Snowman – Java

Part 1 – Game Framework

This project will take several weeks to complete, with enough time for advanced students to customise and extend the solution with extra features.

The project is to build a game that has a pay-to-play feature. The game this tutorial builds is called “Snowman”, a modern variation on the classic “Hangman” game, but slightly less macabre.

The object of the game is to guess a random, secret word, one letter at a time. When the player guesses a letter of the secret word, it’s revealed. If the player reveals the entire secret word, they win. If a player guesses a letter than isn’t in the word then part of the snowman is revealed. If the snowman is completed then the game is lost.

# Game Structure

The game is split in to two classes (or files):

1. A Program class, that is the entry point to the game. An entry point is where the code starts executing.
2. A SnowmanGame class. This is a special type of class that is represents a Graphics User Interface (GUI) component.

We need to start by defining the Program class.

## Program Class

The program class contains lots of Java-specific “boilerplate” code, which unfortunately is necessary for the game to work. For now, just type this in as shown.

**package** elycc.snowman;

**import** java.awt.EventQueue;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**public** **class** Program {

/\*\*

\* The entry point for the program.

\*/

**public** **static** **void** main(String[] args) {

EventQueue.*invokeLater*(**new** Runnable() {

@Override

**public** **void** run() {

// Create a window, or "frame" to put on the screen.

JFrame frame = **new** JFrame("Snowman");

// Create a new SnowmanGame object from the class

JPanel game = **new** SnowmanGame();

// Add the new game object to the frame

frame.add(game);

// Resize the window so that the SnowmanGame object fits

frame.pack();

// Put the frame in the middle of the screen

frame.setLocationRelativeTo(**null**);

// Configure the frame to exit if we click the "X" icon

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

// Finally, show the frame.

frame.setVisible(**true**);

}

});

}

}

Type in the code, but note that it won’t compile yet. There should be a red line underneath the line that creates the new SnowmanGame() instance. This is because we haven’t created this class yet.

## SnowmanGame Class

Create a new class called SnowmanGame, make sure you get the upper and lower-case letters correct and that it matches what you typed in the Program class, above. Type in the following code, which defines the structure of the SnowmanGame class.

While typing, work out the answers to the following questions:

1. What is the calculation for smX doing? Where will the picture appear?
2. Why have we refactored drawLine and drawCircle in to separate methods?
3. What effect does changing the value of scale have on the Snowman

**package** elycc.snowman;

**import** java.awt.BasicStroke;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Font;

**import** java.awt.Graphics;

**import** java.awt.Graphics2D;

**import** java.awt.RenderingHints;

**import** java.awt.Toolkit;

**import** javax.swing.JPanel;

**import** elycc.snowman.SnowmanGame.GameState;

/\*\*

\* A "panel" which fits on a frame

\*

\*/

**public** **class** SnowmanGame1 **extends** JPanel{

**public** SnowmanGame1() {

setBackground(Color.***black***);

setPreferredSize(**new** Dimension(1024, 768));

}

/\*\*

\* Called when Java decides that the window should be repainted.

\*/

@Override

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

drawGraphics(g);

}

**private** **void** drawGraphics(Graphics g) {

g.setColor(Color.***WHITE***);

drawSnowman(g);

Toolkit.*getDefaultToolkit*().sync();

}

/\*\* The scale of the snowman, adjust this for the monitor you're using

\*/

**int** smScale = 20;

/\*\* Where the Snowman will be drawn on the X-axis \*/

**int** smX;

/\*\* Where the Snowman will be drawn on the Y-axis \*/

**int** smY;

**private** **boolean** drawSnowman(Graphics g) {

smY = 10; // A little bit down from the top

smX = getWidth() / 2 - (smScale \* 10);

// Set up a pen

Graphics2D g2d = (Graphics2D) g;

g2d.setRenderingHint(RenderingHints.***KEY\_ANTIALIASING***,

RenderingHints.***VALUE\_ANTIALIAS\_ON***);

g2d.setColor(Color.***WHITE***);

g2d.setStroke(**new** BasicStroke(10));

// Ground

snDrawLine(g, 0, 20, 20, 20);

**return** **false**;

}

/\*\*

\* Draw a line from (x1,y1) to (x2,y2)

\*/

**private** **void** snDrawLine(Graphics g, **int** x1, **int** y1, **int** x2, **int** y2) {

g.drawLine(x1 \* smScale + smX, y1 \* smScale + smY,

x2 \* smScale + smX, y2 \* smScale + smY);

}

/\*\* Draw a circle at (x,y) with the passed radius.

