Project 2

Title

KeyDrop! V.2
Typing Tutor Game

Course

CIS-18

Section

83928

Due Date

February 11, 2006

Author

Angelo Rodriguez

Table of Contents

1 Introduction

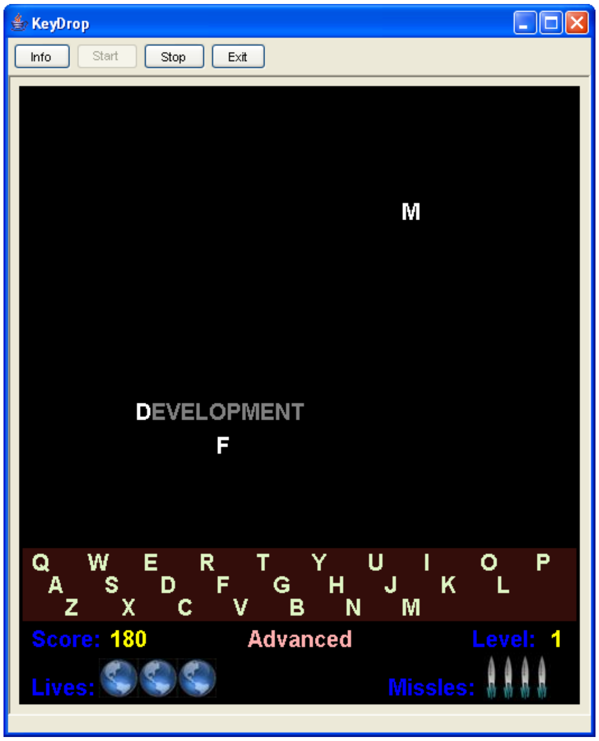
Many developers are often hindered in the speed of they're programming due to the lack of skill in using the keyboard. However, typing classes are not usually required when it comes to obtaining a degree in Computer Programming, Computer Science or Computer Engineering.

Developers are often times too lazy to learn how to type fast not knowing that by learning they'll be more productive and have more spare time for themselves. Often times, the developer's thinking process is significantly faster than the speed in which they can type out code. Sometimes, the developer may even forget key elements mid-stream while typing because they were distracted or they were concentrating more on the keyboard than the algorithm. Developers need to learn how to type fast and it should be as easy as driving.

Learning how to type should be fun though, so I'm creating a typing tutor in a form of a game which is similar to the 80's arcade game missile command. Bombs falling down from the sky will try and hit the corresponding letters on the ground. The objective is to shoot the falling letter bombs using the keyboard before it reaches the ground.

The layout of letters on-screen is similar to the keyboard layout. This visual coordination helps the player remember where on the keyboard a particular letter is. It does not force correct finger placement, however finger placement will be important at higher levels of the game.

2 Game Play and Rules



Score: 180

Advanced

Level: 1

The F2 key will start the game. Hitting F2 again during game play will immediately terminate the game. The F3 key will change the difficultly level of the application. The default is easy, then moderate, then difficult. Hitting the F3 key again will return the difficulty level back to easy. The F1 screen displays the rules and controls of the game.

When the game starts, bombs in the form of letters and/or words will drop down from the top of the screen and try to hit its target at the bottom. Players can shoot the bomb down by hitting the correct key (or key combination if the bomb is a word) on the keyboard. Should the bomb fall below the keyboard layout at the bottom of the screen; the player will lose one life. A total of three lives will be given per game.

You will be given only 5 missiles per life. You lose a missile each time you fire a missile (press a key). You get a bonus missile each time you hit the correct key. If you hit the incorrect key, you basically lose a missile and may eventually run out. If you lose all missiles, the dropping missile will hit its target and you will lose a life.

Each bomb destroyed is 5 points. In the case of word bombs, each letter in the bomb is worth 5 points. The level together with the speed of the game play will increase at regular intervals. An extra life will be given every 500 points. Game play will continue until all lives are lost.

The following table indicates the differences between each difficulty level:

Features	Easy	Moderate	Difficult
Letters	*Yes*	*Yes*	*Yes*
Words	*No*	*Simple*	*Simple and Difficult*
Multiple Bombs	*Highly Unlikely*	*Likely*	*Almost Certainly*

3 Development Summary

Lines of Code	**875**
Comment Lines	**372**
Blank Lines (White space)	**228**
Total Lines of Source File	**1475**

The project is a Java applet which is encapsulated in multiple files and uses several other image and sound files for use in the game. I used the JBuilder IDE for development which helped out a lot in linking me to the online help when I didn't know how to use a particular class or method, or just to find out what else a particular class can do.

This is a continuation of the development of the same applet from project 1 and is a culmination of a total work time of about 40 hours for both design and coding, not including the documentation.

3.1 Version 2 Comments on Development

Two of the features required for this second version required the use of object-orientation to make it a lot easier to deal with. These two features will be explained in the next couple of sections and how object-orientation helped in the enhancement of the program.

3.1.1 Letter and Word Bombs

In the first version of the software, only single character letter bombs were allowed. This made it easy to program because you only needed to keep track of one letter. A new feature, word bombs, could have been accomplished in one of two ways. The first required that for every location in the code that used some aspect of the bomb character is changed to accommodate either a letter or a word. In other words:

```
if (bomb is letter)
      do this.
Else
      do this.
```

There are dozens of these locations and in the future, what happens if a third type is added, or worse a tenth type is added. Object orientation allow me to create a base class bomb which is pretty much handled the same way all throughout the code. The only difference is when the bomb is created. It is either created as a letter bomb or a word bomb. But the interface to access which bomb key or if the bomb is destroyed is the same regardless of the bomb type.

3.1.2 Multiple Bombs

As I've stated before, the original code required very little plumbing to maintain one bomb, what happens when there are multiple bombs, with multiple bomb types. This could be a maintenance nightmare if we set up arrays or int's for locations, arrays of char's or strings for the bomb text, arrays of…you get the point.

Instead, I created these bomb objects and stored them in a collection (ArrayList) that allowed me to keep track of multiple bombs. Each bomb object knows what character needs to be pressed before it's destroyed, its location onscreen, and its type. The KeyDrop game engine just needed to run through the collection each time it had to perform a function on the bomb. I limited the number of bombs to 5, but this could easily be changed to 10 or 100 with a change in just one line of code. I'd like to see a secretary just try that out.

3.1.3 Display Elements

The paint method was a mesh of "draw this here", "draw that there", "use this color", and "do this". If there were more elements put (which I did put in) it would be much worse and pretty soon very hard to maintain.

Instead I created a hierarchy of classes all derived from a base abstract class called DisplayElements. The class hierarchy and definition of each class is listed in section 4.2. The purpose of this design is to create a collection of display elements which the paint method can just loop through to draw.

The logic that was put in the paint method is now distributed within the correct method. This actually is the correct way of handling logic since it is a best practice to separate display code with the logic of the program (separate presentation with business logic).

3.1.4 Other New Features

I also added some other new features to the application including:

- Difficulty Levels – a novice user can start from "Easy" which is pretty much similar to the first version of the application with the occasional second bomb and make their way to "Advanced" which contains multiple words and letters.
- Accuracy – the player's accuracy is now calculated and displayed at the end of the game.

3.2 Version 1 Comments on Development

3.2.1 Timers vs. Threads

Most games have an infinite loop controlling game play which is nothing but:

```
while (!done)
{
    …game play…
}
```

This however is usually running on a thread separate from the main application, or in this case, main applet thread. I research on how to implement a thread in Java, and truthfully it looked fairly easy. All I needed to do was implement the `Runnable` interface with the `Run()` method. This is what will run the game play. Declare a Thread object, having the applet contain the running method and call `start()` on the thread. Unfortunately, I opted on the Timer path. From the book, and from experience, you really should do what you think is simple first. In the future, time permitting, you can either create a more elegant or more efficient solution. In this case, the timer was the easier solution. Threads usually in other platforms inherently have their headaches associated with them.

3.2.2 Graphics Library

Java's graphics library is one of probably a dozen graphics libraries I have encountered through my life and each one has its own methodologies, strengths, nuances, and headaches. From the simple samples in our book that showed how to use `drawString()`, `drawLine()` and `drawRect()`, I had to educate myself on the other aspects of graphics programming using the Java library, including:

- **Changing the Font** – I needed a larger font and I wanted to choose a more pleasing one that the standard font used by the library.
- **Setting colors** – There seemed to be only about 16 colors to choose from which were declared in the Colors class. In addition, I'm used to setting RGB (red, green blue) components for colors instead of HSB (hue, saturation, brightness). I used PhotoShop to tell me the HSB equivalent of the color I wanted.
- **Centering text on the screen** – There a function I wrote, `drawStringCenter()`, which calculates how to center a string on the screen using the size of the string in pixels and the size of the screen. With Win32 API, I'm used to `GetTextExtent()`. In Windows Forms, this is `Graphics.MeasureString()`. In Java, I had to use `FontMetrics.getStringBounds()`.
- **Drawing an image** – Images were relatively easy to import and display, except in the issue I state below. I just needed to use the Java `Toolkit` to load the file to an `Image` class, then pass that to the `Graphics.drawImage()` method.

3.2.3 Sound

What is a game without sound effects? Boring! if you asked me. I needed to add audio effects to the game to make it better. This entailed using `Applet.newAudioClip()` to load the audio file into memory which returns back an `AudioClip`. Now once I had those files loaded into memory, I can actually call `AudioClip.play()` to instantaneous play the sound when I needed it.

The library also allowed multiple sound clips to play concurrently so that one didn't cut another clip in the middle while it was being played. This allowed me to add a rumbling sound in the background to put the player a little more on the edge. I played the rumbling sound in a loop using `AudioClip.loop()` so that it would be constant in the background and I didn't have to worry about replaying it when it was over.

3.2.4 Focus Issues between Main Applet and the JPanel Game

Initially, I designed the applet so that a JPanel extended game component can be used in an application as well. The game panel required focus in order for the keys to work and that presented a UI problem wherein the applet, which had buttons never got focus and was never serviced. You could not start the game at all.

I eventually moved all the code from the component into the applet class and changed the design so that the game play information is in the same screen and drawn (instead of using controls) and game play is now started with the F2 key.

I believe this made for a better interface now that the game is self contained within a single component.

3.2.5 Security Problems with Image and Sound Files

Since the images and sound clips are stored in files, I had to load them from disk. When running the application through the IDE, I did not have much problems displaying the images or showing the files. I simply mimicked examples found in Sun's Java site. However, whey I tried running the application through the browser, I was faced with a security problem where the Java runtime generated an exception when accessing the file. The built-in security was troublesome and I couldn't initially figure out the problem.

Initially, my problem was that the files were in my C:\Temp directory which I found online is not what I'm supposed to do. Java will treat this as another "site" and prevent it from loading. In addition, I had to convert the file load string from a simple "Shoot.wav" to a URL using java.net.URL.getCodeBase().

Here's a sample of the actual working code:

```
    shootSound  = Applet.newAudioClip(new URL(getCodeBase(), "Shoot.wav"));
```

This tells the runtime to get the base code address (directory where the applet is stored) and append the file name to the URL. This fixes the security problem since it now knows that the files comes from the same directory as the launching application.

3.2.6 Flickering and Double Buffering

When the game component functionality was moved to applet, flickering became more evident. I later found out that this is because applets, by default, erase the background using the background color. This is usually gray in most Java Runtime or Browser implementations (I'm not sure which one draws the gray). A simple call to `setBackground()` reduced the flickering, but now I was annoyed.

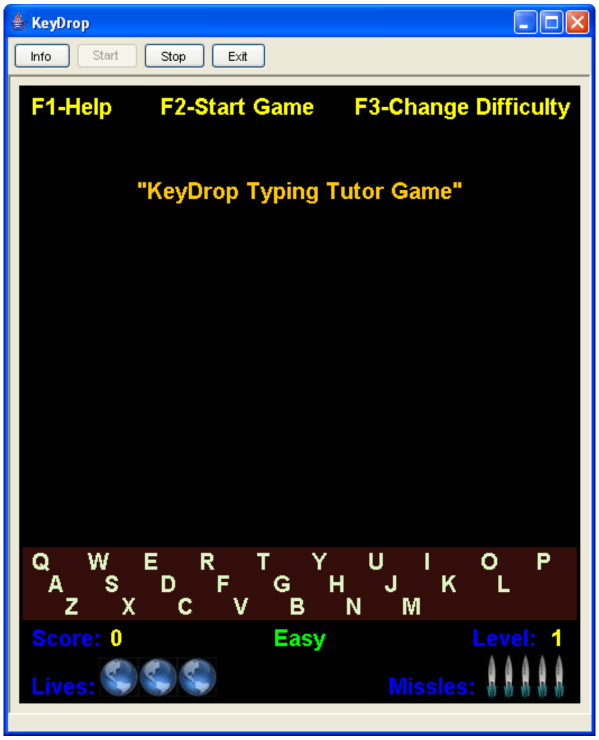
I used `repaint()` to have the game applet refresh the screen and animate the dropping bombs, but I happened to stumble on an overloaded `repaint()` which allowed me to input a rectangular area to repaint instead of doing it for the whole screen. Since only a small portion of the game is animated, I chose to repaint only that small area and this reduced the flicker a lot. There are still times I have to repaint the whole screen but they are less frequent than the dropping bombs.

