Programming Portfolio

HIT2302 Object-Oriented Programming

Author: Charlotte Pierce [7182139]

Tutorial: Friday 1:30

CONTENTS

BRIEF

This portfolio includes a number of projects developed during the completion of HIT2302 Object-Oriented Programming in semester two at Swinburne University of Technology in 2010.

The portfolio comprises of all projects described in the portfolio requirements, including implementation of all recommended optional extensions suggested. Each project included in this document (and on the attached CD) is intended to demonstrate a number of object-oriented concepts, as shown in the Java syntax. The specific concepts demonstrated in each project, as well as the purpose of the project and how to operate the application is included in a file ('README.txt'), located within the directory of each project.

This document contains the summary of each project (i.e. the contents of the README file), and all source code for each project included. All work shown in the document is original, or an extension on a demonstration project included as part of *"Objects First with Java: A Practical Introduction Using BlueJ – Barnes & Kolling"*.

[CD] Clock Display

Summary

Project: Clock Display turned AFL Scoreboard with GUI
Author: Charlotte Pierce

This project is part of the material for the Object-Oriented Programming Portfolio of Charlotte Pierce.

To use this project, create an instance of class ScoreGUI using the main method within. The scores of each team can be modified with the use of the buttons on the GUI.

This project demonstrates Object-Oriented programming with its use of classes, each of which represents one singular task. This demonstrates the principle of cohesion. Each class also demonstrates the use of encapsulation, that is, the principle that all information within the class regarding its implementation is hidden from other classes.

This project fulfils the criteria defined in the portfolio requirements, being the most advanced version of the clock display that was requested. As such, it involves the use of a Swing/AWT graphical user interface, which uses anonymous inner classes for the purpose of event listening on each of the buttons.

The project has basic exception handling included, to catch errors such as NumberDisplay returning a null value rather than a string of text as required.

Code

NumberDisplay

```
/**
 * The NumberDisplay class represents a digital number display that can hold
 * values from zero to a given limit. The limit can be specified when creating
 * the display.
 *
 * @author Charlotte Pierce
 * @version 7/11/2010
 */
public class NumberDisplay
{
   private int limit;
   private int value; //number scored

   /**
    * Constructor for objects of class NumberDisplay.
    * Set the limit at which the display rolls over.
    */
   public NumberDisplay()
   {
      this.limit = 0;
      value = 0;
   }

   /**
    * Creates a NumberDisplay which can display any number from 0 to the
    * sepcified limit.
    * @param limit The highest number the display can show.
    */
   public NumberDisplay(int limit)
   {
      this.limit = limit;
      value = 0;
   }

   /**
    * Return the current value of the display.
    * @return The current value of the display.
    */
   public int getValue()
   {
      return value;
   }

   /**
    * Returns the current value of the display in the form of a string.
    * @return The current valeu of the display.
    */
```

```
    public String getDisplayValue()
    {
        if(value < 10) {
            return "0" + value;
        }
        else {
            return "" + value;
        }
    }

    /**
     * Sets the value of the display, if that value is between 0 and the
     * limit of the display.
     * @param replacementValue The value to set the display to.
     */
    public void setValue(int replacementValue)
    {
        if((replacementValue >= 0) && (replacementValue < limit)) {
            value = replacementValue;
        }
    }

    /**
     * Increments the display value by 1. If the display reaches the limit,
     * the display rolls over to 0.
     */
    public void increment()
    {
        value = (value + 1) % limit;
    }
}
```

ScoreDisplay

```
/**
 * The ScoreDisplay uses two instances of the NumberDisplay class to create a scoreboard for
 * an AFL team.
 *
 * @author Charlotte Pierce
 * @version 7/11/2010
 */

public class ScoreDisplay
{
    private NumberDisplay goals;
    private NumberDisplay behinds;
    private int score;
    private String numDisplay;

    /**
     * Creates a new ScoreDisplay. Instantiates two objects of type NumberDisplay, one for the
```

```
 * goals and one for the behinds. Sets the initial score to 0.
 */
public ScoreDisplay()
{
   goals = new NumberDisplay(99);
   behinds = new NumberDisplay(99);
   score = 0;
   updateNumDisplay();
}

/**
 * Simulates the scoring of a goal.
 */
public void scoreGoal()
{
   goals.increment();
   updateScore();
}

/**
 * Simulates the scoring of a behind.
 */
public void scoreBehind()
{
   behinds.increment();
   updateScore();
}

/**
 * Updates the value of the total score, according to how many goals and how many behinds
 * have been scored.
 */
public void updateScore()
{
   score = (6 * goals.getValue()) + behinds.getValue();
   updateNumDisplay();
}

/**
 * Updates the string representing the score of the team.
 */
public void updateNumDisplay()
{
   numDisplay = goals.getDisplayValue() + "    " + behinds.getDisplayValue() + "    " + score;
}

/**
 * Returns the total score of the team.
 * @return The totals score of the team.
 */
public int getScore()
```

```
    {
        return score;
    }

    /**
     * Returns a summary of the teams performance in the form of a String.
     * Shown in the form "<number of goals> <number of behinds> <total score>"
     * @return The team's score.
     */
    public String getNumDisplay()
    {
        try
        {
            return numDisplay;
        }
        catch(NullPointerException npe)
        {
            return "00 00 0";
        }
    }
}
```

ScoreBoard

```
/**
 * Simulates a ScoreBoard used for an AFL match. Stores the ScoreDisplay for both the Home and
Away teams,
 * and can update and return the values on these displays.
 *
 * @author Charlotte Pierce
 * @version 7/11/2010
 */
public class ScoreBoard
{
    private ScoreDisplay team1;
    private ScoreDisplay team2;
    private String scoreBoard;

    /**
     * Instantiates the two ScoreDisplays, one for each team.
     */
    public ScoreBoard()
    {
        team1 = new ScoreDisplay();
        team2 = new ScoreDisplay();
    }

    /**
     * Returns the scores of both teams, showing which side of the display is for the home team, and
which for the
```

```
     * away. Adds an asterisks next to the score of the team which is currently winning.
     * @return The scores of both teams.
     */
    public String getScores()
    {
        scoreBoard = "[HOME]   ";
        if(team1.getScore() > team2.getScore()){
            scoreBoard += team1.getNumDisplay() + "*  :  " + team2.getNumDisplay();
        }
        else if(team1.getScore() < team2.getScore()){
            scoreBoard += team1.getNumDisplay() + "  :  " + team2.getNumDisplay();
        }
        else{
            scoreBoard += team1.getNumDisplay() + "  :  " + team2.getNumDisplay();
        }
        scoreBoard += "   [AWAY]";
        return scoreBoard;
    }

    /**
     * Simulates the scoring of a goal by the home team.
     */
    public void team1Goal()
    {
        team1.scoreGoal();
    }

    /**
     * Simulates the scoring of a goal by the away team.
     */
    public void team2Goal()
    {
        team2.scoreGoal();
    }

    /**
     * Simulates the scoring of a behind by the home team.
     */
    public void team1Behind()
    {
        team1.scoreBehind();
    }

    /**
     * Simulates the scoring of a behind by the away team.
     */
    public  void team2Behind()
    {
        team2.scoreBehind();
    }
}
```

ScoreGUI

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.Font;

/**
 * The GUI and main controls for the scoreboard system. This class controls the interface
 * for the entire scoreboard system. To run, just run the main method of this class.
 *
 * @author Charlotte Pierce
 * @version 7/11/2010
 */
public class ScoreGUI
{
   private JFrame frame; // the interaction screen
   private JLabel scoresText;
   private ScoreBoard scoreBoard;

   /**
    * Instantiates the fields - creates the ScoreBoard, the frame on which to show GUI controls,
    * and sets the initial value of scoresText.
    */
   public ScoreGUI()
   {
      scoreBoard = new ScoreBoard();
      frame = new JFrame("AFL Score Board"); // creates the main window
      scoresText = new JLabel(scoreBoard.getScores()); // creates label to show the teams' scores
   }

   /**
    * Creates an instance of the GUI and sets up all controls.
    */
   public static void main(String[] args)
   {
      ScoreGUI mainApp = new ScoreGUI();
      mainApp.beginApp();
   }

   /**
    * Sets up all GUI controls and event listeners.
    * Packs the frame and makes it visible.
    */
   private void beginApp()
   {
      Container contentPane = frame.getContentPane(); // gets the content pane so can add stuff to it
      contentPane.setLayout(new BorderLayout()); // set the main window to have a BorderLayout
```

```
    // set and add the scoresText to the contentPane
    scoresText.setHorizontalAlignment(JLabel.CENTER);
    scoresText.setFont(new Font("Calibri", Font.PLAIN, 20));
    contentPane.add(scoresText);

    JPanel scoreControls = new JPanel(new GridLayout(1, 4)); // creates a 1 x 4 grid to store the
buttons on
    // create buttons for scoring goals and behinds
    JButton goal1 = new JButton("Home Goal");
    JButton behind1 = new JButton("Home Behind");
    JButton goal2 = new JButton("Away Goal");
    JButton behind2 = new JButton("Away Behind");
    // add action listeners to the buttons
    goal1.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                           homeGoal();
                        }
                    });
    behind1.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                           homeBehind();
                        }
                    });
    goal2.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                           awayGoal();
                        }
                    });
    behind2.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                           awayBehind();
                        }
                    });
    // add the buttons to the grid
    scoreControls.add(goal1);
    scoreControls.add(behind1);
    scoreControls.add(goal2);
    scoreControls.add(behind2);
    //add the grid to the main window
    contentPane.add(scoreControls, BorderLayout.SOUTH);

    frame.pack();
    frame.setVisible(true);
  }

  /**
```

```
    * Simulates a goal for the home team.
    */
   private void homeGoal()
   {
      scoreBoard.team1Goal();
      updateDisplay();
   }

   /**
    * Simulates a behind for the home team.
    */
   private void homeBehind()
   {
      scoreBoard.team1Behind();
      updateDisplay();
   }

   /**
    * Simulates a goal for the away team.
    */
   private void awayGoal()
   {
      scoreBoard.team2Goal();
      updateDisplay();
   }

   /**
    * Simulates a behind for the away team.
    */
   private void awayBehind()
   {
      scoreBoard.team2Behind();
      updateDisplay();
   }

   /**
    * Updates the JLabel showing the scores to show the most recent score data.
    * Called after every goal/behind.
    */
   private void updateDisplay()
   {
      scoresText.setText(scoreBoard.getScores());
   }
}
```

[MS] MAIL SYSTEM

Summary

Project: Mail System
Author: Charlotte Pierce

This project is part of the material for the Object-Oriented Programming Portfolio of Charlotte Pierce

This project simulates a simple email system. Mail clients simulate email programs of different users. If you create two or more email clients, they can send messages to each other.
This project has been extended to include an Internet class. This allows multiple MailServer objects to exist within the application. Multiple clients can also exist, each with one of any MailServers. If a client attempts to send a message to another client who is on a different server to them, the first client's server forwards the message to the Internet object, who then filters out the intended recipient server and again forwards the message, to the appropriate server. This simulates the actual function of an e-mail system.

To use this project:

- create an instance of the Internet
- create as many MailServer objects as desired
- create as many MailClient objects as desired, and assign each a MailServer
    - Note: the MailClient's e-mail address will be "<name of mailclient>@<name of server>"
- use the 'sendMessage' method of a MailClient to send a message to one of the other MailClients
- use the 'printNextMessage' method of the second MailClient to receive the message.

This project demonstrates the use of object interaction, where objects (MailItem) are passed between other objects (Internet and MailServer). It also demonstrates the use of good program structure, where each class is responsible for a specific function, and each method within each class responsible for one specific task. The project is fully commented using javadoc, in order to facilitate further extensions.

In order to fulfill the requirements defined for the project, it has been extended to include e-mail addresses for each client, including the client's name and their MailServer. It has also been extended to use this e-mail address to filter MailItem's through an Internet object, so that clients can send messages to other clients that are not using thier server. The project folder includes a sequence diagram to demonstrate the interaction between objects in order to achieve this functionality.

The project uses exception handling to output errors in cases such as a MailServer or MailClient not existing.

The project also includes testing, using JUnit, for this extended version of the MailSystem. In addition to testing, the project also includes a file, '[MS]-2 InheritanceExplanation',

which reflects on the ability to refactor the code in the project to use the concept of inheritance.

Code

MailServer

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

/**
 * A simple model of a mail server. The server is able to receive
 * mail items for storage, and deliver them to clients on demand.
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class MailServer
{
    // Storage for the arbitrary number of mail items to be stored
    // on the server.
    private List<MailItem> items;
    //Name of server
    private String name;
    //Reference to internet
    private Internet internet;

    /**
     * Construct a mail server.
     */
    public MailServer(String name, Internet internet)
    {
        items = new ArrayList<MailItem>();
        this.name = name;
        this.internet = internet;
        //'Add' server to the internet
        internet.addServer(this);
    }

    /**
     * Return how many mail items are waiting for a user.
     * @param who The user to check for.
     * @return How many items are waiting.
     */
    public int howManyMailItems(String who)
    {
        int count = 0;
        for(MailItem item : items) {
            if(item.getTo().equals(who)) {
                count++;
            }
        }
        return count;
    }
```

```
    /**
     * Return the next mail item for a user or null if there
     * are none.
     * @param who The user requesting their next item.
     * @return The user's next item.
     */
    public MailItem getNextMailItem(String who)
    {
        Iterator<MailItem> it = items.iterator();
        while(it.hasNext()) {
            MailItem item = it.next();
            if(item.getTo().equals(who)) {
                it.remove();
                return item;
            }
        }
        return null;
    }

    /**
     * Add the given mail item to the message list.
     * @param item The mail item to be stored on the server.
     */
    public boolean post(MailItem item)
    {
        String temp;
        temp = splitServer(item.getTo());
        if(temp.equals(this.getName())){
            items.add(item);
            return true;
        }
        else{
            internet.sendMail(item, temp);
            return false;
        }
    }

    public String getName()
    {
        return name;
    }

    //Splits an email to find its server
    private String splitServer(String email)
    {
        String[] temp;
        temp = email.split("@");
        return temp[(temp.length - 1)];
    }
}
```

MailClient

```
/**
 * A class to model a simple email client. The client is run by a
 * particular user, and sends and retrieves mail via a particular server.
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class MailClient
{
    // The server used for sending and receiving.
    private MailServer server;
    // The user running this client.
    private String user;
    //Name of the client
    private String name;
    //e-mail of the user
    private String email;

    /**
     * Create a mail client run by user and attached to the given server.
     */
    public MailClient(MailServer server, String user, String name)
    {
        this.server = server;
        this.user = user;
        this.name = name;
        email = user + "@" + server.getName();
    }

    /**
     * Return the next mail item (if any) for this user.
     */
    public MailItem getNextMailItem()
    {
        return server.getNextMailItem(user);
    }

    /**
     * Print the next mail item (if any) for this user to the text
     * terminal.
     */
    public boolean printNextMailItem()
    {
        try
        {
            MailItem item = server.getNextMailItem(email);
            System.out.println("From: " + item.getFrom());
            System.out.println("To: " + item.getTo());
            System.out.println("Message: " + item.getMessage());
```

```
            return true;
        }
        catch(NullPointerException npe)
        {
            System.out.println("No new mail.");
            return false;
        }
    }

    /**
     * Send the given message to the given recipient via
     * the attached mail server.
     * @param to The intended recipient. [their email address]
     * @param message The text of the message to be sent.
     */
    public void sendMailItem(String to, String message)
    {
        MailItem item = new MailItem(user, to, message);
        server.post(item);
    }
}
```

MailItem

```
/**
 * A class to model a simple mail item. The item has sender and recipient
 * addresses and a message string.
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class MailItem
{
    // The sender of the item.
    private String from;
    // The intended recipient.
    private String to;
    // The text of the message.
    private String message;

    /**
     * Create a mail item from sender to the given recipient,
     * containing the given message.
     * @param from The sender of this item.
     * @param to The intended recipient of this item.
     * @param message The text of the message to be sent.
     */
    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
```

```
        this.message = message;
    }

    /**
     * @return The sender of this message.
     */
    public String getFrom()
    {
        return from;
    }

    /**
     * @return The intended recipient of this message.
     */
    public String getTo()
    {
        return to;
    }

    /**
     * @return The text of the message.
     */
    public String getMessage()
    {
        return message;
    }
}
```

Internet

```
/**
 * The internet object. The internet object accepts MailItems, searches
 * for the server to which they belong and forwards the MailItem to
 * that server.
 */

import java.util.ArrayList;
import java.util.Iterator;

public class Internet
{
    //Calling server
    private ArrayList<MailServer> servers;

    public Internet()
    {
        servers = new ArrayList<MailServer>();
    }

    /**
```

```
    * Adds a server to the Internet's list of servers
    * @param addserv The server to add.
    * @return Whether the add was successful or not.
    */
   public boolean addServer(MailServer addserv)
   {
      servers.add(addserv);
      return true;
   }

   /**
    * Sends a mail item to the specified server.
    * @param email The mail item to send.
    * @param server The name of the server to send it to.
    * @return True if the sending was successful, else false.
    */
   public boolean sendMail(MailItem email, String server)
   {
      try
      {
         MailServer sendTo = findServer(server);
         sendTo.post(email);
         return true;
      }
      catch(NullPointerException npe)
      {
         System.out.println("Server does not exist!");
         return false;
      }
   }

   /**
    * Finds the server with the given name in the Internet's server list.
    * @param server The name of the server to find.
    * @return The found server.
    */
   private MailServer findServer(String server)
   {
      Iterator<MailServer> it = servers.iterator();
      while(it.hasNext()){
         MailServer serv = it.next();
         if(serv.getName().equals(server)){
            return serv;
         }
      }
      return null;
   }
}
```

MailTesting

```
/**
 * The test class mailtesting.
 *
 * @author  (your name)
 * @version (a version number or a date)
 */
public class mailtesting extends junit.framework.TestCase
{
        private Internet internet1;
        private MailServer mailServ1;
        private MailServer mailServ2;
        private MailClient bob;
        private MailClient fred;

   /**
    * Default constructor for test class mailtesting
    */
   public mailtesting()
   {
   }

   /**
    * Sets up the test fixture.
    *
    * Called before every test case method.
    */
   protected void setUp()
   {
                internet1 = new Internet();
                mailServ1 = new MailServer("mailServ1", internet1);
                mailServ2 = new MailServer("mailServ2", internet1);
                bob = new MailClient(mailServ1, "bob", "mailServ1");
                fred = new MailClient(mailServ2, "fred", "mailServ2");
        }

   /**
    * Tears down the test fixture.
    *
    * Called after every test case method.
    */
   protected void tearDown()
   {
   }

        public void testAddMailServertoInternet()
        {
                assertEquals(true, internet1.addServer(mailServ1));
        }
```

```
	public void testCreateMailItem()
	{
		MailClient bob = new MailClient(mailServ1, "bob", "mailServ1");
		bob.sendMailItem("bob@mailServ1", "hey me");
		assertEquals(0, mailServ1.howManyMailItems("bob"));
		assertEquals(1, mailServ1.howManyMailItems("bob@mailServ1"));
	}

	public void testDirectMail()
	{
		MailServer mailServ2 = new MailServer("mailServ2", internet1);
		MailClient bob = new MailClient(mailServ1, "bob", "mailServ1");
		MailClient fred = new MailClient(mailServ2, "fred", "mailServ1");
		bob.sendMailItem("fred@mailServ1", "wasup fred");
		assertEquals(1, mailServ1.howManyMailItems("fred@mailServ1"));
	}

	public void testSendMailSame()
	{
		MailClient george = new MailClient(mailServ1, "george", "mailServ1");
		bob.sendMailItem("george@mailServ1", "hey bob");
		assertEquals(true, george.printNextMailItem());
	}

	public void testSendMailDifferent()
	{
		bob.sendMailItem("fred@mailServ2", "hey fred");
		assertEquals(true, fred.printNextMailItem());
	}
}
```

World Of Zuul

Summary

Project: World of Zuul
Author: Charlotte Pierce

This project is part of the material for the Object-Oriented Programming portfolio of Charlotte Pierce.

