COMP162 Foundations of Computer Science: Laboratory Manual

Nick Meek Karen Gray

July 9, 2021

Contents

License

Course Information

People

- Steven Livingstone (Course Coordinator): s.livingstone@otago.ac.nz
- Nick Meek (Lab Coordinator): nick.meek@otago.ac.nz
- Karen Gray (Lab Coordinator): karen.gray@otago.ac.nz

Course Structure

The ability to program relies heavily on mastering certain skills and techniques. These skills are cumulative in the sense that mastering skills further on in the material requires that you've fully mastered prior skills. Therefore, the structure of COMP162 is based on a mastery model. In this model, students progress at their own pace and move on to the next topic only when they've mastered the previous topic. Nevertheless, the lectures do follow a fixed pace, and you might find that your lab work is getting either ahead or behind the lecture topics. If you're ahead, that's great. But try not to get too far behind as it will become harder and harder to catch up. Since COMP162 is self-paced, different students will get to different parts of the book by the end of the semester. This is perfectly fine but of course your final grade will be greatly influenced by how many Mastery Tests you pass.

Assessment

- There are 8 Mastery Levels and their associated Preparation Exercises and a Final Exam.
- The Mastery Levels (and their associated preparation exercises) are worth 40% in total.
- The final exam is worth 60%.
- The Preparation Exercises are marked in the lab by a demonstrator and are worth 1% each.
- The Mastery Levels are pass/fail.
- You cannot attempt the next Mastery Level until you have completed the Preparation Exercises.
- You can retry a Mastery Level as many times as you like, but only once per day.
- If you pass a Mastery Level, you get all the marks available for that level.
- If you fail a Mastery Level, you don't get any marks but you can resit any following day.

Table 1: COMP162 Assessment Package

Assessment	Coverage	Worth %
Mastery Test 1 Prep		1 %
Mastery Test 1	Java Basics	4 %
Mastery Test 2 Prep		1 %
Mastery Test 2	OO and Hierarchies	4 %
Mastery Test 3 Prep		1 %
Mastery Test 3	Algorithms	4 %
Mastery Test 4 Prep		1 %
Mastery Test 4	Recursion	4 %
Mastery Test 5 Prep		1 %
Mastery Test 5	Linked Lists	4 %
Mastery Test 6 Prep		1 %
Mastery Test 6	Collections and Data Structures	4 %
Mastery Test 7 Prep		1 %
Mastery Test 7	R Basics	4 %
Mastery Test 8 Prep		1 %
Mastery Test 8	C Basics	4 %
Final Exam	All	60 %

There is one cut-off date:

- Mastery Tests 1 and 2 must be completed before the end of Week 4.

Lectures and Laboratories

- Lectures

  Wednesday at 9am: topics of relevance to labs, mastery levels, practical tests and programming in general.

  Friday at 9am: topics of relevance to labs, mastery levels, practical tests and programming in general.

- Laboratories

  We will be using Lab A in the Owheo Building (133 Union Street East). You will be streamed into two scheduled laboratories each week. Each lab is two hours long. The streaming lists will be posted on the noticeboards near the Computer Science Main Office in the Owheo Building (133 Union Street East) and on eVision.

Labs start in the first half of Week 1.

You should attend your scheduled lab time - we will be scanning you into the lab. You can attend other streams in addition to your streamed session if there are spare seats in the lab.

Table 2: COMP162 Lecture/Lab Schedule

Date	Lecture Title	Number	Lab Title	MT
July 14	Introduction/Ethics	1	Recap COMP161 (Bicycles/ Animals)	(MT1)
July 16	Java basics with Bubblesort	2	Recap COMP161 (Bicycles/ Animals	(MT1))
July 21	Inheritance 1	3	Recap COMP161 (AntTravellers)	(MT1)
July 24	Inheritance 2	4	COMP160 Lab21	(MT2)
July 28	Interfaces	5	COMP160 Lab22	(MT2)
July 30	Algorithms	6	COMP160 Lab23	(MT2)
August 4	Recursion	7	Interfaces	(MT2)
August 6	Analysis of Algorithms	8	Algorithms	(MT3)
August 11	Arrays	9	Recursion	(MT4)
August 13	Lists	10	Catch-up	-
August 18	Linked Lists	11	Arrays - Young Tableaux	-
August 20	Generics and Collections	12	Linked Lists Introduction	(MT5)
August 25	Stacks	13	Linked Lists - Young Tableaux	(MT5)
August 27	Queues	14	Stacks	(MT6)
September 8	Insertion and Selection Sort	15	Catch-up	-
September 10	Quicksort	16	Queues	(MT6)
September 15	-	17	Bucket sort	-
September 17	-	18	Project Euler	-
September 22	Array Programming in R	19	Array Programming in R 1	(MT7)
September 24	-	20	Array Programming in R 2	(MT7)
September 29	-	21	Catch-up	-
October 1	C Programming	22	C Programming	(MT8)
October 6	C Programming	23	C Programming	(MT8)
October 8		24	C Programming	(MT8)
October 13	-	25	Catch-up	-
October 15	-	26	Catch-up	-

Academic Integrity and Academic Misconduct

Academic integrity means being honest in your studying and assessments. It is the basis for ethical decision-making and behaviour in an academic context. Academic integrity is informed by the values of honesty, trust, responsibility, fairness, respect and courage. Students are expected to be aware of, and act in accordance with, the University's Academic Integrity Policy.

