("fanfare", 3);
```

This causes the fanfare.wav clip to play three times, but the function returns immediately without waiting for the playing to finish.

The PlaySound.java example employs two other sound methods:

```
// in PlaySound.java
public static void main(String[] args)
{
  ArrayList<String> fnms = Utils.getSounds();
  int i = 0;
  for(String fnm : fnms) {
    System.out.print(fnm + "  ");
    if ((++i)%5 == 0)
      System.out.println();
  }
  System.out.println();

  int index = (int)(Math.random()*fnms.size());
```



```
  System.out.println("Playing: " + fnms.get(index));
  Utils.playWait(fnms.get(index));
} // end of main()
```

Utils.getSounds() returns a list of all the WAV files currently included with JLEB, and Utils.playWait() is the waiting version of play(). The ".wav" extension can be left off the name passed to play() and playWait(), and if there's no integer argument then the clip is played just once.

2.2. Alarm.java

This example shows how additional sprites can be loaded into the sprite store, and used in an animation. It also illustrates another way of animating a sprite string, as a series of changing pictures (character arrays).

Figure 10 shows Alarm.java executing.

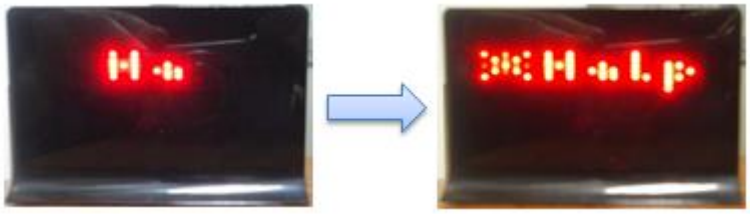

Figure 10. Alarm.java Execution.

The "Help" message reveals itself column by column, from left-to-right, and then an animated sprite appears on the left, which rapidly switches between an "X" and a "+" shape.

The code for Alarm.java:

```
// in Alarm.java
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore(SpriteStore.SMALL);
  ss.loadSprites("sprites1.txt");
  ss.printNames();

  // convert "Help" into a sprite made up of a sequence of arrays
  Sprite helpSeq = Utils.buildRight( ss.buildMsg("Help") );
  helpSeq.setX(6);  // locate sprite at (6,0)
  Utils.showSeq(scr, helpSeq);

  Utils.play("bell.wav");

  // show '+' and 'x' sprites a few times
  Sprite[] sprites = new Sprite[2];
  sprites[0] = ss.getSprite("plus");
  sprites[1] = ss.getSprite("times");
```



```
  Utils.showAll(scr, sprites, 10);

  Utils.waitEnter();  // wait for user to type <ENTER>
  scr.close();
}  // end of main()
```

At the start of Alarm.java, the sprites in sprites1.txt are loaded into the store.

sprites1.txt defines each sprite in terms of its name, the size of its character array, and the contents of that array. For example, the "house" and "envelope" sprites are:

```
house 7 7
   * *
  * **
 *   *
*     *
*  ** *
*  ** *
*  ** *

envelope 5 4
*****
*   *
* * *
*****
```

SpriteStore.printNames() is called in Alarm.java, and reports:

```
Font:
! " # $ % &
' ( ) * + ,
- . / 0 1 2
3 4 5 6 7 8
9 : ; < = >
? @ A B C D
E F G H I J
K L M N O P
Q R S T U V
W X Y Z [ \
] ^ _ ` a b
c d e f g h
i j k l m n
o p q r s sp
t u v w x y
z { | } ~

Sprites:
alarm1 alarm2 alarm3 basket bolt cross
cup downArrow downThumb envelope happy hourGlass
house leftArrow liveSpeaker plus pointFinger rightArrow
sad speaker speech star tick times
twitter upArrow upThumb wifi
```

The character sprite names are listed first, followed by the names of the sprites loaded from sprites1.txt. (Note, that the ' ' character sprite is called "sp".)

The animation of the "help" string is implemented in three lines:


```
// part of Alarm.java
Sprite helpSeq = Utils.buildRight( ss.buildMsg("Help") );
helpSeq.setX(6);  // locate sprite at (6,0)
Utils.showSeq(scr, helpSeq);
```

SpriteStore.buildMsg() creates the sprite string, which is passed to Utils.buildRight(). This creates a new sprite, helpSeq, which contains multiple pictures (character arrays) created from the "Help" character array. Most of the work is performed by ArrUtils.buildRight():

```
// in the Utils class
public static Sprite buildRight(Sprite s)
{ ArrayList<char[][]> pics = ArrUtils.buildRight(s.getCharArray());
  return new Sprite(pics);
}
```

ArrUtils.buildRight() creates a list of new character arrays by iterating over the supplied array copying increasing numbers of columns into each array:

```
// in the ArrUtils class
public static ArrayList<char [][]> buildRight(char[][] charArr)
{
  if (charArr == null) {
    System.out.println("Array is null");
    return null;
  }

  int width = charArr[0].length;
  int height = charArr.length;
  ArrayList<char [][]> pics = new ArrayList<>();

  for (int colEnd = 0; colEnd < width; colEnd++) {
    char[][] out = new char[height][width];
    clear(out);         // copying columns into out[][]
    for (int r = 0; r < height; r++) {
      for (int c = 0; c <= colEnd; c++)
        out[r][c] = charArr[r][c];
    }
    pics.add(out);   // add out[][] to list
  }

  return pics;
}  // end of buildRight()
```

Back in Alarm.java, the multi-picture sprite is positioned at (6,0) on the screen, and then the sprite's pictures are each displayed in turn by Utils.showSeq():

```
// in the Utils class
public static void showSeq(Screen scr, Sprite s)
{  loop(scr, s, 1);  }


public static void loop(Screen scr, Sprite s, int numTimes)
// cycle through all the sprite's pics a number of times
{
```



```
  int numPics = s.numPics();
  for(int i=0; i < (numTimes * numPics); i++) {
    scr.clear(s);
    s.toPic(i%numPics);
    scr.add(s);
    scr.draw();
    delay(200);
  }
}  // end of loop()
```

showSeq() is a wrapper around Utils.loop() which utilizes the same kind of update-draw-delay loop as seen earlier in Utils.moveBetween(). The update clears the sprite from the screen, changes its picture by calling Sprite.toPic(), and then calls add() to add that new picture to the screen. Sprite.toPic() uses an index value to refer to a particular picture, and employs the modulo of the number of pictures to cycle through them.

Alarm.java shows another way of cycling through pictures, by building an array of sprites, and calling Util.showAll():

```
// part of Alarm,java
Sprite[] sprites = new Sprite[2];
sprites[0] = ss.getSprite("plus");
sprites[1] = ss.getSprite("times");
Utils.showAll(scr, sprites, 10);
```

The "plus" and "times" sprites are loaded from the store into an array, which is passed to Utils.showAll():