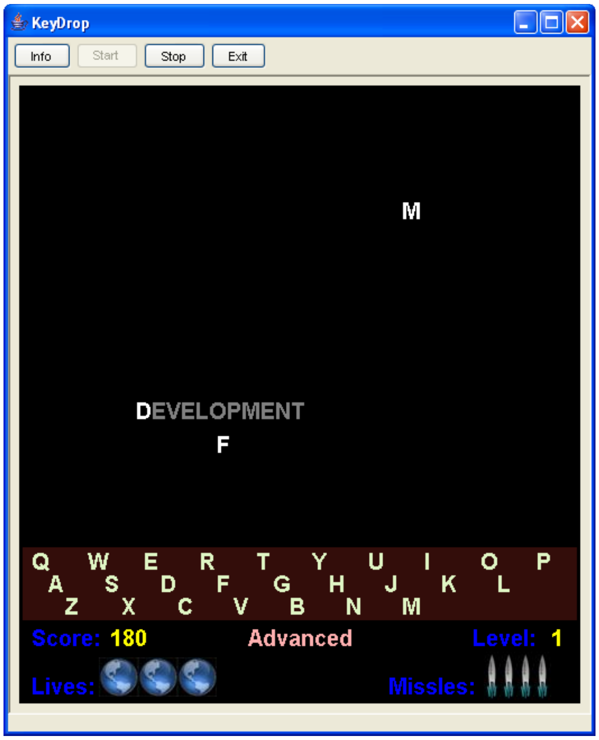
While researching the flickering problem I happened to stumble upon double buffering would have helped in making the applet. Double buffering would have completely eliminated the flicker. However, I believe this is not required at this time, but would be a fun thing to do in the future.

4 Specifications

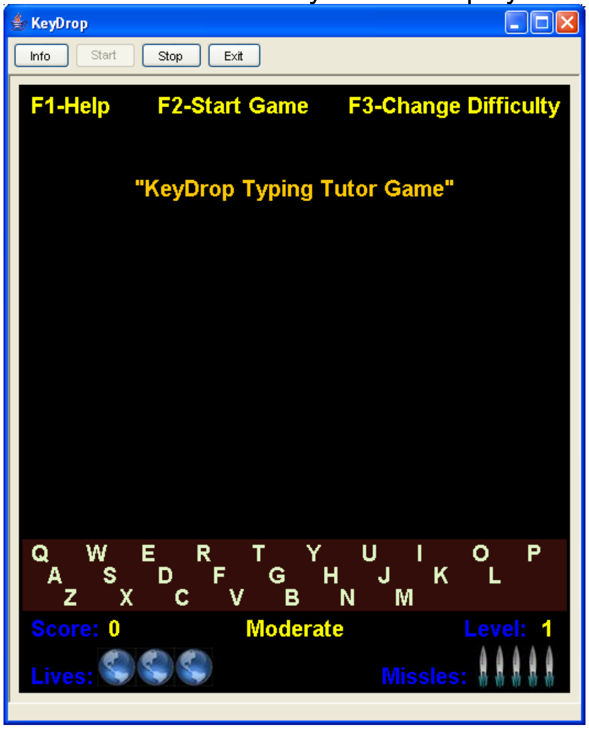
4.1 Sample Inputs/Outputs

Considering this is a game, the inputs are usually UI-centric as opposed to data-centric. The applet requires that the user press the F2 key to start the game. Pressing F2 again will end the game abruptly. Here are screen shots of the application before and after the user hits the F2 key which starts the game:

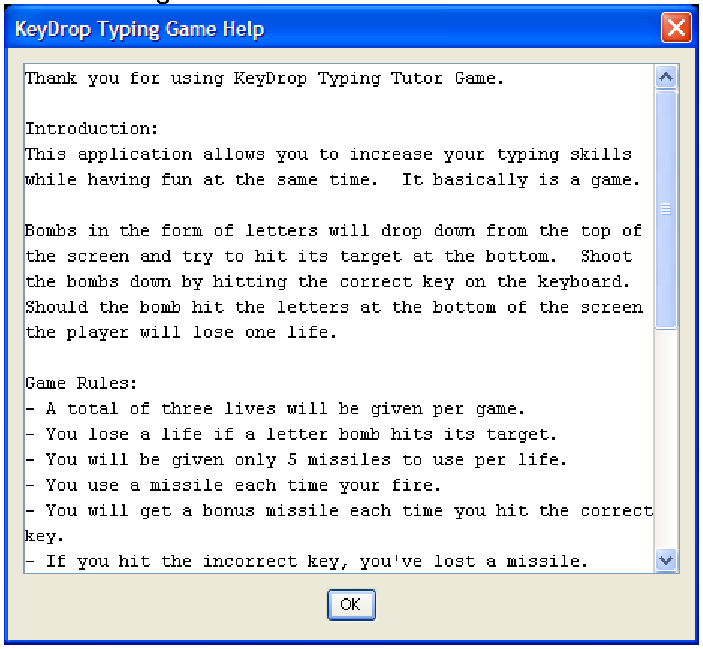




The F3 key is used to change between the different difficulty levels: Easy, Moderate and Advanced. The difficulty level is displayed on the bottom panel centered horizontally.

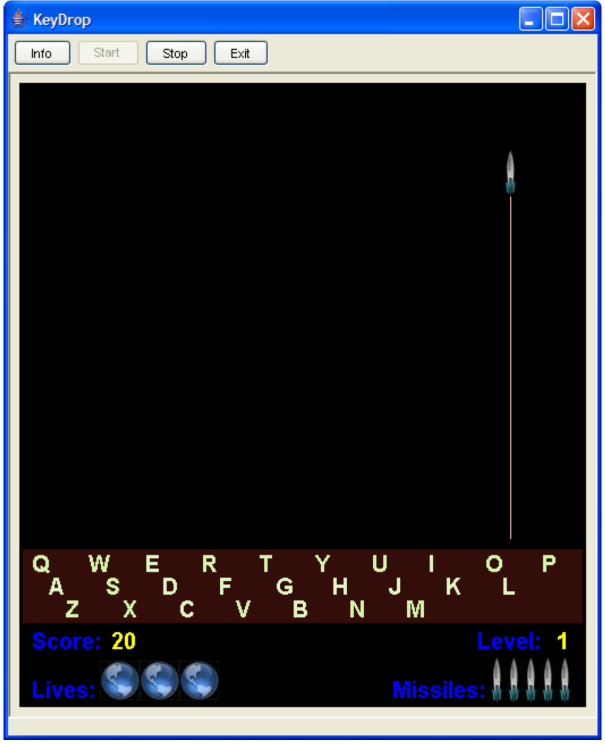


The F1 key is used to display the help screen. The user can close the help screen by clicking OK on the dialog:



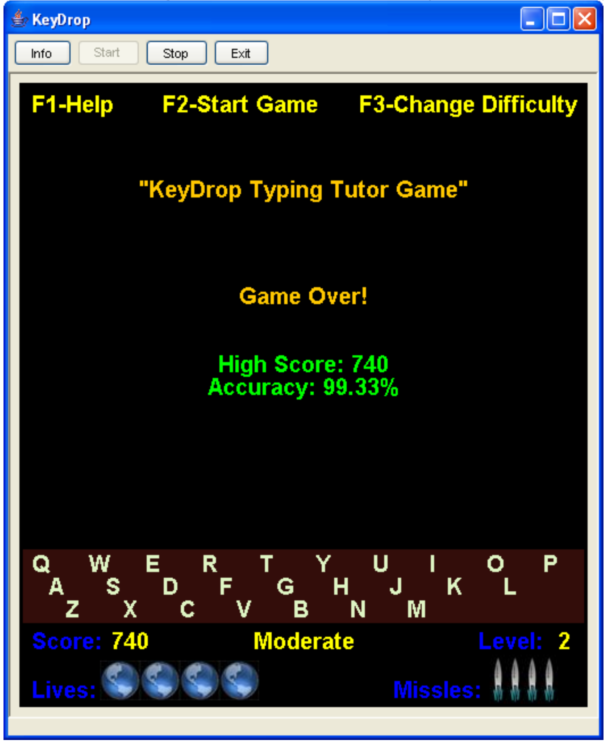
```
Thank you for using KeyDrop Typing Tutor Game.

Introduction:
This application allows you to increase your typing skills
while having fun at the same time.  It basically is a game.

Bombs in the form of letters will drop down from the top of
the screen and try to hit its target at the bottom.  Shoot
the bombs down by hitting the correct key on the keyboard.
Should the bomb hit the letters at the bottom of the screen
the player will lose one life.

Game Rules:
- A total of three lives will be given per game.
- You lose a life if a letter bomb hits its target.
- You will be given only 5 missiles to use per life.
- You use a missile each time your fire.
- You will get a bonus missile each time you hit the correct
key.
- If you hit the incorrect key, you've lost a missile.
```

During game play, the user can type any one of the 26 keys corresponding to the letters of the alphabet. When pressed, a missile will be fired and shown to the player. Once the game is over, the applet will indicate “Game Over” and show the current high score and the player’s accuracy.



Score: 20

Level: 1



"KeyDrop Typing Tutor Game"

Game Over!

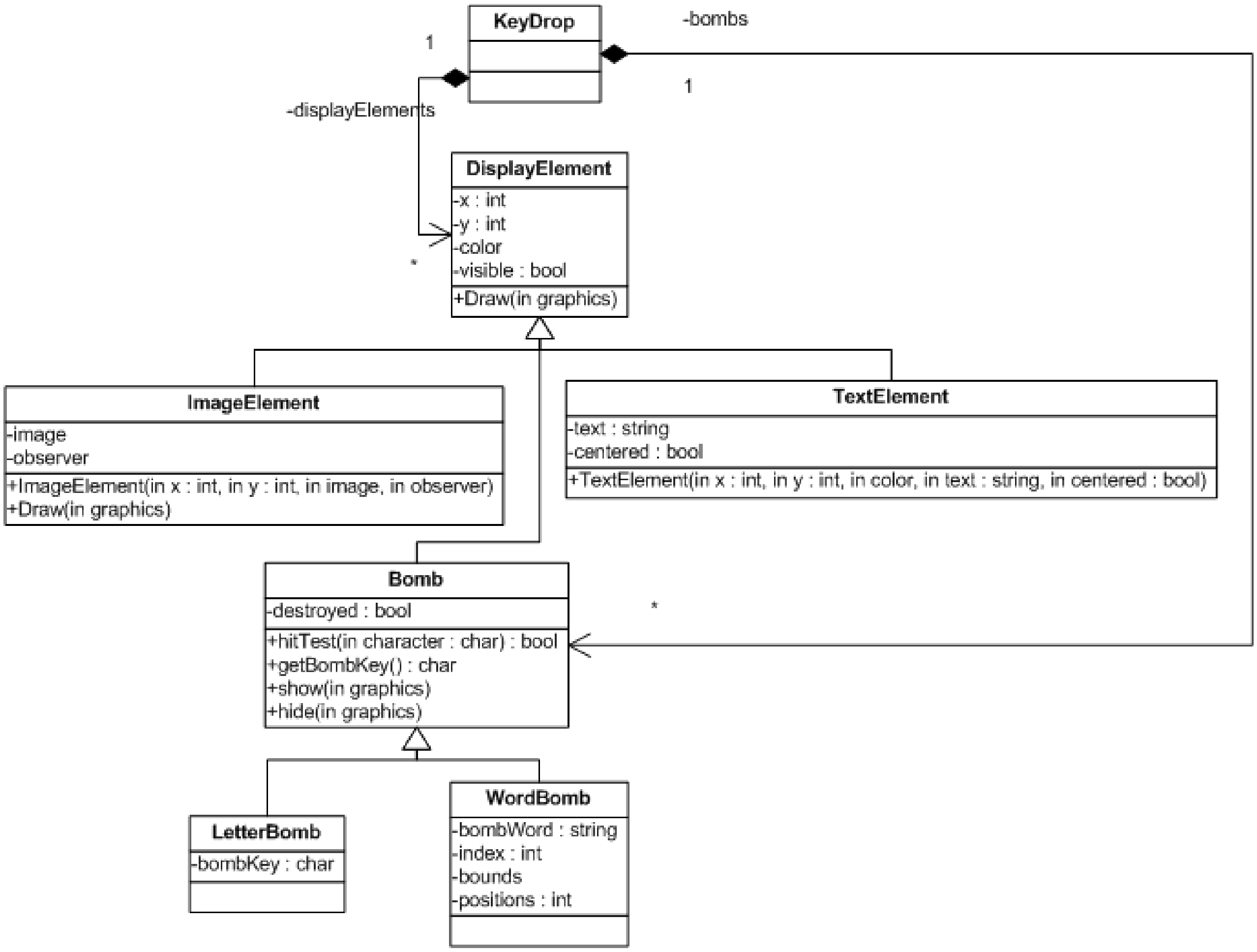
High Score: 740
Accuracy: 99.33%

Score: 740

Moderate

Level: 2

4.2 Class Diagrams



The key drop application contains several classes. Although I had wished to create the application with an object-oriented design from the ground up, I thought it would more useful to show how an existing structured application can be made into an object oriented one.