Academic Misconduct, such as plagiarism or cheating, is a breach of Academic Integrity and is taken very seriously by the University. Types of misconduct include plagiarism, copying, unauthorised collaboration, taking unauthorised material into a test or exam, impersonation, and assisting someone else's misconduct. A more extensive list of the types of academic misconduct and associated processes and penalties is available in the University's Student Academic Misconduct Procedures.

It is your responsibility to be aware of and use acceptable academic practices when completing your assessments. To access the information in the Academic Integrity Policy and learn more, please visit the University's Academic Integrity website at `www.otago.ac.nz/study/academicintegrity` or ask at the Student Learning Centre or Library. If you have any questions, ask your lecturer.

- Academic Integrity Policy (`www.otago.ac.nz/administration/policies/otago116838.html`)
- Student Academic Misconduct Procedures (`http://www.otago.ac.nz/administration/policies/otago116850.html`)

Use of the Labs

The lab has 24 hour, 7 days access.

- Eating is not allowed in the lab, but drinking from spill-proof containers is ok.
- Installing your own software on our machines is forbidden (but installing Python modules is OK).
- Copying software installed on lab machines is forbidden.
- Students streamed to a particular lab session will be given priority for seating and demonstrator time.
- Files not related to your Computer Science course(s) should not be stored in your Computer Science Home directory. You should be aware that Computer Science Systems Administrators can inspect the contents of your home directory at any time.
- Students must behave in a sensible, responsible and adult manner whilst in the lab.
- Please ensure that your behaviour, music or conversation is not adversely affecting others trying to use the lab.

Blackboard

COMP162 will be using Blackboard. You can log-on to Blackboard at:

```
https://blackboard.otago.ac.nz/
```

0.1 F.A.Q.

- Can I work at home?

  Ideally you should attend all your scheduled labs. It is easiest to learn to program when you work in a supportive environment, especially one with friendly demonstrators. Of course you are encouraged to do extra work at home and to stay at home if you are ill, see below.

- What do I do if I am sick?

  Stay home. If you are well enough to work through the course work do so – it's important to not fall behind. If you will be away more than a few days let one of the teaching team know.

- What if COVID returns?

  We are well equipped to teach this under any COVID Level; don't worry, we've got this.

- What do I do if I miss a lab session?

  Catch up on the lab work outside of your scheduled lab time. Although this course is 'self-paced' you really need to keep up with the lecture pace to get good marks.

- Can I come to other lab sessions instead of my own

  NO, this semester we need you to attend your own lab session. Of course if you have some unavoidable appointment or circumstance please get in touch with us and we will do what we can to help you. We will be scanning your ID card at the door so no trying to sneak in.

- What can I do if I am falling behind?

  Don't panic. Set yourself manageable goals. Attend all your lab sessions. Settle in and turn your phone off. Ask questions as often as you need to.

0.2 Rules for use of the COMP162 Laboratory

0.2.1 Software Licences

The software and manuals available in the COMP161 laboratory have been bought under various licences. In general, these licences state that under the Copyright Laws:- No part of this work may be reproduced in any form (in whole or in part) or by any means or used to make a derivative work (such as a translation, transformation or adaptation). Under the law, copying includes translating into another language or format. All rights reserved. Hence, you are NOT permitted to copy the system software or any applications under any circumstances, nor should the systems be used to copy any other licensed or copyright software without the owner's permission. Failure to observe these requirements will be considered a serious breach of University regulations, and will be dealt with under the Discipline Regulations.

0.2.2 Rules

- You must never, ever, ever allow anyone else to use a departmental computer logged in under your account name. Do not log a friend in and let them work/ play beside you.
- Your 24-hour access to the Computer Science Department labs is for you and you alone. You must not allow friends to come into the building with you after-hours.
- Eating is not allowed while sitting at a computer. You may drink out of sipper top bottles, or cups with tight-fitting lids.

- Cell phones must be switched to silent and kept off the desk during streamed lab sessions.
- Your computer science home directory is to be used responsibly and for coursework only.

Lesson 1

Kia ora and welcome

In this chapter and the next we will reaquaint you with JGrasp and the Java concepts and skills you should have gathered in COMP161. In COMP162 we will expect you to be more self-reliant so we will not spell things out in as much detail. We expect that you can look up methods in the Java API or search online for answers. We do still of course encourage you to talk to the demonstrator, just not for every little detail.

1.1 Lab Basics

1.1.1 First Time In The Lab?

If this is the first time you have used a Computer Science laboratory your username will usually[1] be *your first initial followed by your last name*. For example Billie Blogs has the username bblogs, and Edwin Heisenberg has the username eheisenberg. Your password will be your *id number as shown on your student id card*.

Your first action should be to *change your password*. To do this click on your username at top-right of the computer screen. Choose "Users and Groups Preferences", then click on the "Change Password ..." button, and fill out the required entry boxes. Your new password must follow the following convention:

- Include an upper-case letter.
- Include a lower-case letter.
- Include a digit.
- Not be an English word

When you leave the lab remember to log-out. To do this click on the Apple menu (top-left of the Desktop) and choose 'Log-out <your username>' Please don't shut-down the computer as this annoys the system administrators. Please don't lock the screen and then walk away; these are shared machines. If we find a computer with a locked screen we will restart it and any unsaved work will be lost.

In the rest of this book, the action of selecting menu items will be denoted *Apple* $\rightarrow$ *Log Out*.

1.1.2 DemoCall

To call a demonstrator use the DemoCall icon on the desktop. This queues your call and ensures people are seen in the order in which they asked for help.

[1] if the name is not unique, a slightly different one will be created — ask a PPF to check.