```
// in the Utils class
public static void showAll(Screen scr,
                      Sprite[] sprites, int numTimes)
{ for (int i=0; i < numTimes; i++) {
    for (int j=0; j < sprites.length; j++) {
      scr.add(sprites[j]);
      scr.draw();
      Utils.delay(200);
      scr.clear(sprites[j]);
    }
  }
}  // end of showAll()
```

This function employs a slight variation of the update-draw-delay loop, where the clearing of the old sprite is moved to after the delay.

2.3. Alarm2.java

Alarm2 demonstrates another animation using an array of sprites, shows how a multi-picture sprite can be loaded and used, illustrates one of the screen drawing functions, and sprite flashing. Figure 11 has several shots of the Alarm example.

Figure 11. The Alarm2.java Execution.

The code for Alarm2.java:

```
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore();   // load BIG font by default
  ss.loadSprites("sprites1.txt");
  ss.loadMultiSprite("arrows.txt");
  ss.printNames();

  Sprite[] alarms = new Sprite[3];
  alarms[0] = ss.getSprite("alarm1");
  alarms[1] = ss.getSprite("alarm2");
  alarms[2] = ss.getSprite("alarm3");

  Utils.play("romans", 3);

  // iterate through the alarm sprites a few times
  Utils.showAll(scr, alarms, 9);

  // loop through the arrows multi-pic sprite
  Sprite arrows = ss.getSprite("arrows");
  arrows.setPos(9, 1);  // place at (9,1)
  Utils.loop(scr, arrows, 5);

  // draw a rectangle around the cleared display
  scr.clear();
  scr.addRect(0,0, scr.getWidth()-1, scr.getHeight()-1);
  scr.draw();
  Utils.delay(1000);

  // flash an 'N' in the center
  Sprite n = ss.getSprite('N');
  // n.print();
  n.setPos(Sprite.CENTER, scr.getWidth()/2, scr.getHeight()/2);
  Utils.flash(scr, n, 5);

  // leave just the 'N' in place
  scr.clear();
  scr.add(n);
  scr.draw();

  Utils.waitEnter();
  scr.close();
}  // end of main()
```


SpriteStore.loadMultiSprite() loads a multi-picture sprite. In this case, "arrows.txt" contains four images (character arrays) representing an arrow pointing in different directions:

```
arrows 5 5

down
  *
  *
  *
 ***
  *

left

 *
*****
 *

up
  *
 ***
  *
  *
  *


right

   *
*****
   *
```

The sprite's name is "arrow" and each array also has a name. All the pictures must be the same size: 5 x 5 in this case.

The left-most sprite in Figure 11 is animated using the same approach as in Alarm.java. Three different alarm sprites are loaded into an array, and cycled through nine times while a music clip is played three times:

```
// part of Alarm2.java
Sprite[] alarms = new Sprite[3];
alarms[0] = ss.getSprite("alarm1");
alarms[1] = ss.getSprite("alarm2");
alarms[2] = ss.getSprite("alarm3");

Utils.play("romans", 3);
// iterate through the alarm sprites a few times
Utils.showAll(scr, alarms, 9);
```

The multi-picture sprite, arrows, requires less work to animate, since the sprite contains all the pictures, and so there's no need for a Sprite array. The sprite is 'rotated' by being passed to Utils.loop() which was described above:

```
// part of Alarm2.java
Sprite arrows = ss.getSprite("arrows");
```



```
arrows.setPos(9, 1);  // place at (9,1)
Utils.loop(scr, arrows, 5);
```

The Screen class includes functions for drawing lines, rectangles, points, and circles. A border is added to the screen as a large rectangle:

```
// part of Alarm2.java
scr.clear();
scr.addRect(0,0, scr.getWidth()-1, scr.getHeight()-1);
scr.draw();
```

The example ends by adding a large "N" to the center of the screen, which flashes on and off five times:

```
// part of Alarm2.java
Sprite n = ss.getSprite('N');
n.setPos(Sprite.CENTER, scr.getWidth()/2, scr.getHeight()/2);
Utils.flash(scr, n, 5);
```

Usually a sprite is positioned in terms of its top-left corner, but it's also possible to use a 'compass' constant to specify a different position. The various constants are shown in Figure 12.

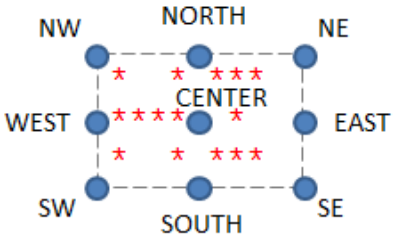

Figure 12. The Compass Positions for a Sprite.

In the code above, the center of the "N" sprite is assigned to the center of the screen.

Utils.flash() uses another variation of update-draw-delay, where the sprite is drawn for 500 ms, then deleted for 200 ms, before repeating:

```
// in the Utils class
public static void flash(Screen scr, Sprite s, int numTimes)
{
  for (int i =0; i < numTimes; i++) {
    scr.add(s);
    scr.draw();
    delay(500);

    scr.clear(s);  // only this sprite is wiped from the screen
    scr.draw();
    delay(200);
  }
}  // end of flash()
```

2.4. Message.java

Message.java shows a flashing envelope sprite in the center of the screen with two arrows that move towards it from the left and right edges. This example illustrates yet another way of animating, by applying an inverse transformation to the character array inside the sprite. Figure 13 shows a few shots of the program executing.

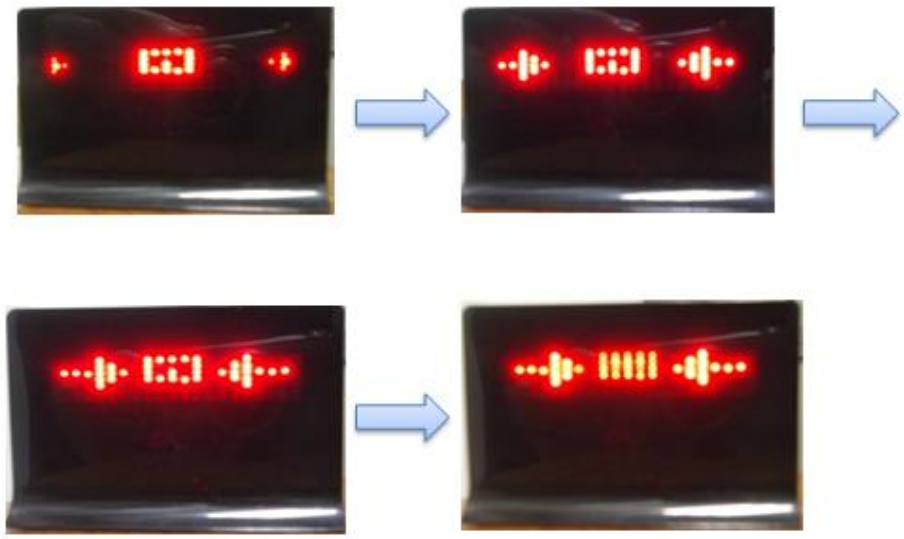

Figure 13. Message.java Executing.

The code for Message.java:

```
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore(SpriteStore.SMALL);
  ss.loadSprites("sprites1.txt");
  ss.printNames();

  // left arrow off left edge
  Sprite rightArrow = ss.getSprite("rightArrow");  // -->
  rightArrow.setPos(Sprite.NE, -1, 0);