Each new class is described in the following sections. The class that is similar in both version 1 and 2 is the KeyDrop class which serves as both the extended JApplet class and the game engine as well. Other than the object oriented approach, the interface and most of the functionality has not changed much.

4.2.1 DisplayElement

This is the main abstract base class in which all elements on the screen will be used for display. It contains both the `x` and `y` coordinates so that the object will know where to draw itself on screen, as well as the color which is used by both the `TextElement` and the `Bomb` derived classes.

The class also has a `visible` field which can be used to temporarily hide the object even though it is still contained within the collection.

`Draw()` accepts a graphics object and is an abstract method which needs to be implemented by a concrete derived type. This is what allows an object to show text or images.

4.2.2 TextElement

The `TextElement`, which represents any form of text that will be displayed on screen, derives from `DisplayElement`. It contains a `text` field which holds the data to display onscreen as well as a `centered` flag. Setting the `centered` flag to `true` will indicate to the object that the text needs to be centered on screen when displaying. The `x` field must be set to the width of the display for the `TextElement` to center correctly.

All of these parameters can be set initially through an overloaded constructor.

4.2.3 ImageElement

An image will be referenced in the `image` field. A reference is used (instead of the object reading directly from file) for performance reasons. It helps eliminate unnecessary file reads and additional use of memory for same images between different objects, as in our case where we can display five or six missiles at a time and up to 7 lives.

An `observer` field is also referenced so that it can be notified when the `drawImage()` method is called. Similar to the `TextElement`, all parameters can be set initially through an overloaded constructor.

4.2.4 Bomb

This is another abstract base class, which is derived from `DisplayElement`. The purpose of this is to allow the application to create one or more bombs which are either of type `LetterBomb` or `WordBomb` without the application writing specific code for each.

There is a `destroyed` field which will be set to `true` if the bomb has been destroyed, that is if the character has been pressed in a letter bomb or if the whole word has been sequentially typed in a word bomb.

The `hitTest()` method returns `true` if the next key required to destroy the bomb has been pressed, otherwise it returns `false`. `hitTest()` is the actual method that determines if the bomb has been destroyed or not. This method is abstract and must be declared in a concrete derived class.

`getBombKey()` indicates to the application which character needs to be pressed next. This method is also abstract and requires an implementation in a derived class.

`show()` and `hide()` are functions that control the display of the bomb. I have determined that adding the bomb to the collection of display elements slows down the applet, so I decided to list keep the display of the bombs separate.

4.2.5 LetterBomb

A `LetterBomb` is a concrete derived type of `Bomb` and implements only one attribute, `bombKey` which is the letter needed to destroy the bomb. This letter is randomly generated when `LetterBomb` is created.

4.2.6 WordBomb

A `WordBomb` is another concreted derived type of `Bomb` and implements several attributes. `bombWord` is a randomly generated word from a dictionary of around 100 words. There are actually two dictionaries, one containing easy to spell words and another containing harder to spell words.

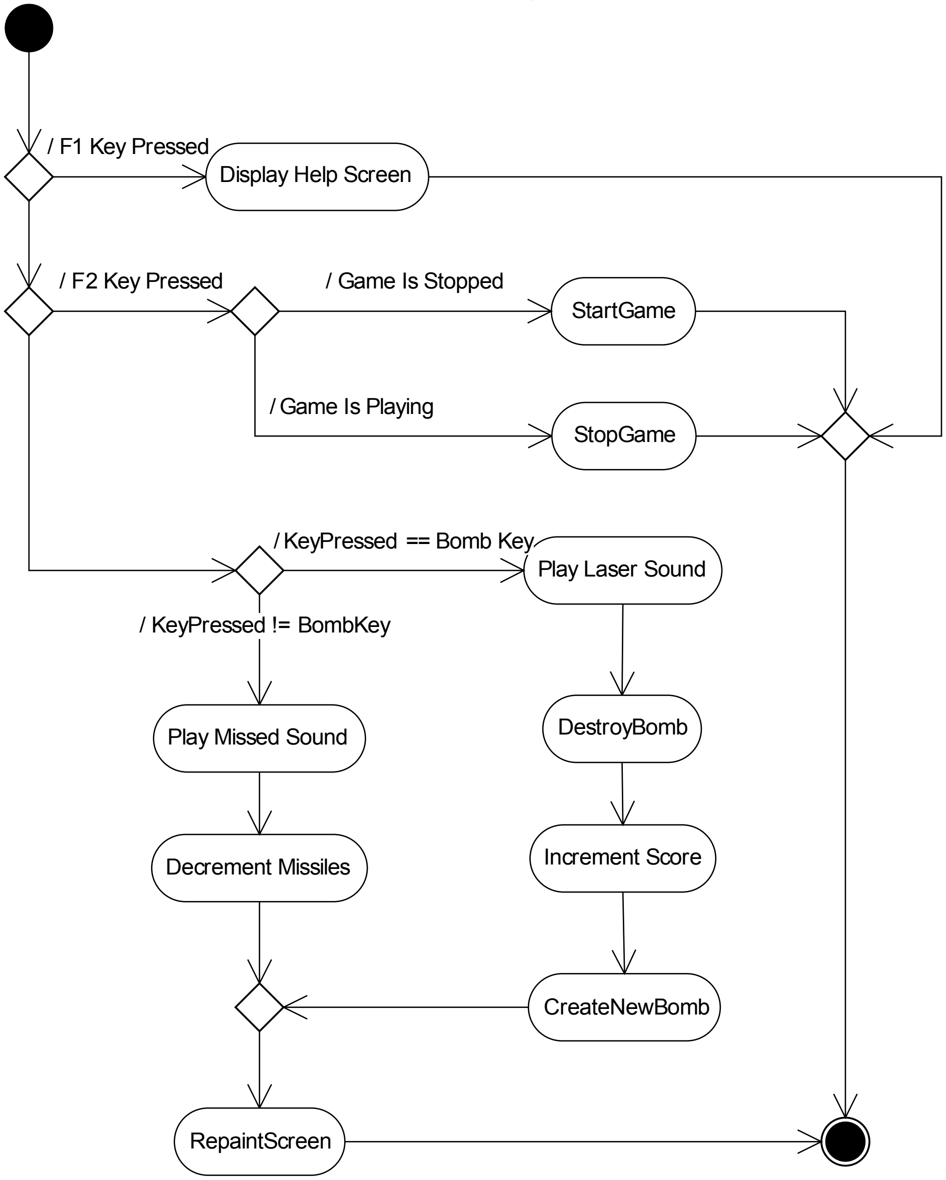
`index` determines the index of the next letter required to press in the sequence of letters within the word. `bounds` and `positions` indicate the bounding rectangle and horizontal positions for each rectangle. This is required to highlight the next letter to press when drawing the word on screen.

4.3 Activity Diagrams

I've created activity diagrams for 4 of the most important methods within the applet, `keyPressed`, `actionPerformed` ~ `timerElapsed`, `addPoint`, and `paint`. Two other methods which require mention is `startGame()` and `stopGame()` which starts game play or ends the game play respectively. Game play variables are initialized in `startGame()` and the game timer triggering the animation starts as well. In `stopGame()`, the game timer is stopped and the high score is calculated.

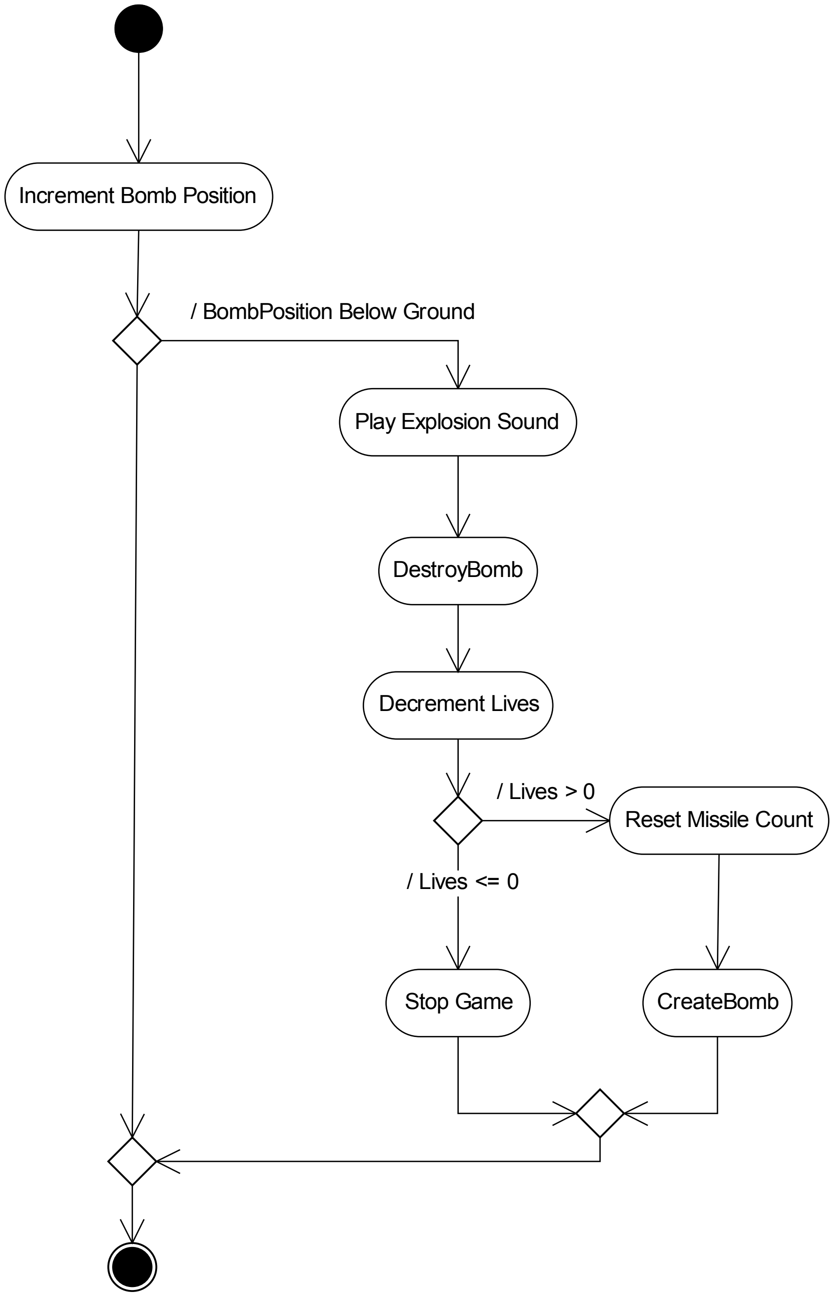
4.3.1 keyPressed

`keyPressed` is triggered whenever the user presses a key. Key events which this method processes are F1, F2, and the alphabet keys.



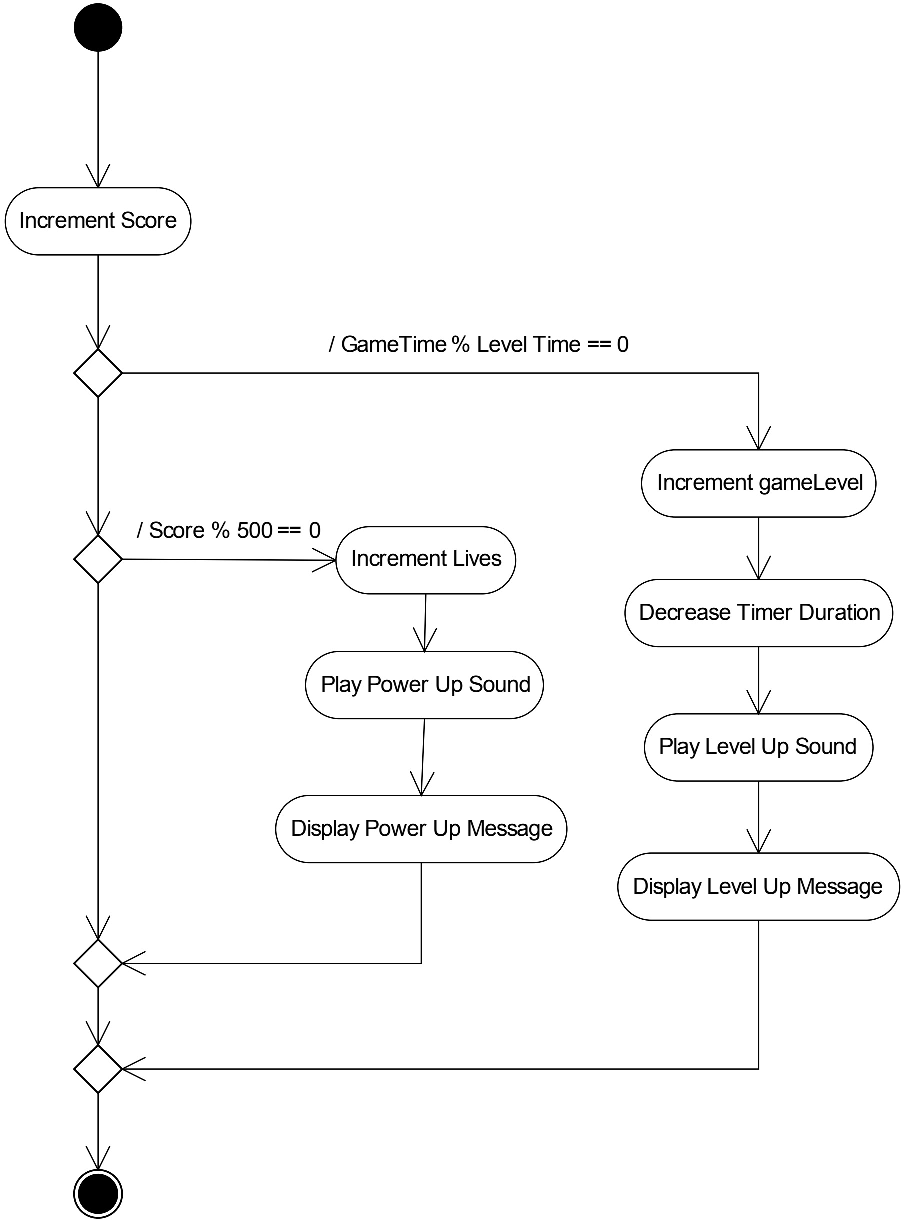
4.3.2 actionPerformed ~ timerElapsed

`actionPerformed` processes only one event and that is from the timer. So in essence, this is more of a `timerElapsed` method. The method is triggered whenever the delay time set for the timer has elapsed. The initial delay value is 60 and will decrease by 5 milliseconds every game level.