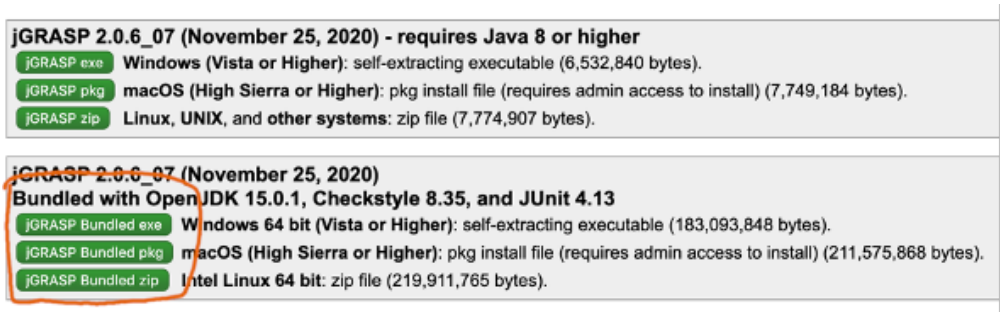


Figure 1.1: Downloading the bundled version of JGrasp

1.1.3 Course Files

Keeping all the files you create during this course organized is a must. At the beginning of each laboratory you should create a folder (directory) in your Home > COMP162 directory for that lab, e.g. Lab01, Lab02, Lab03 etc. You could make some of those now. Sometimes we provide files, you will find these on Blackboard in the **Lab Files** folder.

1.1.4 Moodle

We will (again) be using Moodle for the Mastery Tests. You should log-on now just to make sure it works for you. If it doesn't let a demonstrator know. Moodle is at http://progress.otago.ac.nz/. Your username and password are the same as for Blackboard and eVision.

1.1.5 JGrasp

JGrasp is a free, cross-platform, open-source application; it is already installed on all the lab computers. JGrasp is available for Windows and Mac, and is free to download and install (https://www.jgrasp.org/). If you do install it on your own machine (and we would encourage you to do so) then ensure you get the 'Bundled Version' as shown in the picture below. It additionally contains a recent version of the Java JDK and other packages that will be needed later.

There is a JGrasp Tutorial on Blackboard under Course Documents. We strongly encourage you to read it in the next few days as it will provide a lot more detail than we provide in the labs on the features of JGrasp. We will refer to the Toolbar buttons and other JGrasp features by name in the lab instructions so you will need to refer to the pictures below to see which button/pane etc. we are talking about.

There is one final piece of setup you need to do and then you are ready to start writing programs. Java is constantly being updated. The most current version as I write this is version 15. To make sure we are all using the same version of Java (and if you install JGrasp on your own machine you should definitely do this as well) we will specify that JGrasp use its own bundled version of Java.

1. From the Settings menu choose JGRASP Startup Settings.
2. If there is more than one version of Java to choose from in the drop-down menu choose the one that has 'bundled' in the path, as shown.
3. Click OK.
4. You are ready to start programming!

1.1.6 Good Programming Practices

There are a few practices that we will encourage in the lab because they will make you a better, more efficient and more employable programmer.

Incremental Development

Incremental Development means that at every stage of development your program compiles and works as much as it can. If you are developing a program incrementally you will typically write one or two lines of code, and then see if your program still compiles. You might print out a value or two to check that the expected values are being produced. Programming this way ensures errors are caught early when they are easy to locate and fix.

Javadoc Comments

Few people really enjoy writing comments but they are a key part of writing (maintainable) programs. Every class and every method **must** have a short Javadoc-style comment explaining what this class or method does. You should include @param and @return lines as in the example:

```
/**
* A method to calculate the number of seconds in a given
* number of hours and minutes.
* @param hours The number of hours
* @param minutes The number of minutes
* @return The number of seconds
*/
```

1.2 What now?

This is a great opportunity to discuss any code from last semester (or over the break) that you wanted to talk about. If you have time left in this lab, start looking ahead to the next lab.

Lesson 2

Java Recap

The exercises in this lab are designed to refresh your Java skills. Please, please, please talk to a demonstrator about your solutions.

Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

2.1 ** Bicycles

Bicycle
- wheelDia:int - electric:boolean - colour: String
+ Bicycle() + Bicycle(wheelDia:int, electric:boolean) + setColour():void + isElectric():boolean + getImperial():double + getMetric():int + toString():String

1. Write a Java class called Bicycle according to the UML specifications in the diagram above, and described below:
   (a) It has 3 data fields, 1 is an int value representing the bicycle's wheel diameter in mm, 1 is a boolean and 1 is a String.
   (b) It has a constructor which takes an int input parameter and a boolean input parameter, and initialises the corresponding datafields.
   (c) It has a method setColour that uses a Scanner object to ask the user for the bicycle's colour, e.g. "What colour is this 697mm bicycle?" and uses the value typed in to set the colour data field. The constructor should call this method.
   (d) It has an accessor isElectric which returns the value stored in the electric data field.
   (e) It has a getImperial method which returns a double representing the bicycle's wheel diameter in Imperial measurement. (To convert mm to inches divide the length value by 25.4)
   (f) It has a getMetric method which returns an int representing the bicycle's wheel diameter in mm.

(g) It has a toString method which returns a String describing the bicycle's colour and wheel size. If the bike is electric, use imperial measurements, otherwise use metric (mm). Label the unit used, and describe the power method. See the output suggested in 2 b) for what is expected.