\*/

**private** **void** snDrawCircle(Graphics g, **int** x, **int** y, **double** radius) {

g.drawOval((**int**) (smX + x \* smScale - radius \* smScale),

(**int**) (smY + y \* smScale - radius \* smScale),

(**int**) (radius \* 2 \* smScale),

(**int**) (radius \* 2 \* smScale));

}

}

When you run this code, it will draw a simple, horizontal line on the screen. You need to design your snowman now (or whatever other picture you want to draw) by adding calls to snDrawLine and snDrawCircle to the drawSnowman method.

If you get stuck, use the following code to draw a simple snowman then adapt it to your own image.

// Body

snDrawCircle(g, 10, 15, 5);

// Head

snDrawCircle(g, 10, 7, 3);

// Hat

snDrawLine(g, 8, 1, 12, 1);

snDrawLine(g, 12, 1, 12, 3);

snDrawLine(g, 12, 3, 14, 3);

snDrawLine(g, 8, 1, 8, 3);

snDrawLine(g, 6, 3, 8, 3);

snDrawLine(g, 6, 3, 6, 4);

snDrawLine(g, 14, 3, 14, 4);

snDrawLine(g, 6, 4, 14, 4);

// Buttons

snDrawCircle(g, 10, 12, 1);

snDrawCircle(g, 10, 15, 1);

snDrawCircle(g, 10, 18, 1);

// Arms

snDrawLine(g, 2, 15, 5, 15);

snDrawLine(g, 15, 15, 17, 15);

// Eyes

snDrawCircle(g, 11, 6, 0.3);

snDrawCircle(g, 9, 6, 0.3);

Part 2 – Game Mechanics

Now we have created the image, it’s time to start putting some of the game mechanics around it. The image should gradually be displayed as the player incorrected guesses letters that make up the secret word.

The way games work is that every time something changes typically the entire screen is redrawn, like drawing a cartoon frame by frame. So, every time something changes in our game (such as a letter is guessed and either a letter is revealed or another part of the snowman is drawn) we will redraw everything.

Of course, this is easy in Java, because as long as we have classes and methods to do so, it’s not really any more work for you, the programmer, to make this happen.

## Gradually drawing the Snowman

This means that our drawSnowman routine can be fairly simply modified. It just needs to draw more and more of the snowman as the number of “bad guesses” increases.

We’re going to need to keep track of the number of bad guesses made by the player so we know how much of it to draw, so add a new *instance* variable, near the top of the code (underneath the class declaration but outside of any method):

**int** badGuesses; // Track the number of bad guesses the player has made.

This needs to be reset to zero at the start of each game. We’ll be doing more initialisation later, create a method to initialise the game as follows:

**private** **void** initGame() {

badGuesses = 0;

}

Add a call to initGame() from the constructor (this is the method with the same name as the class).

Now the drawSnowman(Graphics g) method can be updated to draw more and more of the snowman as the number of badGuesses increases. Also, the game will have to know whether the snowman has been completed, so the method will return true to indicate that the snowman is complete (i.e. game over) or false (for more bad guesses allowed).

At the beginning of your drawSnowman method, initialise a counter. This is going to increase as more of the snowman is drawn:

int count = 0;

At different points in your drawSnowman method, add the following snippet:

**if** (count++ == badGuesses) **return** **false**;

Work out what this code does. This is actually a very concise was of doing the following:

**if** (count == badGuesses) {

**return** **false**;

}

count = count + 1;

So, note that the ++ operator is applied *after* the test to see if it’s the same as badGuesses.

Use either snippet to break up the drawing of the Snowman in to different stages then test out your code by setting badGuesses to different numbers to see how changing it changes the amount of your snowman that is drawn.

At the end of the method (when the drawing is complete), put a final return statement to return true instead of false. This is how the method will indicate that the drawing is complete and therefore, it’s “Game Over!”

## Selecting the Secret Word and Available Letters

For the game to work, we need two lists of letters:

1. The letters that make up the secret word.
2. The letters that the player can guess from.

Create a new instance variable called word, as follows:

/\*\* The secret word \*/

String word;

We’re going to choose a word randomly from a list of words. For now, let’s create a hard-coded array of words:

/\*\* The words that the program will select from \*/

String[] words = { "POINTLESS", "COMPUTER", "DECORATE", "CONSTRUCT", "PERIMETER" };

Add as many words as you like, but make sure you use block capitals.