  // envelope in center
  Sprite envelope = ss.getSprite("envelope");
  envelope.setPos(Sprite.NORTH, scr.getWidth()/2, 0);

  // right arrow off right edge
  Sprite leftArrow = ss.getSprite("leftArrow");    // <--
  leftArrow.setPos(Sprite.NW, scr.getWidth()+1, 0);

  scr.add(envelope);
  scr.draw();
  Utils.play("notify");

  // move arrows in towards the envelope
  for (int i=0; i < 8; i++) {
```



```
    scr.clear();
    rightArrow.xStep(1);
    leftArrow.xStep(-1);
    scr.add(leftArrow);
    scr.add(rightArrow);
    scr.add(envelope);
    scr.draw();
    Utils.delay(200);
  }

  ArrUtils.printTransforms();

  // 'flash' the envelope by inverting its picture
  Utils.flashInvert(scr, envelope, 5);

  Utils.waitEnter();
  scr.close();
}  // end of main()
```

The code begins by placing the "rightArrow", "envelope", and "leftArrow" sprites in their starting positions. Inside the loop, the arrows are moved by calling Sprite.xStep().

ArrUtils.printTransforms() lists the character array transformation functions in ArrUtils. Currently, the output is:

```
Transformation names:
  copy  flipHoriz  flipVert  invert
  rotLeft  rotRight  trim
```

Utils.flashInvert() utilizes the "invert" transformation:

```
// in the Utils class
public static void flashInvert(Screen scr, Sprite s, int num)
{
  int numTimes = num*2;
  // s.backup();
  for(int i=0; i < numTimes; i++) {
    scr.clear(s);
    s.transform("invert");
    scr.add(s);
    scr.draw();
    Utils.delay(200);
  }
  // s.restore();
}  // end of flashInvert()
```

A transformation changes the contents of the sprite's character array, and so it's sometimes useful to call Sprite.backup() beforehand so that Sprite.restore() can be used to restore the original array at the end. That isn't needed here since inversion is called an even number of times which leaves the array as it started.

2.5. Clock.java

Clock.java displays the current time, as shown in Figure 14, which changes every minute.

Figure 14. Clock.java Executing.

This example differs from earlier ones in that it must run continuously, ideally without causing the calling program to wait forever. Implementing the clock as an infinite update-draw-delay loop isn't sufficient; the loop must also be embedded in a thread. This coding pattern is useful for any long-lived display application.

The Clock class extends Thread so that the update-draw-delay loop can be executed inside the thread's run() method. The loop will continue so long as an isRunning boolean is true. The code for the class:

```
public class Clock extends Thread
{
  private static final int REFRESH_DELAY = 1000 * 20;
                                          // 20 secs in ms
  private volatile boolean isRunning = false;
                              // used to stop the thread
  private Screen scr;
  private SpriteStore ss;


  public Clock(Screen scr, SpriteStore ss)
  { this.scr = scr;
    this.ss = ss;
    this.start();
  }  // end of Clock


  public void run()
  {
    isRunning = true;
    while(isRunning) {
      scr.clear();
      scr.add( ss.buildMsg(getTime()) );
      scr.draw();
      Utils.delay(REFRESH_DELAY);
    }
  } // end of run()


  public void end()   // end the update-draw-delay loop
```



```
  {  isRunning = false;   }


  private String getTime()
  // return time as a formatted string: <hours> : <mins>
  {
    Calendar cal = new GregorianCalendar();
    int hours = cal.get(Calendar.HOUR); // 0..11
    int mins = cal.get(Calendar.MINUTE); // 0..59
    return String.format("%d:%02d", hours, mins);
  }  // end of getTime()

}  // end of Clock class
```

A test-rig for executing the Clock class:

```
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore(SpriteStore.MEDIUM);
         // font must be small or medium so that all the
         // time text will be visible at once
  Clock c = new Clock(scr, ss);

  Utils.waitEnter(); // wait a while before killing the clock...
  c.end();
  scr.close();
}  // end of main()
```

The Screen and SpriteStore objects are passed to the Clock class' constructor which starts the thread running.

The run() method in this kind of application will always have the general form:

```
public void run()
{
  isRunning = true;
  while(isRunning) {
    // update the screen

    scr.draw();
    Utils.delay(REFRESH_DELAY);
  }
} // end of run()
```

Clock updating clears the screen. then adds the current time as a sprite string:

```
// update part of Clock.run()
scr.clear();
scr.add( ss.buildMsg(getTime()) );
```

getTime() is a standard bit of Java which returns the current hour and minutes as a string.

Note that isRunning is declared as volatile since its value can be changed by a call to Clock.isEnd() which is executed outside the running thread.

Utils.delay() uses Java's sleep() function which isn't all that accurate. To avoid any errors, delay() sleeps for 20 seconds at a time rather than in 1 minute steps. Also when isRunning is set to false, execution may be paused inside delay() which will mean that the thread won't terminate for at most REFRESH_DELAY milliseconds. It's often better to make the delay period smaller to reduce this wait time.

2.6. BarsClock.java

This example was inspired by the unusual Matrix M6001 wrist watch, which displays the time as a series of bars rather than using hours and minutes hands (e.g. see https://www.engadget.com/2008/01/18/matrix-m6001-watch-uses-bars-not-hands/). The Dream Cheeky version of it is shown in Figure 15.

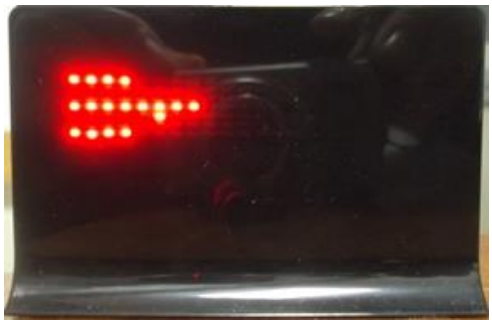

Figure 15. BarsClock.java Executing.

Figure 15 shows the same time as Figure 14 (4.44pm). The three lines represent the number of hours (4), the number of 5-minute intervals (8), and the last line is for any extra minutes after the last 5-minute interval (4). The dot below the 5-minutes line marks 30 minutes (i.e. six 5-minute intervals), and the hours line will also have added dots after 5 and 10 hours have passed.

BarsClock is almost identical to the earlier Clock class: it subclasses Thread, and executes an update-draw-delay loop inside run(). It differs in the details of how the screen is updated:

```
// run() inside BarsClock.java
public void run()
{
  isRunning = true;
  while(isRunning) {
    scr.clear();
    addTime(scr);
    scr.draw();
    Utils.delay(REFRESH_DELAY);
  }
} // end of run()
```


addTime() get the current time in hours and minutes, and converts the minutes value into two integers representing the number of 5-minute intervals, and the remaining number of minutes.

Three lines are added to the screen using the three integers to calculate their lengths; dots may be included to make it easier to visually judge the length of long lines.