2. Write an application class which contains a main method which:

   (a) Makes an array of 3 Bicycle objects and stores in it the following:
      i. an instance of an electric bicycle with wheel diameter 697.
      ii. an instance of a pedal-powered bicycle with wheel diameter 660.
      iii. an instance of a pedal-powered bicycle with wheel diameter 610.

   (b) Writes a loop which displays data returned by the toString method of every bicycle.

```
Bicycle is red. Wheel size is 27.440944881889767 inches. Electric
Bicycle is blue. Wheel size is 660 mm. Push-bike
Bicycle is pink. Wheel size is 610 mm. Push-bike
```

3. Write a second method in the application class. The array will need to be sent to the method as an argument (via the parameter list). The method should use a loop to determine the bicycle with the largest wheel, then display its data. Call this method from the main method. The output should like like this:

```
Largest bicycle: Bicycle is red. Wheel size is 27.440944881889767 inches. Electric
```

2.2 ** 2-D Arrays

```
public class MyClass{
    public static void main(String[]args){
        String [][] words = {
        {"Apple","Aardvark","Albatross"},
        {"Bear","Baby","Box"},
        {"Candy","Cave","Cube"},
        {"Dog","Desk"}};
    }
}
```

1. Type in the application class listed above. It has initialised a 2 dimensional array of String objects.

   (a) In the main method, write a nested loop which displays all the items of each inner array on the same line, separated by a space. The output should look like this:

```
Apple Aardvark Albatross
Bear Baby Box
Candy Cave Cube
Dog Desk
```

(b) Write another method in the class (a suitable name for this method would be findChar). This method should take a char and a 2 dimensional array of Strings as parameters. The purpose of the method is to display every word in the array which contains the character. Write nested loops which access every element in the array. Search every element for the character. If the character is found, display the word.

Call this method from the main method. The method call findChar('v',words); should produce the output Finding letter v: Aardvark Cave

The method call findChar('b',words); should produce the output Finding letter b: Albatross Bear Baby Box Cube

2.3 ** Coin Tosses - Part One

The data we'll be working with in these exercises represent the results of a series of coin tosses. There are various ways we could represent them, but one of the simplest is as an array of boolean values (i.e., **`boolean[]`**) using **`true`** to represent 'heads' and **`false`** to represent 'tails'.

In the directory `LabFiles/Lab01` on Blackboard, you will find some code which you should use as your starting point when completing this lab.

The skeleton code provided includes some data fields and a basic constructor. To this, add two functions:

`countHeads()` A method that returns an **`int`** which is the number of occurrences of 'heads' in the coin tosses.

`toString()` A method that returns a **`String`** representation of the coin tosses, using **`H`** to represent heads and **`T`** to represent tails for example **`HHTHHHTTT`**.

You can progressively test that each method you write is working as it should. One way of doing this is by adding code to a main method which exercises the other methods as you write them. For example, once you have written your **`countHeads()`** method you could write a simple **`main()`** to use it like so:

```
public class CoinsApp{
   public static void main(String[] args){
      boolean[] coinTosses = {true, false, true, true, false};

     Coins c = new Coins(coinTosses);
     System.out.println(c.countHeads();
  }
}
```

This creates a **`boolean`** array to pass to the provided constructor. It then creates a new **`Coins`** instance using that constructor. And finally, prints out the result of calling **`countHeads()`** on that instance. If our **`countHeads()`** works correctly then we should see the number 3 printed out when we run our program.

2.4 ** Coin Tosses - Part two

Now add two more constructors:

`Coins(String c)` Creates a **`Coins`** object from a **`String`** consisting entirely of the characters **`H`** and **`T`** (i.e., the result of applying **`toString()`** to the constructed object should be the original string **`c`**).

`Coins(int length)` Constructs a **`Coins`** object consisting of a series of **`length`** coins – the value of each coin should be determined by a random coin toss.

Also add one more method:

`countRuns()` Returns an **`int`** which is the number of runs in this sequence of coins (a run is a block of coins all showing the same face, so for example in **`HHTHHHTTT`** there are four runs namely **`HH`**, **`T`**, **`HHH`**, and **`TTT`**.

2.5 Reflection and extension

- Why is the **`Coins`** representation using an internal boolean array more convenient than just using a **`String`** consisting of **`'H'`** and **`'T'`** characters? Or is it?
- How might you use the classes and methods that have been written here to investigate the questions:
  - What is the average number of runs when 1000 coins are tossed?
  - What is the average length of the longest run when 1000 coins are tossed, and how are the lengths of the longest runs distributed?

  Of course '1000' here is just a placeholder for "an arbitrary integer n".
- Someone suggests that 'a coin might land on its edge'. How could you accommodate his astute observation?

Lesson 3

Java Recap (2)

Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

3.1 ** Java Recap Exercise

Some scientists are studying the behaviour of ants. For part of the study they want to track where and how far an 'explorer ant' travels. They do this by fitting tiny GPS tracking devices to the ants as they leave the nest. Every time the ant changes direction it's location is stored. When the ant returns to the colony the data is retrieved from the tracking device. You have been asked to write a program that allows the scientists to work with the data. See 3.1 for the tracks of four ants code-name "Red Ant", "Blue Ant", "Green Ant" and "Purple Ant".

3.1.1 Make a class from a UML

1. Make two classes as described by the UMLs below. Note the static method in the Point class.
2. Write the code for the methods - most of it is pretty simple and shouldn't pose any problems.