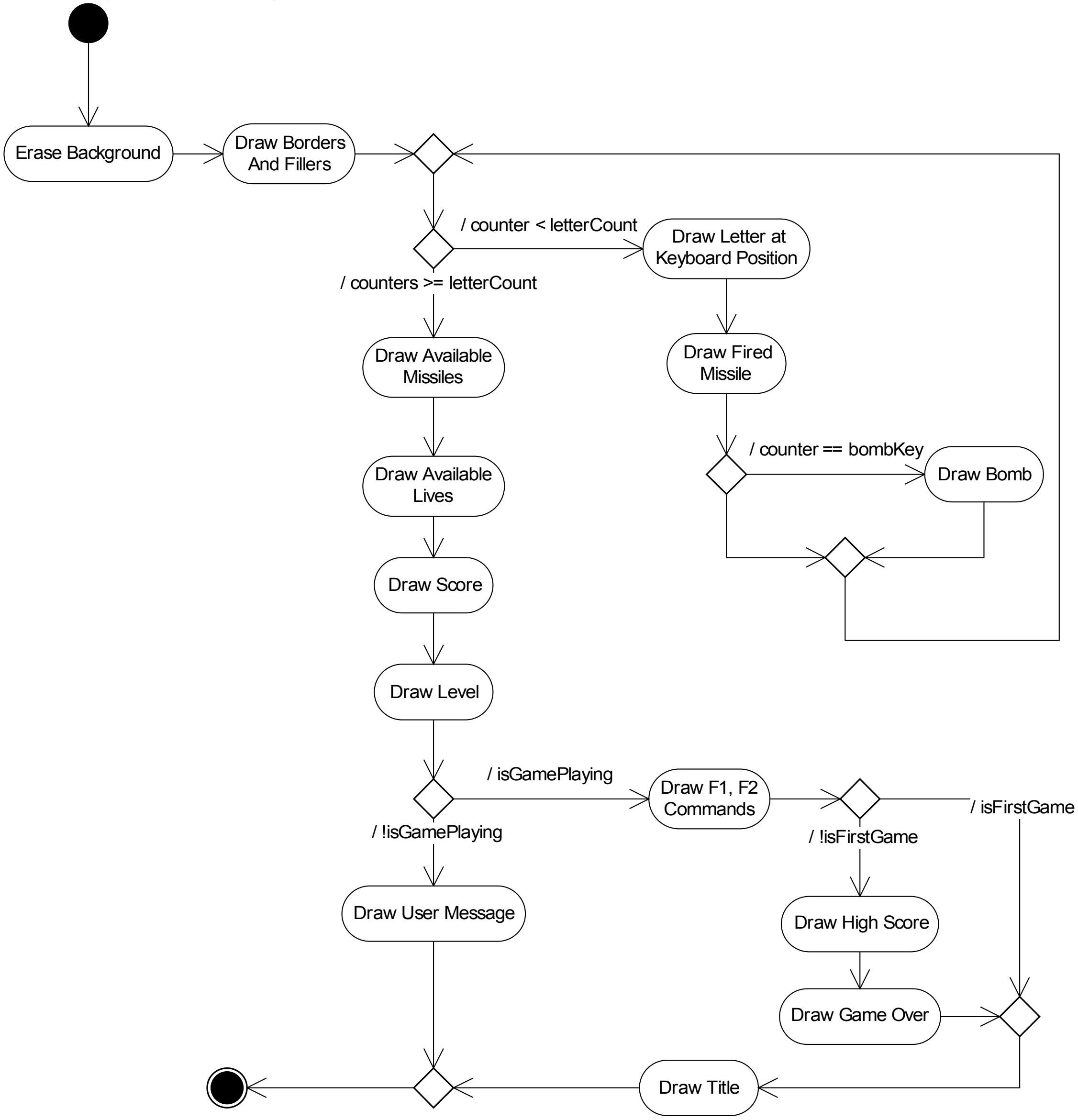
4.3.3 addPoint

Although the basic functionality of this method is just to increment the score, the applet uses it to determine if its time to level and if the player deserves a bonus life.



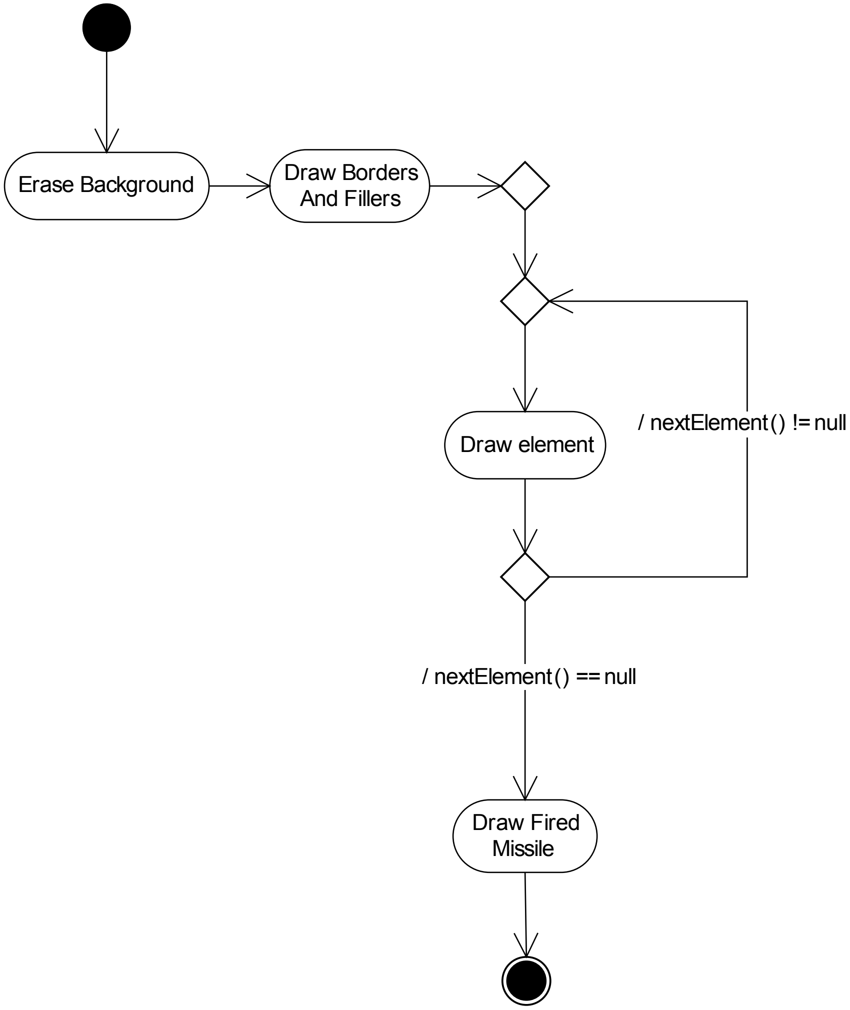
4.3.4 paint (Old)

`paint` is where all the action takes place. This is where all the drawing occurs. There really isn't much logic in paint, which should be the case. It simply reacts to variables set by the game play, such as the number of lives and missiles to display, whether or not to display the laser, or where to display the dropping bomb.



4.3.5 paint (New)

`paint` is still where all the action takes place in version 2 of the application, however instead of elaborating how each element on the screen is displayed, this task is delegated to objects of different types. Although a single function is called – `draw()` for each display object, it still somehow manages to display each element correctly and in a more elegant and collected fashion. It clearly shows by comparing the diagram below with that from the old version of paint() the benefit of a object-oriented approach using inheritance and polymorphism.



4.4 Pseudocode

4.4.1 Version 1 Pseudocode

The following Pseudocode was developed prior to the development of the application. Although the general idea behind several of the methods is still intact in the final product, there were several additions that are not reflected.

```
Destroy Missle:
    hide the missile

Create Missle:
    missile.key = randomly generated key
    missile.position = top of screen
    draw the missile
```

Add Point To Score:

```
    Add 1 point to player’s score
    if (score modulus 100 is 0)
        decrease timeout of missile movement
    if (score modules 200 is 0)
        Add 1 to player’s life count
```

Process Keystroke:

```
    play shooting sound
    deduct 1 from player’s missile count

    If (keystroke is the same as the missile.key)
    {
        draw animation of player’s missile to dropping missile
        play explosion sound

        destroy Missile
        Add Point To Score

        add 1 to player’s missile count
        create Missile
    }
    else
    {
        Draw animation of player’s missile to top of screen
    }
```

Move Missle:

```
    Hide the missile
    Move missile.position down by one
    Draw the missile

    if (missile position >= ground position) // below ground level
    {
        Play big explosion sound
        Draw animation of letter exploding
        Destroy Missle

        Deduct 1 life from player
        if (number of lives > 0)
        {
             Set player’s missiles to 5
             Create Missile
        }
    }
```

Main application:

```
    Set players lives to 3
    Draw entire screen
    Create Missile

    while (Still Alive)
    {
        wait for keystroke or timeout
        switch (action)
        {
            case keystroke:
                Process Keystroke
                break
            case timeout:
                Move Missile
                break
        }
    }
```

4.4.2 Version 2 Pseudocode

The display elements on screen were really simply many different text and image data that were just stored as strings, integers and images. What needed to happen was to create a collection of display elements and have the paint method draw all elements within the collection. Here's the Pseudocode I wrote for the second version of the application:

```
Initialization:
      Add Title Text to display container
      Add F1, F2 text to display container
      Add High Score text to display container
      Add Letters to display container
      Add Score label and value to display container
      Add level and value to display container
      Add Lives and missiles label to display container
      Add lives and missiles images to display container

Paint
      Clear screen
      Draw borders        // This may also become a display element
      foreach (display container element)
            element.draw()
```

Also, I wanted to add a feature where bombs can not only be letters, but full words so I needed to create a mechanism to allow words or letters. In addition, I wanted to be able to display more than 1 bomb at the same time:

```
CreateBomb:
      CreateBomb = Random choice
            Case Easy = 0.01% - create new bomb
            Case Moderate =  0.1% - create a new bomb
            Case Difficult = 1% - create a new bomb

      if (bomb count < 5)
      {
            Bomb = Random choice
                  25% - create word bomb
                  75% - create letter bomb
      }
```

Timer event:

```
      Foreach (Bomb)
```

```
            Bomb.move( distance based on level);
```

Although things may seem simplistic, putting the infrastructure of all the display elements may be a task.

4.5 Variables

I'll separate this section into three: constant fields, game play fields, and resources. All fields defined are class level and are considered the more important variables in the applet. Local variables are seldom important in the overall scheme of applications, especially this one.