```
private void addTime(Screen scr)
{
  Calendar cal = new GregorianCalendar();
  int hours = cal.get(Calendar.HOUR); // 0..11
  int mins = cal.get(Calendar.MINUTE); // 0..59
  int num5s = mins/5;
  int units = mins - (5*num5s);

  // draw hours line
  scr.addLine(0,0, hours-1, 0);
  if (hours > 4)
    scr.set(4,1, true);    // to mark 5 hours
  if (hours > 9)
    scr.set(9,1, true);    // to mark 10 hours

  // draw 5 mins line
  if (num5s > 0)
    scr.addLine(0,3, num5s-1, 3);
  if (num5s > 6)         // to mark 6 lots of 5 mins i.e. 30 mins
    scr.set(5,4, true);

  if (units > 0)     // draw units line
    scr.addLine(0,6, units-1, 6);
}  // end of addTime()
```

The arguments passed to Screem.addLine() are the starting and ending (x, y) coordinates of the line. Screen.set() is passed the coordinate of a point, and true or false to indicate whether the point should be drawn (i.e. LED lit) or not drawn (LED off).

2.7. TimeLeft.java

TimeLeft.java displays the number of seconds left between now and a time entered on the command line when the program is called. When the time left reaches 0, the message "Time!!" is displayed, and a sound played. Figure 15 shows the display counting down.

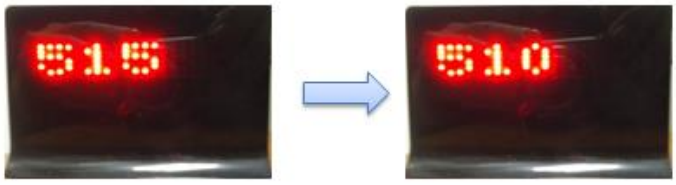

Figure 15. TimeLeft.java Executing.

The main() function passes the deadline time string to the TimeLeft constructor, which utilizes Java 8's much improved time classes to record the deadline and calculate the time remaining:

```
// globals
private Screen scr;
private SpriteStore ss;
private LocalTime deadline;


public TimeLeft(Screen scr, SpriteStore ss, String deadlineStr)
{
  this.scr = scr;
  this.ss = ss;

  try {
    deadline = LocalTime.parse(deadlineStr);
    this.start();
  }
  catch(DateTimeParseException e)
  {  System.out.println("Incorrect time form; use 24-hour HH:MM"); }
}  // end of TimeLeft()
```

run() can terminate in two ways: either the isRunning boolean can be set to false, as in earlier examples, or the deadline can be reached:

```
// globals
private static final int REFRESH_DELAY = 1000;  // 1 sec in ms
private volatile boolean isRunning = false;


public void run()
{
  isRunning = true;
  while(isRunning) {
    long secs = secsLeft();
    if (secs <= 0)  // deadline reached; might be negative
      break;

    scr.clear();
    scr.add(ss.buildMsg("" + secs));
    scr.draw();
    Utils.delay(REFRESH_DELAY);
  }

  // time has run out
  Utils.play("buzz", 4);
  scr.clear();
  scr.add(ss.buildMsg("Time!!"));
  scr.draw();
} // end of run()
```

The secsLeft() function calculates the time in seconds between now and the deadline:

```
private long secsLeft()
```



```
// return seconds left until the deadline
{
  LocalTime now = LocalTime.now();
  return now.until(deadline, ChronoUnit.SECONDS);
}
```

Back in run(), the time left is converted to a sprite string and added to the screen:

```
scr.add(ss.buildMsg("" + secs));
```

When the update-draw-delay loop ends, the run() method draws "Time!!" to the screen and emits a buzzing sound.

2.8. Date.java

Date.java displays the date once, then exits, and so there's no need to use a thread. The program uses a standard bit of Java to obtain the date as a string, and calls Utils.scrollLeft() to display it as in Figure 16.

Figure 16. Date.java Executing

Utils.scrollLeft() is utilized because the date's length is wider than the board. The code:

```
// in Date.java
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore(SpriteStore.MEDIUM);

  Sprite today = ss.buildMsg(getDate());
  // today.print();   // for debugging
  Utils.play("gong");
  Utils.scrollLeft(scr, today);
  scr.close();
} // end of main()


private static String getDate()
// return today's date as a formatted string
{
  Calendar cal = Calendar.getInstance();
  Format formatter = new SimpleDateFormat("EEE, d MMM yyyy");
  return formatter.format(cal.getTime());
}  // end of getDate()
```


When the call to today.print() is uncommented, the following is printed:

```
Pos: (0, 0)
Pic nm: "Wed, 19 Apr 2017"
Size (w x h): 66 x 7
----------------------------------------------------------------------
|*  *          *         *   **      **                 **   *   *  ****|
|*  *          *        **  *  *    *  *                *  * * * **     *|
|*  *  **   ***          *  *  *    *  * ***  ***        * * *  *     * |
|**** * ** *  *          *   ***    **** *  * *  *      *  * *  *     * |
|**** **   *  *  **      *     *    *  * *  * *        *   * *  *   *   |
|*  *  **   ***  *     ***  **     *  * ***  *       ****  *  ***  *   |
|               *                     *                               |
----------------------------------------------------------------------
```

2.9. Walker.java

Walker.java is another multi-picture sprite example, this time of a 'stick man' who walks left to right across the screen, passing in front of a house. Figure 17 shows the program executing.

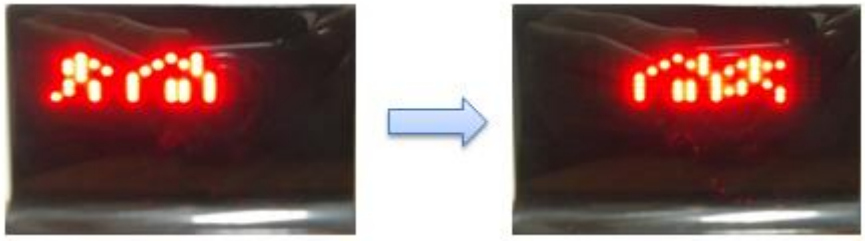

Figure 17. Walker.java Executing.

The Walker.java code:

```
public static void main(String[] args)
{
  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore();
  ss.loadSprites("sprites1.txt");
  ss.loadMultiSprite("stickWalk.txt");  // stick-man multi-pic
  // ss.printNames();

  Sprite house = ss.getSprite("house");
  house.setX(6);   // located at (6,0)

  Sprite walker = ss.getSprite("walker");
  int startPos = -walker.getWidth();
  walker.setX(startPos);  // start off left side of screen

  int no = 0;  // the picture num.
  while(true) {   // man never finishes walking
    scr.clear();
    scr.add(house);
```



```
    walker.xStep(1);      // move right
    walker.toPic(no);     // switch to picture num.
    boolean isVis = scr.add(walker);
    no = (no+1) % walker.numPics();  // next picture num.

    scr.draw();
    if (!isVis) {   // has walker left the board?
      Utils.play("teleport.wav");
      walker.setX(startPos);   // reset to starting position
    }
    Utils.delay(200);
  }
}  // end of main()
```

SpriteStore.loadMultiSprite() loads the definition of the multi-picture stick man. "stickWalk.txt" contains eight character arrays for different walking positions, starting with:

```
walker 5 7

w0
  *
 ***
* * *
  *
 * *
 *  *
*  *

w1
  *
 ***
 ***
  *
 * *
** *
  *
```

As with the earlier "arrows.txt" multi-picture file, each array has a name, and are the same size (in this case, 5 columns by 7 rows).

Walker.java animates the stick man in two ways – its cycles through the multiple pictures so that the man's limbs appear to move, and also translates the sprite across the screen.

When the man has left the right side of the screen, a sound clip is played, and the sprite is reset to just off the left edge of the screen to repeat its journey past the house. The reset is triggered by examining the result of Screen.add() which returns false if the sprite isn't added to a visible part of the screen.

The update-draw-delay loop never terminates, so the user must type ctrl-c at the command line to stop the program.

2.10. Cycle.java

Cycle.java is a shorter example of multi-picture sprites. A multi-picture sprite name is supplied on the command line, and its corresponding text file is loaded. The sprite is animated by cycling through all of its pictures five times. Figure 18 shows screenshots of the execution of the "pattern" sprite.

Figure 18. Cycle.java Animating "pattern".

The Cycles.java code:

```
public static void main(String[] args)
{
  if (args.length != 1) {
    System.out.println("Usage: run Cycle <multi-pic sprite-name>");
    return;
  }

  Screen scr = new Screen();
  SpriteStore ss = new SpriteStore();
  ss.loadMultiSprite(args[0]+".txt");  // load sprite's text file

  Sprite s = ss.getSprite(args[0]);
  Utils.loop(scr, s, 5);