To calculate the distance between two points use the following:
Distance between point1 and point2 = $Sqrt((point1\_X - point2\_X)^2 + (point1\_Y - point2\_Y)^2)$

Point
- x:int - y:int
+ Point() + Point(x:int, y:int) + getX():int + getY():int + setX(x:int):void + setY(y:int):void + distanceToOtherPoint(p:Point):double + distanceBetweenTwoPoints(p1:Point, p2:Point):double + toString():String

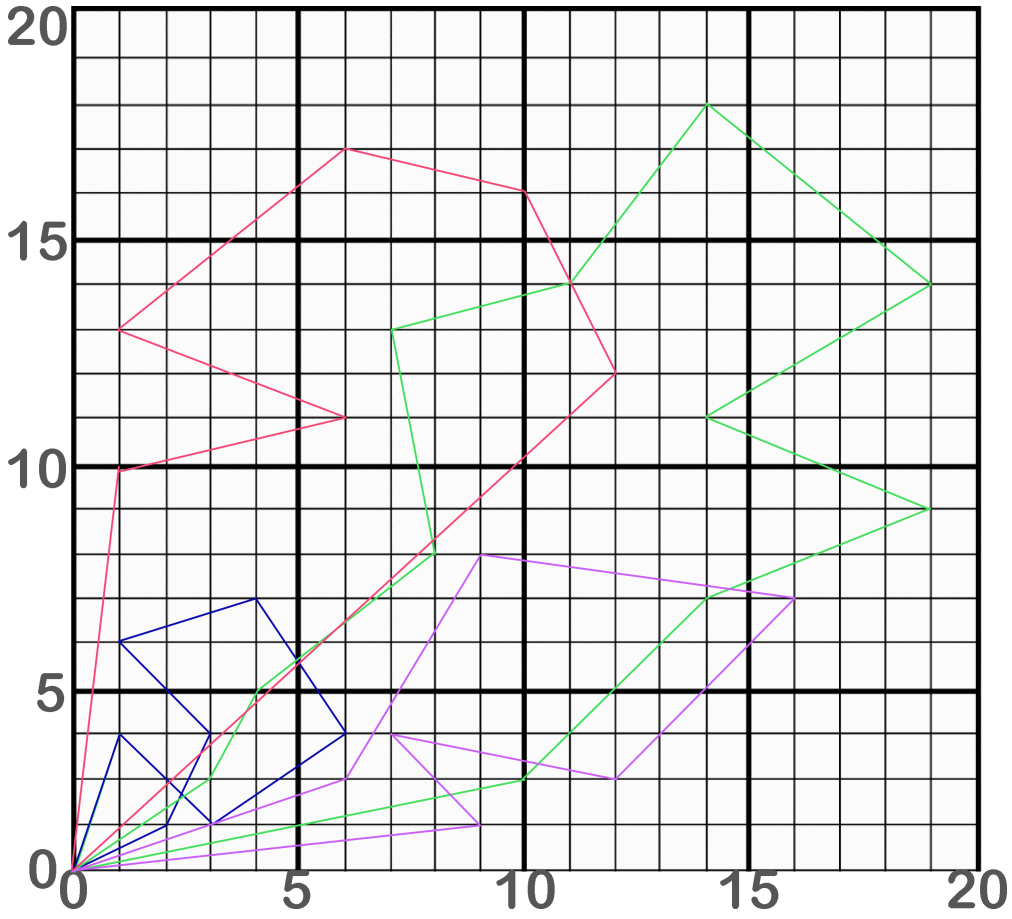


Figure 3.1: The paths of various ants

You can test your point class in the Interactions Pane like so:

```
Point p1 = new Point(0, 0)
Point p2 = new Point(3, 4)
Point.distanceBetweenTwoPoints(p1, p2)
5.0
p1.distanceToOtherPoint(p2)
5.0
```

Ant
- antId:String - journey:Point[]
+ Ant() + Ant(id:String, numPoints:int) + getId():String + setId(id:String):void + getDistanceTravelled():double + getShortestLeg():double + getLongestLeg():double + addPointVisited(p:Point):void + toString():String

3.1.2 Make an Application class

When you have the two classes complete create an application class (AntStudy.java) and add code to:

1. Create a new instance of the Ant class with id = ”Blue Ant”.
2. Add each point that the ant visited using the **`addPointVisited()`** method. The first and last Points are always at position (0, 0).
3. Call the **`getShortestLeg(), getLongestLeg() and getDistanceTravelled()`** methods and print out the results. See below for what your output should look like. Use **`DecimalFormat`** to format the values in the output.

```
Details for: Blue Ant
Shortest Leg: 2.236
Longest Leg: 4.243
Total Distance: 26.966
```

1. Add three new instances of the Ant class to your application class and add their respective journeys.

3.1.3 Manage an ArrayList of Objects

Rather than having to create a new variable to hold each instance of the Ant class it would be much better to use an Array, actually we will use an ArrayList.

1. Modify your application class so that rather than having four individually named instances of the Ant class make an ArrayList of Ant in the application class and add each instance of Ant to it.
2. Now add code to print out the details for all the Ants.

3.2 Extension Exercise - Read the Data from a File

All this entering data manually in the application class is very tedious and not at all scalable. In the Lab Files/Lab 03 you will find some text files that contain the journeys of a number of ants. Each line of the file contains the points that a single ant visited for a single journey. Each line consists of the following:

1. The id of the ant followed by a colon ':'
2. pairs of integers separated by a comma that represent a point visited.
3. If there is more than one point visited points are separated by colons.

For example the journey of the Blue Ant would be represented thus:
Blue Ant: 2,1: 3,4: 1,6: 4,7: 6,4: 3,1

Add code to your main method to read data from a file and print out the results as in the previous exercise. The file BlueAntJourney just has the points of the Blue Ant from the previous exercise to make testing easier. You may hard-code the file name if you wish.