This application originated as a text-based adventure game, which has been extended to include a graphical user interface. The game places players in the 'World of Zuul', a game where Zuul, an evil scientist, is threatening to take over the world. It is up to the player to infiltrate Zuul's laboratory facilities, find the key to his master control room, and shut down his operations.

When playing the game, the player can move around the facility, pick up and drop items, use potions to restore their health, and attack monsters (using items they have picked up).

To use this application, run the main method in the Game class. Then use the buttons on the GUI to navigate through the game.

This application demonstrates many Object-Oriented concepts and principles. Demonstrating encapsulation, each class hides it's implementation from all other classes. The application also demonstrates cohesion, as each class is responsible for one function, and each method within each class is responsible for one specific task. The structure of the project was designed with the goal of making classes as loosely coupled as possible, so that the project can be more easily extended or modified in the future. In another effort to assist further extensions on the project, every class has been extensively documented using javadoc style.

The project demonstrates some more advanced Object-Oriented concepts, such as inheritance. This is used by the IntelligentBeing and Room classes, each of which is extended (IntelligentBeing into Monster and Player classes; Room into TransporterRoom). The IntelligentBeing class also demonstrates the use of an abstract class (IntelligentBeing), which is used to stop direct instances of the class from being created, and only allow its subclasses to be instantiated.

Polymorphism is also used in the project, where instances of the TransporterRoom class are assigned to the static type of Room, in order for it to be used without the need to design special methods which perform idential functions (i.e. when an exit points to a TransporterRoom, the TransporterRoom is treated as an instance of Room, rather than specifically a TransporterRoom).

The application fulfills the criteria of the portfolio, first by implementing the extensions of incorporating an inventory, and the ability to pick up and drop items as was suggested. It also includes the extension of adding health and strength fields to every monster and player (which was refactored to be included in IntelligentBeing), which defines the damage that monster/player does in combat. As such, the application was also extended to include combat, and potions which restore health.

The project was then extended to incorporate a GUI, which enables players to navigate the game using buttons, rather than typing in textual commands. These buttons perform functions with the use of anonymous inner classes being used as event listeners.

Finally, the project was included to allow the player to save and load their game. The relevant details regarding the player's game (i.e. their inventory, the current room, the player's health) are stored in a text file in whichever directory the game file is located. The player is invited to name this file. Should the player wish to load a previous game, they just have to select 'Load' from the 'File' menu, and type in the name of the file they saved to. The loading methods assume that the file being loaded is stored in the same directory as the game file, unless the user specifically types in a concrete address. When loaded, the game changes the value of the player's health and current moves to those stored, and reinstates the inventory and current room that the player had/was in when they saved the game.

Exception handling is also included in this version of the project, primarily to avoid the issue of a null pointer being returned when the player is asked to specify an item to drop or pick up, or a monster to attack. Due to the exception handling, instead of returning an error and terminating the execution of the application, the game instead displays a message to the player indicating their lack of choice, and continues running as normal.

Code

Game

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;
import java.util.ArrayList;

/**
 *  This class is the main class of the "World of Zuul" application.
 *  "World of Zuul" is a text based adventure game.  Users
 *  can walk around some scenery, pick up items and attack monsters.
 *  The goal for the user is to find the master key, and take it to the control room.
 *  There is a limit on the number of moves a player can make to reach this goal.
 *
 *  To play this game, create an instance of this class and call the "play"
 *  method.
 *
 *  This main class creates and initialises all the others: it creates all
 *  rooms, creates the parser and starts the game.  It also evaluates and
 *  executes the commands that the parser returns.
 *
 * @author  Charlotte Pierce
 * @version 1/10/2010
 */

public class Game
{
   private Parser parser;
   private Room currentRoom;
   private Room goalRoom;
   private int maxMoves;
   private int maxWeight;
   private Player player1;
   private RoomRandomiser randomiser;
   private JFrame frame;
   private JTextArea mainText; // the place to show all text to the user
   private Command nextCommand;
   private ArrayList<Item> allItems;

   public static void main(String args[]){
      Game g = new Game();
      g.play();
   }

   /**
    * Create the game and initialise its internal map.
    */
   public Game()
   {
```

```
        initialiseGame();
        parser = new Parser();
        player1 = new Player();
        maxMoves = 25;
        maxWeight = 100;
        setUpGUI();
    }

    /**
     * Create all the rooms and link their exits together.
     * Create items and monsters and place them in appropriate rooms.
     */
    private void initialiseGame()
    {
        allItems = new ArrayList<Item>();
        randomiser = new RoomRandomiser();
        // initialise rooms - give them descriptions and add them to the randomiser
        Room outside = new Room("outside", "\n \n \n You are outside a huge lab facility. Inpenetrable
jungle is all around, \n with the only exits being the trailing path you can from, and the entry to \n
the facility your options. It looks like Zuul knew you were coming, and has \n abandoned the
building. \n However, you know he will have left nasties within.", randomiser);
        TransporterRoom foyer = new TransporterRoom("foyer", "\n \n \n You are in what appears to
be a foyer. It looks like \n a pretty standard room. The exits look strange, almost as if they are
portals \n somwhere. You think it's a trap, but you have no options. You know that \n trying to exit
the room could leave you anywhere.", randomiser);
        Room waitingRoom1 = new Room("waitingRoom1", "\n \n \n You walk slowly into what you
decide must be a waiting room. Apart \n from the chairs and tables, you see nothing out of the
ordinary.", randomiser);
        Room teaRoom = new Room("teaRoom", "\n \n \n Success! You have found the tea room!
Perhaps now you can get something \n to refresh you after your long journey...", randomiser);
        Room entertainmentRoom = new Room("entertainmentRoom", "\n \n \n Seems you have found
a room where Zuul's scientists allow their \n test subjects to rest a while.", randomiser);
        Room waitingRoom2 = new Room("waitingRoom2", "\n \n \n This room looks exactly the same
as the waiting room you found eariler \n in the adventure. *Exactly* the same. Creepy...",
randomiser);
        Room hiddenRoom = new Room("hiddenRoom", "\n \n \n This does not look like a room people
are supposed to find...", randomiser);
        Room controlRoom = new Room("controlRoom", "\n \n \n This looks like the room you have
been looking for.", randomiser);
        Room testRoom1 = new Room("testRoom1", "\n \n \n Scientific things all around, looks like
some sort of testing takes place \n here, a smear that looks suspiciously like blood is on the wall.",
randomiser);
        Room testRoom2 = new Room("testRoom2", "\n \n \n Scientific things all around, a testing
room you think.", randomiser);
        Room storage1 = new Room("storage1", "\n \n \n This room looks less organised than the
others. It's filled with cleaning products.", randomiser);
        Room storage2 = new Room("storage2", "\n \n \n This room seems out of character with the
others in the facility, there's no \n scientific equipment to be seen, no hidden monitors, just endless
\n rags and cleaning products.", randomiser);
        Room observationRoom1 = new Room("observationRoom1", "\n \n \n This room is filled with
monitors looking into four smaller rooms. \n Possibly an observation room?", randomiser);
```

```
    Room observationRoom2 = new Room("observationRoom2", "\n \n \n You see lots of
equipment in this room. Through the glass you can see \n a couple of smaller rooms. It looks like
they are used for some \n form of testing.", randomiser);
    Room lab2A = new Room("lab2A", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '2A'.", randomiser);
    Room lab2B = new Room("lab2B", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '2B'.", randomiser);
    Room lab1A = new Room("lab1A", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '1A'.", randomiser);
    Room lab1B = new Room("lab1B", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '1B'.", randomiser);
    Room lab1C = new Room("lab1C", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '1C'.", randomiser);
    Room lab1D = new Room("lab1D", "\n \n \n A small room, with a few pieces of equipment and a
single chair. \n You see a big sign on the roof which reads '1D'.", randomiser);

    // initialise room exits
    outside.setExit("north", foyer);
    foyer.setExit("south", outside);
    foyer.setExit("north", waitingRoom1);
    waitingRoom1.setExit("south", foyer);
    waitingRoom1.setExit("north", teaRoom);
    waitingRoom1.setExit("east", storage2);
    waitingRoom1.setExit("west", lab1C);
    teaRoom.setExit("south", waitingRoom1);
    teaRoom.setExit("north", entertainmentRoom);
    teaRoom.setExit("east", storage1);
    entertainmentRoom.setExit("south", teaRoom);
    entertainmentRoom.setExit("north", waitingRoom2);
    entertainmentRoom.setExit("east", testRoom2);
    waitingRoom2.setExit("south", entertainmentRoom);
    waitingRoom2.setExit("north", hiddenRoom);
    waitingRoom2.setExit("east", testRoom1);
    waitingRoom2.setExit("west", lab2B);
    hiddenRoom.setExit("south", waitingRoom2);
    controlRoom.setExit("west", lab2B);
    testRoom1.setExit("west", waitingRoom2);
    testRoom1.setExit("south", testRoom2);
    testRoom2.setExit("west", entertainmentRoom);
    testRoom2.setExit("north", testRoom1);
    storage1.setExit("west", teaRoom);
    storage1.setExit("south", storage2);
    storage2.setExit("north", storage1);
    storage2.setExit("west", waitingRoom1);
    lab2A.setExit("east", lab2B);
    lab2A.setExit("south", observationRoom2);
    lab2B.setExit("north", controlRoom);
    lab2B.setExit("east", waitingRoom2);
    lab2B.setExit("west", lab2A);
    lab2B.setExit("south", observationRoom2);
    observationRoom2.setExit("north", lab2A);
```

```
observationRoom2.setExit("south", lab1B);
lab1A.setExit("north", observationRoom2);
lab1A.setExit("south", observationRoom1);
lab1A.setExit("east", lab1B);
lab1B.setExit("west", lab1A);
lab1B.setExit("south", lab1C);
lab1C.setExit("north", lab1B);
lab1C.setExit("south", lab1D);
lab1D.setExit("north", lab1C);
observationRoom1.setExit("north", lab1A);

//create items
Item tree = new Item("tree", 200, 50, 5);
allItems.add(tree);
Item flower = new Item("flower", 1, 5, 1);
allItems.add(flower);
Item chair = new Item("chair", 25, 10, 10);
allItems.add(chair);
Item table = new Item("table", 100, 15, 5);
allItems.add(table);
Item potion = new Potion();
allItems.add(potion);
Item television = new Item("television", 110, 250, 8);
allItems.add(television);
Item bat = new Item("bat", 30, 5, 20);
allItems.add(bat);
Item speaker = new Item("speaker", 5, 10, 10);
allItems.add(speaker);
Item laptop = new Item("laptop", 50, 50, 15);
allItems.add(laptop);
Item acid = new Item("acid", 5, 10, 60);
allItems.add(acid);
Item broom = new Item("broom", 5, 5, 5);
allItems.add(broom);

//add items to rooms
outside.addItem(tree);
outside.addItem(flower);
waitingRoom1.addItem(chair);
waitingRoom1.addItem(table);
teaRoom.addItem(potion);
entertainmentRoom.addItem(television);
waitingRoom2.addItem(bat);
observationRoom2.addItem(speaker);
observationRoom1.addItem(laptop);
testRoom2.addItem(acid);
storage1.addItem(broom);

//create monsters
Monster rat1 = new Monster("rat", 5, 10);
Monster rat2 = new Monster("rat", 5, 10);
```

```
        Monster drone = new Monster("drone", 20, 10);
        Monster mutantB = new Monster("mutantB", 15, 10);
        Monster mutantL = new Monster("mutantL", 25, 50);
        Monster mutant3 = new Monster("mutant", 100, 100);
        Monster spider = new Monster("spider", 5, 15);
        Monster skeleton = new Monster("skeleton", 20, 5);

        //add monsters to rooms
        foyer.addMonster(rat1);
        storage2.addMonster(rat2);
        teaRoom.addMonster(drone);
        lab1D.addMonster(mutantB);
        lab2A.addMonster(mutantL);
        hiddenRoom.addMonster(mutant3);
        lab1A.addMonster(spider);
        observationRoom2.addMonster(skeleton);

        //assign current room and goal room
        currentRoom = outside;
        goalRoom = controlRoom;
    }

    /**
     *  Main play routine. Executes one 'move' of play.
     */
    public void play()
    {
        boolean finished = false;
        if (!finished) {
            if(nextCommand != null){
                finished = processCommand(nextCommand);
                //only check if move limit reached if user hasn't already quit
                //nested if's because only want to check once (otherwise finished gets re-written to true by
reachedGoa())
                if(!finished){
                    finished = checkMoves();
                    if(!finished){
                        finished = !checkAlive();
                    }
                    if(!finished){
                        finished = reachedGoal();
                    }
                }
            }
        }
        if(finished){
            if(checkMoves()){
                mainText.setText("!!! Move limit reached. !!!");
            }
            else if(!checkAlive()){
                mainText.setText("!!! You died. !!!");
```

```
        }
        else if(reachedGoal()){
            mainText.setText("\n!!! You reached the control room with the master key!\n You destroy
Zuul's work, and his evil plans. \nContratulations! !!!\n");
        }
        else{
            mainText.setText("");
        }
        mainText.append("\n Thank you for playing.  Goodbye. \n You may now click 'Quit' in the file
menu to quit.");
      }
   }

   /**
    * Return a string of the opening message for the player.
    * @return The opening message for the player
    */
   private String printWelcome()
   {
      String returnString = "";
      returnString += "Welcome to the World of Zuul!";
      returnString += "\n Your entire town was abducted by a scientist known as 'Zuul'.";
      returnString += "\n You have followed Zuul and his team back to his secret laboratory";
      returnString += "\n in the jungle. You know there is little hope for you friends and family,";
      returnString += "\n but you know that if you can get to the control room in this research
facility,";
      returnString += "\n with the key to the master files, you can thwart Zuul's evil plans and save";
      returnString += "\n other towns from going through what you had to endure.";
      returnString += "\n Your journey has been long and tough, and you are already feeling its effects
\n on your health... \n \n";
      returnString += "\n Click the '" + CommandWord.HELP + "' button if you need help.";
      returnString += "\n" + currentRoom.getLongDescription() + "\n\n";
      return returnString;
   }

   /**
    * Given a command, process (that is: execute) the command.
    * @param command The command to be processed.
    * @return true If the command ends the game, false otherwise.
    */
   private boolean processCommand(Command command)
   {
      boolean wantToQuit = false;

      CommandWord commandWord = command.getCommandWord();

      if(commandWord == CommandWord.UNKNOWN) {
         mainText.append("I don't know what you mean...");
         return false;
      }
```

```
    if (commandWord == CommandWord.HELP) {
        printHelp();
    }
    else if (commandWord == CommandWord.GO) {
        goRoom(command);
    }
    else if (commandWord == CommandWord.QUIT) {
        wantToQuit = quit(command);
    }
    else if (commandWord == CommandWord.GET){
        getItem();
    }
    else if (commandWord == CommandWord.DROP){
        getDropItem();
    }
    else if (commandWord == CommandWord.SHOW){
        showInventory();
    }
    else if (commandWord == CommandWord.ATTACK){
        getAttackMonster();
    }
    else if (commandWord == CommandWord.POTION){
        drinkPotion();
    }
    // else command not recognised.
    return wantToQuit;
}

// implementations of user commands:

/**
 * Print out some help information.
 * Here we print some stupid, cryptic message and a list of the
 * command words.
 */
private void printHelp()
{
    mainText.setText("You are lost. You are alone. You wander");
    mainText.append("\n around at the university. \n");
    mainText.append("\n Your command words are: \n");
    mainText.append(parser.showCommands());
}

/**
 * Try to go to one direction. If there is an exit, enter the new
 * room, otherwise print an error message.
 */
private void goRoom(Command command)
{
    String direction = command.getSecondWord();
```

```
    // Try to leave current room.
    Room nextRoom = currentRoom.getExit(direction);

    if (nextRoom == null) {
        mainText.append("\n There is no door!");
    }
    else {
        currentRoom = nextRoom;
        player1.addMove();
        mainText.setText(currentRoom.getLongDescription());
    }
}

/**
 * "Quit" was entered. Check the rest of the command to see
 * whether we really quit the game.
 * @return true, if this command quits the game, false otherwise.
 */
private boolean quit(Command command)
{
    if(command.hasSecondWord()) {
        mainText.append("Quit what?");
        return false;
    }
    else {
        return true;  // signal that we want to quit
    }
}