  Utils.waitEnter();
  scr.close();
}  // end of main()
```

3. JLEB Classes

This section describes the public methods and constants for the JLEB classes: Screen, Sprite, SpriteStore, Utils, and ArrUtils. Examples of how these methods are used were presented in the previous section.

3.1. The Screen Class

The public methods are divided into six groups, listed in the following tables:

- Table 1. Basic Screen Methods.
- Table 2. Screen Information.
- Table 3. Screen Testing.
- Table 4. Screen Clearing.
- Table 5. Adding Sprites/Arrays/Points to the Screen.

- Table 6. Adding Shapes to the Screen.

Method	Purpose
Screen();	Constructor; create a Screen object which may refer to a Swing window or to the Dream Cheeky LED message board.
void draw();	Draw the contents of the screen to the Swing window or the message board.
void close();	Close the connection to the Swing window or message board.

Table 1. Basic Screen Methods.

Method	Purpose
int getWidth();	Return the width of the Swing window or message board.
int getHeight();	Return the height of the Swing window or message board.
void print();	Print the contents of Swing window or message board. A lit LED is represented by '*', an unlit LED by ' '. A border is placed around the screen using '-' and '|'.

Table 2. Screen Information.

Method	Purpose
Rectangle intersection(Sprite s);	Return the intersection rectangle of the sprite with the screen's drawing area. (May be null.)
boolean intersects(Sprite s);	Does the sprite intersect the drawing area of the screen? In other words, is the sprite visible on the board?
boolean onScreen(Point p);	Is the coordinate p inside the drawing area of the screen? In other words, is the point visible on the board?
boolean onScreen(int x, int y);	Is the coordinate (x, y) inside the drawing area of the screen?

Table 3. Screen Testing.

Method	Purpose
void clear();	Clear the screen.
void clear(int x, int y, int w, int h);	Clear the screen rectangle defined by the top-left coordinate (x, y) and width w and height h.
void clear(Rectangle r);	Clear the screen rectangle defined by r.

void clear(Sprite s);	Clear the screen rectangle define by the current position of the sprite, and its width and height.

Table 4. Screen Clearing.

Method	Purpose
boolean add(Sprite s);	Add a sprite to the screen. Return true if some or all of the sprite is in the drawing area of the screen, false otherwise. This method does not draw the screen; for that use Screen.draw().
boolean add(int x, int y, char[][] chArr);	Add the character array to the screen with its top-left corner located at (x, y). Return true if some or all of the array is in the drawing area of the screen, false otherwise. This method does not draw the screen; for that use Screen.draw().
void set(int x, int y, boolean isOn);	Depending on the boolean argument either turn on or off the LED located at (x, y). This method does not draw the screen; for that use Screen.draw().
boolean isSet(int x, int y);	Is the LED at (x, y) turned on?

Table 5. Adding Sprites/Arrays/Points to the Screen.

Method	Purpose
void addLine(int x0, int y0, int x1, int y1);	Add a line to the screen defined by the end-points (x0, y0) and (x1, y1). This method does not draw the screen; for that use Screen.draw().
void addRect(int x, int y, int w, int h);	Add an unfilled rectangle to the screen defined by the top-left corner (x, y), width w, and height h. This method does not draw the screen; for that use Screen.draw().
void addCircle(int x, int y, int radius);	Add an unfilled circle to the screen centered at (x, y), with the specified radius. This method does not draw the screen; for that use Screen.draw().

Table 6. Adding Shapes to the Screen.

3.2. The Sprite Class

The Sprite class has nine public constants: NORTH, NE, EAST, SE, SOUTH, SW, WEST, NW, and CENTER, which refer to different compass points on the sprite's picture (see Figure 12).

The public methods are divided into three groups, listed in the following tables:

- Table 7. Sprite Creation.
- Table 8. Position Methods.
- Table 9. Picture Methods.

Method	Purpose
Sprite(char[][] charArr);	Constructor; create a sprite using the character array. It will have the default name "noname$$", and its top-left corner will be at (0, 0).
Sprite(String name, char[][] charArr);	Constructor; create a sprite called name using the character array. Its top-left corner will be at (0, 0).
Sprite(ArrayList<char[][]> charArrs);	Constructor; create a sprite using the character array list. The arrays will be called "noname$$" followed by a number starting at 0. The sprite will use the first array as its default picture, and its top-left corner will be at (0, 0).
Sprite(ArrayList<String> names, ArrayList<char[][]> charArrs);	Constructor; create a sprite using the named character arrays in the list. The sprite will use the first array as its default picture, and its top-left corner will be at (0, 0).
void print();	Print details about the sprite including its current position, picture name, and character array.

Table 7. Sprite Creation.

Method	Purpose
int getX();	Get the x-axis position of the top-left corner of the sprite.
int getY();	Get the y-axis position of the top-left corner of the sprite.
Point getPos();	Get the (x, y) coordinate of the top-left corner of the sprite.
Rectangle getRect();	Get the sprite's rectangle, defined by its top-left corner, and its width and height.
void setX(int x);	Set the x-axis position of the top-left corner of the sprite.

void setY(int y);	Set the y-axis position of the top-left corner of the sprite.
void setPos(int x, int y);	Set the coordinate of the top-left corner of the sprite to (x, y)
void setPos(Point p);	Set the coordinate of the top-left corner of the sprite to p.
void setPos(int compass, int x, int y);	Set the coordinate of the compass point of the sprite to (x, y). For a list of the compass point constants see Figure 12.
void xStep(int xStep);	Move the sprite by adding xStep to its x-axis position.
void yStep(int yStep);	Move the sprite by adding yStep to its y-axis position.

Table 8. Position Methods.

Method	Purpose
void addPic(String name, char[][] charArr);	Add the specified named picture (character array) to the sprite.
int getCurrPicNo();	Return the index of the current picture being used by the sprite.
int numPics();	Return the number of pictures in the sprite.
String getNames();	Return a single string containing all the names of the pictures in the sprite, separated by spaces.
String getName();	Return the name of the current picture being used by the sprite.
String getName(int i);	Return the name of the picture with index number i.
int getWidth();	Return the width of the current picture being used by the sprite.
int getHeight();	Return the height of the current picture being used by the sprite.
char[][] getCharArray();	Return a copy of the current picture being used by the sprite.
char[][] getCharArray(int i);	Return a copy of the picture with index number i.
void backup();	Make a backup copy of the current picture being used by the sprite.
void restore();	Assign the backup copy to the current picture being used by the sprite.
void appendPic(Sprite s);	Append the picture (character array) in sprite s to the end of this sprite's current picture, with a blank column between them.
void trim();	Remove the starting and ending blank columns from the sprite's current picture (character array).
char[][] getRegion(Rectangle	Return a copy of the character array covered by the

rect);	rectangle, or null.
void transform(String methodNm);	Transform the current picture (character array).The transformation names are: copy, invert, rotRight, rotLeft, flipVert, flipHoriz, and trim.
boolean toPic(String nm);	Change the current picture used by the sprite based on its name.
boolean toPic(int i);	Change the current picture used by the sprite based on its index.
boolean nextPic();	Change to the next picture, if one exists.

Table 9. Picture Methods.

3.3. The SpriteStore Class

The sprite store maintains two maps of name-sprite pairs, a font map for character sprites, and a sprites map for other sprites.

There are three public constants: BIG, MEDIUM, and SMALL, which refer to different sized fonts.

The public methods are divided into four groups, listed in the following tables:

- Table 10. SpriteStore Creation.
- Table 11. SpriteStore Printing.
- Table 12. Sprite Retrieval.
- Table 13. Sprite Loading.

Method	Purpose
SpriteStore();	Create a sprite store with the BIG font loaded into the font map, and no sprites in the sprites map.
SpriteStore(int fontType);	Create a sprite store with the specified font loaded into the font map, and no sprites in the sprites map.
SpriteStore(String fnm);	Create a sprite store with the font loaded from the specified text file into the font map. The format of the text file is descibed in section 5.1. There are no sprites in the sprites map

Table 10. SpriteStore Creation.

Method	Purpose
void printNames();	Print the names of all the characters in the font map, and all the sprites in the sprite

	map.
void print();	Print the names *and* character arrays of all the characters in the font map, and all the sprites in the sprite map. This will generate a lot of output.

Table 11. SpriteStore Printing.

Method	Purpose
Sprite getSprite(String nm);	Retrieve the named sprite from the sprites map.
Sprite getSprite(char ch);	Retrieve the named character sprite from the font map.
Sprite buildMsg(String s);	Create a new sprite by combining the arrays for all the characters in the supplied string. The combination ensures that there's a blank column between each character.
Sprite buildMsg(String[] spriteNms);	Create a new sprite by combining the pictures for all the sprites named in the array. The combination ensures that there's a blank column between each sprite's picture.

Table 12. Sprite Retrieval.

Method	Purpose
void loadSprites(String fnm);	Load sprites from a file into the sprites map. The file format is explained in section 5.2.
void loadMultiSprite(String fnm);	Load a single multi-picture sprite from a file into the sprites map. The file format is explained in section 5.2.