Lesson 4

A Graphical User Interface

This lab develops the GUI for a program that will allow us to explore Hierarchies and Abstract classes. It is taken from the first half of Lab23 in COMP161. If you have already coded it - congratulations! You are significantly ahead of the game.

To help you prepare for the Hierarchy aspects that we will get to in lab 6 we have included some important prep exercises for you to do.

Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

4.1 ** Preparation

1. The class Art is extended by 2 classes, Sculpture and Print. The Sculpture class has its call to the super constructor written. Complete the Print constructor so all its data fields are correctly instantiated.

2. If there were to be other data fields, year and creator, which class should they be in?

   _____

3. Finish the foreach loop in the application class so the total value of the collection is correctly calculated.

```
public class Art{
    protected int height;
    protected int width;
    protected int value;

    public Art(int wd, int ht, int valu){
        width = wd;
        height = ht;
        value = valu;
    }
}
```

```
    public class Sculpture extends Art{
        private int weight;
        private int depth;

        public Sculpture(int wd, int ht, int dpth, int wt, int value){
            super(wd, ht, value);
            depth = dpth;
            weight = wt;
        }
    }
```

```
    public class Print extends Art{
        private int numberMade;

        public Print( int wd, int ht, int num, int value ){



        }
    }
```

```
    public class ArtApp{
        public static void main(String[]args){
        Art[] collection = {new Sculpture(200,500,300,25,25000),
                            new Print(800,900,40,400),
                            new Sculpture(350, 835, 553, 150,4500)};
        for(Art work: collection){


        }
        System.out.println("total value $$$" + totalValue);
    }
}
```

4.2 ** Lab Work: Displaying Shapes

4.2.1 The app class

You will need an application class that creates an instance of **JFrame** and adds an instance of **ShapePanel** to its content pane.

4.2.2 The ShapePanel class

Examine the UML and the containment hierarchy diagrams shown. Ignore the greyed out data fields until lab 5.

In addition:

- The **controlPanel** should be 100 x 400 pixels.

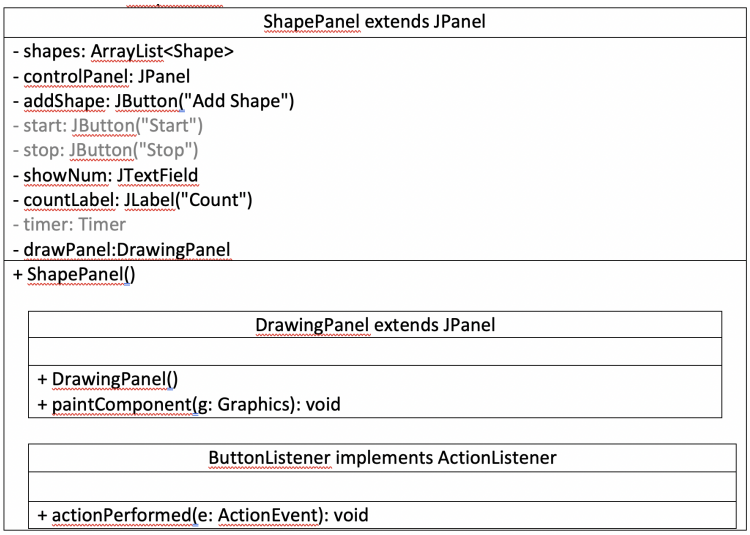


Figure 4.1: Initial UML for ShapePanel class

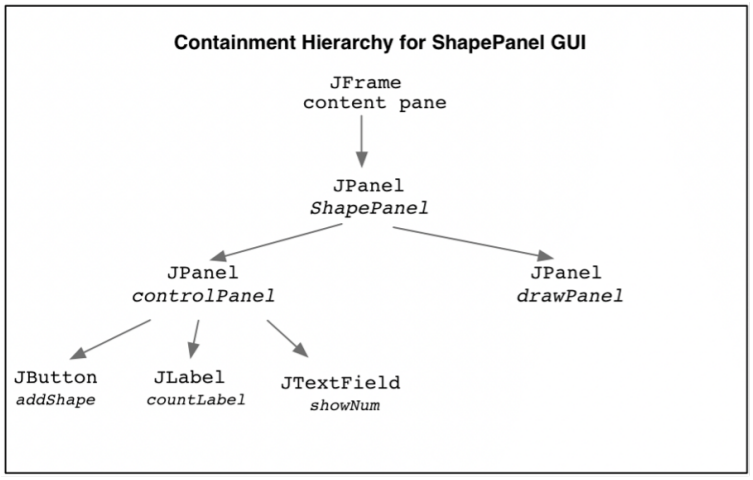


Figure 4.2: Initial Containment Hierarchy:

- The **JTextField** should have space for 2 characters.
- The **drawPanel** should be 400 x 400 pixels and have a specified background colour.