/**
 * Checks if user has reached the move limit.
 * @return True if user has reached limit, otherwise false.
 */
private boolean checkMoves()
{
    if(player1.getMoves() == maxMoves){
        return true;
    }
    else{
        return false;
    }
}

/**
 * Try to pick up an item. Will print a message if the item cannot be picked up.
 * @param itemName The name of the item to try to pick up.
 */
private void pickUpItem(String itemName)
{
    try
    {
```

```
        Item item = currentRoom.findItem(itemName);
        int itemWeight = currentRoom.findItem(itemName).getWeight();
        if(player1.canPickUp(itemWeight, maxWeight)){
            player1.addItem(item);
            currentRoom.removeItem(itemName);
            player1.addWeight(item.getWeight());
            mainText.append(itemName + " picked up!   ");
        }
        else{
            mainText.append("Item too heavy!");
        }
    }
    catch(NullPointerException npe)
    {
        mainText.append("No item selected!");
    }
}

/**
 * Drops the specified item from inventory. Checks if item is in inventory,
 * and if so adds the item to the current room, then removes it from the inventory.
 * @param itemName Name of item to drop.
 */
private void dropItem(String itemName)
{
    Item item = player1.findItem(itemName);
    try
    {   if(item != null){
            for(int i = 0; i < player1.getInventory().size(); i++){
                currentRoom.addItem(item);
                if(player1.getInventory().get(i) == item){
                    player1.removeItem(i);
                }
            }
            player1.removeWeight(item.getWeight());
            mainText.append(itemName + " dropped!");
        }
        else{
            mainText.append("Item not in inventory!  ");
        }
    }
    catch(NullPointerException npe)
    {
        mainText.append("No item selected!");
    }
}

/**
 * Sets player to attack the specified monster. Player automatically uses
 * heaviest item in their inventory, for maximum damage.
 */
```

```
    private void attackMonster(String monstName)
    {
        Monster m = currentRoom.findMonster(monstName);
        double playerAttack = player1.getAttackStr();
        double monstAttack = m.getStrength();
        try
        {
            if(m.getHealth() < 1){
            mainText.append("Monster is dead!");
            }
            else{
                if(playerAttack >= monstAttack){
                    m.getHit(playerAttack);
                    mainText.setText("You hit monster for " + playerAttack + " damage! \n");
                    mainText.append(m.getDescription());


                    if(m.getHealth() < 1){
                        mainText.append(m.getName() + " is dead!");
                        if(m.getName().equals("mutant")){
                            mainText.append("\n The mutant dropped a potion, \n and a mysterious looking
key!");
                            Item pot = new Potion();
                            currentRoom.addItem(pot);
                            Item key = new Item("key", 1, 0, 0);
                            currentRoom.addItem(key);
                        }
                    }
                }
                else{
                    player1.getHit(monstAttack);
                    mainText.setText("Monster hit you for " + monstAttack + " damage!");
                    mainText.append("Your health: " + player1.getHealth());
                }
            }
        }
        catch(NullPointerException npe)
        {
            mainText.append("No monster selected!");
        }
    }

    /**
     * Checks if the player is still alive.
     * @return False if player is dead, else true.
     */
    private boolean checkAlive()
    {
        if(player1.getHealth() < 1){
            return false;
        }
```

```
    else{
        return true;
    }
}

/**
 * Checks if player has reached the goal room.
 * @return True if in the goal room, with the key, else false.
 */
private boolean reachedGoal()
{
    if((currentRoom == goalRoom) && (player1.checkInventory("key"))){
        return true;
    }
    else{
        return false;
    }
}

/**
 * The player drinks a potion.
 */
private void drinkPotion()
{
    mainText.append(player1.drinkPotion());
}

/**
 * Shows the inventory of the player.
 */
private void showInventory()
{
    mainText.setText(player1.showInventory(maxWeight));
}

/**
 * Starts the process for a user to pick up an item in-game.
 * Opens a dialog and asks user which item, then filters that name through  to
 * the regular pickUpItem() method.
 * @param frame The frame to which to associate the dialog.
 */
private void getItem()
{
    final JDialog dialog = new JDialog(frame, "Choose Item", true);
    JPanel dialogPanel = new JPanel();

    dialog.setLayout(new BorderLayout());
    dialogPanel.setLayout(new GridLayout(1, 2));

    JLabel itemLabel = new JLabel("Name of item: ");
    dialogPanel.add(itemLabel);
```

```
    final JTextArea itemInput = new JTextArea(1, 10); //needs to be final so it can be accessed by
the listener
    dialogPanel.add(itemInput);

    JButton submitButton = new JButton("Ok");
    submitButton.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                          pickUpItem(itemInput.getText());
                          dialog.setVisible(false);
                        }
                      });
    dialog.add(submitButton, BorderLayout.SOUTH);

    dialog.add(dialogPanel, BorderLayout.CENTER);
    dialog.setSize(new Dimension(200, 80));

    dialog.setVisible(true);
  }

  /**
   * Asks the user which item they would like to drop, and then drops it from their inventory,
   * and adds it to the room the player is currently in.
   */
  private void getDropItem()
  {
    final JDialog dialog = new JDialog(frame, "Choose Item", true);
    JPanel dialogPanel = new JPanel();

    dialog.setLayout(new BorderLayout());
    dialogPanel.setLayout(new GridLayout(1, 2));

    JLabel itemLabel = new JLabel("Name of item: ");
    dialogPanel.add(itemLabel);

    final JTextArea itemInput = new JTextArea(1, 10); //needs to be final so it can be accessed by
the listener
    dialogPanel.add(itemInput);

    JButton submitButton = new JButton("Ok");
    submitButton.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                          dropItem(itemInput.getText());
                          dialog.setVisible(false);
                        }
                      });
    dialog.add(submitButton, BorderLayout.SOUTH);

    dialog.add(dialogPanel, BorderLayout.CENTER);
    dialog.setSize(new Dimension(200, 80));
```

```
        dialog.setVisible(true);
    }

    /**
     * Asks the user which monster they would like to attack, and then proceeds with the attack
process.
     */
    private void getAttackMonster()
    {
        final JDialog dialog = new JDialog(frame, "Choose Monster", true);
        JPanel dialogPanel = new JPanel();

        dialog.setLayout(new BorderLayout());
        dialogPanel.setLayout(new GridLayout(1, 2));

        JLabel itemLabel = new JLabel("Name of monster: ");
        dialogPanel.add(itemLabel);

        final JTextArea monsterInput = new JTextArea(1, 10); //needs to be final so it can be accessed by
the listener
        dialogPanel.add(monsterInput);

        JButton submitButton = new JButton("Ok");
        submitButton.addActionListener(new ActionListener(){
                                    public void actionPerformed(ActionEvent e){
                                        attackMonster(monsterInput.getText());
                                        dialog.setVisible(false);
                                    }
                                });
        dialog.add(submitButton, BorderLayout.SOUTH);

        dialog.add(dialogPanel, BorderLayout.CENTER);
        dialog.setSize(new Dimension(250, 80));

        dialog.setVisible(true);
    }

    /**
     * Creates the GUI for the Zuul game.
     */
    private void setUpGUI()
    {
        mainText = new JTextArea(printWelcome());
        mainText.setEditable(false);
        frame = new JFrame("World of Zuul");
        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new BorderLayout()); // create the border layout
        contentPane.add(mainText, BorderLayout.CENTER); // add the main text label to the center
area

        //create grid for the 6 buttons in the south of the screen
```

```
JPanel southButtonGrid = new JPanel();
southButtonGrid.setLayout(new GridLayout(2, 3));
// create buttons for the south button grid
JButton showInventory = new JButton("Show Inventory");
JButton moveUp = new JButton("Up");
JButton getItem = new JButton("Get...");
JButton moveLeft = new JButton("Left");
JButton moveDown = new JButton("Down");
JButton moveRight = new JButton("Right");
//add event listeners for each button in the south grid
showInventory.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                           nextCommand = parser.getCommand("show", null);
                           play();
                        }
                     });
moveUp.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("go", "north");
                       play();
                    }
                 });
getItem.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("get", null);
                       play();
                    }
                 });
moveLeft.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("go", "west");
                       play();
                    }
                 });
moveDown.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("go", "south");
                       play();
                    }
                 });
moveRight.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("go", "east");
                       play();
                    }
                 });
//add buttons to the grid layout
southButtonGrid.add(showInventory);
southButtonGrid.add(moveUp);
southButtonGrid.add(getItem);
southButtonGrid.add(moveLeft);
```

```
southButtonGrid.add(moveDown);
southButtonGrid.add(moveRight);
//add the gridpanel to the contentpane
contentPane.add(southButtonGrid, BorderLayout.SOUTH);

//create grid for 5 buttons on left-hand-side of the screen
JPanel leftButtonGrid = new JPanel();
leftButtonGrid.setLayout(new GridLayout(4, 1));
//create buttons for the leftbuttongrid
JButton attack = new JButton("Attack");
JButton potion = new JButton("Drink Potion");
JButton dropItem = new JButton("Drop...");
JButton help = new JButton("Help");
//add event listeners for each button in the left grid
attack.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("attack", null);
                       play();
                    }
                 });
potion.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("potion", null);
                       play();
                    }
                 });
dropItem.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("drop", null);
                       play();
                    }
                 });
help.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent e){
                       nextCommand = parser.getCommand("help", null);
                       play();
                    }
                });
//add buttons to the grid layout
leftButtonGrid.add(attack);
leftButtonGrid.add(potion);
leftButtonGrid.add(dropItem);
leftButtonGrid.add(help);
//add the gridpanel to the contentpane - first add it to a container panel with flow layout
//so that it does not make the buttons super tall to fill up the entire height of the window
JPanel leftGridContainer = new JPanel();
leftGridContainer.setLayout(new FlowLayout());
leftGridContainer.add(leftButtonGrid);
contentPane.add(leftGridContainer, BorderLayout.WEST);

//create the menu bar
```

```
    JMenuBar menuBar = new JMenuBar();
    frame.setJMenuBar(menuBar);

    // create file menu
    JMenu fileMenu = new JMenu("File");
    menuBar.add(fileMenu);

    // create save and load buttons
    JMenuItem saveMenuItem = new JMenuItem("Save");
    JMenuItem loadMenuItem = new JMenuItem("Load");
    JMenuItem quitMenuItem = new JMenuItem("Quit");
    // add menu items to file menu
    fileMenu.add(saveMenuItem);
    fileMenu.add(loadMenuItem);
    fileMenu.add(quitMenuItem);
    // add event listeners for menu items
    saveMenuItem.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                           saveGameDialog();
                        }
                    });
    loadMenuItem.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
          loadGameDialog();
       }
    });
    quitMenuItem.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                           System.exit(0);
                        }
                    });
    frame.pack();
    frame.setVisible(true);
}

/**
 * Gets the location at which to save the player's game. Sends this location to another method
 * which does the actual saving process.
 */
private void saveGameDialog()
{
    final JDialog dialog = new JDialog(frame, "Save Game", true);
    JPanel dialogPanel = new JPanel();

    dialog.setLayout(new BorderLayout());
    dialogPanel.setLayout(new GridLayout(1, 2));

    JLabel itemLabel = new JLabel("Save name(not includeing .txt): ");
    dialogPanel.add(itemLabel);
```

```
      final JTextArea fileNameInput = new JTextArea(1, 10); //needs to be final so it can be accessed
by the listener
      dialogPanel.add(fileNameInput);

      JButton submitButton = new JButton("Ok");
      submitButton.addActionListener(new ActionListener(){
                                 public void actionPerformed(ActionEvent e){
                                    saveGame(fileNameInput.getText());
                                    dialog.setVisible(false);
                                 }
                              });
      dialog.add(submitButton, BorderLayout.SOUTH);

      dialog.add(dialogPanel, BorderLayout.CENTER);
      dialog.setSize(new Dimension(400, 80));

      dialog.setVisible(true);
   }

   /**
    * Performs the action of saving the player's game. Saves all relevant data to a text file.
    * Saves in the same directory as the game file.
    * @param fileName The name of the file to save to.
    */
   private void saveGame(String fileName)
   {
      try
      {
         String saveGameName = fileName + ".txt";
         File myFile = new File(saveGameName);
         FileWriter fw = new FileWriter(myFile);
         BufferedWriter out = new BufferedWriter(fw);
         String writeText = currentRoom.getName() + System.getProperty("line.separator") +
player1.getCurrMoves() + System.getProperty("line.separator") + player1.getCurrWeight() +
System.getProperty("line.separator") + player1.getHealth();
         ArrayList<Item> playerInventory = player1.getInventory();
         for(Item i: playerInventory){
            writeText += (System.getProperty("line.separator") + i.getName());
         }
         out.write(writeText);
         out.close();
      }
      catch(IOException ioe)
      {
         mainText.append("Save error!");
      }
   }

   /**
    * Gets the location at which the player's game is saved. Sends this location to another method
    * which loads the data. Assumes that the save is in the same directory as the game file.
```

```
    */
    private void loadGameDialog()
    {
        final JDialog dialog = new JDialog(frame, "Load Game", true);
        JPanel dialogPanel = new JPanel();

        dialog.setLayout(new BorderLayout());
        dialogPanel.setLayout(new GridLayout(1, 2));

        JLabel itemLabel = new JLabel("Save name (not includeing .txt): ");
        dialogPanel.add(itemLabel);

        final JTextArea fileNameInput = new JTextArea(1, 10); //needs to be final so it can be accessed
by the listener
        dialogPanel.add(fileNameInput);

        JButton submitButton = new JButton("Ok");
        submitButton.addActionListener(new ActionListener(){
                                    public void actionPerformed(ActionEvent e){
                                        loadGame(fileNameInput.getText());
                                        dialog.setVisible(false);
                                    }
                                });
        dialog.add(submitButton, BorderLayout.SOUTH);

        dialog.add(dialogPanel, BorderLayout.CENTER);
        dialog.setSize(new Dimension(400, 80));

        dialog.setVisible(true);
    }

    /**
     * Performs the action of loading the player's game.
     * @param saveName The name of the game save.
     */
    private void loadGame(String saveName)
    {
        try
        {
            String fileName = saveName + ".txt";
            BufferedReader in = new BufferedReader(new FileReader(fileName));
            String nextLine = in.readLine();
            currentRoom = randomiser.findRoom(nextLine);
            nextLine = in.readLine();
            player1.setCurrMoves(Integer.parseInt(nextLine));
            nextLine = in.readLine();
            player1.setWeight(Integer.parseInt(nextLine));
            nextLine = in.readLine();
            player1.setHealth(Double.parseDouble(nextLine));

            nextLine = in.readLine(); // assign here to prepare for loop
```

```
            while(nextLine != null){
                player1.addItem(findItemForLoad(nextLine));
                nextLine = in.readLine();
            }

             in.close();
        }
        catch(FileNotFoundException notFound)
        {
            mainText.append("Loading error!");
        }
        catch(NullPointerException npe)
        {
            mainText.append("Loading error!");
        }
       catch(IOException ioe)
       {
            mainText.append("Loading error!");
       }
       finally
       {
          mainText.setText(currentRoom.getLongDescription());
       }
    }

    /**
     * Finds the specified item - used for loading
     * @param itemName The name of the item to find.
     * @return The item found.
     */
    private Item findItemForLoad(String itemName)
    {
        for(Item i: allItems){
            if(i.getName().equals(itemName)){
                return i;
            }
        }
        return null;
    }
}
```

Room

```
import java.util.Set;
import java.util.HashMap;
import java.util.Iterator;
import java.util.ArrayList;

/**
 * Class Room - a room in an adventure game.
```

```
 *
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * A "Room" represents one location in the scenery of the game.  It is
 * connected to other rooms via exits.  For each existing exit, the room
 * stores a reference to the neighboring room.
 *
 * @author  Charlotte Pierce
 * @version 6/11/2010
 */

public class Room
{
   private String name;
   private String description;
   private HashMap<String, Room> exits;        // stores exits of this room.
   private ArrayList<Item> items;
   private ArrayList<Monster> monsters;
   private RoomRandomiser roomRandomiser;

   /**
    * Create a room described "description". Initially, it has
    * no exits. "description" is something like "a kitchen" or
    * "an open court yard".
    * @param description The room's description.
    * @param name The room's name.
    * @param rand The room randomiser.
    */
   public Room(String name, String description, RoomRandomiser rand)
   {
      this.description = description;
      exits = new HashMap<String, Room>();
      items = new ArrayList<Item>();
      monsters = new ArrayList<Monster>();
      rand.addRoom(this);
      roomRandomiser = rand;
      this.name = name;
   }

   /**
    * Define an exit from this room.
    * @param direction The direction of the exit.
    * @param neighbor  The room to which the exit leads.
    */
   public void setExit(String direction, Room neighbor)
   {
      exits.put(direction, neighbor);
   }

   /**
```

```
 * @return The short description of the room
 * (the one that was defined in the constructor).
 */
public String getShortDescription()
{
   return description;
}

/**
 * Return a description of the room in the form:
 *     You are in the kitchen.
 *     Exits: north west
 * @return A long description of this room
 */
public String getLongDescription()
{
   return  description + ".\n" + getExitString() + "\n" + getItemsDesc() + getMonstersDesc();
}

/**
 * Return a string describing the room's exits, for example
 * "Exits: north west".
 * @return Details of the room's exits.
 */
private String getExitString()
{
   String returnString = "Exits:";
   Set<String> keys = exits.keySet();
   for(String exit : keys) {
      returnString += " " + exit;
   }
   return returnString;
}

/**
 * Return the room that is reached if we go from this room in direction
 * "direction". If there is no room in that direction, return null.
 * @param direction The exit's direction.
 * @return The room in the given direction.
 */
public Room getExit(String direction)
{
   return exits.get(direction);
}

/**
 * Finds and returns an item if it is in the room.
 * @param itemName Name of the item to find.
 * @return The item found.
 */
public Item findItem(String itemName)
```

```
{
   for(Item item: items){
      if(itemName.equals(item.getName())){
         return item;
      }
   }
   return null;
}

/**
 * Finds and returns a monster if it is in the room.
 * @param monsterName Name of the monster to find.
 * @return The monster found.
 */
public Monster findMonster(String monstName)
{
   for(Monster m: monsters){
      if(monstName.equals(m.getName())){
         return m;
      }
   }
   return null;
}