4.5.1 Constant Fields

I made it easy to change some of the aspects of the game play and locations of the labels and components by declaring a list of constant fields. Here is a list, with comments of the fields defined in this section (the updates for version 2 are marked in blue):

```
// Defines Game Defaults
private final static byte PLAYER_LIVES    = 3;    // Initial number of lives per game
private final static int  TIMER_UPDATE    = 60;   // Initial time out for next animation
private final static int  TIMER_INCREMENT = 5;    // Amount to decrease timeout per level
private final static byte MISSILE_COUNT   = 5;    // Initial number of missiles per life
private final static byte LIFE_COUNT      = 7;    // Maximum number of lives a player can have
private final static byte SCORE_INCREMENT = 5;    // Points per hit
private final static byte LEVEL_UP_TIME   = 30;   // Seconds interval between each level up
private final static int  MESSAGE_TIME    = 2;    // Seconds to display user message
private final static int  FIRE_TIME       = 103;  // Milliseconds to display the laser fired
private final static int  POWER_UP_SCORE  = 500;  // Score interval required for power up
private final static int  MAX_BOMB_COUNT  = 5;    // Maximum number of bombs that can drop


// Defines Positions and increments
private final static byte TOTAL_ROWS      = 3;   // Total number of rows to display
private final static int  INIT_X_POSITION = 10;  // Initial horizontal position to display letter.
private final static int  INIT_Y_POSITION = 430; // Initial vertical position to display letters.
private final static int  X_INCREMENT     = 50;  // Distance between each row.
private final static int  Y_INCREMENT     = 20;  // Distance between each letter.
private final static int  ROW_X_OFFSET    = 15;  // Offset to simulate diagonal keyboard layout.
private final static int  DROP_INCREMENT  = 2;   // Amount in pixels to increment drop each level.
```

4.5.2 Game Play Fields

The game play fields are variables that pertain to the game in particular such as number of lives and score. Here is the list with comments (the items removed are crossed out, while the new updated fields are marked in blue):

```
// Game States
private boolean gamePlaying  = false;   // Is game ongoing?
~~private boolean firstGame    = true;    // Determines if first game has been played~~

// Dropping bomb parameters, put this in a structure or class later.
~~private char bombKey;                   // The currently dropping bomb letter~~
~~private int  bombPosition;              // The current position of the dropping bomb~~

// Dropping bomb parameters, put this in a structure or class later.
private ArrayList bombs = new ArrayList();  // Holds the the current bomb object

// Display elements with time limits
~~private String userMessage;             // Message to display to the user during game play~~
private int    messageTime;             // Timeout for the message.
private int    fireTime;                // Timeout for missile fire.
private char   fireLetter;              // Letter which was fired.
private int    firePosition;            // Bomb position at time fired.

// Game play
private byte    playerLives  = PLAYER_LIVES;   // Number of remaining lives
private byte    missileCount = MISSILE_COUNT;  // Number of remaining missiles
private int     score        = 0;              // Player's current score
private byte    gameLevel    = 1;              // Player's current level
private int     highScore    = 0;              // Current high Score
private int     timerCount   = 0;              // Elapsed game time.
```

4.5.3 Resource and Plumbing Fields

Visual components, audio clips, images, timers and other fields required for the plumbing of methods are listed here. Also included are fields which will hold the letters displayed in order and their positions (the items removed are crossed out, while the new updated fields are marked in blue):

```
// Define the keys and holders for their positions.
// Letters and order to display
~~private String rowKeys[]         = new String[TOTAL_ROWS];~~
// Calculated positions of letters
~~private Point  letterPositions[] = new Point[26];~~

// Declare resources used
// Timer event which triggers the animation
private Timer  gameTimer         = new Timer(TIMER_UPDATE, this);
// Font to be used on screen instead of the lame default
private Font   keyFont           = new Font("Arial", Font.BOLD, 20);
// Help screen text area, less processing if kept at class level.
private JTextArea outputTextArea = new JTextArea(20, 60);

// Declare images and sound clips used.
private Image     missileImage;    // Missile image on bottom right
private Image     lifeImage;       // World image on bottom left
private AudioClip shootSound;      // Played when bomb is intercepted
private AudioClip explosionSound;  // Played when bomb falls to ground
private AudioClip levelUpSound;    // Played when player levels up
private AudioClip emptySound;      // Played when missileCount is 0
private AudioClip rumbleSound;     // Looped in the background
private AudioClip missedSound;     // Played when player presses wrong key
private AudioClip powerUpSound;    // Played when player gets extra life

// Create a container for all display elements that will be shown on the screen
private Hashtable displayElements = new Hashtable(30);
private ArrayList idleElements    = new ArrayList(3);

// Name some invidually accessible displayElements
private TextElement scoreValue;       // Displays what the players current score
private TextElement levelValue;       // Displays the players current level
private TextElement highScoreValue;   // Displays the games current high score
private TextElement userMessage;      // Message to display to the user during game play
private TextElement difficultyText;   // Displays the level of difficulty
private TextElement accuracyValue;    // Displays the player's level of accuracy

private ImageElement[] missileImages; // Missile images on bottom right
private ImageElement[] lifeImages;    // World images on bottom left
```

4.6 Concepts Used

Chapter 7 Concepts

- Arrays
- References
- ~~Passing Arrays to Methods~~
- ~~Sorting~~
- ~~Searching Arrays~~
- ~~Multidimensional Arrays~~
- ~~Collaboration Diagrams~~

Chapter 8 Concepts

- Object-based Programming
- Class Scope – Private and Public
- ~~Referring to Current Object's members with this~~
- Constructors
- Set – Mutators
- Get – Accessors
- Composition
- ~~Garbage Collection~~
- Static Class Members
- Final Instance Variables
- ~~Packages~~
- Data Abstraction and Encapsulation

Chapter 9 Concepts

- Inheritance
- Superclasses and Subclasses
- Protected Members
- Three-Level Inheritance
- ~~Finalizers~~

Chapter 10 Concepts

- Polymorphism
- Abstract Classes
- ~~Final Methods and Classes~~
- ~~Runtime Type Identification – getClass & instanceOf~~
- Interfaces
- ~~Application Frames using JFrame~~
- ~~Nested, Inner Classes~~
- ~~Anonymous Inner Classes~~
- ~~Design Patterns~~

5 References

Sound Effects are taken from the following sites:

- http://www.a1freesoundeffects.com/noflash.htm
- http://www.koumis.com/soundfx.htm

Most of the research and reference material used is through Sun's Java site (http://sun.java.com). Here I took most referred to the online library as well as tutorial and sample programs to develop the applet. No code however was copied from this site.

Google Initiated Sites

In instances where Sun did not provide a helpful or clear example (or didn't provide an example of use at all), I searched Google on references to specific code. However, I didn't take notes of the sites where examples were helpful. Regardless, code was not copied from these sites. They were simply used in research and as part of the learning process.

I also employed the use of GeroneSoft Code Counter Pro Version 1.23 (http://www.investph.com/geronesoft) to generate the statistics for me on the applet's source code.