Now, inside initGame, initialise word to be a random choice from words.

Random r = **new** Random();

word = words[r.nextInt(words.length)];

To allow you to easily test your game, print out the word in initGame using:

System.out.println(“Secret word is: “ + word);

Run your code to make sure that it works.

Now, let’s render the letters the player can choose from. At the beginning all 26 letters are available, but we’ll need to keep track of them as the player uses them up.

Add a new instance variable to hold all the letters.

/\*\* The letters available to the player. \*/

HashSet<Character> availableLetters;

A “Set” in Java is a collection of unique objects. The “Hash” refers to how those objects are found within the set on lookup.

To initialise the list of available letters, you’ll need to set them up in initGame.

availableLetters = **new** HashSet<Character>();

Work out how to add the letters A to Z to the set, using the following hints:

1. You’ll need a for loop.
2. An easy way to get a set of all the letters is "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray().
3. There’s a method called add on the set, to add a new member.

So that the player knows what letters are available, we need to print them on the screen. Create a new method:

**private** **void** drawAvailableLetters(Graphics g) {

**int** single = getWidth() / 26;

**int** posX = 10;

}

The first line calculates how many pixels is 1/26 of the screen (as there should be spaces for 26 letters). posX is going to keep track of where to draw each letter.

Add a for loop to the method using the same code as you did above in initGame. In the body of the loop, you want to draw the letter **only if** is the letter **is not** in availableLetters. Fortunately, there’s a method called contains on the object which returns true if the letter is in the set and false otherwise. The call to draw a string looks like this:

g.drawString(letter.toString(), posX, 600);

Remember, each time you draw a letter, you’ll need to add single to posX, or the letters will all appear on top of each other.

Also, remember that in order to see your work, you’ll need to call this new method, so add a call inside drawGraphics:

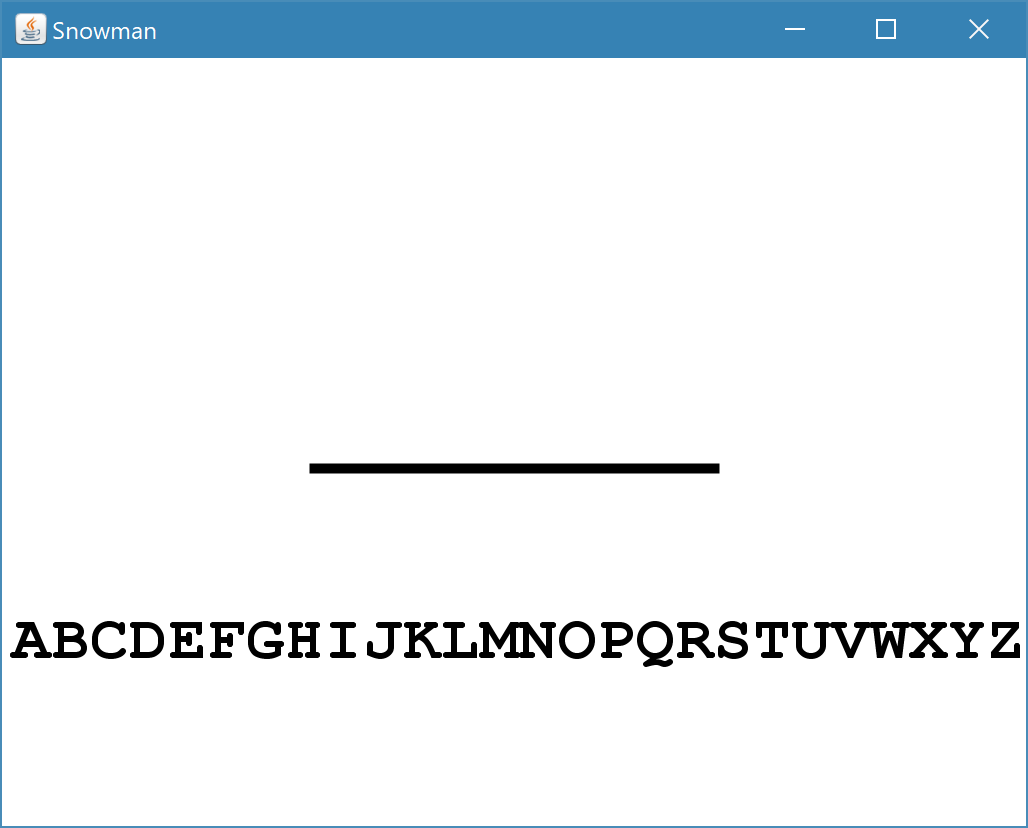
drawAvailableLetters(g);

Finally, for this part, if your letters look a little small, add a call like this to the drawAvailableLetters.

g.setFont(**new** Font("Monospaced", Font.***BOLD***, 64));

Where 64 is the size of the point, try different sizes until you find one that you like. You can also change the name of the font by replacing the word Monospaced with something else.