/**
 * Find the item with the specified name and removes it from the room.
 * @param itemName The name of the item to remove.
 */
public void removeItem(String itemName)
{
   for(int i = 0; i < items.size(); i++){
      if(items.get(i).getName().equals(itemName)){
         items.remove(i);
      }
   }
}

/**
 * Adds an item to the room.
 * @param item The item to add.
 */
public void addItem(Item item)
{
   items.add(item);
}

/**
 * Returns a String representing a description of all items in the room.
 * @return Description of all the items in the room.
 */
private String getItemsDesc()
```

```
{
   String returnString = " Items: ";
   if(items.size() == 0){
      returnString += "\n   No items!";
   }
   else{
      for(Item item: items){
         returnString += item.getDescription();
      }
   }
   return returnString;
}

/**
 * Returns a String representing a description of all monsters in the room.
 * @return Description of all the monsters in the room.
 */
private String getMonstersDesc()
{
   String returnString = "\n Monsters: ";
   if(monsters.size() == 0){
      returnString += "\n   No monsters!";
   }
   else{
      for(Monster m: monsters){
         returnString += m.getDescription();
      }
   }
   return returnString;
}

/**
 * Adds a monster to the room.
 * @param m The monster to add.
 */
public void addMonster(Monster m)
{
   monsters.add(m);
}

/**
 * Returns the room randomiser (so that TransporterRoom class can use it)
 * @return The RoomRandomiser obejct.
 */
public RoomRandomiser getRandomiser()
{
   return roomRandomiser;
}

/**
 * Returns the name of the room.
```

```
     * @return The name of the room.
     */
    public String getName()
    {
        return name;
    }
}
```

TransporterRoom

```
/**
 * A transporter room which can be included in the game. When a player tries to leave a
 * transporter room, they arrive in a random room from the game.
 *
 * @author Charlotte Pierce
 * @version 6/11/2010
 */
public class TransporterRoom extends Room
{
    /**
     * Creates a room.
     * @param name Then name of the room.
     * @param description The description of the room.
     * @param rand The room randomiser.
     */
    public TransporterRoom(String name, String description, RoomRandomiser rand)
    {
        super(name, description, rand);
    }

    /**
     * Uses the Randomiser to return a random room to which the player is directed.
     */
    public Room getExit(String direction)
    {
        return (getRandomiser()).getRandomRoom();
    }
}
```

RoomRandomiser

```
import java.util.Random;
import java.util.ArrayList;

/**
 * A class which stores all the rooms in the game, and can return a random one if called for.
 *
 * @author Charlotte Pierce
```

```
 * @version 6/11/2010
 */
public class RoomRandomiser
{
   private Random rand;
   private ArrayList<Room> rooms;

   public RoomRandomiser()
   {
      rand = new Random();
      rooms = new ArrayList<Room>();
   }

   /**
    * Returns a random room from the list of rooms.
    * Makes sure a viable room is chosen, by limiting the scope of the output
    * to any number between 0 and rooms.size()
    * @return Room A random room from within the game.
    */
   public Room getRandomRoom()
   {
      return (rooms.get(rand.nextInt(rooms.size())));
   }

   /**
    * Adds a room to the randomisers' list.
    */
   public void addRoom(Room toAdd)
   {
      rooms.add(toAdd);
   }

   /**
    * Finds the room in the list of rooms with the specified string.
    */
   public Room findRoom(String find)
   {
      for(Room r: rooms){
         if(r.getName().equals(find)){
            return r;
         }
      }
      return null;
   }
}
```

Item

```
/**
 * An item that can be interacted with in the game.
 *
 * @author Charlotte Pierce
 * @version 1/10/2010
 */
public class Item
{
   private String name;
   private int weight;
   private int value;
   private int damage;

   /**
    * Constructor for objects of class Item
    * @param aName Name of the item.
    * @param aWeight Weight of the item.
    * @param aValue Value of the item.
    * @param aDamage Damage of the item.
    */
   public Item(String aName, int aWeight, int aValue, int aDamage)
   {
      name = aName;
      weight = aWeight;
      value = aValue;
      damage = aDamage;
   }

   /**
    * Returns a string representing the name of the item.
    * @return The name of the item.
    */
   public String getName()
   {
      return name;
   }

   /**
    * Returns an int representing the weight of the item.
    * @return The weight of the item.
    */
   public int getWeight()
   {
      return weight;
   }

   /**
```

```
    * Returns an int representing the value of the item.
    * @return The value of the item.
    */
   public int getValue()
   {
      return value;
   }

   /**
    * Returns an int representing the damage of the item.
    * @return The damage of the item.
    */
   public int getDamage()
   {
      return damage;
   }

   /**
    * Returns a String description of the item.
    * @return Description of the item.
    */
   public String getDescription()
   {
      String returnString = "";
      returnString += "\n " + name + "(" + weight + " weight) (" + value + " value) (" + damage + "
dmg)";
      return returnString;
   }
}
```

Potion

```
/**
 * A potion. Restores the drinker's health by 50 points.
 *
 * @author Charlotte Pierce
 * @version 6/11/2010
 */public class Potion extends Item
{
   private int restoreVal;

   /**
    * Constructor for objects of class Potion
    */
   public Potion()
   {
      super("potion", 25, 500, 0);
      restoreVal = 50;
   }
```

```
    public int getRestoreVal()
    {
        return restoreVal;
    }
}
```

IntelligentBeing

```
/**
 * A class representing an object within the game that can be perceived as being 'alive'.
 * This would most likely be a human, a monster/enemy (even monsters which within the context of
the
 * game are of a robotic nature), or the player character.
 * Every being has strength and health. Strength affects their damage in battle.
 *
 * @author Charlotte Pierce
 * @version 5/11/2010
 */
public abstract class IntelligentBeing
{
    private int strength; //the strength which the being can hit with
    private double health;

    protected IntelligentBeing(int strength, double health)
    {
        this.strength = strength;
        this.health = health;
    }

    /**
     * Returns the being's strength.
     * @return The being's strength.
     */
    protected int getStrength()
    {
        return strength;
    }

    /**
     * Returns the being's current health.
     * @return The being's current health.
     */
    protected double getHealth()
    {
        return health;
    }

    /**
```

```
    * Set the health of the being.
    * @param hp Health value to assign.
    */
   protected void setHealth(double hp)
   {
      this.health = hp;
   }

   /**
    * Being gets 'hit' with the specified strength. Health goes down by that strength value.
    * @param strength Strength hit with.
    */
   public void getHit(double howMuch)
   {
      health -= howMuch;
   }

   /**
    * Adds a certain amount of health to the being's current health.
    * @param howMuch The amount of health points to add.
    */
   public void addHealth(double howMuch)
   {
      health += howMuch;
   }

   /**
    * Calculates and returns the strength of the being's attack.
    * @return The being's attack strength.
    */
   public abstract double getAttackStr();
}
```

Player

```
import java.util.ArrayList;

/**
 * Player class for the text game. This class holds information about the player, such as the number
 * of moves they have made, the items they are currently holding, and the weight of those items.
 *
 * @author Charlotte Pierce
 * @version 1/10/2010
 */
public class Player extends IntelligentBeing
{
   private int currMoves;
   private int currWeight;
   private ArrayList<Item> inventory;
```

```
/**
 * Initialises the player. Health is set to 5, and strength is 10.
 * The number of moves taken, and the weight of items the player holds is
 * set to zero (just to make sure).
 * The ArrayList for the inventory is also initialised.
 */
public Player()
{
   super(10, 5);
   currMoves = 0;
   currWeight = 0;
   inventory = new ArrayList<Item>();
}

/**
 * Prints a string describing the players current inventory.
 * @return A string describing the players inventory.
 */
public String showInventory(int maxWeight)
{
   String printString = "Current Items:";
   if(inventory.size() == 0){
      printString += " no items!";
   }
   else{
      for(Item item: inventory){
         printString += item.getDescription();
      }
   }
   printString += "\n Weight: " + currWeight + "/" + maxWeight;
   return printString;
}

 /**
 * Checks if an item is in the inventory.
 * @param itemName Item to check for.
 * @return True if item is held, else false.
 */
public boolean checkInventory(String itemName)
{
   for(Item item: inventory){
      if(itemName.equals(item.getName())){
         return true;
      }
   }
   return false;
}

/**
 * Adds one to the number of moves taken by the player.
 */
```

```
public void addMove()
{
   currMoves += 1;
}

/**
 * Adds the specified weight to the player.
 * @param weight The weight to add.
 */
public void addWeight(int weight)
{
   currWeight += weight;
}

/**
 * Removes the specified weight to the player.
 * @param weight The weight to remove.
 */
public void removeWeight(int weight)
{
   currWeight -= weight;
}

/**
 * Get the number of moves the player has made.
 * @return Number of moves made by the player.
 */
public int getMoves()
{
   return currMoves;
}

/**
 * Get the current total weight the player is carrying.
 * @return Total weight player is carrying.
 */
public int getWeight()
{
   return currWeight;
}

/**
 * Adds item to the players' inventory.
 * @param item The item to add to the inventory.
 */
public void addItem(Item item)
{
   inventory.add(item);
}

/**
```

```
 * Removes item from player's inventory.
 * @param item The index of the item to remove.
 */
public void removeItem(int index)
{
   inventory.remove(index);
}

/**
 * Returns the player's inventory.
 * @return The player's inventory.
 */
public ArrayList<Item> getInventory()
{
   return inventory;
}

   /**
 * Checks whether the item can be picked up or not.
 * @param item Item to try to pick up.
 * @return True if item can be picked up, otherwise false.
 */
public boolean canPickUp(int itemWeight, int maxWeight)
{
   if((currWeight + itemWeight) <= maxWeight){
      return true;
   }
   else{
      return false;
   }
}

/**
 * Finds and returns an item if it is in the player's inventory.
 * @param itemName Name of the item to find.
 * @return The item found.
 */
public Item findItem(String itemName)
{
   for(Item item: inventory){
      if(itemName.equals(item.getName())){
         return item;
      }
   }
   return null;
}

/**
 * Calculates and returns the strength of the player's attack. This is based on the
 * player's strength, and the most damaging item they are carrying.
 * @return The players attack strength.
```

```
 */
public double getAttackStr()
{
   Item best = findMostDamaging();
   double attackStr = getStrength();
   if(best != null){
      attackStr += best.getDamage() * 1.5;
   }
   return attackStr;
}

/**
 * Finds the most damaging item in the character's inventory and returns it.
 * @return The most damaging item in the inventory.
 */
private Item findMostDamaging()
{
   if(inventory.size() > 0){
      Item best = inventory.get(0);
      for(Item item: inventory){
         if(item.getDamage() > best.getDamage()){
            best = item;
         }
      }
      return best;
   }
   else{
      return null;
   }
}

/**
 * Player drinks a potion to restore some of their health. Prints a message if player
 * is not currently holding a potion, otherwise removes potion from inventory, and
 * adds to players' health.
 * @return The status of the player after drinking, or a message saying there are no
 * potions in the inventory.
 */
public String drinkPotion()
{
   Item item = findItem("potion");
   Potion pot = (Potion) item;
   String returnString = "";
   if(pot != null){
      addHealth(pot.getRestoreVal());
      for(int i = 0; i < inventory.size(); i++){
         if(inventory.get(i) == pot){
            removeItem(i);
         }
      }
      returnString += "Drank a potion! \n Health is now: " + getHealth();
```

```
        }
        else{
            returnString += "No potions in inventory! \n Health is: " + getHealth();
        }
        return returnString;
    }

    /**
     * Returns the number of moves the player has made.
     * @return The number of moves the player has made.
     */
    public int getCurrMoves()
    {
        return currMoves;
    }

    /**
     * Returns the current weight held by the player.
     * @return The current weight held by the player.
     */
    public int getCurrWeight()
    {
        return currWeight;
    }

    /**
     * Sets the value for currMoves - used for loading
     * @param curr The new value for currMoves.
     */
    public void setCurrMoves(int curr)
    {
        currMoves = curr;
    }

    /**
     * Sets the value for currWeight - used for loading
     * @param newWeight The new value for currWeight.
     */
    public void setWeight(int newWeight)
    {
        currWeight = newWeight;
    }
}
```

Monster

```
/**
 * A hostile creature found in the game world. Player can attack the monster
 *
 * @author Charlotte Pierce
 * @version 1/10.2010
 */
public class Monster extends IntelligentBeing
{
   private String name;

   /**
    * Constructor for objects of class Monster.
    * @param name The name of the monster.
    * @param strength The strength of the monster.
    * @param health The health of the monster.
    */
   public Monster(String name, int strength, double health)
   {
      super(strength, health);
      this.name = name;
   }

   /**
    * Returns a string representing a description of the monster.
    * @return A description of the monster.
    */
   public String getDescription()
   {
      String returnString = "";
      returnString += "\n" + name + " (" + getHealth() + "hp) (" + getStrength() + " str)";
      return returnString;
   }

   /**
    * Returns the monsters name.
    * @return The monsters' name.
    */
   public String getName()
   {
      return name;
   }

   /**
    * Calculates and returns the strength of the monster's attack.
    * @return The monster's attack strength.
```

```
    */
   public double getAttackStr()
   {
      return getStrength();
   }
}
```

Parser

```
/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * This parser reads user input and tries to interpret it as an "Adventure"
 * command. Every time it is called it reads a line from the terminal and
 * tries to interpret the line as a two-word command. It returns the command
 * as an object of class Command.
 *
 * The parser has a set of known command words. It checks user input against
 * the known commands, and if the input is not one of the known commands, it
 * returns a command object that is marked as an unknown command.
 *
  * @author  Charlotte Pierce
 * @version 6/11/2010
 */
public class Parser
{
   private CommandWords commands;  // holds all valid command words

   /**
    * Create a parser to read from the terminal window.
    */
   public Parser()
   {
      commands = new CommandWords();
   }

   /**
    * Creates and returns a command, as if the command words 'word1' and 'word2' were typed into
the system
    * @return The next command from the user.
    */
   public Command getCommand(String aWord1, String aWord2)
   {
      String word1 = aWord1;
      String word2 = aWord2;
      return new Command(commands.getCommandWord(word1), word2);
   }

   /**
```

```
    * Return out a list of valid command words.
    * @return A list of all valid command words.
    */
   public String showCommands()
   {
      return commands.showAll();
   }
}
```

CommandWords

```
import java.util.HashMap;

/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * This class holds an enumeration of all command words known to the game.
 * It is used to recognise commands as they are typed in.
 *
* @author  Charlotte Pierce
 * @version 6/11/2010
 */

public class CommandWords
{
   // A mapping between a command word and the CommandWord
   // associated with it.
   private HashMap<String, CommandWord> validCommands;

   /**
    * Constructor - initialise the command words.
    */
   public CommandWords()
   {
      validCommands = new HashMap<String, CommandWord>();
      for(CommandWord command : CommandWord.values()) {
         if(command != CommandWord.UNKNOWN) {
            validCommands.put(command.toString(), command);
         }
      }
   }

   /**
    * Find the CommandWord associated with a command word.
    * @param commandWord The word to look up.
    * @return The CommandWord correspondng to commandWord, or UNKNOWN
    *         if it is not a valid command word.
    */
   public CommandWord getCommandWord(String commandWord)
```

```
    {
        CommandWord command = validCommands.get(commandWord);
        if(command != null) {
            return command;
        }
        else {
            return CommandWord.UNKNOWN;
        }
    }

    /**
     * Check whether a given String is a valid command word.
     * @return true if it is, false if it isn't.
     */
    public boolean isCommand(String aString)
    {
        return validCommands.containsKey(aString);
    }

    /**
     * Return all valid commands as a string.
     * @return All valid commands.
     */
    public String showAll()
    {
        String returnString = "";
        for(String command : validCommands.keySet()) {
            returnString += (command + "  ");
        }
        returnString += "\n";
        return returnString;
    }
}
```

Command

```
/**
 * This class is part of the "World of Zuul" application.
 * "World of Zuul" is a very simple, text based adventure game.
 *
 * This class holds information about a command that was issued by the user.
 * A command currently consists of two parts: a CommandWord and a string
 * (for example, if the command was "take map", then the two parts
 * are TAKE and "map").
 *
 * The way this is used is: Commands are already checked for being valid
 * command words. If the user entered an invalid command (a word that is not
 * known) then the CommandWord is UNKNOWN.
 *
 * If the command had only one word, then the second word is <null>.
```

```
 *
* @author  Charlotte Pierce
 * @version 6/11/2010
 */

public class Command
{
   private CommandWord commandWord;
   private String secondWord;

   /**
    * Create a command object. First and second words must be supplied, but
    * the second may be null.
    * @param commandWord The CommandWord. UNKNOWN if the command word
    *                was not recognised.
    * @param secondWord The second word of the command. May be null.
    */
   public Command(CommandWord commandWord, String secondWord)
   {
      this.commandWord = commandWord;
      this.secondWord = secondWord;
   }

   /**
    * Return the command word (the first word) of this command.
    * @return The command word.
    */
   public CommandWord getCommandWord()
   {
      return commandWord;
   }

   /**
    * @return The second word of this command. Returns null if there was no
    * second word.
    */
   public String getSecondWord()
   {
      return secondWord;
   }

   /**
    * @return true if this command was not understood.
    */
   public boolean isUnknown()
   {
      return (commandWord == CommandWord.UNKNOWN);
   }

   /**
    * @return true if the command has a second word.
```

```
    */
    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }
}
```

CommandWord

```
/**
 * Representations for all the valid command words for the game
 * along with a string in a particular language.
 *
 * @author Charlotte Pierce
 * @version 6/11/2010
 */
public enum CommandWord
{
    // A value for each command word along with its
    // corresponding user interface string.
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?"), GET("get"), DROP("drop"), SHOW("show"),
ATTACK("attack"), POTION("potion");

    // The command string.
    private String commandString;