Table 13. Sprite Loading.

3.4. The Utils Class

The Utils class contains static methods for sprites and sounds. The utility methods for character arrays are in ArrUtils.java (see section 3.5).

The public methods are divided into six groups, listed in the following tables:

- Table 14. Sprite Effects.
- Table 15. Multi-picture Sprite Effects.
- Table 16. Animation by Changing Position.
- Table 17. Generating Points in a Shape.

- Table 18. Sound Methods.
- Table 19. Assorted.

Method	Purpose
static void flash(Screen scr, Sprite s, int numTimes);	Make this sprite appear/disappear a number of times.
static void flashInvert(Screen scr, Sprite s, int numTimes);	Make this sprite flash by inverting its picture a number of times.
static void showAll(Screen scr, Sprite[] sprites, int numTimes);	Display each sprite in the array in turn, repeating the specified number of times.

.Table 14. Sprite Effects.

Method	Purpose
static void loop(Screen scr, Sprite s, int numTimes);	Cycle through the sprite's pictures a specified number of times
static void showSeq(Screen scr, Sprite s);	Cycle through the sprite's pictures once.
static Sprite buildUp(Sprite s);	Create a multi-picture sprite by generating pictures from the character array in the supplied sprite. Each picture is a slice of the array by increasing the number of rows in the slice counting from the bottom of the array.
static Sprite buildRight(Sprite s);	Create a multi-picture sprite by generating pictures from the character array in the supplied sprite. Each picture is a slice of the array by increasing the number of columns in the slice counting from the left of the array.

.Table 15. Multi-picture Sprite Effects.

Method	Purpose
static void scrollLeft(Screen scr, Sprite s);	Move the sprite right to left, starting beyond the right edge and finishing beyond the left edge.
static void scrollUp(Screen scr, Sprite s);	Move the sprite upwards, starting below the bottom edge and finishing above the top edge. The sprite is positioned in the middle of the x-axis.

static void moveBetween(Screen scr, Sprite s, int x0, int y0, int x1, int y1);	Generate a list of points in a line between (x0,y0) and (x1,y1), which are used to change the position of the sprite incrementally.

Table 16. Animation by Changing Position.

Method	Purpose
static ArrayList<Point> calcLine(int x0, int y0, int x1, int y1);	Return a list of points making up a line connecting (x0,y0) to (x1,y1). Uses the Bresenham line algorithm.
static ArrayList<Point> calcRect(int x, int y, int w, int h);	Return a list of points making up the four sides of the rectangle whose top-left corner is at (x, y), and has width w and height h.
static ArrayList<Point> calcCircle(int x, int y, int r);	Return a list of points making up the circumference of the circle centered at (x, y) with radius r.

Table 17. Generate Points in a Shape.

Method	Purpose
static ArrayList<String> getSounds();	Return a list of the ".wav" filenames stored in the sounds/ directory inside the JLEB JAR.
static void play(final String nm);	Play the named sound *without waiting* for it to finish. The name can include a ".wav" extension or not, and the clip will be loaded from the JLEB JAR.
static void play(final String nm, final int numTimes);	Play the named sound the specified number of times *without waiting* for it to finish. The name can include a ".wav" extension or not, and the clip will be loaded from the JLEB JAR.
static void playWait(String nm);	Play the named sound *and wait* for it to finish. The name can include a ".wav" extension or not, and the clip will be loaded from the JLEB JAR.
static void playWait(String nm, int numTimes);	Play the named sound the specified number of times *and wait* for it to finish. The name can include a ".wav" extension or not, and the clip will be loaded from

	the JLEB JAR.

Table 18. Sound Methods.

Method	Purpose
static void delay(int ms);	Delay execution for the specified number of milliseconds.
static void waitEnter();	Delay execution until the use presses <enter>.

Table 19. Assorted.

3.5. The ArrUtils Class

ArrUtils contains static methods for manipulating character arrays. Methods for sprites and sounds are in Utils.java (see section 3.4 above).

The public methods are divided into seven groups, listed in the following tables:

- Table 20. Create Multiple Arrays from One Array.
- Table 21. Transform a Character Array.
- Table 22. Clear a Character Array.
- Table 23. Character Array Containment.
- Table 24. Extend a Character Array.
- Table 25. Reduce a Character Array.
- Table 26. Print a Character Array.

Method	Purpose
static ArrayList<char [][]> buildUp(char[][] charArr);	Create a list by generating multiple arrays from the supplied character array. Each new array is a slice of the input array by increasing the number of rows in the slice counting from the bottom of the array.
static ArrayList<char [][]> buildRight(char[][] charArr);	Create a list by generating multiple arrays from the supplied character array. Each new array is a slice of the input array by increasing the number of columns in the slice counting from the left of the array.

Table 20. Create Multiple Arrays from One Array.

Method	Purpose
static char[][] transform(String methodNm, char[][] charArr);	The supplied character array is used to generate a new array by calling the function named in methodNm. The function must be in this class. Currently the functions are copy, flipHoriz, flipVert, invert, rotLeft, rotRight, and trim.
static ArrayList<String> getTransforms();	Return a list of the transformation functions defined in this class. A transformation function takes a character array as input and returns a new array.
static void printTransforms();	Print a list of the transformation functions defined in this class. A transformation function takes a character array as input and returns a new array.
static char[][] copy(char[][] charArr);	Transformation function: copy the array without change.
static char[][] rotRight(char[][] charArr);	Transformation function: the new array contains the input values rotated clockwise by 90 degrees.
static char[][] rotLeft(char[][] charArr);	Transformation function: the new array contains the input values rotated counter-clockwise by 90 degrees.
static char[][] flipVert(char[][] charArr);	Transformation function: the new array contains the input values flipped vertically.
static char[][] flipHoriz(char[][] charArr);	Transformation function: the new array contains the input values flipped horizontally.
static char[][] invert(char[][] charArr);	Transformation function: the new array contains the input values inverted so that 'on' is 'off', and vice versa. 'On' is coded as '*', 'off' as ' '.

Table 21. Transform a Character Array.

Method	Purpose
static void clear(char[][] charArr);	Fill the array with spaces.
static void clear(char[][] charArr, Rectangle r);	Fill the part of array that intersects with the rectangle with spaces.
static void clear(char[][] charArr, int x, int y, int w, int h);	Fill the array with spaces starting from row x, column y, extending for w

	columns and h rows.

Table 22. Clear a Character Array.

Method	Purpose
static boolean inside(char[][] charArr, Point p);	Is point p inside the character array?
static boolean inside(char[][] charArr, int x, int y);	Is coordinate (x, y) inside the character array?

Table 23. Character Array Containment.

Method	Purpose
static boolean add(char[][] out, int x, int y, char[][] charArr);	Add the character array to the out array with its top-left corner located at (x, y). Ignore the parts of the character array which extend beyond the edges of the out array.
static boolean append(char[][] out, int x, char[][] charArr);	Add the character array to the out array with its top-left corner located at (x,0). Ignore the parts of the character array which extend beyond the edges of the out array.

Table 24. Extend a Character Array.

Method	Purpose
static char[][] copyRegion(Rectangle rect, Point pos, char[][] charArr);	Copy the contents of the char array covered by the rectangle. The rectangle's (x, y) is in screen coordinates, and the top-left corner of the array (i.e. charArr[0][0]) is assumed to be located at position pos on the screen.
static char[][] trim(char[][] charArr);	Copy the char array, removing the starting and ending blank columns. This is also a transformation function (see Table 21).
static int getFirstTextCol(char[][] charArr);	Return the index of the first column with text.
static int getLastTextCol(char[][]	Return the index of the last column with

charArr);	text.
static boolean isBlankCol(char[][] charArr, int col);	Is this column of the char array all spaces?

Table 25. Reduce a Character Array.

Method	Purpose
static void print(char[][] charArr);	Print the array with a border around it, preceded by its size.

Table 26. Print a Character Array.

4. Support Tools

Aside from the support classes described in the previous section, the JLEB JAR contains three standalone tools for designing new sprites:

- ToChars.java
- ToHex.java
- DrawBoard.java, which utilizes DrawPanel.java

These tools can be executed most easily using their corresponding ".bat" files which are located in the jlebTests/ folder.

It's also possible to test the Dream Cheeky LED message board without writing a JLEB program by calling the HIDScreen.bat script.

4.1. ToChars.java

ToChars converts the image in the supplied file into a character array.

The user can optionally supply width and height arguments, and the image will be scaled before being converted. The scaling always retains the original width/height ratio of the image, so the scaled image may not exactly match the dimensions supplied by the user.

The image is converted to black and white, and the character array is generated by mapping each black pixel to '*', and each white pixel to ' '. The resulting array is printed to standard output.

The mapping to black and white does not work well on images with alpha channels (i.e. translucent parts), so any alpha parts should be converted to a solid color (e.g. white) before ToChars.java is called on the image.

Usage examples:

```
ToChars.bat p2_01.bmp
ToChars.bat blueTele.png 20 20
```


4.2. ToHex.java

ToHex converts character arrays read from a file into hexadecimals suitable for reading in as a font text file in the SpriteStore constructor. The hexadecimals encode each array horizontally, where one hexadecimal corresponds to one row. Each character array must have 7 rows, and at most 5 columns. This means that 7 hexadecimals are output for each array.

For example, ToHex.java is capable of translating the arrays in chars1.txt, which begins like so:

```
$$ /

    *
   *
  *
 *
*


// ---------------------------

$$ .




 **
 **

// ---------------------------

$$ A
 **
*  *
****
*  *
*  *


// ------------------------------
```

"$$" lines are used to assign a character name to the array that follows. This part of chars1.txt defines three arrays for the letters "/", ".", and "A".

Calling:

```
ToHex.bat chars1.txt
```

produces:

```
/   0x00  0x08  0x04  0x02  0x01  0x00  0x00
.   0x00  0x00  0x00  0x00  0x06  0x06  0x00
A   0x06  0x09  0x0F  0x09  0x09  0x00  0x00
```

This data can be used to construct a font text file, whose format is explained in the section 5.1.

4.3. DrawBoard.java

DrawBoard.java and DrawPanel.java are a Swing application, which displays a 21 by 7 set of empty rectangles, as in Figure 19.

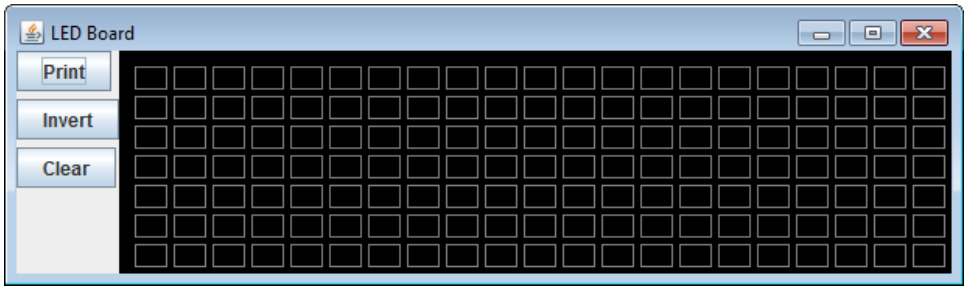

Figure 19. DrawBoard at Start-up.

The user can click on the rectangles to turn them 'on' (and again to turn them 'off'). When a suitable pattern has been created (as in Figure 20), the user can click on the "Print" button, to print the pattern as characters.

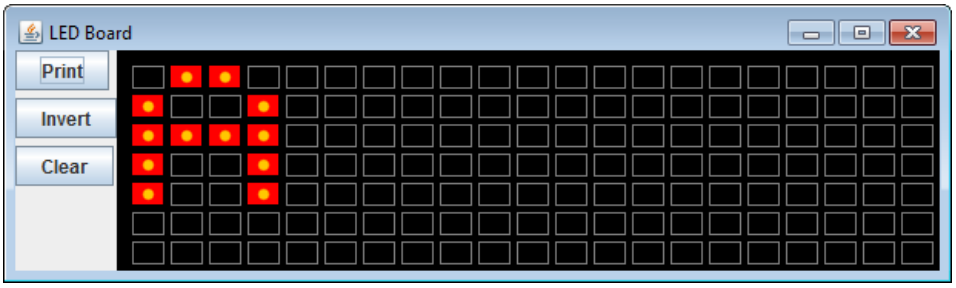

Figure 20. DrawBoard with a Pattern.

For instance, the output for Figure 20 would be:

```
-------- 21x7 -------
 **
*  *
****
*  *
*  *


---------------------
```

This data can be used to create a file suitable for ToHex.java (e.g. see chars1.txt), or a sprite file that can be loaded by SpriteStore (the format for SpriteStore files are described in the section 5.2).

4.4. The HIDScreen.bat Script

The HIDScreen.bat script calls JLEBs low-level HIDScreen class, which includes a main() function for testing the functionality of the Dream Cheeky LED message board without loading the rest of the JLEB API. If the script fails then there's a problem with the message board and/or javahidapi (the Java API for HID communication).

If the board is working correctly, then HIDScreen will print a list of detected HID devices, which should mention the message board. For example, the following is printed on one of my test machines:

```
HID Devices:
0. HIDDeviceInfo
[path=\\?\hid#vid_046d&pid_c315#6&32f3a27f&0&0000#{4d1e55b2-f16f-
11cf-88cb-001111000030}, vendor_id=1133, product_id=49941,
serial_number=?, release_number=10240, manufacturer_string=Logitech,
product_string=Logitech USB Keyboard, usage_page=1, usage=6,
interface_number=-1]

1. HIDDeviceInfo
[path=\\?\hid#vid_1d34&pid_0013#6&3457654c&0&0000#{4d1e55b2-f16f-
11cf-88cb-001111000030}, vendor_id=7476, product_id=19,
serial_number=1, release_number=1, manufacturer_string=Dream Link,
product_string=USB LED Message Board v1.0, usage_page=65280, usage=1,
interface_number=-1]
```

Then the class attempts to open a link to the message board, and light up a few of its LEDs. The output to the command window will be:

```
HID Library available
Opened HID device:
  Manufacturer: Dream Link
  Product: USB LED Message Board v1.0
  Serial Number: 1
Released HID device
Finished.
```

The board is kept running for a few seconds, so the user can check that it looks like Figure 21.

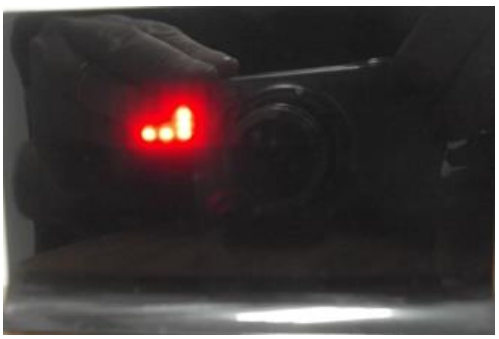

Figure 21. HIDScreen's Board Output.

If the board is unavailable then the script will report:


```
HID Devices:
  Unable to list devices: java.lang.NullPointerException
HID Library available
Could not open HID device
```

This tells us that the HID library (i.e. javahidapi from https://github.com/torbjornvatn/javahidapi) is installed but the Dream Cheeky hardware cannot be found.

Another possible error message is:

```
Could not find HID library
```

This means that javahidapi has not been installed correctly; my code assumes it is in D:\javahidapi.

5. File Formats Used by JLEB

SpriteStore is capable of loading four different text file formats: two font formats and two sprite formats.

5.1. Two Font Encodings

A character sprite can be encoded either vertically or horizontally as hexadecimals. In both cases the sprite is assumed to consist of 7 rows by 5 columns of '*' and ' ' characters.

The vertical encoding converts each column into a hexadecimal, producing five hexadecimal digits. The encoding is explained in detail at https://helpfactory.wordpress.com/tag/ascii/, and is used for JLEB's bigFont.txt font.

The horizontal encoding converts each row into a hexadecimal, producing seven hexadecimal digits. The encoding is implemented by ToHex.java included with the JLEB download, and is used for the mediumFont.txt and smallFont.txt fonts. I 'borrowed' the encoding from the dcled tool (http://www.last-outpost.com/~malakai/dcled/).

5.2. Two Sprite Encodings

The two encodings represent sprites that consist of a single character array picture, and sprites that contain multiple pictures.

sprites1.txt in the jlebTests/ folder is an example of how sprites with one picture are stored in a text file. Each sprite consists of a name, width and height (for the array), and the character array. For example:

```
downThumb 4 6
 ***
****
****
****
  *
```



```
  *

upArrow  5 7
  *
 ***
*****
 ***
 ***
 ***
 ***
```

The "downThumb" sprite consists of 4 column and 6 rows, while "upArrow" is 5 columns by 7 rows.

Examples of multi-pictures sprites can be found in stickWalk.txt, line.txt, pattern.txt, robot.txt, and arrows.txt in the jlebTests/ folder. Each file defines a single sprite, which consists of multiple character arrays. For example, the start of stickWalk.txt is:

```
walker 5 7

w0
  *
 ***
* * *
  *
 * *
 *  *
*  *

w1
  *
 ***
 ***
  *
 * *
** *
  *

// more lines; not shown...
```

stickWalk.txt contains eight character arrays, which are labeled as w0 to w7. The first line of the file gives the sprite's name and the dimensions of all the arrays (in this case 5 columns by 7 rows). Each character array is preceded by a name for the array, which must start with a lower-case letter.