Now, when you run your code, you should see output like this:



## Guessing letters

So now we have out secret word and a set of letters to guess from. The next step is to allow the player to guess from the available letters and display the word as they gradually guess correctly or draw more of the snowman if they don’t.

The first step is draw the letters that the player has correctly guessed, along with a short line underneath where each letter will appear so the player knows how letters remain unguessed.

Create a new method, called drawWord as follows:

**boolean** drawWord(Graphics g) {

}

Add a call to it from within the existing paintComponent method, below where you call drawSnowman but above the call to Toolkit.getDefaultToolkit().sync().

Go back to the drawWord method and consider the tasks we have to complete:

1. Draw a set of marks for each letter.
2. Draw any letters in the word that have been found.
3. Return true if all the letters have been guessed or false if there are still some letters missing.

We’re going to draw each letter and its corresponding mark one at a time. So we’ll need a for loop that operates over all of the letters, but first we need to create some variables:

1. Create an integer (whole number) variable to track how many letters we’ve found, called lettersGuessed. We’ll increment this each time we come across a guessed letter.
2. Create another integer to record how far across the screen we’ve drawn, call this xPos.
3. Finally, create a third integer to remember how far down the screen to draw, call this yPos and set it to 500.

Java won’t let us compile and run our code though, until we return a Boolean value from the method, so add a final line to the method like this:

**return** lettersGuessed == word.length();

Work out what this does and ask if you’re unsure.

Now above the return statement but below the variable declarations, create the for loop:

**for** (**int** i = 0; i < **this**.word.length(); i++) {

g.drawString("\_", xPos, yPos + 5);

xPos += 64;

}

This for loop is going to draw one letter mark (an underscore character) for each letter in the word, moving the value of xPos on by 64 pixels for each mark. Depending on the screen resolution and font size you may want to alter this number later.

Whether we draw a letter or not is going to depend whether the letter is available for guesses. Remember that we have an instance variable called availableLetters, which contains a list of all the letters that *haven’t* been guessed.

So, if a letter *isn’t* in availableLetters, we want to draw it and add 1 to lettersGuessed.

The logic to do this looks like this:

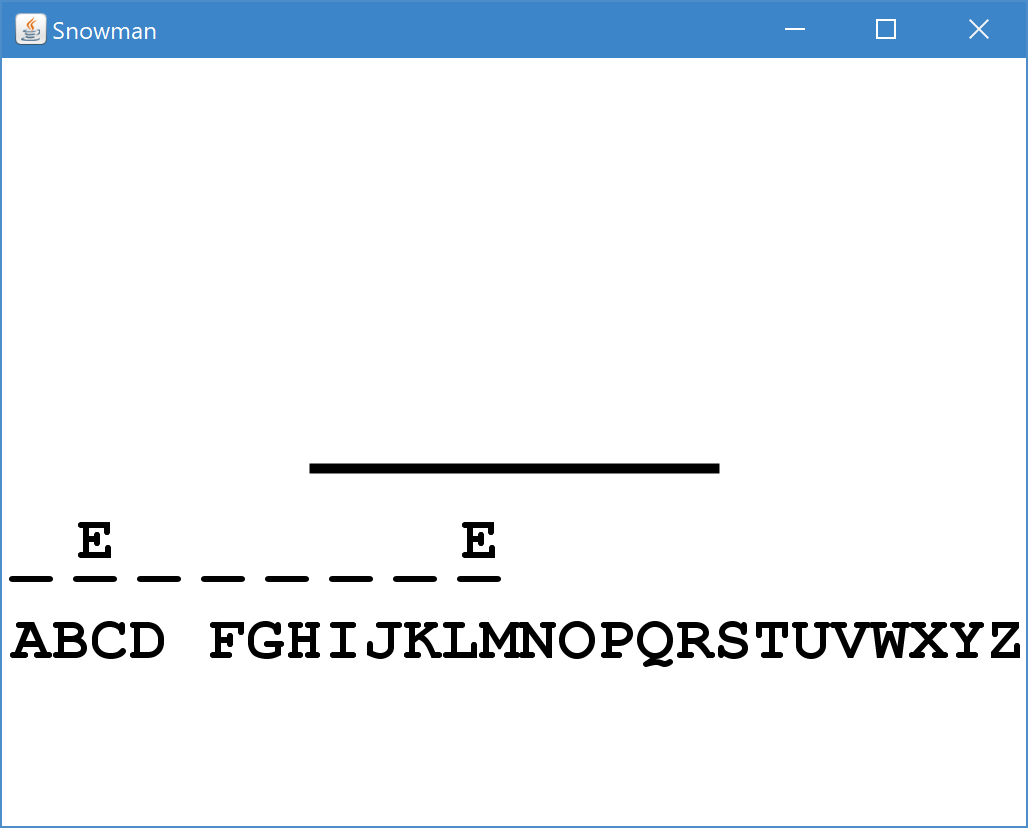
**if** (!availableLetters.contains(word.charAt(i))) {