    /**
     * Initialise with the corresponding command word.
     * @param commandWord The command string.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * @return The command word as a string.
     */
    public String toString()
    {
        return commandString;
    }
}
```

RANDOM TESTER

Summary

Project: Random Tester
Author: Charlotte Pierce

This project allows the user to generate some random numbers. Each random number generate must have a limit specified, which defines the range of numbers that the generator could return (range = 1 -> limit). The user can then request a certain number of random numbers. The application will then generate that number of random digits, keeping track of how often each digit is returned. It then prints the details of the number generation, showing what numbers were generated, how many times each number was generated, and the percentage of all the numbers generated that the frequency represents.
Note: if the number was not generated at all, it is not shown in the output, for the purpose of brevity.

To use this application:
- create an instance of RandomTester with a certain limit
- use the generate() method, specifying the number of random digits to generate
Alternatively:
Run the main method in the Test class, which shows a number of possible uses for the random numeber generator.

This project demonstrates the use of the Random class. It is fully documented using javadoc style. The project implements all of the required features. It also includes a table showing the tests performed on the RandomTester class, to see how accurate its number generation is.

Experiment Table

Experiment	Expected Outcome	Actual Outcome
Limit: 10 **Numbers generated: 10**	Each number is generated once.	3: 2 [0.2%] 4: 2 [0.2%] 5: 3 [0.3%] 9: 1 [0.1%] 10: 3 [0.3%]
Limit: 10 **Number generated: 20**	Each number is generated twice.	1: 1 [0.05%] 2: 2 [0.1%] 3: 1 [0.05%] 4: 2 [0.1%] 5: 2 [0.1%] 6: 2 [0.1%] 7: 4 [0.2%] 8: 2 [0.1%] 9: 4 [0.2%] 10: 1 [0.05%]
Limit: 10 **Number generated: 30**	Each number is generated three times.	1: 2 [0.06666666666666667%] 2: 1 [0.03333333333333333%] 3: 5 [0.16666666666666666%] 4: 5 [0.16666666666666666%] 5: 2 [0.06666666666666667%] 6: 3 [0.1%] 7: 3 [0.1%] 8: 5 [0.16666666666666666%] 9: 3 [0.1%] 10: 2 [0.06666666666666667%]
Limit: 10 **Number generated: 40**	Each number is generated four times.	1: 5 [0.125%] 2: 3 [0.075%] 3: 3 [0.075%] 4: 3 [0.075%] 5: 5 [0.125%] 6: 5 [0.125%] 7: 5 [0.125%] 8: 2 [0.05%] 9: 6 [0.15%] 10: 4 [0.1%]
Limit: 10 **Number generated: 50**	Each number is generated five times.	1: 3 [0.06%] 2: 8 [0.16%] 3: 3 [0.06%] 4: 4 [0.08%] 5: 6 [0.12%] 6: 5 [0.1%] 7: 6 [0.12%] 8: 3 [0.06%] 9: 8 [0.16%] 10: 5 [0.1%]
Limit: 5 **Number generated: 50**	Each number is generated ten times.	1: 7 [0.14%] 2: 12 [0.24%] 3: 10 [0.2%]

		4: 12 [0.24%] 5: 10 [0.2%]
Limit: 5 **Number generated: 100**	Each number is generated twenty times.	1: 20 [0.2%] 2: 25 [0.25%] 3: 18 [0.18%] 4: 17 [0.17%] 5: 21 [0.21%]
Limit: 5 **Number generated: 1000**	Each number is generated two-hundred times.	1: 198 [0.198%] 2: 224 [0.224%] 3: 191 [0.191%] 4: 194 [0.194%] 5: 194 [0.194%]

The tests show that the accuracy of the number generation, according to logical conventions (i.e. each number has the exact same chance of being generated, thus each number will be generated exactly the same number times), increases with the number of digits that are generated.

Code

RandomTester

```
import java.util.Random;

/**
 * Write a description of class RandomTester here.
 *
 * @author Charlotte Pierce
 * @version 4/9/2010
 */
public class RandomTester
{
   private int limit; //the range of random numbers that will be generated
   private Random generator;
   private int[] frequencies;
   private int requested; //number of random values requested to be generated

   /**
    * Constructor for objects of class RandomTester
    */
   public RandomTester(int lim)
   {
      limit = lim;
      generator = new Random();
      frequencies = new int[limit];
   }

   /**
    * Generates the specified number of random numbers, from 1 to limit (defined in constructor)
    * Increments appropriate value in frequencies as numbers are generated.
    * @param amount The number of numbers to generate.
    */
   public void generate(int amount)
   {
      requested = amount;
      for(int i = 0; i < (requested + 1); i++){
         int tempInt = generator.nextInt(limit) + 1; //get the 'right' number
         frequencies[tempInt - 1]++; //increase frequencies
      }
      printDetails();
   }

   /**
    * Displays in the console the number of values requested, and the number of times each value
from
    * 1 to limit occurred, as well as the percentage of total numbers generated this represents.
    * Does not print details of numbers which were not generated at all.
    */
   private void printDetails()
   {
```

```
      System.out.println("FORMAT - <Number>: <Frequency> [<Percentage>]");
      for(int i = 0; i < limit; i++){
         double percent = (double)frequencies[i] / (double)requested; //get percentage.
         int number = i + 1;
         if(percent > 0){
            System.out.println(number + ": " + frequencies[i] + "  [" + percent + "%]");
         }
      }
   }
}
```

Test

```
/**
 * Tests the random number generator by calling for
 * some outputs.

 * @author Charlotte Pierce
 * @version 4/9/2010
 */

public class Test
{
   public static void main(String[] args)
   {
      System.out.println("10 DICE ROLLS:");
      RandomTester dice = new RandomTester(6);
      dice.generate(10);
      System.out.println("100 COIN TOSSES:");
      RandomTester coin = new RandomTester(2);
      coin.generate(100);
      System.out.println("100,000 COIN TOSSES:");
      RandomTester coin2 = new RandomTester(2);
      coin.generate(100000);
   }
}
```

EX1.16 – SUNSET

Summary

Project "picture"
Authors: Michael Kolling and David J. Barnes with extension by Charlotte Pierce

The extension in this project is part of the material for the Object-Oriented programming portfolio of Charlotte Pierce.

This project demonstrates the ability to read, understand, and modify other's code in order to perform a new function. In this application, the new function is causing the sun object to move down the screen in the manner of a sunset.

To see this extension:
- create an instance of Picture
- execute the draw() method
- execute the sunset() method

Code

Picture

```
/**
 * This class represents a simple picture. You can draw the picture using
 * the draw method. But wait, there's more: being an electronic picture, it
 * can be changed. You can set it to black-and-white display and back to
 * colors (only after it's been drawn, of course).
 *
 * This class was written as an early example for teaching Java with BlueJ, and
 * extended as part of the weekly lab work.
 *
 * @author  Charlotte Pierce
 * @version 13/8/2010
 */
public class Picture
{
   private Square wall;
   private Square window;
   private Triangle roof;
   private Circle sun;
   private Circle sun2;

   /**
    * Constructor for objects of class Picture
    */
   public Picture()
   {
      // nothing to do... instance variables are automatically set to null
   }

   /**
    * Draw this picture.
    */
   public void draw()
   {
      wall = new Square();
      wall.moveVertical(80);
      wall.changeSize(100);
      wall.makeVisible();

      window = new Square();
      window.changeColor("black");
      window.moveHorizontal(20);
      window.moveVertical(100);
      window.makeVisible();

      roof = new Triangle();
      roof.changeSize(50, 140);
      roof.moveHorizontal(60);
      roof.moveVertical(70);
```

```
    roof.makeVisible();

    sun = new Circle();
    sun.changeColor("blue"); //exercise 1.13
    sun.moveHorizontal(180);
    sun.moveVertical(-10);
    sun.changeSize(60);
    sun.makeVisible();

    //exercise 1.14
    sun2 = new Circle();
    sun2.changeColor("yellow");
    sun2.moveHorizontal(150);
    sun2.moveVertical(-10);
    sun2.changeSize(50);
    sun2.makeVisible();

    //exercise 1.15
    //sun2.slowMoveVertical(200);
}

/**
 * Change this picture to black/white display
 */
public void setBlackAndWhite()
{
    if(wall != null)   // only if it's painted already...
    {
        wall.changeColor("black");
        window.changeColor("white");
        roof.changeColor("black");
        sun.changeColor("black");
    }
}

//exercise 1.16
/**
 * Move the sun down to simluate a sunset.
 */
public void sunset()
{
    sun2.slowMoveVertical(200);
}

/**
 * Change this picture to use color display
 */
public void setColor()
{
    if(wall != null)   // only if it's painted already...
    {
```

```
            wall.changeColor("red");
            window.changeColor("black");
            roof.changeColor("green");
            sun.changeColor("yellow");
        }
    }
}
```

Triangle

```
import java.awt.*;

/**
 * A triangle that can be manipulated and that draws itself on a canvas.
 *
 * @author  Michael Kolling and David J. Barnes
 * @version 2008.03.30
 */

public class Triangle
{
    private int height;
    private int width;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new triangle at default position with default color.
     */
    public Triangle()
    {
        height = 30;
        width = 40;
        xPosition = 50;
        yPosition = 15;
        color = "green";
        isVisible = false;
    }

    /**
     * Make this triangle visible. If it was already visible, do nothing.
     */
    public void makeVisible()
    {
        isVisible = true;
        draw();
    }
```

```
/**
 * Make this triangle invisible. If it was already invisible, do nothing.
 */
public void makeInvisible()
{
   erase();
   isVisible = false;
}

/**
 * Move the triangle a few pixels to the right.
 */
public void moveRight()
{
   moveHorizontal(20);
}

/**
 * Move the triangle a few pixels to the left.
 */
public void moveLeft()
{
   moveHorizontal(-20);
}

/**
 * Move the triangle a few pixels up.
 */
public void moveUp()
{
   moveVertical(-20);
}

/**
 * Move the triangle a few pixels down.
 */
public void moveDown()
{
   moveVertical(20);
}

/**
 * Move the triangle horizontally by 'distance' pixels.
 */
public void moveHorizontal(int distance)
{
   erase();
   xPosition += distance;
   draw();
}
```

```
/**
 * Move the triangle vertically by 'distance' pixels.
 */
public void moveVertical(int distance)
{
   erase();
   yPosition += distance;
   draw();
}

/**
 * Slowly move the triangle horizontally by 'distance' pixels.
 */
public void slowMoveHorizontal(int distance)
{
   int delta;

   if(distance < 0)
   {
      delta = -1;
      distance = -distance;
   }
   else
   {
      delta = 1;
   }

   for(int i = 0; i < distance; i++)
   {
      xPosition += delta;
      draw();
   }
}

/**
 * Slowly move the triangle vertically by 'distance' pixels.
 */
public void slowMoveVertical(int distance)
{
   int delta;

   if(distance < 0)
   {
      delta = -1;
      distance = -distance;
   }
   else
   {
      delta = 1;
   }
```

```
    for(int i = 0; i < distance; i++)
    {
        yPosition += delta;
        draw();
    }
}

/**
 * Change the size to the new size (in pixels). Size must be >= 0.
 */
public void changeSize(int newHeight, int newWidth)
{
    erase();
    height = newHeight;
    width = newWidth;
    draw();
}

/**
 * Change the color. Valid colors are "red", "yellow", "blue", "green",
 * "magenta" and "black".
 */
public void changeColor(String newColor)
{
    color = newColor;
    draw();
}

/**
 * Draw the triangle with current specifications on screen.
 */
private void draw()
{
    if(isVisible) {
        Canvas canvas = Canvas.getCanvas();
        int[] xpoints = { xPosition, xPosition + (width/2), xPosition - (width/2) };
        int[] ypoints = { yPosition, yPosition + height, yPosition + height };
        canvas.draw(this, color, new Polygon(xpoints, ypoints, 3));
        canvas.wait(10);
    }
}

/**
 * Erase the triangle on screen.
 */
private void erase()
{
    if(isVisible) {
        Canvas canvas = Canvas.getCanvas();
        canvas.erase(this);
    }
```

```
    }
}
```

Square

```
import java.awt.*;

/**
 * A square that can be manipulated and that draws itself on a canvas.
 *
 * @author  Michael Kolling and David J. Barnes
 * @version 2008.03.30
 */

public class Square
{
    private int size;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new square at default position with default color.
     */
    public Square()
    {
        size = 30;
        xPosition = 60;
        yPosition = 50;
        color = "red";
        isVisible = false;
    }

    /**
     * Make this square visible. If it was already visible, do nothing.
     */
    public void makeVisible()
    {
        isVisible = true;
        draw();
    }

    /**
     * Make this square invisible. If it was already invisible, do nothing.
     */
    public void makeInvisible()
    {
        erase();
        isVisible = false;
```

```
}

/**
 * Move the square a few pixels to the right.
 */
public void moveRight()
{
   moveHorizontal(20);
}

/**
 * Move the square a few pixels to the left.
 */
public void moveLeft()
{
   moveHorizontal(-20);
}

/**
 * Move the square a few pixels up.
 */
public void moveUp()
{
   moveVertical(-20);
}

/**
 * Move the square a few pixels down.
 */
public void moveDown()
{
   moveVertical(20);
}

/**
 * Move the square horizontally by 'distance' pixels.
 */
public void moveHorizontal(int distance)
{
   erase();
   xPosition += distance;
   draw();
}

/**
 * Move the square vertically by 'distance' pixels.
 */
public void moveVertical(int distance)
{
   erase();
   yPosition += distance;
```

```
    draw();
}

/**
 * Slowly move the square horizontally by 'distance' pixels.
 */
public void slowMoveHorizontal(int distance)
{
    int delta;

    if(distance < 0)
    {
        delta = -1;
        distance = -distance;
    }
    else
    {
        delta = 1;
    }

    for(int i = 0; i < distance; i++)
    {
        xPosition += delta;
        draw();
    }
}

/**
 * Slowly move the square vertically by 'distance' pixels.
 */
public void slowMoveVertical(int distance)
{
    int delta;

    if(distance < 0)
    {
        delta = -1;
        distance = -distance;
    }
    else
    {
        delta = 1;
    }

    for(int i = 0; i < distance; i++)
    {
        yPosition += delta;
        draw();
    }
}
```

```
    /**
     * Change the size to the new size (in pixels). Size must be >= 0.
     */
    public void changeSize(int newSize)
    {
        erase();
        size = newSize;
        draw();
    }

    /**
     * Change the color. Valid colors are "red", "yellow", "blue", "green",
     * "magenta" and "black".
     */
    public void changeColor(String newColor)
    {
        color = newColor;
        draw();
    }

    /**
     * Draw the square with current specifications on screen.
     */
    private void draw()
    {
        if(isVisible) {
            Canvas canvas = Canvas.getCanvas();
            canvas.draw(this, color,
                    new Rectangle(xPosition, yPosition, size, size));
            canvas.wait(10);
        }
    }

    /**
     * Erase the square on screen.
     */
    private void erase()
    {
        if(isVisible) {
            Canvas canvas = Canvas.getCanvas();
            canvas.erase(this);
        }
    }
}
```

Circle

```
import java.awt.*;
import java.awt.geom.*;

/**
 * A circle that can be manipulated and that draws itself on a canvas.
 *
 * @author  Michael Kolling and David J. Barnes
 * @version 2008.03.30
 */

public class Circle
{
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new circle at default position with default color.
     */
    public Circle()
    {
        diameter = 30;
        xPosition = 20;
        yPosition = 60;
        color = "blue";
        isVisible = false;
    }

    /**
     * Make this circle visible. If it was already visible, do nothing.
     */
    public void makeVisible()
    {
        isVisible = true;
        draw();
    }

    /**
     * Make this circle invisible. If it was already invisible, do nothing.
     */
    public void makeInvisible()
    {
        erase();
        isVisible = false;
```

```
}

/**
 * Move the circle a few pixels to the right.
 */
public void moveRight()
{
   moveHorizontal(20);
}

/**
 * Move the circle a few pixels to the left.
 */
public void moveLeft()
{
   moveHorizontal(-20);
}

/**
 * Move the circle a few pixels up.
 */
public void moveUp()
{
   moveVertical(-20);
}

/**
 * Move the circle a few pixels down.
 */
public void moveDown()
{
   moveVertical(20);
}

/**
 * Move the circle horizontally by 'distance' pixels.
 */
public void moveHorizontal(int distance)
{
   erase();
   xPosition += distance;
   draw();
}

/**
 * Move the circle vertically by 'distance' pixels.
 */
public void moveVertical(int distance)
{
   erase();
   yPosition += distance;
```

```
    draw();
}

/**
 * Slowly move the circle horizontally by 'distance' pixels.
 */
public void slowMoveHorizontal(int distance)
{
    int delta;

    if(distance < 0)
    {
        delta = -1;
        distance = -distance;
    }
    else
    {
        delta = 1;
    }

    for(int i = 0; i < distance; i++)
    {
        xPosition += delta;
        draw();
    }
}

/**
 * Slowly move the circle vertically by 'distance' pixels.
 */
public void slowMoveVertical(int distance)
{
    int delta;

    if(distance < 0)
    {
        delta = -1;
        distance = -distance;
    }
    else
    {
        delta = 1;
    }

    for(int i = 0; i < distance; i++)
    {
        yPosition += delta;
        draw();
    }
}
```

```
    /**
     * Change the size to the new size (in pixels). Size must be >= 0.
     */
    public void changeSize(int newDiameter)
    {
        erase();
        diameter = newDiameter;
        draw();
    }

    /**
     * Change the color. Valid colors are "red", "yellow", "blue", "green",
     * "magenta" and "black".
     */
    public void changeColor(String newColor)
    {
        color = newColor;
        draw();
    }

    /**
     * Draw the circle with current specifications on screen.
     */
    private void draw()
    {
        if(isVisible) {
            Canvas canvas = Canvas.getCanvas();
            canvas.draw(this, color, new Ellipse2D.Double(xPosition, yPosition,
                                        diameter, diameter));
            canvas.wait(10);
        }
    }

    /**
     * Erase the circle on screen.
     */
    private void erase()
    {
        if(isVisible) {
            Canvas canvas = Canvas.getCanvas();
            canvas.erase(this);
        }
    }
}
```

Canvas

```
import javax.swing.*;
import java.awt.*;
import java.util.List;
import java.util.*;

/**
 * Canvas is a class to allow for simple graphical drawing on a canvas.
 * This is a modification of the general purpose Canvas, specially made for
 * the BlueJ "shapes" example.
 *
 * @author: Bruce Quig
 * @author: Michael Kolling (mik)
 *
 * @version 2008.03.30
 */
public class Canvas
{
    // Note: The implementation of this class (specifically the handling of
    // shape identity and colors) is slightly more complex than necessary. This
    // is done on purpose to keep the interface and instance fields of the
    // shape objects in this project clean and simple for educational purposes.