1. Create the graphical components for the project as described. Your program should produce something like figure 4.3
2. Now that the graphical structure is written, the next step is to write the Shape class as described by the UML diagram, see figure 4.4.
3. In the constructor:
   - **width** is set to a random value between 10 and 30
   - **height** is the same as width

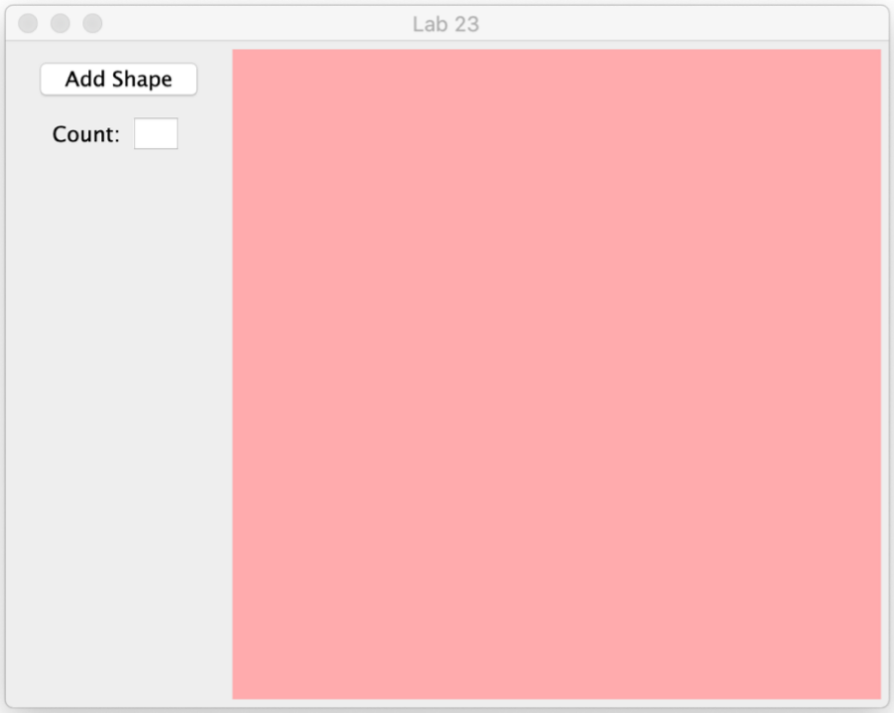


Figure 4.3: The Graphical Components

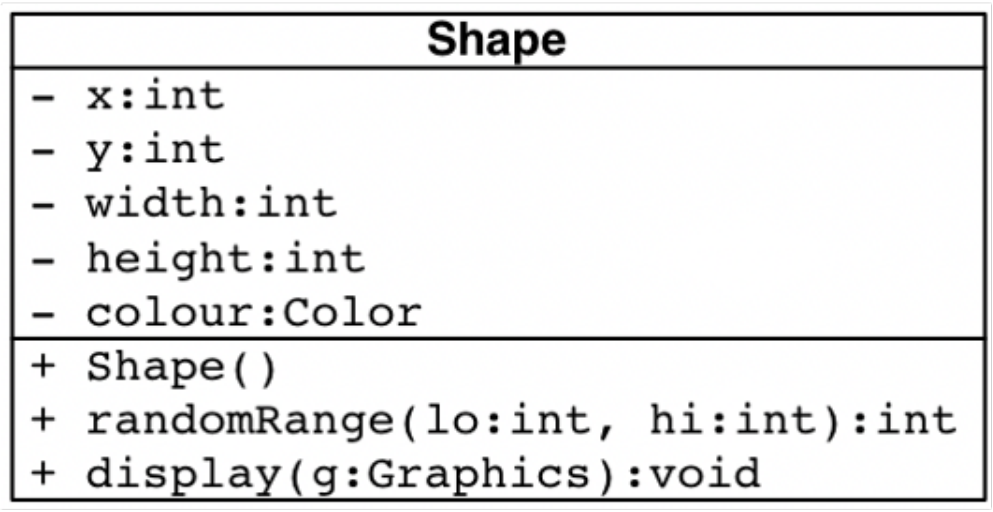


Figure 4.4: UML diagram for the Shape class

- **x** is a random value between 0 and the width of the **DrawingPanel** (400) minus the width of the Shape.
- **y** is a random value between 0 and the height of the **DrawingPanel** (400) minus the height of the

Shape.

- **colour** is set to a random RGB value. A new Color can be created using the syntax new Color(red, green, blue) where red, green and blue are each integer values between 0 and 255.
- The **display** method should set the Graphics colour, and draw a solid circle using the information stored in the data fields for size and location.

4. Test your class by adding a **Shape** data field to **ShapePanel**. Don't forget to instantiate a new Shape object e.g. Shape shape = new Shape(). In the **paintComponent** method call the **display** method on your Shape object. If things are set up correctly, each time you run the code you should see a circle of varying size in a random colour in a random position.

5. Adapt your code so that a different Shape will be instantiated and displayed each time the user presses the **addShape** button. This will involve registering your **JButton** with a listener and making a new Shape in the **actionPerformed** method. Don't forget to import java.awt.event.* at the top of your class. The screen will need to be refreshed each time there is a different shape to display. Use the repaint() method. (The repaint method automatically calls the **paintComponent** method, which begins by redrawing the super **JPanel** thereby removing the previous shape.)

6. To complete part 1, rather than have just one Shape, the **ShapePanel** class should have a shapes data field which holds a reference to an ArrayList of Shape objects. A new Shape will be added each time the user presses the **addShape** button. The current size of shapes should be displayed on the **showNum** JTextField. A for-each loop in the **paintComponent** method should iterate over every Shape in the ArrayList and call its display method.

7. Lastly, tidy up your code and ensure you have written good java doc style comments (/** */) above all method headers!

Lesson 5

Animation

Now you are going to get your spots moving. The Shape class will need a move method which updates the x and y (location) data fields by some formula each time it is called.