}

Inside the if statement add the code to draw the letter and add 1 to lettersGuessed. I’m not going to give the code, it’s very similar to the drawString call you’ve already written. To get a single letter from the word, use the following logic method call: word.substring(i, i + 1).

The easiest way to test this is to remove some letters from the availableLetters variable set up in the initGame method so they’re not available any more, so will appear in the word.

Remove some letters and run your code, it should like something like this:



## Guessing letters

It’s time to start reading some input from the player, allowing them to guess letters, reveal the secret word or the snowman.

All widgets (things that can be drawn on the screen) can respond to mouse and keyboard “events” (clicks and pressed). In order to do so, you have to provide a separate class inside your SnowmanGame class.

Put the following class declaration outside the class

/\*\*

\* Event handler for key presses.

\*/

**private** **class** TAdapter **extends** KeyAdapter {

@Override

**public** **void** keyPressed(**final** KeyEvent e) {

}

}

In order for it to be used, you have to register the TAdapter with the SnowmanGame class (which is a type of widget). Do this as the first line of the constructor (the method called SnowmanGame).

addKeyListener(**new** TAdapter());

When the player presses a key, Java will call the keyPressed method for you. We can now examine what key has been pressed and take appropriate action.

So, inside the keyPressed method, get the key that’s been pressed from the parameter and store it in a variable, like this:

Character letter = Character.*toUpperCase*(e.getKeyChar());

First things first, if the player has already guessed this letter then do nothing (bit unfair to draw more snowman if press a key for a letter that’s already been guessed).

**if** (availableLetters.contains(letter)) {

}

Inside this if statement then remove from availableLetters (this will cause the letter to appear in the secret word if it’s there). There’s a handy method called remove on availableLeters!

If the letter isn’t in the word then we need to register a bad guess, i.e. increment badGuesses.

**if** (!word.contains(letter.toString())) {

badGuesses++;

}

Finally, we need to tell Java to repaint the whole game:

repaint();

Test your code and you can now essentially play the game: the secret word and the snowman are gradually revealed as the player guesses at letters.

However, the game doesn’t acknowledge when the player has won or lost, so that’s the last thing we need to do!

Part 3 – Game Over!

In order to manage the end of the game you need to introduce the idea of a “state” to the game. The state that the game is in will tell it what to draw. The game is in one of three states:

1. The player is playing the game.
2. The player has won the game (guessed all the letters of the secret word).
3. The player has lost the game (the snowman has been completely drawn).

In order to model this in the game, we need to introduce an *enumeration*. An enumeration is a special type of variable that can only hold values that you define. This is different from, say, an integer (int) that can take any number.

Outside of your methods but inside the class, declare an enum called GameState with three states along with an instance of GameState called gameState (lower-case g).

// Create an enumeration type to store game state

**enum** GameState {

***IN\_GAME***, ***GAME\_LOST***, ***GAME\_WON***

}

// We start the game by playing it

GameState gameState = GameState.***IN\_GAME***;

We only want to let the player guess at letters when they’re playing the game, so the first change to make is in the keyPressed method inside the TAdapter class.