    private static Canvas canvasSingleton;

    /**
     * Factory method to get the canvas singleton object.
     */
    public static Canvas getCanvas()
    {
        if(canvasSingleton == null) {
            canvasSingleton = new Canvas("BlueJ Shapes Demo", 300, 300,
                                         Color.white);
        }
        canvasSingleton.setVisible(true);
        return canvasSingleton;
    }

    //  ----- instance part -----

    private JFrame frame;
    private CanvasPane canvas;
    private Graphics2D graphic;
    private Color backgroundColor;
    private Image canvasImage;
    private List<Object> objects;
    private HashMap<Object, ShapeDescription> shapes;
```

```
/**
 * Create a Canvas.
 * @param title    title to appear in Canvas Frame
 * @param width    the desired width for the canvas
 * @param height   the desired height for the canvas
 * @param bgColor the desired background color of the canvas
 */
private Canvas(String title, int width, int height, Color bgColor)
{
   frame = new JFrame();
   canvas = new CanvasPane();
   frame.setContentPane(canvas);
   frame.setTitle(title);
   canvas.setPreferredSize(new Dimension(width, height));
   backgroundColor = bgColor;
   frame.pack();
   objects = new ArrayList<Object>();
   shapes = new HashMap<Object, ShapeDescription>();
}

/**
 * Set the canvas visibility and brings canvas to the front of screen
 * when made visible. This method can also be used to bring an already
 * visible canvas to the front of other windows.
 * @param visible  boolean value representing the desired visibility of
 * the canvas (true or false)
 */
public void setVisible(boolean visible)
{
   if(graphic == null) {
      // first time: instantiate the offscreen image and fill it with
      // the background color
      Dimension size = canvas.getSize();
      canvasImage = canvas.createImage(size.width, size.height);
      graphic = (Graphics2D)canvasImage.getGraphics();
      graphic.setColor(backgroundColor);
      graphic.fillRect(0, 0, size.width, size.height);
      graphic.setColor(Color.black);
   }
   frame.setVisible(visible);
}

/**
 * Draw a given shape onto the canvas.
 * @param  referenceObject  an object to define identity for this shape
 * @param  color           the color of the shape
 * @param  shape           the shape object to be drawn on the canvas
 */
// Note: this is a slightly backwards way of maintaining the shape
// objects. It is carefully designed to keep the visible shape interfaces
```

```
 // in this project clean and simple for educational purposes.
public void draw(Object referenceObject, String color, Shape shape)
{
   objects.remove(referenceObject);   // just in case it was already there
   objects.add(referenceObject);      // add at the end
   shapes.put(referenceObject, new ShapeDescription(shape, color));
   redraw();
}

/**
 * Erase a given shape's from the screen.
 * @param  referenceObject  the shape object to be erased
 */
public void erase(Object referenceObject)
{
   objects.remove(referenceObject);   // just in case it was already there
   shapes.remove(referenceObject);
   redraw();
}

/**
 * Set the foreground color of the Canvas.
 * @param  newColor   the new color for the foreground of the Canvas
 */
public void setForegroundColor(String colorString)
{
   if(colorString.equals("red")) {
      graphic.setColor(Color.red);
   }
   else if(colorString.equals("black")) {
      graphic.setColor(Color.black);
   }
   else if(colorString.equals("blue")) {
      graphic.setColor(Color.blue);
   }
   else if(colorString.equals("yellow")) {
      graphic.setColor(Color.yellow);
   }
   else if(colorString.equals("green")) {
      graphic.setColor(Color.green);
   }
   else if(colorString.equals("magenta")) {
      graphic.setColor(Color.magenta);
   }
   else if(colorString.equals("white")) {
      graphic.setColor(Color.white);
   }
   else {
      graphic.setColor(Color.black);
   }
}
```

```
/**
 * Wait for a specified number of milliseconds before finishing.
 * This provides an easy way to specify a small delay which can be
 * used when producing animations.
 * @param  milliseconds  the number
 */
public void wait(int milliseconds)
{
   try
   {
      Thread.sleep(milliseconds);
   }
   catch (Exception e)
   {
      // ignoring exception at the moment
   }
}

/**
 * Redraw ell shapes currently on the Canvas.
 */
private void redraw()
{
   erase();
   for(Object shape : objects) {
      shapes.get(shape).draw(graphic);
   }
   canvas.repaint();
}

/**
 * Erase the whole canvas. (Does not repaint.)
 */
private void erase()
{
   Color original = graphic.getColor();
   graphic.setColor(backgroundColor);
   Dimension size = canvas.getSize();
   graphic.fill(new Rectangle(0, 0, size.width, size.height));
   graphic.setColor(original);
}


/************************************************************************
 * Inner class CanvasPane - the actual canvas component contained in the
 * Canvas frame. This is essentially a JPanel with added capability to
 * refresh the image drawn on it.
 */
private class CanvasPane extends JPanel
{
```

```
    public void paint(Graphics g)
    {
      g.drawImage(canvasImage, 0, 0, null);
    }
  }

  /**********************************************************************
   * Inner class CanvasPane - the actual canvas component contained in the
   * Canvas frame. This is essentially a JPanel with added capability to
   * refresh the image drawn on it.
   */
  private class ShapeDescription
  {
    private Shape shape;
    private String colorString;

    public ShapeDescription(Shape shape, String color)
    {
      this.shape = shape;
      colorString = color;
    }

    public void draw(Graphics2D graphic)
    {
      setForegroundColor(colorString);
      graphic.fill(shape);
    }
  }

}
```

EX2.58 – TICKET MACHINE

Summary

Project: Ticket Machine
Author: Charlotte Pierce

This project is part of the material for the Object-Oriented Programming portfolio of Charlotte Pierce.

This project is a simple ticket machine. The ticket machine offers the choice of three possible ticket types ("short", "half-day" and "daily"), each of which has a different cost associated (the value of 'price' is altered appropriately when a ticket type is selected). The machine allows the user to insert an amount of money in order to pay for their ticket. If the amount input is not enough, it displays a message and can refund the user's money.

To use this application:
- create an instance of TicketMachine
- use the selectTicket method, and enter either 'short', 'half-day' of 'daily'
- insert some money (cents)
- attempt to print a ticket using printTicket()

This project demonstrates the use of standard conditional statements in the java language. It is fully documented using the javadoc style.

Extension Summary

There would need to be a method through which the user could choose their ticket, such as *selectTicket*. This method could take a parameter through which the user can choose a type of ticket, and then have the value of the chosen ticket assigned to *price*. The only method which would need to be altered, is printTicket, to ensure that a ticket type has been selected. The other methods would not need to be changed, as they would use the new value of price to calculate the amount left to pay, and other relevant values, as normal. However, the constructor would need to be altered so that a ticket price is not input at the creation of the TicketMachine object.

Code

TicketMachine

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * Instances will check to ensure that a user only enters
 * sensible amounts of money, and will only print a ticket
 * if enough money has been input.
 * Class was extended for exercise 2.58, Object-Oriented Programming
 * lab 2.
 *
 * @author Charlotte Pierce
 * @version 13/8/2010
 */
public class TicketMachine
{
   // The price of a ticket from this machine.
   private int price;
   // The amount of money entered by a customer so far.
   private int balance;
   // The total amount of money collected by this machine.
   private int total;

   /**
    * Create a machine that issues tickets.
    */
   public TicketMachine()
   {
      price = 0;
      balance = 0;
      total = 0;
   }

   /**
    * @Return The price of a ticket.
    */
   public int getPrice()
   {
      return price;
   }

   /**
    * Return The amount of money already inserted for the
    * next ticket.
    */
   public int getBalance()
   {
      return balance;
   }
```

```
/**
 * Receive an amount of money in cents from a customer.
 * Check that the amount is sensible.
 * @param amount The amount of money to insert.
 */
public void insertMoney(int amount)
{
   if(amount > 0) {
      balance = balance + amount;
   }
   else {
      System.out.println("Use a positive amount: " +
                         amount);
   }
}

/**
 * Print a ticket if enough money has been inserted, and
 * reduce the current balance by the ticket price. Print
 * an error message if more money is required.
 */
public void printTicket()
{
   if(price != 0){
      if(balance >= price) {
         // Simulate the printing of a ticket.
         System.out.println("####################");
         System.out.println("# The BlueJ Line");
         System.out.println("# Ticket");
         System.out.println("# " + price + " cents.");
         System.out.println("####################");
         System.out.println();

         // Update the total collected with the price.
         total = total + price;
         // Reduce the balance by the prince.
         balance = balance - price;
      }
      else {
         System.out.println("You must insert at least: " +
                            (price - balance) + " more cents.");

      }
   }
   else{
      System.out.println("You must select a ticket first.");
   }
}

/**
```

```
     * Return the money in the balance.
     * The balance is cleared.
     */
    public int refundBalance()
    {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }

    /**
     * Select one of the three ticket types.
     * Accepts "short", "half-day" and "daily" ticket types
     * @param ticketType The type of ticket to select.
     */
    public void selectTicket(String ticketType)
    {
        if(ticketType == "short"){
            price = 100;
        }
        else if(ticketType == "half-day"){
            price = 200;
        }
        else if(ticketType == "daily"){
            price = 300;
        }
        else{
            System.out.println("Unsupported type entered, ticket price 350");
            price = 350;
        }
    }
}
```

Ex2.84 – Heater

Summary

Project: Heater
Author: Charlotte Pierce

This project is a simple heater. The heater can be created with a minimum and maximum value of heat which it can hold. It can also become warmer or colder by a certain amount, defined by the user (unless the calculation would send the value of the heater to be outside the limits also defined by the user).

To use this application:
- create an instance of Heater, selecting two numbers for the minimum and maximum values
- set a value by which to increment the temperature of the heater using setIncrement (can be left with the default of 5 if desired)
- use the warmer() and cooler() methods
- use the getTemp() method to see how the temperature changes

This project demonstrates the use of conditional statements to keep a value within certain boundaries. It is fully documented using javadoc style.

Code

Heater

```
/**
 * A heater. Holds a certain temperature, which must be within
 * a range of temperatures defined by the minimum and maximum
 * fields.
 * Heater can be cooled/warmed by the amount held in increment.
 *
 * @author Charlotte Pierce
 * @version 14/8/2010
 */

public class Heater
{
   private int temperature;
   private int min; //minimum temperature
   private int max; //maximum temperature
   private int increment; //amount to change temperature by

   /**
    * Instantiates the heater with minimum and maximum temperatures.
    * @param minSet The minimum temperature for the heater.
    * @param maxSet The maximum temperature for the heater.
    */
   public Heater(int minSet, int maxSet)
   {
      temperature = 15;
      min = minSet;
      max = maxSet;
      increment = 5;
   }

   /**
    * Increases the temperature of the heater by the value in
    * increment, unless that modification would take the
    * temperature above the maximum value allowed.
    */
   public void warmer()
   {
      if((temperature + increment) > max){
         //do nothing...
      }
      else
      {
         temperature = temperature + increment;
      }
   }

   /**
    * Decreases the temperature of the heater by the value in
```

```
     * increment, unless that modification would take the
     * temperature below the minimum value allowed.
     */
    public void cooler()
    {
        if((temperature - increment) < min){
            //do nothing...
        }
        else{
            temperature = temperature - increment;
        }
    }

    /**
     * Used to change the value by which the temperature of the
     * heater is incremented.
     * @param inc The value to change increment to.
     */
    public void setIncrement(int inc)
    {
        if(inc < 0){
            System.out.println("Invalid value to increment.");
        }
        else{
            increment = inc;
        }
    }

    /**
     * Returns the temperature of the heater.
     * @return The temperature of the heater.
     */
    public int getTemp()
    {
        return temperature;
    }
}
```

Ex4.24 – 4.35 – Club Demo

Summary

Project: Club Demo
Author: Charlotte Pierce

This project is material for the Object-Oriented Programming portfolio of Charlotte Pierce.

This application simulates a club. The club can have a number of memberships. The club can return the number of members in the club, the number that joined in any given month, as well as adding members, and removing members (members can be removed based on the month and year of their joining the club).

To use this application:
- create as many instances of Club as desired
- create as many instances of Membership as desired
- add each Membership to a club as desired
- use each method within Club and observe it's functionality
Alternatively, peruse the code of ClubDemo, which creates two memberships and adds them to a club as a demonstration of the Club and Membership classes.

This application demonstrates the use of collections within a project. It implements all associated challenge exercises. It is fully documented using javadoc style.

Code

ClubDemo

```
/**
 * Provide a demonstration of the Club and Membership
 * classes.
 *
 * @author Charlotte Pierce
 * @version 29/8/2010
 */
public class ClubDemo
{
   // instance variables - replace the example below with your own
   private Club club;

   /**
    * Constructor for objects of class ClubDemo
    */
   public ClubDemo()
   {
      club = new Club();
   }

   /**
    * Add some members to the club, and then
    * show how many there are.
    * Further example calls could be added if more functionality
    * is added to the Club class.
    */
   public void demo()
   {
      club.join(new Membership("David", 2, 2004));
      club.join(new Membership("Michael", 1, 2004));
      System.out.println("The club has " +
                         club.numberOfMembers() +
                         " members.");
   }
}
```

Club

```
/**
 * Defines a club. Every club can have a number of members.
 *
 * @author Charlotte Pierce
 * @version 29/8/2010
 */

import java.util.ArrayList;
```

```
import java.util.Iterator;

public class Club
{
   ArrayList<Membership> members;

   public Club()
   {
      members = new ArrayList<Membership>();
   }

  //Add a new member to the club's list of members.
   public void join(Membership member)
   {
      members.add(member);
   }

   //The number of members (Membership objects) in the club.
   public int numberOfMembers()
   {
      return members.size();
   }

   /*
    *Determine the number of members who joined in the
    *given month.
    *@param month The month we are interested in.
    *@return The number of members joinng in month
   */
   public int joinedInMonth(int month)
   {
      int count = 0;
      Iterator<Membership> it = members.iterator();
      if(month < 1 || month > 12){
         System.out.println("Invalid month");
         return 0;
      }
      else{
         while(it.hasNext()){
            Membership mem = it.next();
            if(mem.getMonth() == month){
               count += 1;
            }
         }
         return count;
      }
   }

   /*
    * Remove from the club's collection all members who
    * joined in the given month, and return them stored
```

```
     * in a  separate collection object.
     * @param month The month of the Membership.
     * @param year The year of the Membership.
     * @return The members who joined in the given month and year.
     */
    public ArrayList<Membership> purge(int month, int year)
    {
        ArrayList<Membership> purged = new ArrayList<Membership>();
        Iterator<Membership> it = members.iterator();
        if(month < 1 || month > 12){
            System.out.println("Invalid month entered!");
            return null;
        }
        else{
            while(it.hasNext()){
                Membership mem = it.next();
                if(mem.getMonth() == month && mem.getYear() == year){
                    purged.add(mem);
                    it.remove();
                }
            }
            return purged;
        }
    }
}
```

Membership

```
/**
 * Store details of a club membership.
 *
* @author Charlotte Pierce
 * @version 29/8/2010
 */
public class Membership
{
    // The name of the member.
    private String name;
    // The month in which the membership was taken out.
    private int month;
    // The year in which the membership was taken out.
    private int year;

    /**
     * Constructor for objects of class Membership.
     * @param name The name of the member.
     * @param month The month in which they joined. (1 ... 12)
     * @param year The year in which they joined.
     */
    public Membership(String name, int month, int year)
```

```
      throws IllegalArgumentException
   {
      if(month < 1 || month > 12) {
         throw new IllegalArgumentException(
            "Month " + month + " out of range. Must be in the range 1 ... 12");
      }
      this.name = name;
      this.month = month;
      this.year = year;
   }

   /**
    * @return The member's name.
    */
   public String getName()
   {
      return name;
   }

   /**
    * @return The month in which the member joined.
    *         A value in the range 1 ... 12
    */
   public int getMonth()
   {
      return month;
   }

   /**
    * @return The year in which the member joined.
    */
   public int getYear()
   {
      return year;
   }

   /**
    * @return A string representation of this membership.
    */
   public String toString()
   {
      return "Name: " + name +
           " joined in month " +
           month + " of " + year;
   }
}
```

EX4.36 – 4.40 – STOCK DEMO

Summary

Project: Stock Demo
Author: Charlotte Pierce

This project is material for the Object-Oriented Programming portfolio of
Charlotte Pierce.

This application simulates a stock room. The stock room can have a number of products.
The stock room can add products, receive a delivery of that product, find a given
product within the stock room and return the quantity held of a particular product.

To use this application:
- create an instance of StockManager
- create as many instances of Product as desired
- add the products to the StockManager
- use the methods in StockManager and observe the functionality
Alternatively, create an instance of the StockDemo class, which
demonstrates some of this functionality.

This application demonstrates the use of collections within a project. It also demonstrates
the use of an iterator. It implements all associated challenge exercises.
It is fully documented using javadoc style.