6 Program Listing

6.1 KeyDrop.java

```
import javax.swing.JApplet;
import javax.swing.Timer;
import java.applet.AudioClip;
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import java.awt.Toolkit;
import java.awt.Font;
import java.awt.Color;
import java.net.URL;
import java.net.*;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.ArrayList;
import java.text.DecimalFormat;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JOptionPane;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: KeyDrop</p>
 * <p>Class Description: Main applet form as well as the game engine.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/

public class KeyDrop extends JApplet implements KeyListener, ActionListener {
    // Defines Game Defaults
    private final static byte PLAYER_LIVES    = 3;    // Initial number of lives per game
    private final static int  TIMER_UPDATE    = 60;   // Initial time out for next animation
    private final static int  TIMER_INCREMENT = 5;    // Amount to decrease timeout per level
    private final static byte MISSILE_COUNT   = 5;    // Initial number of missiles per life
    private final static byte LIFE_COUNT      = 7;    // Maximum number of lives a player can
have
    private final static byte SCORE_INCREMENT = 5;    // Points per hit
    private final static byte LEVEL_UP_TIME   = 30;   // Seconds interval between each level up
    private final static int  MESSAGE_TIME    = 2;    // Seconds to display user message
    private final static int  FIRE_TIME       = 103;  // Milliseconds to display the laser fired
    private final static int  POWER_UP_SCORE  = 500;  // Score interval required for power up
    private final static int  MAX_BOMB_COUNT  = 5;    // Maximum number of bombs that can drop

    // Defines Positions and increments
    private final static byte TOTAL_ROWS      = 3;   // Total number of rows to display
    private final static int  INIT_X_POSITION = 10;  // Initial horizontal position to display
letter.
    private final static int  INIT_Y_POSITION = 430; // Initial vertical position to display
letters.
    private final static int  X_INCREMENT     = 50;  // Distance between each row.
    private final static int  Y_INCREMENT     = 20;  // Distance between each letter.
    private final static int  ROW_X_OFFSET    = 15;  // Offset to simulate diagonal keyboard
layout.
    private final static int  DROP_INCREMENT  = 2;   // Amount in pixels to increment drop each
level.

    // Difficulty Level strings
    private final static String[] DIFFICULTY_TEXT = new String[]{"Easy", "Moderate",
"Advanced"};
    private final static Color[] DIFFICULTY_COLOR = new Color[]{Color.green, Color.yellow,
Color.pink};

    // Create a container for all display elements that will be shown on the screen
    private Hashtable displayElements = new Hashtable(30);
    private ArrayList idleElements    = new ArrayList(3);

    // Name some invidually accessible displayElements
    private TextElement scoreValue;       // Displays what the players current score
    private TextElement levelValue;       // Displays the players current level
    private TextElement highScoreValue;   // Displays the games current high score
    private TextElement userMessage;      // Message to display to the user during game play
    private TextElement difficultyText;   // Displays the level of difficulty
    private TextElement accuracyValue;    // Displays the player's level of accuracy

    private ImageElement[] missileImages; // Missile images on bottom right
    private ImageElement[] lifeImages;    // World images on bottom left

    // Declare resources used
    // Timer event which triggers the animation
    private Timer  gameTimer          = new Timer(TIMER_UPDATE, this);
    // Font to be used on screen instead of the lame default
    private Font   keyFont            = new Font("Arial", Font.BOLD, 20);
    // Help screen text area, less processing if kept at class level.
    private JTextArea outputTextArea  = new JTextArea(20, 60);

    // Declare images and sound clips used.
    private AudioClip shootSound;      // Played when bomb is intercepted
    private AudioClip explosionSound;  // Played when bomb falls to ground
    private AudioClip levelUpSound;    // Played when player levels up
    private AudioClip emptySound;      // Played when missileCount is 0
    private AudioClip rumbleSound;     // Looped in the background
    private AudioClip missedSound;     // Played when player presses wrong key
    private AudioClip powerUpSound;    // Played when player gets extra life
    private Image     missileImage;    // Missile images on bottom right
    private Image     lifeImage;       // World images on bottom left

    // Declare game fields.
    // Game States
    private boolean gamePlaying  = false;   // Is game ongoing?

    // Dropping bomb parameters, put this in a structure or class later.
    private ArrayList bombs = new ArrayList();  // Holds the the current bomb object

    // Display elements with time limits
    private int    messageTime;             // Timeout for the message.
    private int    fireTime;                // Timeout for missile fire.
    private char   fireLetter;              // Letter which was fired.
    private int    firePosition;            // Bomb position at time fired.
    private int    lastBombX;               // Horizontal position of bomb when fired

    // Game play
    private byte   playerLives  = PLAYER_LIVES;   // Number of remaining lives
    private byte   missileCount = MISSILE_COUNT;  // Number of remaining missiles
    private int    score        = 0;              // Player's current score
    private int    gameLevel    = 1;              // Player's current level
    private int    highScore    = 0;              // Current high Score
    private int    timerCount   = 0;              // Elapsed game time.
    private byte   difficulty   = 0;              // 0-Easy; 1-Moderate; 2-Difficult
    private int    shotsFired   = 0;              // Total number of shots fired
    private int    shotsHit     = 0;              // Total number of shots hit
    //--------------------------------------------------------------------------
    // Constructor
    public KeyDrop() {
    }


    //--------------------------------------------------------------------------
    // init is called by Swing library when the object is created.
    public void init()
    {
        this.setBackground(Color.black);

        // Initialize all audio and graphics for this applet
        initGraphicsAndAudio();

        // Initialize the game instructions
        initGameInstructions();

        // This applet will be receiving keystrokes
        this.addKeyListener(this);
        requestFocus();
    }

    //--------------------------------------------------------------------------
    // Initialize all visual components, since this is the first time that
    // the applet coordinates are available.
    public void start()
    {
        // Define the keys and holders for their positions.
        initDisplayElements();
    }

    //--------------------------------------------------------------------------
    // Initialize all the letter objects
    private void initDisplayElements()
    {
        // Letters and order to display
        String rowKeys[] = new String[TOTAL_ROWS];

        // Initialize the row keys.
        rowKeys[0] = "QWERTYUIOP";
        rowKeys[1] = "ASDFGHJKL";
        rowKeys[2] = "ZXCVBNM";

        // Initialize Vertical position
        int yPosition = INIT_Y_POSITION;

        // Create all letter objects and store them in the display elements container
        // Calculate positions, determine row position first.
        for (int lineCount = 0; lineCount < TOTAL_ROWS; lineCount++)
        {
            // Initialize horizontal position
            int xPosition = INIT_X_POSITION + (ROW_X_OFFSET * lineCount);

            // Calculate each letter position by iterating through each character in the row
string
            for (int keyCount = 0; keyCount < rowKeys[lineCount].length(); keyCount++)
            {
                // Determine the character and create the correspoding letter object.
                String character = Character.toString(rowKeys[lineCount].charAt(keyCount));
                TextElement letter = new TextElement(xPosition, yPosition,
                                       Color.getHSBColor(0.241f, 0.20f, 0.96f),
                                       character);

                // Add the new object to the container.
                this.displayElements.put(character, letter);

                // Prepare position for next letter
                xPosition += X_INCREMENT;
            }
            // Increment row position and the horizontal offset
            yPosition += Y_INCREMENT;
        }

        // Add missles, lives, score and level labels.
        this.displayElements.put("MissleLabel", new TextElement(this.getWidth() - 170,
                this.getHeight() - 7, Color.blue, "Missles:"));
        this.displayElements.put("LivesLabel", new TextElement(10, this.getHeight() - 7,
                Color.blue, "Lives:"));
        this.displayElements.put("ScoreLabel", new TextElement(10, this.getHeight() - 50,
                Color.blue, "Score:"));
        this.displayElements.put("LevelLabel", new TextElement(this.getWidth() - 95,
                this.getHeight() - 50, Color.blue, "Level:"));

        // Add values for score and level
        this.scoreValue = new TextElement(80, this.getHeight() - 50, Color.yellow,
                                   Integer.toString(getScore()));
        this.displayElements.put("Score", scoreValue);
        this.levelValue = new TextElement(this.getWidth() - 25, this.getHeight() - 50,
Color.YELLOW,
                              Integer.toString(getLevel()));
        this.displayElements.put("Level", this.levelValue);
        this.difficultyText = new TextElement(this.getWidth(), getHeight() - 50,
                              DIFFICULTY_COLOR[this.difficulty],
                              DIFFICULTY_TEXT[this.difficulty], true);
        this.displayElements.put("Difficulty", this.difficultyText);

        // Add Help, Start, Difficulty and Title text
        this.idleElements.add(new TextElement(10, 25, Color.yellow, "F1-Help"));
        this.idleElements.add(new TextElement(125, 25, Color.yellow,
                                              "F2-Start Game"));
        this.idleElements.add(new TextElement(this.getWidth() - 200, 25, Color.yellow,
                                              "F3-Change Difficulty"));
        this.idleElements.add(new TextElement(this.getWidth(), 100, Color.orange,
                                              "\"KeyDrop Typing Tutor Game\"", true));

        // Draw game over and high score but make them initially invisible.
        this.highScoreValue = new TextElement(this.getWidth(), this.getHeight() / 2 - 20,
                                              Color.green, "High Score: ", true);
        this.highScoreValue.setVisible(false);
        this.idleElements.add(this.highScoreValue);
        TextElement element = new TextElement(this.getWidth(), this.getHeight() / 2 - 80,
                              Color.orange, "Game Over!", true);
        element.setVisible(false);
        this.idleElements.add(element);
        this.accuracyValue = new TextElement(this.getWidth(), this.getHeight() / 2,
                             Color.green, "Accuracy: ", true);
        this.accuracyValue.setVisible(false);
        this.idleElements.add(this.accuracyValue);

        // Also add it to the display element list.
        for (int count = 0; count < this.idleElements.size(); count++)
        {
            this.displayElements.put(this.idleElements.get(count).toString(),
                                     this.idleElements.get(count));
        }

        // Create the user message, but don't add it to the list yet
        userMessage = new TextElement(this.getWidth(),  this.getHeight() / 2 - 80, Color.yellow,
"", true);

        // Create the display elements for all the missiles
        this.missileImages = new ImageElement[MISSILE_COUNT];
        for (int missile = 0; missile < MISSILE_COUNT; missile++)
        {
            ImageElement imageElement = new ImageElement(
                this.getWidth() - 85 + (missile * this.missileImage.getWidth(this)),
                this.getHeight() - 45, this.missileImage, this);
            this.displayElements.put("Missile_" + missile, imageElement);
            this.missileImages[missile] = imageElement;
        }

        // Create and display element for all lives
        this.lifeImages = new ImageElement[LIFE_COUNT];
        for (int playerLifeCount = 0; playerLifeCount < LIFE_COUNT; playerLifeCount++)
        {
            ImageElement imageElement = new ImageElement(
                    70 + (playerLifeCount * this.lifeImage.getWidth(this)),
                    this.getHeight() - 40, this.lifeImage, this);
            this.displayElements.put("Lives_" + playerLifeCount, imageElement);
            this.lifeImages[playerLifeCount] = imageElement;

            // Hide if more than PLAYER_LIVES
            if (playerLifeCount >= PLAYER_LIVES)
                imageElement.setVisible(false);
        }
    }


    //---------------------------------------------------------------------------
    // Loads all necessary graphics and audio files
    private void initGraphicsAndAudio()
    {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        MediaTracker tracker = new MediaTracker(this);

        // Initialize the audio and image files
        try {
            this.shootSound     = Applet.newAudioClip(new URL(getCodeBase(), "Shoot.wav"));
            this.explosionSound = Applet.newAudioClip(new URL(getCodeBase(), "Explosion.wav"));
            this.levelUpSound   = Applet.newAudioClip(new URL(getCodeBase(), "LevelUp.wav"));
            this.emptySound     = Applet.newAudioClip(new URL(getCodeBase(), "Empty.wav"));
            this.rumbleSound    = Applet.newAudioClip(new URL(getCodeBase(), "Rumble.wav"));
            this.missedSound    = Applet.newAudioClip(new URL(getCodeBase(), "Missed.wav"));
            this.powerUpSound   = Applet.newAudioClip(new URL(getCodeBase(), "PowerUp.wav"));

            this.missileImage   = toolkit.getImage(new URL(getCodeBase(), "Missile.jpg"));
            tracker.addImage(this.missileImage, 1);
            this.lifeImage      = toolkit.getImage(new URL(getCodeBase(), "World.JPG"));
            tracker.addImage(this.lifeImage, 2);
        } catch (MalformedURLException ex) {
        }

        // Wait for all images to load.
        try
        {
           tracker.waitForAll();
        }
        catch(Exception e)
        {
           System.err.println("Unknown error while loading images");
        }
    }

    //---------------------------------------------------------------------------
    // Initialize the data for the game instructions
    private void initGameInstructions()
    {
        this.outputTextArea.setWrapStyleWord(true);
        this.outputTextArea.setLineWrap(true);
        this.outputTextArea.setEditable(false);

        this.outputTextArea.append("Thank you for using KeyDrop Typing Tutor Game.\n\n");
        this.outputTextArea.append("Introduction:\n");
        this.outputTextArea.append("This application allows you to increase your typing skills
while having fun at the same time.  It basically is a game.\n\n");
        this.outputTextArea.append("Bombs in the form of letters or words will drop down from
the top of the screen and try to hit its target at the bottom.  ");
```

```
        this.outputTextArea.append("Shoot the bombs down by hitting the correct key on the
keyboard correspoding the letter or the highlighted letter in the word.  ");
        this.outputTextArea.append("Should the bomb hit the letters at the bottom of the screen
the player will lose one life.\n\n");
        this.outputTextArea.append("There are three difficulty levels:");
        this.outputTextArea.append("   Easy - only letters will fall, multiple bombs are
unlikely" );
        this.outputTextArea.append("   Moderate - letters and easy to spell words, multiple
bombs possible");
        this.outputTextArea.append("   Advance - letters and hard to spell words, multiple bombs
most certainly");
        this.outputTextArea.append("Game Rules:\n");
        this.outputTextArea.append("- A total of three lives will be given per game.\n");
        this.outputTextArea.append("- You lose a life if a letter bomb hits its target.\n");
        this.outputTextArea.append("- You will be given only 5 missiles to use per life.\n");
        this.outputTextArea.append("- You use a missile each time you fire.\n");
        this.outputTextArea.append("- You will get a bonus missile each time you hit the correct
key.\n");
        this.outputTextArea.append("- If you hit the incorrect key, you've lost a missile.\n");
        this.outputTextArea.append("- If you lose all missiles, the dropping missile will
eventually hit its target and you will lose a life.\n");
        this.outputTextArea.append("- Each bomb destroyed is 5 points.  In the case of a word
bomb, each letter in the word is 5 points.\n");
        this.outputTextArea.append("- The speed and level of the game play will increase at
regular intervals.\n");
        this.outputTextArea.append("- An extra life will be given every 500 points.\n");
        this.outputTextArea.append("- Game play will continue until all lives are lost.\n\n");
        this.outputTextArea.append("Controls:\n");
        this.outputTextArea.append("F1 - Opens this screen\n");
        this.outputTextArea.append("F2 - Start/End Game\n\n");
        this.outputTextArea.append("F3 - Change Difficulty");
        this.outputTextArea.append("About KeyDrop Typing Tutor Game:\n");
        this.outputTextArea.append("Author: Angelo Rodriguez\n");
        this.outputTextArea.append("Date: February 11, 2005\n");
        this.outputTextArea.append("This program was developed as a school project for CIS-18A
(Java: Objects) under Dr. Mark Lehr at Riverside Community College.");
    }

    //--------------------------------------------------------------------------
    // Mutator for score
    private void setScore(int score)
    {
        this.score = score;
        this.scoreValue.setText(Integer.toString(this.score));
    }

    //--------------------------------------------------------------------------
    // Accessor for score
    private int getScore()
    {
        return this.score;
    }

    //--------------------------------------------------------------------------
    // Mutator for level field
    private void setLevel(int level)
    {
        this.gameLevel  = level;
        this.levelValue.setText(Integer.toString(this.gameLevel));
    }

    //--------------------------------------------------------------------------
    // Accessor for level field
    private int getLevel()
    {
        return this.gameLevel;
    }

    //--------------------------------------------------------------------------
    // Mutator for High Score field
    private void setHighScore(int highScore)
    {
        this.highScore = highScore;
        this.highScoreValue.setText("High Score: " + this.highScore);
    }

    //--------------------------------------------------------------------------
    // Accessor for High Score field
    private int getHighScore()
    {
        return this.highScore;
    }

    //--------------------------------------------------------------------------
    // Accessor for game playing field
    private boolean isGamePlaying()
    {
        return this.gamePlaying;
    }

    //--------------------------------------------------------------------------
    // Mutator for game playing field
    private void setGamePlaying(boolean gamePlaying)
    {
        this.gamePlaying = gamePlaying;

        // Switch elements which are valid or invalid for display.
        for (int count = 0; count < this.idleElements.size(); count++)
        {
            DisplayElement element = (DisplayElement)this.idleElements.get(count);
            element.setVisible(!this.gamePlaying);
        }

        // Make sure that the user message is not on the list if the game has stopped
        if (!this.gamePlaying)
        {
            if (displayElements.containsKey("User"))
                displayElements.remove("User");
        }
    }

    //--------------------------------------------------------------------------
    // Mutator for missile count field.
    private void setMissileCount(byte missileCount)
    {
        this.missileCount = missileCount;

        for (int missile = 0; missile < MISSILE_COUNT; missile++)
        {
            missileImages[missile].setX(this.getWidth() - 85 +
                                        (missile * missileImage.getWidth(this)));
            missileImages[missile].setVisible(missile < this.missileCount);
        }
    }

    //--------------------------------------------------------------------------
    // Accessor for the missile count field
    private byte getMissileCount()
    {
        return this.missileCount;
    }


    //--------------------------------------------------------------------------
    // Mutator for player life count
    private void setPlayerLives(byte playerLives)
    {
        this.playerLives = playerLives;

        for (int playerLifeCount = 0; playerLifeCount < LIFE_COUNT; playerLifeCount++)
        {
            lifeImages[playerLifeCount].setVisible(playerLifeCount < this.playerLives);
            lifeImages[playerLifeCount].setX(70 + (playerLifeCount * 
this.lifeImage.getWidth(this)));
        }
    }

    //-------------------------------------------------------------------------
    // Accessor for player life count
    private byte getPlayerLives()
    {
        return this.playerLives;
    }

    //-------------------------------------------------------------------------
    // Mutator for difficulty field
    private void setDifficulty(byte difficulty)
    {
        this.difficulty = difficulty;
        // Check if we need to wrap around the text.
        if (difficulty >= 3)
            this.difficulty = 0;

        // Configure the appropriate display elements
        this.difficultyText.setText(DIFFICULTY_TEXT[this.difficulty]);
        this.difficultyText.setColor(DIFFICULTY_COLOR[this.difficulty]);
    }

    //-------------------------------------------------------------------------
    // Accessor for difficulty field
    private byte getDifficulty()
    {
        return this.difficulty;
    }

    //-------------------------------------------------------------------------
    // Starts the timer that controls the game
    public void startGame()
    {
        // Initializes all game variables.
        this.timerCount = 0;
        this.shotsFired = 0;
        this.shotsHit   = 0;
        setPlayerLives(PLAYER_LIVES);
        setMissileCount(MISSILE_COUNT);
        setLevel(1);
        setScore(0);

        // Create the first bomb.
        createBomb(true);

        // Start the game timer.
        this.gameTimer.setDelay(TIMER_UPDATE);
        this.gameTimer.start();

        // Keep track if game play is ongoing.
        setGamePlaying(true);

        // Start rumbling sound
        this.rumbleSound.loop();

        this.repaint();
    }


    //-------------------------------------------------------------------------
    // Stops the timerr that controls the game
    public void stopGame()
    {
        DecimalFormat percentage = new DecimalFormat("0.00%");
        // Calculate accuracy
        double accuracy = (double)shotsHit / (double)shotsFired;
        this.accuracyValue.setText("Accuracy: " + percentage.format(accuracy));

        // Clear all bombs
        this.bombs.clear();

        // Stop the rumbling sound
        this.rumbleSound.stop();

        // Signal the game to stop
        setGamePlaying(false);

        // Stop timer
        this.gameTimer.stop();

        // Keep track of high schore
        if (getScore() > getHighScore())
            setHighScore(getScore());

        // Redraw the entire game screen
        this.repaint();
    }


    //----------------------------------------------------------------------------
    // Implementation for ActionListener, used on timer events.
    public void actionPerformed(ActionEvent e)
    {
        // Detect if the timer event sent the message.
        if (e.getSource() == this.gameTimer)
        {
            // Move missile.position down by one
            for (int bombCounter = 0; bombCounter < bombs.size(); bombCounter++)
            {
                Bomb bomb = (Bomb)this.bombs.get(bombCounter);
                bomb.move(DROP_INCREMENT * getLevel(), getGraphics());
            }

            // Increment timerCount so that we know time elapsed since the
            // beginning of the game.
            this.timerCount += this.gameTimer.getDelay();

            // Reduce user message timer if available, make sure it's positive
            this.messageTime -= this.gameTimer.getDelay();
            if (messageTime < 0)
            {
                this.messageTime = 0;
                displayElements.remove("User");
            }

            // Reduce the laser timer if available, make sure it's positive
            if (this.fireTime > 0)
            {
                this.fireTime -= this.gameTimer.getDelay();
                if (this.fireTime < 0) {
                    this.fireTime = 0;
                    this.repaint();
                }
            }

            // Determine if the dropping bomb is below the line.
            for (int bombCounter = 0; bombCounter < bombs.size(); bombCounter++)
            {
                Bomb bomb = (Bomb)bombs.get(bombCounter);
                if (bomb.getBombPosition() >= INIT_Y_POSITION) {
                    // Play big explosion sound
                    this.explosionSound.play();
                    // Draw animation of letter exploding

                    // Destroy all bombs, start anew
                    destroyBomb(null);

                    // Deduct 1 life from player
                    setPlayerLives((byte) (getPlayerLives() - 1));

                    // Determine if play should continue.
                    if (getPlayerLives() > 0) {
                        // Reset Missile Count and create new bomb.
                        setMissileCount(MISSILE_COUNT);
                        createBomb(true);
                    } else {
                        // Otherwise, show game over and Stop the game timer.
                        stopGame();
                    }
                }
            }

            // Randomly see if we need to create another bomb
            createBomb(false);

            // Draw the entire screen if there isn't a bomb
            if (this.bombs.size() == 0)
                this.repaint();
        }
    }


    //---------------------------------------------------------------------------
    // Draws the playing field.
    public void paint(Graphics g)
    {
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, this.getWidth(), this.getHeight());

        // Change the color to gray and draw borders.
        g.setColor(Color.getHSBColor(0.01f, 0.80f, 0.20f));
        g.fillRect(2, INIT_Y_POSITION - Y_INCREMENT, this.getWidth() - 4, Y_INCREMENT *
TOTAL_ROWS + 5);

        // Make the font bigger and change the color to white.
        g.setColor(Color.getHSBColor(0.241f, 0.20f, 0.96f));
        g.setFont(keyFont);

        // Draw all static element on screen..
        for (Enumeration e = this.displayElements.elements(); e.hasMoreElements();)
        {
            DisplayElement element = (DisplayElement)e.nextElement();
            element.draw(g);
        }

        // Draw the missile if instructed to do so by timer
        if (this.fireTime > 0)
        {
            TextElement letter =
(TextElement)this.displayElements.get(Character.toString(this.fireLetter));

            if (letter != null)
            {
                g.setColor(Color.pink);
                g.drawLine(lastBombX + 7, this.firePosition + 20,
                           letter.getX() + 7, INIT_Y_POSITION - 30);
                g.drawImage(this.missileImage, lastBombX,
                            this.firePosition, this);
            }
        }

        // Focus needs to be kept on the applet so that keyboard input works.
        requestFocus();
    }


    //--------------------------------------------------------------------------
    // Creates a new bomb to shoot.
    private void createBomb(boolean forceCreate)
    {
        // Don't create more bombs than limit
        if (this.bombs.size() >= MAX_BOMB_COUNT)
            return;

        // Random generator of bombs
        if (!forceCreate)
        {
            int nGenerate = 0;

            // Randomly generate one from time to time
            // Factor in that we have bombs already, make it harder to create
            // bombs when more are on screen
            switch (this.difficulty)
            {
                case 0: nGenerate = (int)(Math.random() * 50000 * this.bombs.size());
                case 1: nGenerate = (int)(Math.random() * 5000 * this.bombs.size());
                case 2: nGenerate = (int)(Math.random() * 500 * this.bombs.size());
            }

            if (nGenerate != 100)
                return;
        }

        boolean createWord    = false;
        boolean difficultWord = false;

        switch (this.difficulty)
        {
            // case 0: This will only create letters, similar to the old game in project 1.
            case 1:
                // Create an easy word 50% of the time
                createWord = (int)(Math.random() * 2) != 1;
                break;
            case 2:
                // Create a word 50% of the time
                createWord = (int)(Math.random() * 2) != 1;

                // Generate difficult word 90% of the time.
                difficultWord = (int)(Math.random() * 10) != 1;
                break;
        }

        // Determine whether we will display a letter or character.
        if (createWord)
        {
            // Randomly generate a word bomb.
            this.bombs.add(new WordBomb(this.getWidth(), difficultWord));
        }
        else
        {
            // Randomly generate a letter bomb.
            Bomb bomb = new LetterBomb();
            this.bombs.add(bomb);

            // Align the bomb with the letter on the keyboard.
            TextElement letter =
(TextElement)this.displayElements.get(Character.toString(bomb.getBombKey()));
            bomb.setX(letter.getX());
        }

        // Redraw
        this.repaint();
    }


    //----------------------------------------------------------------------------
    // Destroys the current bomb.
    private void destroyBomb(Bomb bomb)
    {
        if (bomb != null)
        {
            // Make the bomb key invalid for drawing.
            this.bombs.remove(bomb);
        }
        else
        {
            // Clear all bombs
            this.bombs.clear();
        }
        // Redraw
        this.repaint();
    }


    //----------------------------------------------------------------------------
    // Increment the players score and also determine whether to level up and
    // power up.
    private void addPoint()
    {
        setScore(getScore() + SCORE_INCREMENT);

        // Increase the speed, and level every 30 seconds.
        if (this.timerCount / (LEVEL_UP_TIME * 1000) != getLevel() - 1)
        {
            this.gameTimer.setDelay(this.gameTimer.getDelay() - TIMER_INCREMENT);
            setLevel(getLevel() + 1);

            // Play new level sound
            this.levelUpSound.play();

            // Tell user that he has leveled up
            displayUserMessage("Level " + getLevel() + "!", MESSAGE_TIME);
        }

        // Increase players life every 200 points.
        if (this.getScore() % POWER_UP_SCORE == 0)
        {
            setPlayerLives((byte)(getPlayerLives() + 1));

            // Play new life sound
            this.powerUpSound.play();

            // Tell user that he has an extra life
            displayUserMessage("Power Up! Extra Life!", MESSAGE_TIME);
        }
    }


    //----------------------------------------------------------------------------
    // Instructs the screen to display a user message for a specified amount of
    // time in milliseconds.
    private void displayUserMessage(String message, int timeOut)
    {
        this.userMessage.setText(message);

        // Set the time limit for the message to display.
        this.messageTime = timeOut * 1000;

        // Add the message to the list of elements to draw.
        this.displayElements.put("User", this.userMessage);
    }
    //--------------------------------------------------------------------------
    // Implementation for key listener, triggered when key is pressed.
    public void keyPressed(KeyEvent e) {
        switch (e.getKeyCode())
        {
        case KeyEvent.VK_F1:
            // Bring up help screen
            if (!isGamePlaying())
            {
                // Declare a scroll pane to allow the text area to scroll
                JScrollPane scrollPane = new JScrollPane(outputTextArea,
                    JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                    JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

                // Output the game play information.
                JOptionPane.showMessageDialog(null, scrollPane,
                    "KeyDrop Typing Game Help", JOptionPane.PLAIN_MESSAGE);
            }
            break;
        case KeyEvent.VK_F2:
            // Start/Stop the game
            if (isGamePlaying())
                stopGame();
            else
                startGame();
            break;
        case KeyEvent.VK_F3:
            // Set Difficulty
            setDifficulty((byte)(getDifficulty() + 1));
            this.repaint();
            break;
        case KeyEvent.VK_F8:
            // Pause game, undocumented.  Just using it for screen captures.
            if (isGamePlaying())
            {
                if (this.gameTimer.isRunning())
                    this.gameTimer.stop();
                else
                    this.gameTimer.start();
            }
            break;
        default:
            // Some other key was pressed, lets see if its our key.
            if (isGamePlaying() && this.gameTimer.isRunning()) {
                if (getMissileCount() > 0) {
                    // Increment total number of shots fired for statistics
                    this.shotsFired++;

                    // Deduct 1 from player’s missile count since he fired.
                    setMissileCount((byte)(getMissileCount() - 1));

                    // Triggers the animation of player’s missile to dropping bomb
                    this.fireTime = FIRE_TIME;
                    this.fireLetter = Character.toUpperCase(e.getKeyChar());

                    // Determine if the keystroke is correct
                    Bomb bomb = null;

                    for (int bombCounter = 0; bombCounter < bombs.size() && bomb == null;
bombCounter++)
                    {
                        if (((Bomb)(this.bombs.get(bombCounter))).hitTest(this.fireLetter))
                             bomb = (Bomb)this.bombs.get(bombCounter);
                    }

                    if (bomb != null)
                    {
                        // Set the total number of shots hit
                        this.shotsHit++;
                        // Remember where the bomb is
                        this.lastBombX = bomb.getX();

                        // Play shooting sound
                        this.shootSound.play();

                        // The missile will be displayed next to the letter.
                        this.firePosition = bomb.getBombPosition();

                        // Add 1 to player’s missile count since he hit it
                        setMissileCount((byte)(getMissileCount() + 1));

                        // Add Point To Score
                        addPoint();

                        if (bomb.isDestroyed())
                        {
                            // Destroy the dropping bomb
                            destroyBomb(bomb);

                            // Create a new Missile
                            createBomb(true);
                        }
                    } else {
                        // Calculate position of letter.
                        TextElement letter =
(TextElement)this.displayElements.get(Character.toString(this.fireLetter));
                        this.lastBombX = letter.getX();

                        // Play missed sound
                        this.missedSound.play();

                        // The missile will be displayed near the top.
                        this.firePosition = -20;
                    }
                } else {
                    // Play empty barrel click
                    this.emptySound.play();
                }
            }

            this.repaint();
        }
    }


    //------------------------------------------------------------------------
    // Implementation for key listener, triggered when key is released.
    public void keyReleased(KeyEvent e) {
    }


    //------------------------------------------------------------------------
    // Implementation for key listener, triggered when key is typed.
    public void keyTyped(KeyEvent e) {
    }
}
```