**if** (gameState == GameState.***IN\_GAME***) {

}

Between the curly brackets leave the lines of code that handle removing a letter from the list of available letters.

When gameState changes to GAME\_WON or GAME\_LOST we’re going to want to print that in the window. Create a new method to draw a string in the middle of the window in the colour of your choice.

**private** **void** drawMiddleString(Graphics g, String string, Color colour) {

g.setFont(**new** Font("Monospaced", Font.***BOLD***, 64));

g.setColor(colour);

FontMetrics fm = g.getFontMetrics();

**int** middleX = (getWidth() - fm.stringWidth(string)) / 2;

**int** middleY = (getHeight() - fm.getHeight()) / 2;

g.drawString(string, middleX, middleY);

}

The rest of the changes are required in the drawGraphics routine. You already have calls to drawAvailableLetters, drawSnowman and drawWord, but add variables if you need to, to capture the state of the game.

drawAvailableLetters(g);

**boolean** gameWon = drawWord(g);

**boolean** gameLost = drawSnowman(g);

Then add code underneath to draw a message to the play telling them whether they’ve won or lost.

**if** (gameLost) {

gameState = GameState.***GAME\_LOST***;

drawMiddleString(g, "GAME OVER", Color.***RED***);

}

**if** (gameWon) {

gameState = GameState.***GAME\_WON***;

drawMiddleString(g, "CONGRATULATIONS!", Color.***GREEN***);

}

And you’re done! (Once all the bugs are out 😊). You now have a fully working Snowman game.

# Extension 1: Truly random words

As the program currently stands, you only have a short list of words from which the secret word is selected:

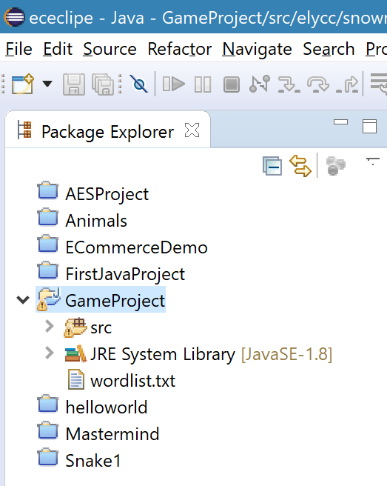
/\*\* The words that the program will select from \*/

String[] words = { "POINTLESS", "COMPUTER", "DECORATE", "CONSTRUCT", "PERIMETER" };

What would make the game much more challenging would be a much larger selection of words. Download the 10,000 most common English words from here (be careful to get the upper and lower-case letters right, just one capital letter, the L):

http://bit.ly/2zL8emh

Cut and paste the list of words in to Notepad and save the file in to the **root directory of your project** as wordlist.txt. If it’s in the right place, Eclipse will look like this (except that *GameProject* will have whatever name your Snowman game project has):



Now we have a list of words to choose from, we need to replace the words member variable with this list of 10,000 words.

Start by creating the shell of a new method to load the list.

**private** String[] loadWords() {

Set<String> wordList = **new** HashSet<String>();

// Load the list of words here

**return** wordList.toArray(**new** String[0]);

}

Where the comment in the middle is, this is where we’ll add the code to read from a file:

try {

// Open the file, ready for reading

BufferedReader rdr = new BufferedReader(new FileReader("wordlist.txt"));

String word;

do {

// Read the next line of the file

word = rdr.readLine();

// If we read a word (null means end of file) and it has more

// than 3 letters then add it to the list.

if (word!=null && word.length()>3) {

wordList.add(word.toUpperCase());

}

} while(word!=null);

// Close the file so it's not in use any more

rdr.close();

} catch(IOException ioe) {

// This is called if there's a problem loading the file.

// Print out details of the error.

System.***err***.println("Couldn't load word list: " + ioe);

**throw** **new** RuntimeException(ioe);

}

Read the comments carefully so you understand what’s going on. It uses a do… while loop, which is something we haven’t seen before. It’s basically a loop that runs only while the condition in the while clause is true.

It also uses a try… catch block as well! This is required in Java for certain operations that *throw exceptions*. If anything goes wrong with the commands that read from the file the computer will jump immediately to the catch block and execute it. The last line of the catch block re-throws the exception. Because no other code “catches” the exception it will cause the program to terminate.

Finally, remove the declaration of words from the source code and update initGame to add a new declaration of words, it with a call to loadWords().

This should be enough for your program to use the downloaded list of words. You now have a proper challenge to beat the game!