Code

StockDemo

```
/**
 * Demonstrate the StockManager and Product classes.
 * The demonstration becomes properly functional as
 * the StockManager class is completed.
 *
 * @author Charlotte Pierce
 * @version 2008.03.30
 */
public class StockDemo
{
   // The stock manager.
   private StockManager manager;

   /**
    * Create a StockManager and populate it with a few
    * sample products.
    */
   public StockDemo()
   {
      manager = new StockManager();
      manager.addProduct(new Product(132, "Clock Radio"));
      manager.addProduct(new Product(37,  "Mobile Phone"));
      manager.addProduct(new Product(23,  "Microwave Oven"));
   }

   /**
    * Provide a very simple demonstration of how a StockManager
    * might be used. Details of one product are shown, the
    * product is restocked, and then the details are shown again.
    */
   public void demo()
   {
      // Show details of all of the products.
      manager.printProductDetails();
      // Take delivery of 5 items of one of the products.
      manager.delivery(132, 5);
      manager.printProductDetails();
   }

   /**
    * Show details of the given product. If found,
    * its name and stock quantity will be shown.
    * @param id The ID of the product to look for.
    */
   public void showDetails(int id)
   {
      Product product = getProduct(id);
      if(product != null) {
```

```
            System.out.println(product.toString());
        }
    }

    /**
     * Sell one of the given item.
     * Show the before and after status of the product.
     * @param id The ID of the product being sold.
     */
    public void sellProduct(int id)
    {
        Product product = getProduct(id);

        if(product != null) {
            showDetails(id);
            product.sellOne();
            showDetails(id);
        }
    }

    /**
     * Get the product with the given id from the manager.
     * An error message is printed if there is no match.
     * @param id The ID of the product.
     * @return The Product, or null if no matching one is found.
     */
    public Product getProduct(int id)
    {
        Product product = manager.findProduct(id);
        if(product == null) {
            System.out.println("Product with ID: " + id +
                               " is not recognised.");
        }
        return product;
    }

    /**
     * @return The stock manager.
     */
    public StockManager getManager()
    {
        return manager;
    }
}
```

StockManager

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Manage the stock in a business.
 * The stock is described by zero or more Products.
 *
 * @author Charlotte Pierce
 * @version 30/8/2010
 */
public class StockManager
{
   // A list of the products.
   private ArrayList<Product> stock;

   /**
    * Initialise the stock manager.
    */
   public StockManager()
   {
      stock = new ArrayList<Product>();
   }

   /**
    * Add a product to the list. Does not add if ID is already
    * in use.
    * @param item The item to be added.
    */
   public void addProduct(Product item)
   {
      if(findProduct(item.getID()) == null){
         stock.add(item);
      }
      else{
         System.out.println("Product ID already exists!");
      }
   }

   /**
    * Receive a delivery of a particular product.
    * Increase the quantity of the product by the given amount.
    * @param id The ID of the product.
    * @param amount The amount to increase the quantity by.
    */
   public void delivery(int id, int amount)
   {
```

```
    Product item = findProduct(id);
    if(item == null){
        System.out.println("No such product!");
    }
    else{
        item.increaseQuantity(amount);
    }
}

/**
 * Try to find a product in the stock with the given id.
 * @return The identified product, or null if there is none
 *         with a matching ID.
 */
public Product findProduct(int id)
{
    Iterator<Product> it = stock.iterator();
    while(it.hasNext()){
        Product p = it.next();
        if(p.getID() == id){
            return p;
        }
    }
    return null;
}

/**
 * Locate a product with the given ID, and return how
 * many of this item are in stock. If the ID does not
 * match any product, return zero.
 * @param id The ID of the product.
 * @return The quantity of the given product in stock.
 */
public int numberInStock(int id)
{
    Product item = findProduct(id);
    if(item == null){
        System.out.println("No such product!");
        return 0;
    }
    else{
        return item.getQuantity();
    }
}

/**
 * Print details of all the products.
 */
public void printProductDetails()
{
    Iterator<Product> it = stock.iterator();
```

```
        while(it.hasNext()){
            Product p = it.next();
            System.out.println(p.toString());
        }
    }

    /**
     * Finds a returns a specified product.
     * @param name The name of the product to find.
     * @return The product.
     */
    public Product findProduct(String name)
    {
        Iterator<Product> it = stock.iterator();
        while(it.hasNext()){
            Product p = it.next();
            if(p.getName().equals(name)){
                return p;
            }
        }
        return null;
    }
}
```

Product

```
/**
 * Model some details of a product sold by a company.
 *
 * @author Charlotte Pierce
 * @version 29/8/2010
 */
public class Product
{
    // An identifying number for this product.
    private int id;
    // The name of this product.
    private String name;
    // The quantity of this product in stock.
    private int quantity;

    /**
     * Constructor for objects of class Product.
     * The initial stock quantity is zero.
     * @param id The product's identifying number.
     * @param name The product's name.
     */
    public Product(int id, String name)
    {
        this.id = id;
```

```
    this.name = name;
    quantity = 0;
}

/**
 * @return The product's id.
 */
public int getID()
{
    return id;
}

/**
 * @return The product's name.
 */
public String getName()
{
    return name;
}

/**
 * @return The quantity in stock.
 */
public int getQuantity()
{
    return quantity;
}

/**
 * @return The id, name and quantity in stock.
 */
public String toString()
{
    return id + ": " +
           name +
           " stock level: " + quantity;
}

/**
 * Restock with the given amount of this product.
 * The current quantity is incremented by the given amount.
 * @param amount The number of new items added to the stock.
 *               This must be greater than zero.
 */
public void increaseQuantity(int amount)
{
    if(amount > 0) {
        quantity += amount;
    }
    else {
        System.out.println("Attempt to restock " +
```

```
                    name +
                    " with a non-positive amount: " +
                    amount);
        }
    }

    /**
     * Sell one of these products.
     * An error is reported if there appears to be no stock.
     */
    public void sellOne()
    {
        if(quantity > 0) {
            quantity--;
        }
        else {
            System.out.println(
                "Attempt to sell an out of stock item: " + name);
        }
    }
}
```

EX8.16 LAB CLASS

Summary

Project: Lab Classes
Author: Charlotte Pierce

This project simluates a lab class. Each lab class can have one instructor, and any number of students. The number of students allowed in a class is specified by the user.

To use this application:

- create an Instructor
- create a LabClass (specifying the maximum number of students and the instructor for the class)
- create as many Student objects as desired
- add students to the LabClass
- use the methods in LabClass to set the room and time for the class, and return the number of students enrolled in that class
- use the method printList() in LabClass to print all the details of that class

This project demonstrates the use of collections. It also demonstrates the use of an abstract class (Person), which Instructor and Student classes extend. This stops the class Person from being instantiated directly, allowing only its subclasses to be created. This was done in order to allow both the Instructor and Student classes, which share similar code, to be able to access their shared code from one common class, decreasing the need for code duplication, and making it easier for extensions or changes to the project. The project is fully documented using javadoc style.

Code

LabClass

```
import java.util.*;

/**
 * The LabClass class represents an enrolment list for one lab class. It stores
 * the time, room and participants of the lab, as well as the instructor's name.
 *
 * @author Charlotte Pierce
 * @version 3/11/2010
 */
public class LabClass
{
   private String instructor;
   private String room;
   private String timeAndDay;
   private List<Student> students;
   private Instructor teacher; // can only have one instructor per lab class
   private int capacity;

   /**
    * Create a LabClass with a maximum number of enrolments. All other details
    * are set to default values.
    * @param maxNumberOfStudents The maximum number of students.
    * @param teacher The instructor for the class.
    */
   public LabClass(int maxNumberOfStudents, Instructor teacher)
   {
      instructor = "unknown";
      room = "unknown";
      timeAndDay = "unknown";
      students = new ArrayList<Student>();
      this.teacher = teacher;
      capacity = maxNumberOfStudents;
   }

   /**
    * Add a student to this LabClass.
    * @param newStudent The student to add.
    */
   public void enrollStudent(Student newStudent)
   {
      if(students.size() == capacity) {
         System.out.println("The class is full, you cannot enrol.");
      }
      else {
```

```
        students.add(newStudent);
    }
}

/**
 * Return the number of students currently enrolled in this LabClass.
 * @return The number of students in the class.
 */
public int numberOfStudents()
{
    return students.size();
}

/**
 * Set the room number for this LabClass.
 * @param roomNumber The room number for the class.
 */
public void setRoom(String roomNumber)
{
    room = roomNumber;
}

/**
 * Set the time for this LabClass. The parameter should define the day
 * and the time of day, such as "Friday, 10am".
 * @param timeAndDayString The time for the lab class.
 */
public void setTime(String timeAndDayString)
{
    timeAndDay = timeAndDayString;
}

/**
 * Set the name of the instructor for this LabClass.
 * @param instructorName The name of the instructor for the class.
 */
public void setInstructor(String instructorName)
{
    instructor = instructorName;
}

/**
 * Print out a class list with other LabClass details to the standard
 * terminal.
 */
public void printList()
{
    System.out.println("Lab class " + timeAndDay);
    System.out.println("Instructor: " + instructor + "   room: " + room);
    System.out.println("Class list:");
    for(Student student : students) {
```

```
            student.print();
        }
        System.out.println("Number of students: " + numberOfStudents());
    }
}
```

Instructor

```
/**
 * An instructor who takes a lab class. Extends from Person.
 *
 * @author Charlotte Pierce
 * @version 3/11/2010
 */
public class Instructor extends Person
{
    private int salary;

    /**
     * Creates an instructor.
     * @param name The name of the instructor.
     * @param address The address of the instructor.
     * @param phone The phone of the instructor.
     * @param salary The salary of the instructor.
     * @param id The id of the instructor.
     */
    public Instructor(String name, String address, String phone, int salary, String id)
    {
        super(name, address, phone, id);
        this.salary = salary;
    }
}
```

Student

```
/**
 * The Student class represents a student in a student administration system.
 * It holds the student details relevant in our context.
 * Students inherit the fields 'name', 'address' and 'phone' from class Person.
 *
 * @author Charlotte Pierce
 * @version 3/11/2010
 */
public class Student extends Person
{
    // the amount of credits for study taken so far
    private int credits;
```

```
    /**
     * Create a new student.
     * @param fullName The name of the student.
     * @param address The address of the student.
     * @param phone The phone of the student.
     * @param studentID The id of the student.
     */
    public Student(String fullName, String address, String phone, String studentID)
    {
        super(fullName, address, phone, studentID);
        credits = 0;
    }

    /**
     * Return the student ID of this student.
     * @return The student's ID.
     */
    public String getStudentID()
    {
        return id;
    }

    /**
     * Add some credit points to the student's accumulated credits.
     * @param additionalPoints The number of credit points to add.
     */
    public void addCredits(int additionalPoints)
    {
        credits += additionalPoints;
    }

    /**
     * Return the number of credit points this student has accumulated.
     * @return The number of credit points the student has.
     */
    public int getCredits()
    {
        return credits;
    }
}
```

Person

```
/**
 * Abstract class Person - can be extended into an instructor or a student to be
 * included in a lab class.
 *
 * @author Charlotte Pierce
```

```
 * @version 3/11/2010
 */
public abstract class Person
{
   protected String name;
   protected String address;
   protected String phone;
   protected String id;

   /**
    * Create the person.
    * @param aName The name of the person.
    * @param anAddress The address of the person.
    * @param aPhone The phone number of the person.
    * @param id The ID of the person.
    */
   public Person(String aName, String anAddress, String aPhone, String id)
   {
      name = aName;
      address = anAddress;
      phone = aPhone;
      this.id = id;
   }

   /**
   * Return the name of this person.
   * @return The name of the person.
   */
   protected String getName()
   {
      return name;
   }

   /**
    * Set a new name for this person.
    * @param replacementName The new name for the person.
    */
   protected void changeName(String replacementName)
   {
      name = replacementName;
   }

   /**
    * Return the login name of this person.
    * @return The login name for the person.
    */
   public String getLoginName()
   {
      return name.substring(0,4) + id.substring(0,3);
   }
```

```
    /**
    * Print the person's name and ID number to the output terminal.
    */
    public void print()
    {
        System.out.println(name + " (" + id + ")");
    }
}
```

APPENDIX A – JAVA SYNTAX SUMMARY

General Definitions

Abstraction: the ability to ignore details and focus attention on higher levels of a problem.

Modularisation: the process of dividing something into parts, each of which can be built separately.

Overloading: when multiple constructors and/or methods in the same class have the same name. They are then defined by their parameter(s).

Access Modifiers: Used for fields, constructors and methods. 'public' is used for methods which are accessible by other classes; 'private' methods are only accessible within that class.

Information Hiding: the principle stating that internal details of a class's implementation should be hidden from other classes.

Coupling: the level to which changes in one class mean changes are required in another class. Loose coupling is desired. Coupling is highly affected by cohesion, responsibility-driven design and encapsulation.

Implicit Coupling: when one class depends on internal information from another, but the dependence is not immediately obvious (class will often still compile when changes are made, just return the wrong data).

Cohesion: how well a unit of code maps to a logical task. Good cohesion means that **one** unit (method or class) is responsible for **one** well-defined task. Cohesion can be achieved with Responsibility-Driven Design.
Cohesion helps readability, and the potential for reuse of code.

Responsibility-Driven Design: the process of designing classes by assigning each specific responsibilities. Each class should be responsible for handling its own data, and methods manipulating that data should be stored in that class.

Code Duplication: when the same segment of code is present more than once in an application; an indication of bad design (bad cohesion).

Encapsulation: a guideline suggesting that only information about **what** a class can do should be visible to the outside, not **how** is does that. If implemented, the implementation of a class, and how it stores its data, can be changed without affecting any other classes (known as *Localising Change*).

Refactoring: restructuring of existing classes and methods to adapt them to changed functionality and requirements.

Inheritence: the extension of one class into another.

Polymorphism: 'many types'; where one object can be used in situations where a different type to its

declared type is called for (e.g. polymorphic variables - when subclass can be used when superclass is called for).

Autoboxing: performed automatically when a primitive type is used in a context requiring a wrapper type.

Type Checking: uses an objects static type; when the compiler checks code to see if it is feasible (e.g. if the object has the method called).

Java Programs

Java programs are created through the primary use of classes, objects and methods.
Classes are used to define the type, or general cases for objects, and the actions that can be performed with those objects. From a class, one can instantiate an instance of that class, with specific attributes (such as dimensions and colour) that the programmer has decided upon; this is known as an 'object'. The object can then be interacted with using the methods defined in its class, changing its *state* (the set of all attributes defining that object – Java refers to attributes as *fields*).
Methods are are series of instructions, separated into blocks of code with curly brackets, each individual statement terminated with the use of a semi-colon.
The line of code
public static void main(String[] args)
denotes the start of the 'actual' program, that is, the steps that will be executed during runtime.

Classes

A class definition follows the general form
<access modifier(s)> class <identifier>
Each class can have fields, which define the state of each object created from it. Fields are set in the class' *constructor*, which is a 'special' method defined as such:
public <class name>(<parameter(s)>)
Within the constructor the default values of each of the fields should be set, using parameters if the user needs to define any of these values. Methods can then be written to use the fields in the class.

Class Variable

Indicated by the keyword static, it is a field stored in the class itself, therefore there is only ever one copy of the variable at any time, regardless of the number of instances of the class that exist.

Class Method

Can be invoked without an instance of the class. Denoted by the keyword *static*. The method can then be called with <class name>.<method>. Most commonly used as the main method. A class method cannot access any instance fields (because instance fields are associated with individual instances of the class); can only access class variables. A class method may also not call an instance method from the class (because it has no object to call it from), it can only call other class methods.

Constants

Indicated by the keyword final. They are like variables, but cannot change their value during the execution of the application. Must be declared with a value. Often class variables are constant. By

convention, constants are usually all caps.

Generic Classes
A class which requires a second parameter to define types of its fields and/or other variables. e.g. an ArrayList requires another parameter to define what type of objects the list will store.

Class Interface
Name of class, general purpose, constructors, methods, parameters and return types. Everything to use the class, but not know how it specifically works.

Inner Class
A class that is declared inside another class.

Class Implementation
Source code of a class.

Immutable Object
An object where the contents or state cannot be changed once it has been created (e.g. String).

Mutable Object
An object where the contents or state can be changed after its creation.
Note: multiple constructors can exist (known as 'overloading'). The constructor will then be chosen based on the parameter(s) input.
Note2: classes can be input from packages with the syntax import <class name>;
Note3: classes can be imported using import <qualifiedclassname>.

Abstract Class:
a class that is not intended for creating objects. Its purpose is to serve as a superclass for other classes. Can have a mixture of abstract and concrete methods. Defined with the keyword 'abstract'. A subclass of an abstract class must implement all of the abstract class' methods, or it too is an abstract class.
e.g. public abstract class MyClass

Concrete Class
A class which does have an implementation for every method.

Access Modifiers
Used to define what objects (if any) can access an element in a class. Access modifiers can be applied to an objects methods and fields.

Public
Anything can access it.

Private
Only the encapsulating class/object can access it, for fields use getter/setter methods.

Protected

Somewhere between public and private; allows access from within the encapsulating class, and any subclasses, but no other classes. Usually reserved for methods and constructors (applying protected access to fields rather than private ruins encapsulation).

Object Equality

Reference equality

When two variables reference the exact same object (i.e. the same memory location); does not take into account the content, but only the memory location. Test for reference equality with double equals, '=='.
The .equals() method inherited from the object class uses reference equality, unless overridden to do differently.

Content Equality

Checks whether two objects are the same internally; checks whether the two states of the objects match (i.e. all the fields are the same); does not take into account the objects memory location.
Note: if writing your own content equality .equals() method, first include a check to make sure the object passed as a parameter is of the right type (i.e. are they both students/vehicles?) before checking the fields – otherwise you'll likely be calling fields that don't even exist.
Note: if the .equals() method is overridden, you should also override the hashCode() method.

hashCode()

Needs to be overridden if equals() is, because two objects which are equal according to equals() must have the same return value from hashCode(). hashCode() returns an integer value that represents an object.

Object Types

A way of differentiating between the type of the variable, or the type of the object stored within; caters for issues raised by inheritance.

Static Type

The type of an object, as declared in the source code; will never change.

Dynamic Type

The type of object currently stored (e.g. takes into account changes to the object, and casting etc.; changes depending on dynamic behaviour of the program throughout runtime.
e.g. Vehicle c1 = new Car();
Static type is Vehicle, dynamic type at the time of creation is Car, but may change.
e.g. for(Vehicle v: garage)
Static type is vehicle, dynamic type could be any subtype of vehicle.
Note: static type is used for type checking.

Methods

A method signature is the combination of the methods name and its parameters. Access modifiers and return types are not officially recognised as part of the method signature, though some references do include them as being so. For example, the line of code
public void addNumbers(int X, int Y)

would have the method signature of addNumbers(int, int) under the official definition, with some references also including 'public void'.

Accessor Method
This is a method which returns a piece of information about the state of an object.

Mutator Method
A method which changes the state of its object.

Internal Method Call
A method call made by another method in the same class. <method name> (<parameter(s)>)

External Method Call
A method call made to a method in a different class. <object name>.<method name>(<parameter(s)>) - known as *dot notation*.

Method Polymorphism
The same method call may at different times execute different methods, depending on the dynamic type of the variable used to make that call. e.g. how a different method is called when inheritance is used, depending on the subtype called.

Abstract Method
A method consisting of only the method signature, with no implementation. Defined with the keyword abstract, and a semi-colon at the end of the definition.
e.g. abstract public void myMethod();

Concrete Method
A method which does have an implementation.

Overriding
If subtypes and supertypes have the same method, with the exact same signature, the subtype is described as having overridden the method in the supertype. The overriding method takes precedence when that method is called on the subtype object – this is because at runtime, methods from the dynamic type of an object are executed, not the static type.
You can, however, call an overridden method with the use of the keyword 'super'.
e.g. super.print()
Will call the print method from a supertype, even if that method was overridden in the calling subtype.
Note: if a method is called that doesn't exist in the subtype, the compiler searches the supertype, and continues up the class hierarchy until the method is found.

Parameters
Parameters are a way of 'sending' data to a method. Parameters are declared in the parentheses of the method, the same way regular variables are, with an identifier and data type. When the method is called, the parameter must be given a value, by including a form of data within the parentheses of the method call. Throughout the method, the data 'sent' to the method through the parameter(s) can be accessed through the chosen identifier.
e.g.
public void multiply(int X, int Y)
has two parameters, 'X' and 'Y', of type *integer*. When calling this method, X and Y can be given values in the following manner:

multiply(2, 3)
Note: Methods can have any number of parameters, each declaration separated by a comma.
Note2: Names of parameters can be referred to as *formal parameters* (which are limited in scope to their defining constructor or method), whereas the actual value referred to as *actual parameters*.

Functions

As well as taking input, methods can also return a value. This functionality must be declared in the method header, with the data type of the return value. For example,
string getName()
denotes a method which returns a value of the type 'string'. If a method does not return any value, the word 'void' must be included in its header to indicate so. For example, to return the value stored in a variable called 'name', the line
return name;
would be used.

Conditional Statements/Loops

If Statement

General form:
if(<condition>) {
<statement(s)>
}
else {
<statement(s)>
}

For Loop

Executes statements a set number of times.
for(<counter>; <condition>; <counter changes>){<instructions>}
The first part (counter) is often a variable declaration. (e.g. “int I = 0”).

For Each Loop

for(<element type> <element identifier>: <collection>{instructions}
For each element in collection, execute the commands in the curly brackets.

While Loop

while(<condition>){instructions}
Continue to execute the instructions, so long as the condition remains true (pre-condition loop).

Do-While Loop

Similar to the While loop, but the condition is not tested until the statements have run at least once.