6.2 DisplayElement.java

```
import java.awt.Graphics;
import java.awt.Color;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
```

```java
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: DisplayElemnt</p>
 * <p>Class Description: Base class for all display elements within the system.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/

public abstract class DisplayElement {
    private int x;             // Horizontal position of display element
    private int y;             // Vertical position of the element
    private Color color;       // Color of the element, ignored if image.

    private boolean visible  = false;     // Determines whether to draw the item or not.

    //--------------------------------------------------------------------------
    // Mutator for x field.
    public void setX(int x)
    {
        this.x = x;
    }

    //--------------------------------------------------------------------------
    // Accessor for x field
    public int getX()
    {
        return this.x;
    }

    //--------------------------------------------------------------------------
    // Mutator for y field
    public void setY(int y)
    {
        this.y = y;
    }

    //--------------------------------------------------------------------------
    // Accessor for y field
    public int getY()
    {
        return this.y;
    }

    //--------------------------------------------------------------------------
    // Accessor for the color field.
    public Color getColor()
    {
        return color;
    }

    //--------------------------------------------------------------------------
    // Mutator for the color field.
    public void setColor(Color color)
    {
        this.color = color;
    }

    //--------------------------------------------------------------------------
    // Mutator for the visible field.
    public void setVisible(boolean visible)
    {
        this.visible = visible;
    }

    //--------------------------------------------------------------------------
    // Accessor for the visible field.
    public boolean isVisible()
    {
        return this.visible;
    }

    //------------------------------------------------------------------------------
    // draw() will display, polymorphicly, this display element.
    public abstract void draw(Graphics g);
}
```

6.3 TextElement.java

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.FontMetrics;
import java.awt.geom.Rectangle2D;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: TextElement</p>
 * <p>Class Description: Concrete derived class which can be used to display
 *     any text.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/

public class TextElement extends DisplayElement {
    protected String text      = "";      // String containing the text to display.
    protected boolean centered = false;   // Determines whether to horizontally center the item
on screen.

    //------------------------------------------------------------------------------
    // Constructor
    public TextElement(int x, int y, Color color, String text)
    {
        this(x, y, color, text, false);
    }

    //------------------------------------------------------------------------------
    // Constructor with centered parameter.
    public TextElement(int x, int y, Color color, String text, boolean centered)
    {
        setX(x);
        setY(y);
        setColor(color);
        setText(text);
        setCentered(centered);
        setVisible(true);
    }

    //------------------------------------------------------------------------------
    // Accessor to the letter field
    public String getText()
    {
        return text;
    }

    //------------------------------------------------------------------------------
    // Mutator for the letter field
    public void setText(String text)
    {
        this.text = text;
    }

    //------------------------------------------------------------------------------
    // Mutator for centered field.
    public void setCentered(boolean centered)
    {
        this.centered = centered;
    }

    //---------------------------------------------------------------------------
    // Accessor for centered field.
    public boolean isCentered()
    {
        return this.centered;
    }

    //---------------------------------------------------------------------------
    // Implementation for toString()
    public String toString()
    {
        return getText();
    }

    //---------------------------------------------------------------------------
    // Implementation for the draw method
    public void draw(Graphics g) {
        if (isVisible())
        {
            g.setColor(getColor());
            if (isCentered())
                drawStringCenter(g);
            else
                g.drawString(getText(), getX(), getY());
        }
    }

    //---------------------------------------------------------------------------
    // Draw a string centererd horizontally on the screen, but using a
    // paramenter for y.
    private void drawStringCenter(Graphics g)
    {
        // We need to font metrics to center text on screen
        FontMetrics fontMetrics = g.getFontMetrics();

        // Calculate the size of the string.
        Rectangle2D rectSize = fontMetrics.getStringBounds(getText(), g);

        // Center the text horizontally, but using passed in y parameter for vertical position.
        g.drawString(getText(), getX() / 2 - (int)rectSize.getCenterX(), getY());
    }
}
```

6.4 *ImageElement.java*

```
import java.awt.Image;
import java.awt.image.ImageObserver;
import java.awt.Graphics;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: ImageElement</p>
 * <p>Class Description: Concrete derived class which can be used to display
 *     any image.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/

public class ImageElement extends DisplayElement {
    Image image;
    ImageObserver observer;

    //--------------------------------------------------------------------------
    // Constructor
    public ImageElement(int x, int y, Image image, ImageObserver observer)
    {
        setX(x);
        setY(y);
        setImage(image);
        setObserver(observer);
        setVisible(true);
    }

    //--------------------------------------------------------------------------
    // Mutator for image field
    public void setImage(Image image)
    {
        this.image = image;
    }

    //--------------------------------------------------------------------------
    // Accessor for image field
    public Image getImage()
    {
        return this.image;
    }

    //--------------------------------------------------------------------------
    // Mutator for observer field
    public void setObserver(ImageObserver observer)
    {
        this.observer = observer;
    }

    //--------------------------------------------------------------------------
    // Implementation for drawing element
    public void draw(Graphics g)
    {
        if (isVisible())
            g.drawImage(getImage(), getX(), getY(), this.observer);
    }
}
```

6.5 Bomb.java

```
import java.awt.Graphics;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: Bomb</p>
 * <p>Class Description: Abstract base class class which represents a bomb.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/
public abstract class Bomb extends DisplayElement {
    private boolean destroyed = false;    // Determines if the bomb has been destroyed

    //--------------------------------------------------------------------------
    // Displays the bomb using the graphics engine.
    public void show(Graphics g)
    {
        setVisible(true);
    }
    //--------------------------------------------------------------------------
    // Hides the bomb using the graphics engine.
    public void hide(Graphics g)
    {
        setVisible(false);
    }

    //--------------------------------------------------------------------------
    // Determines if it's a hit
    public abstract boolean hitTest(char character);

    //--------------------------------------------------------------------------
    // Gets the next bomb key for this bomb.
    public abstract char getBombKey();

    //--------------------------------------------------------------------------
    // Returns the current position of the bomb
    public int getBombPosition()
    {
        return getY();
    }

    //--------------------------------------------------------------------------
    // Accessor to the destroyed field.
    public boolean isDestroyed()
    {
        return this.destroyed;
    }

    //--------------------------------------------------------------------------
    // Mutator to the destroyed field.
    public void setDestroyed(boolean destroyed)
    {
        this.destroyed = true;
    }

    //--------------------------------------------------------------------------
    // Move is polymorphic, hiding the bomb first before with the derived
    // class method Hide(), changing the position, the making it visible again
    // with the derived class' Show() method.
    public void move(int height, Graphics g)
    {
        hide(g);

        setY(getY() + height);

        show(g);
    }

    //--------------------------------------------------------------------------
    // Implementation of the draw function.
    public void draw(Graphics g)
    {
        if (isVisible())
            show(g);
    }
}
```

6.6 *LetterBomb.java*

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
```

```java
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: LetterBomb</p>
 * <p>Class Description: Concrete derived class which represents a bomb with one letter.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/

public class LetterBomb extends Bomb {
    private char bombKey;                   // The currently dropping bomb letter

    // Font to be used on screen instead of the lame default
    private Font keyFont = new Font("Arial", Font.BOLD, 20);

    //----------------------------------------------------------------------------
    // Constructor
    public LetterBomb()
    {
        this.bombKey = (char)('A' + (char)(Math.random() * 26));

        this.setX(0);
        this.setY(0);
    }

    //----------------------------------------------------------------------------
    // Retrieves the bomb key
    public char getBombKey()
    {
        return bombKey;
    }

    //----------------------------------------------------------------------------
    // Displays the letter bomb
    public void show(Graphics g)
    {
        super.show(g);

        g.setColor(Color.white);
        g.setFont(keyFont);
        g.drawString(Character.toString(getBombKey()), getX(), getBombPosition());
    }

    //----------------------------------------------------------------------------
    // Hides the letter bomb
    public void hide(Graphics g)
    {
        super.hide(g);

        g.clearRect(getX(), getY() - 20, 30, 30);
    }

    //----------------------------------------------------------------------------
    // Determine if this bomb was hit with specified character.
    public boolean hitTest(char character) {
        boolean hit = false;

        if (character == this.bombKey)
        {
            hit = true;
            setDestroyed(true);
        }

        return hit;
    }
}
```

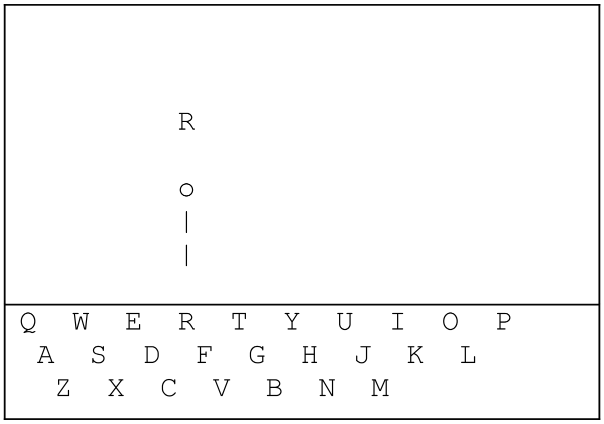
6.7 WordBomb.java

```java
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
import java.awt.geom.Rectangle2D;
import java.awt.FontMetrics;

/**
 * <p>Title: KeyDrop Typing Tutor Applet (Project 2)</p>
 * <p>Description: A missile command simulation game which drops letters instead
 *     of missiles, shoot them down using keys on the keyboard.</p>
 * <p>Class Name: WordBomb</p>
 * <p>Class Description: Concrete derived class which represents a bomb in the word format.</p>
 * <p>Class: CIS-18A 83928</p>
 * <p>Due Date: February 11, 2005</p>
 * @author Angelo Rodriguez
*/


public class WordBomb extends Bomb {
    private String bombWord;                    // The currently dropping bomb letter
    private int index = 0;
    private int[] positions = null;
    private Rectangle2D bounds;

    // Font to be used on screen instead of the lame default
    private Font keyFont = new Font("Arial", Font.BOLD, 20);

    private final static String[] EASY_WORDS =  new String[]
    {
        "again", "animal", "avenue", "become", "believe", "bid", "boot", "bright", "burst",
        "bury", "bushes", "busy", "cast", "chest", "chili", "coffee", "crate", "curl", "dainty",
        "dinner", "dress", "drink", "drug", "duty", "easy", "enough", "excuse", "fight",
"fired",
        "flow", "flower", "glass", "governor", "how", "issue", "lecture", "light", "lion",
"lock",
        "log", "mess", "narrate", "only", "open", "path", "pear", "pint", "planet", "poise",
"pour",
        "reach", "remain", "row", "rush", "saddle", "sandwich", "should", "shy", "slept",
        "slipper", "slippery", "slow", "smart", "soft", "spark", "stand", "stood", "stump",
        "tax", "tip", "toad", "toy", "trim", "tug", "walk", "wheel", "wild", "write", "young",
        "zebra", "hello", "when",  "arise", "banjo", "adopt", "solo", "who", "what", "where"
    };

    private final static String[] DIFFICULT_WORDS = new String[]
    {
        "acclimate", "agitation", "allowance", "asylum", "autonomy", "bounty", "bumptious",
        "civilian", "clemency", "confetti", "conglomerate", "conservation", "conundrum",
        "copious", "culminate", "development", "dimension", "disown", "disseminate", "engulf",
        "extremely", "heartily", "henna", "incantation", "inhibit", "insularity",
"intentionally",
        "intonation", "jobber", "minimize", "modest", "naturalize", "overhead", "perverse",
        "physician", "pollen", "prose", "qualify", "readily", "recommendation", "redundant",
        "reliable", "replacement", "replica", "resign", "respectfully", "rigid", "rigorous",
        "sever", "strumpet", "surpass", "sycophant", "syndicate", "television", "unappealing",
        "undecided", "unfortunately", "utilitarian", "western", "difficult", "resource",
"steady",
        "introduce", "accident", "appreciate", "sculptor", "illustration",  "perish",
"escapade",
        "serious", "preference", "efficient", "cabinet", "trousers", "division", "violence"
    };


    //------------------------------------------------------------------------
    // Constructor, accepts the width of the window to display and whether or
    // not to generate a difficult word.
    public WordBomb(int width, boolean difficult)
    {
        if (difficult)
```

```
            bombWord = DIFFICULT_WORDS[(int)(Math.random() *
DIFFICULT_WORDS.length)].toUpperCase();
        else
            bombWord = EASY_WORDS[(int)(Math.random() * EASY_WORDS.length)].toUpperCase();

        // Generate random position for X
        setX((int)(Math.random() * width));

        // Ensure that the word isn't outside the window
        if (getX() + (bombWord.length() * 20) > width)
        {
            // Move the word, the word lengths away from the right edge, this
            // is better than just right alignment, we really don't want
            // it to be always right-aligned.
            setX(getX() - (bombWord.length() * 20));
        }
    }

    //--------------------------------------------------------------------------
    // Displays the word with the next character to type in white, the rest in gray
    public void show(Graphics g) {
        super.show(g);

        // Set the font for this bomb
        g.setFont(keyFont);

        // We need to font metrics to center text on screen
        FontMetrics fontMetrics = g.getFontMetrics();

        // Calculate position of each character
        if (positions == null)
        {
            positions = new int[bombWord.length()];
            positions[0] = 0;
            for (int count = 1; count < bombWord.length(); count++)
            {
                // Calculate the size of the string.
                Rectangle2D rectSize = fontMetrics.getStringBounds(bombWord.substring(0, count),
g);

                positions[count] = (int)rectSize.getWidth();

                // Store the bounds of the last entire word
                bounds = rectSize;
            }
        }

        // Draw the entire word in gray first.
        g.setColor(Color.gray);
        g.drawString(bombWord, getX(), getBombPosition());

        // Draw the letter that needs to be hit in white
        g.setColor(Color.white);
        g.drawString(Character.toString(bombWord.charAt(index)),
                     getX() + positions[index], getBombPosition());
    }

    //--------------------------------------------------------------------------
    // Hides the current word
    public void hide(Graphics g) {
        super.hide(g);

        // Clear the area where the word was shown.
        if (bounds != null)
            g.clearRect(getX(), getY() - 20, getX() + (int)bounds.getWidth(), 30);
    }

    //--------------------------------------------------------------------------
    // Returns the next key to type
    public char getBombKey() {
        return bombWord.charAt(index);
    }

    //-------------------------------------------------------------------------
    // Detects whether or not the character typed has been hit, also determines
    // if the bomb has been destroyed and sets a flag.
    public boolean hitTest(char character) {
        boolean hit = false;

        // Check if the bomb character matches keystroke character
        if (bombWord.charAt(index) == character)
        {
            hit = true;
            index++;
        }

        // If all letters have been typed, flag the bomb for destruction
        if (index >= bombWord.length())
            setDestroyed(true);

        return hit;
    }
}
```

7 Copy of Original Proposal

Typing Game
I'd like to create a text based (possibly graphical, but not necessary) typing game that has a randomly falling letter. The object of the game is to type the letter (and shoot it down) before it reaches the line, if it does, the player loses a life. There will be 3 lives per game. Here's a conceptual screen shot:



```
             R

             o
             |
             |
Q   W   E   R   T   Y   U   I   O   P
  A   S   D   F   G   H   J   K   L
    Z   X   C   V   B   N   M
```

Features that I may add if time permits:

- Scoring – 1 point per letter
- Levels – increasing falling speed per level
- Case – lower- and uppercase letters
- Multiple letters – more than one letter on screen

Possibly, the additional features could be my second project, in addition to converting it into a purely OO design.

Concepts that will probably be used:

- Arrays
- Conditional statements
- Loops
- Random Number Generator
- Methods

Concepts I’ll have to learn:

- Timers
- Keyboard Input
- Text graphics (It may actually be easier to implement using just DrawString)