```
Do {
        <statement(s)>;
} while (<condition>);
```

Switch (like a case select)

switch (<variable identifier>){
case <value>: <statement>; break;
case <value>: <statement>; break;
}
Note: without 'break' at the end of each line, the compiler 'falls over' into the next line and continues executing statements until it reaches a break, regardless of whether the variable meets the case requirement.
e.g.
switch (number){
case 1: System.out.println("The number was 1.");
case 2: System.out.println("The number was 2."); break;
}
Would output "The number was 1.The number was 2.", if the number was 1.

InstanceOf

obj instanceof Class
This checks if the 'obj' is an instance of the 'Class'.
Returns true if the dynamic type obj is Class, or any of Class' subtypes.

Data Types

Note: Headings written with the format <full name> (<name used when declaring>)
Data types define the way in which data can be manipulated.

Integer (int)

Whole numbers.

String (string)

Denotes a section of text. When assigning text to a string variable, the text must be enclosed in double quotes.
Note: when comparing two strings, always use:
<first string>.equals("second string");

Boolean (boolean)

A variable that can hold only two values – 'true' or 'false'.

Collections:

A group of objects. When defining, we must declare the type of collection and what type of elements it contains. The data type contained within a collection is defined between angular (<>) brackets.

Array

A fixed size collection. Can 'directly' store primitive types *and* objects.
Declared with:

<data type>[] <identifier>;
and then created with
<identifier> = new <data type>[<number of items>];
Values in an array can be accessed by:
<identifier>[<index>]
where indexes go from 0 – the number of items – 1.
Note: all arrays contain a field *length*.

ArrayList

A flexible sized collection. Can store only 'directly' store objects (auto-boxing allows it to store primitive types).
Must import java.util.ArrayList to get access to the class.
ArrayList<<data type>>();
A list of items of the defined data type, of any length. Use add() to add items, use size() to return the number of items in the list. Each item is referred to by its index (starts at 0; removing items is messy, as all the items after it move down an index), and can be returned by its index with <identifier>.get(index).
Note: known as a 'collection' object (objects which can store an arbitrary number of objects).
Note2: each ArrayList has an iterator (of type Iterator) which can be accessed with <identifier>.iterator();

Iterator

Must import java.util.Iterator to get access to the class.
Has two super-useful methods: hasNext() and next().
Iterator<element type> <identifier> = <arraylist identifier>.iterator();
hasNext() returns true/false, depending on whether there is another element for the iterator to move to, and next() moves the iterator to the next index.
iterator.remove() should be used to remove elements from a collection, as you are cycling through (but normal ArrayList remove method should be used for everything else). This avoids exceptions caused by removing an index and confusing the iterator, because the iterator performing the removal means that it knows what's going on.

HashMap

A collection which stores pairs of objects, being keys and values. Values can be looked up by using the key. Ideal for one way lookup (where they key is always known), but terrible for reverse lookup (using the value to find the key). Defined with:
HashMap<<key data type>, <value data type>>
'put' and 'get' are the most useful methods.
<identifier>.put(<key>, <value>);
<identifier>.get(<key>);

Set

A set is much like a list, except that it only allows unique elements (i.e. the same element cannot be stored twice).
Set<<data type>> <identifier>;

Enumerator

A set of items, which must be specified, and then cannot be changed. An enumerator is essentially a name for a group of classes (the enumerator values). Defined as follows:
public enum <enumerator identifier>
{
<value1>, <value2>, <value3>;
}
This code is treated as if it were an entire class. By convention, the values are capitalised.
Enumerators are not objects, but its type values are objects (so could be compared with value1 == value2).
Enumerators can be defined with a parameter value (e.g. GO("go")).
Enumerators can have a constructor (this is not defined as public or private, because the enumerator itself is not a class), which is used for the initialisation of the enumerated values (which are objects).
The enumerator can also have methods, which can be used on its objects.
Values in an enumerator can be iterated over like so:
for (<enum identifier> <identifier> : <enum identifier>.values()){ do stuff...}

Declaring Variables

The declaration of variables follows the following general form
<public or private> <data type> <identifier>
where the data type is one of the recognised and support Java data types, and the identifier a name of the programmers choosing, which can be used thereon in to refer to the data stored in that variable.

e.g.
Private String Variable With Identifier 'firstname'
private string firstname;

Public Integer Variable With Identifier 'age'
public int age;

Note: variables are only available to the method or constructor that declared them. Fields are available throughout the class; *local variables* are those declared within a method (commonly given a value in the same line as their declaration).
Local variable Declaration:
<data type> <identifier> = <value (optional)>;
Note: local variables do not have an access modifier.

Class and Object Diagrams

Class Diagram

Shows the relationship between classes; a *static* view.

Object Diagram

Shows objects (and their field types and field values) at one particular moment during runtime; a *dynamic* view.

Fields

Fields are the 'attributes' of an object, which make up an objects state. There are two types of fields:

Primitive Field:

A field of a data type predefined by the java language (e.g. int, string); stored as a literal value.

Object Field:

A field whose type is a class defined by the programmer; stored as a reference to an object.
Note: if a field is an object field, it is not enough to simply define it as being of that class' type. You must initialise it to be a *new* object (e.g. new <class name>)
Syntax:
private <field type> <identifier>;

Inheritance

public class Class1 extends Class2
The first class has all the functionality of the second; referred to as Class1 inheriting from Class2. This is helpful for avoiding code duplication and maintenance issues. If one wants to use methods from the first class, refer to it using super.methodname.
In a class diagram, inheritance is shown with an arrow with a hollow head extending from the subclass to the superclass.
Inheritance is know as an 'is-a' relationship; Class1 'is-a' Class2, and can be used as such.
Class1 is known as a 'subtype' of Class2.
Variables may hold objects of their declared type, or any subtypes of that declared type.
An *inheritance hierarchy* is a group of linked classes formed through inheritance relationships.
A subclass cannot access private methods of its superclass, but other classes can access public methods of both sub and superclasses. A superclass' public methods can be called by the subclass as if they were directly part of the subclass.

Substitution:

When a subtype is used where an instance of a supertype is expected; does not work both ways (cannot used or declare supertype when subtype is expected).
Note: Execution of the superclass' constructor must be the first line of the subclass' constructor, with the line super(<superclass constructor parameters>).
Note: all classes with no explicit superclass inherit from **Object**; thus, all classes on some level extend from object.
Note: Java allows a class to inherit through 'extends' from only one class; it does, however, allow the implementation of as many interfaces as the programmer wants.

Interfaces

A specification of a type, including the type name and a set of methods, but does not include any implementation for any of those methods. Using an interface ensures that the class implementing that interface includes a certain set of methods. Indicated with the keyword 'interface'.
e.g. public interface myInterface{}

Any class can 'implement' this interface, but it must override all the methods defined in the interface, with its own implementation.
e.g. public class myClass implements myInterface{}
An interface defines a type, thus polymorphism applies the same it would as if the interface were a superclass.
Note: interfaces are usually preferable to an abstract class if given a choice.

Casting

When the type of a variable is explicitly stated with that variable.
(<cast type>) <variable name>
Explicitly states that variable to be of the defined type. Can be used to get around the restriction of using a supertype where subtype is called for, but only if the programmer knows that supertype will *always* be an instance of the subtype whenever the code is called (otherwise errors).
The compiler can not do this for you, because it suffers from 'type loss', where it only knows the declared type of a variable, as it interprets code line-by-line.

Autoboxing

Performed automatically when a primitive type value is used in a context requiring a wrapper type. Compiler automatically wraps the primitive type in its appropriate wrapper object so that a primitive type can be directly added to a collection without extra effort by the programmer.
e.g. int, boolean and char are not object types, but primitive types; therefore, because they do not extend from object, it is impossible to add them to a collection.

Wrapper Classes

Every primitive type in Java has a corresponding wrapper class. A wrapper class holds a primitive type, but represents it as an object.
Note: the reverse to autoboxing, **unboxing**, is also performed automatically.

Graphical User Interfaces (GUI)

Components

Individual parts the GUI is built from (e.g. button, menu, slider); placed on a frame by either adding to the *menu bar* of the *content pane*.

Layout

How to arrange components on the screen; achieved by using a layout manager.

Event Handling

The task of redirecting events (e.g. mouse click), so that something happens when they trigger it. If a user activates a component, the system generates an event; this sends a notification to the appropriate part of the program (the event listener), which then triggers the appropriate action.

GUI Libraries

AWT (Abstract Window Toolkit; import java.awt.*) and Swing (import javax.swing.*). Swing builds on the work in AWT, but still makes use of some AWT classes. For overridden classes, we use the Swing

variation, which is denoted by a 'J' at the beginning of the class name (e.g. Frame is AWT; JFrame is Swing).
Note: custom Swing components can be made by creating a class which extends **JComponent**.
JFrame: a top-level window which contains all other components; can be resized and moved.
frame.getContentPane() returns the content pane, to which things can be added
e.g. Container contentPane = frame.getContentPane();
frame.pack(); // makes the window fit the preferred size and layout; always call after adding/resizing components
frame.setVisible(true); // make frame visible on screen
Note: it is popular to extend a class from JFrame, so that the JFrame does not have to be instantiated, but the methods can be called as if they were internal.
JLabel: a component that can display text and/or an image.
e.g. Container contentPane = frame.getContentPane();
JLabel label = new Jlabel("hello world.");
contentPane.add(label);
JMenuBar: only ever one per window; holds JMenu and JMenuItem components.
e.g. JMenuBar menubar = new JMenuBar();
frame.setJMenuBar(menubar);
JMenu: represent a single menu (e.g. 'File'); often held in a menu bar
e.g. JMenu filemenu = new JMenu("File");
menubar.add(filemenu);
JMenuItem: and single menu item (e.g. 'Open'); held inside a menu.
e.g. JMenuItem openitem = new JmenuItem("Open");
filemenu.add(openitem);

Event Listeners

An object can listen to component events by implementing the appropriate *event listener* interface (must implement the right interface given the component it wants to listen to – e.g. *JMenuItem* components raise *ActionEvents*, so a listener must implement the *ActionListener* interface.). Any object can become an event listener for any event. If it does, it will receive a notification about any event it listens to.
When a button is clicked/menu item selected, the component raises an **ActionEvent**. When a mouse is clicked or moved, a **MouseEvent** is raised. When a frame is closed, a **WindowEvent** is generated. There are a many types of event.
Can have a single object listen for multiple events, or have a separate event listener for each object.

One Listener for them All:

If implementing one event listener for all objects, the frame is often used. The class header needs to implement the ActionListener interface, and implement the method public void actionPerformed(ActionEvent e). This class can then be registered as an action listener to an object by defining it as such in that object (e.g. openItem.addActionListener(this);). If the class is the central ActionListener, it must first figure out what event triggered the method call before it can respond.

Inner Classes:

This is a class that is declared inside another class. Instances of an inner class are attached to instances of the enclosing class. The inner class can see and access private fields/methods from its enclosing class, as it is considered part of the enclosing class just like any other method.
Therefore, the programmer could define separate inner classes for each event possible, and define that inner class to be the action listener for the event. This avoids the need to search for which event happened, but still places all the action listeners in the same central location.
e.g. class MyClass
{

```
class OpenActionListener implements ActionListener
{
public void actionPerformed(ActionEvent e){...}
}
class SaveActionListener implements ActionListener
{
public void actionPerformed(ActionEvent e){...}
}
}
JMenuItem open = new JMenuItem("Open");
open.addActionListener(new OpenActionListener());
JMenuItem save = new JMenuItem("Save");
save.addActionListener(new SaveActionListener());
```

Anonymous Inner Classes:

A class without a name, declared as a parameter within another class. Very useful for implementing event listeners, as they are used where only a single instance of the implementation is required (like with menu items). They essentially provide the same functionality as inner classes, but with a syntactical shortcut because the class is not created as specifically, as it does not have a name, and is defined closer to the registration of its 'listener' status.

Anonymous inner classes are created without naming the class, and with the immediate creation of a single instance of the class.

```
e.g. JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
openFile();
}
})
```

Instead of a name, the anonymous class names only its supertype (often an abstract class or interface), instantiates that supertype, then defines a block of implementation of its supertypes abstract methods, as is done with any other class.

Anonymous inner classes are able to access fields/methods of their enclosing class, and local variables/parameters of the method in which they are defined (however, any local variables accessed must be declared as **final** variables).

Layout

Swing uses layout managers to arrange components in a frame. Each separate container holding components has an associated layout manager. Layouts can be nested for more sophisticated interfaces. Layout managers are objects in their own rights, and need to be instantiated before being set to a component.

Layout is defined on a contentPane with:

```
conentPane.setLayout(new <layouttype>());
```

FlowLayout: Arranges all components sequentially from left to right; leaves components at their preferred size, and centers them horizontally – if horizontal space is not enough, the components wrap around to a second line. Can be set to align components left or right.

BorderLayout: places up to five components in an arranged pattern (center, top, bottom, right, left) – more components can be added by adding them to a single JPanel, then adding that panel to one of the positions. Position is defined as being CENTER, NORTH, SOUTH, EAST or WEST. When resized, middle component is stretch both directions; east and west change in height, and north and south change in only width.

```
e.g. label = new JLabel();
contentPane.add(label, BorderLayout.NORTH);
```

BoxLayout: lays out components either horizontally or vertically; does not wrap when resized.
GridLayout: lays components in an evenly spaced grid. Number of rows and columns can be specified, and the layout manager will keep all components the same size.

Containers

Containers appear to other components to be a single component, but they can contain multiple components within them. Each container has its own layout manager attached. The most useful/used container is the JPanel.

JPanel:

Can be inserted as a component into a frames content pane, and then have its own components assigned to it. Various JPanel's, each with their own layout, can be combined to make sophisticated layouts.

Borders

Can be used to group components or add space between them. Every Swing component can have a border.

- BevelBorder
- CompoundBorder
- EmptyBorder
- EtchedBorder
- TitledBorder

e.g. JPanel contentPane = (JPanel) frame.getContentPane();
contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));
Note: you must cast the contentPane as being a JPanel, as Container does not have the setBorder method, but JPanel does.
Note2: can only implement borders after importing the border package (javax.swing.border).

Dialog

Modal Dialog: blocks all interaction with other parts of an application until the dialog is closed; forces user to deal with the dialog.
Non-Modal Dialog: allows user to ignore the dialog and continue interaction with other aspects of the application.
Dialogs are often implemented using the **JDialog** class. Modal dialogs with a standard structure can be implmented using convenience methods in JOptionPane.
JOptionPane: includes three standard dialogs; these are in static methods, so do not need to be instantiated.
Message Dialog: displayed a message with an OK button
Confirm Dialog: displays some text and selection buttons (e.g. yes, no, cancel)
Input Dialog: displays some text and a text field for the user to respond.
e.g. JoptionPane.showMessageDialog(frame, “Some Text”, JoptionPane.<message type (e.g. INFORMATION_MESSAGE) – changes icon in message window>);

General Syntax

Assignment Statement
<variable to assign to> = <expression>;

Print To Console
System.out.println("<text>" + <variable>);
Note: if the parameter to this method (or System.out.print()) is not a String object, the method automatically calls the objects toString() method.

Operators
+: addition (and string concatenation – if one argument in an addition is a string, all with be string)
-: subraction
*: multiplication
/: division
%: modulus (remainder after division)

Logic Operators
&&: and
| |: or
!: not

Substring
Returns a string containing the characters from the lowest index, to the last index (does not return the last index) in the nominated string, the first character in the string being index 0..
<string identifier>.substring(<lowest index>, <last index>);
e.g. Calling name.substring(0, 4) on "Benjamin", would return "Benj".

String Length
length(<string>) returns an integer representing the length of the nominated string.

This
this.<some identifier>
This is used to specifically call an object, field, method or other item from within the same class.

Javadoc
The accepted method of commenting java programs. A javadoc comment is shown by an extra asterisk.
e.g.
/**
*Javadoc comment.
*/
Key Symbols:
@version

@author
@param
@return
- information should go after the symbol for proper formatting.