In order to get the animation working, the ShapePanel class will declare an instance of Timer which automatically generates an ActionEvent at regular intervals (possibly fast). This means that the actionPerformed method will get called at regular intervals. If we put code that updates and redraws the locations of Shapes inside the actionPerformed method, each time the Shapes are redrawn they will be in a different place, creating the illusion of movement (in the same way that motion pictures display the individual frames of a film).

The Timer class has start and a stop method which can be used to control the action. You will add a button to start them moving, and a button to stop them moving.
Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.First, some more preparation around hierarchies and how they work.

5.1 ** Preparation

1. Examine the class hierarchy in the diagram, then answer the questions below.

   (a) There are three different travel methods, A B and C. Which one does Kiwi use?

   (b) Which one does LittleBlue use?

   (c) Which one does Albatross use?

   (d) List all the methods that Kiwi can use?

   (e) Can LittleBlue use the getWingSpan() method?

   (f) Can Albatross moult()?

   (g) Write a statement for the Kiwi class so it can access access Bird's nests() method.

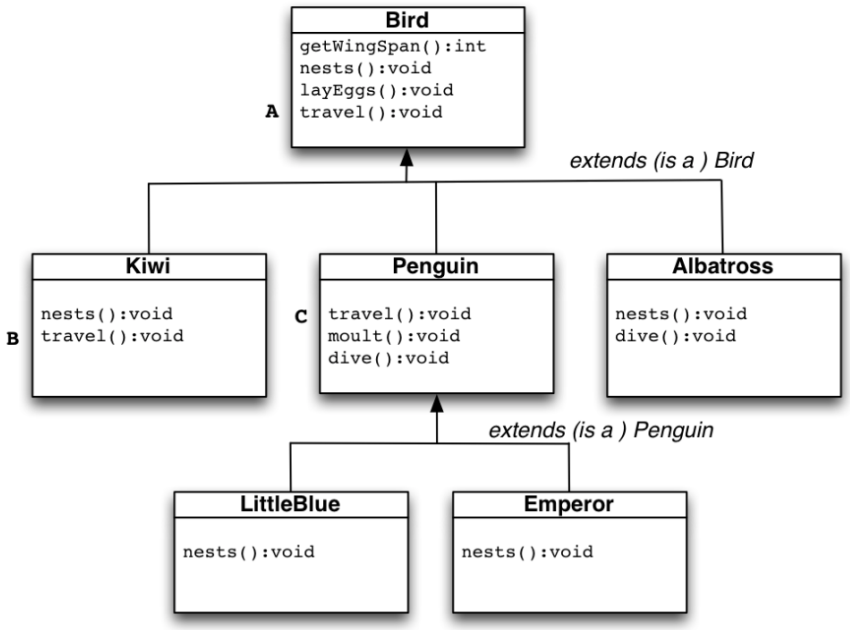


Figure 5.1: The Bird hierarchy

(h) Declare an array which may hold up to 5 Bird elements. This array will be able to store references to any type of Bird (Kiwi OR Albatross OR Penguin OR LittleBlue OR Emporer OR Bird). Write another statement which makes the first element of the array a reference to a new instance of LittleBlue.

2. Why is inheritance useful?

3. What is over-riding? Give an example from the Bird hierarchy.

5.2 ** The Shape class

1. Declare two int data fields called moveX and moveY in the Shape class. Set their initial values to 1.

2. Write a void method called move. In this method, add moveX to the value of x, and add moveY to the value of y. This changes the location at which the shape would be drawn by 1 pixel to the right and 1 pixel down every time it is called.

5.3 ** The ShapePanel class

1. Declare two new JButtons as data fields in the ShapePanel class, one to start and one to stop. Make these buttons capable of event handling. Add them to the controlPanel.

2. Declare a data field of type Timer called timer, and declare a final int data field DELAY initialised to 10 (milliseconds).

```
Timer timer;
private final int DELAY = 10;
```

There are three Timer classes in the Java API. The Timer we are using is a class in the javax.swing package. If you get a multiple Timer error when using Timer, you can fix this by prefixing every use of the class name Timer with the full package name i.e. javax.swing.Timer

3. In the ShapePanel constructor, set the variable timer to be a new instance of Timer, and send it the DELAY and the ButtonListener as parameters.

```
timer = new Timer(DELAY, listener);// or whatever you have called your ActionListener
```

4. In the actionPerformed method, begin with an if block for when the source of the event is timer and an else for when the source of the event is the addShape button. For the timer, each valid element in the ArrayList should have its move method called in turn. This will repeat the x,y updating of each Shape at a rate controlled by the DELAY parameter. Make sure the repaint method is being called at the end of the actionPerformed method, not in an if block. The repaint method will redraw the panel (every 10 milliseconds) with the Shape objects in their new position.

5. In the actionPerformed method, call the method timer.stop() if the stop button is the source, and timer.start() if the start button is the source.

6. Compile and run your code.

If all is well, your shapes will quietly slip off the right or bottom of the panel, never to be seen again. You need to keep them in view by bouncing them off the sides of the panel. The move method will need to check whether the current x or y location has put the shape on the edge of the drawing panel. If it is on the edge, either the horizontal (moveX) or vertical (moveY) direction should be reversed. How can you determine whether the shape is on the edge? Take a look at the diagram below, imagining that the shape on the left edge is travelling left (moveX = -1), and the shape on the right edge is travelling right (moveX = 1).

7. Write an if statement in the move method which checks to see whether x is on or over the edge of the panel. If so, change moveX to -moveX, toggling the direction of the horizontal movement. Then (whether or not you have had to change direction) carry on and make the change to x.

8. Write another statement to look after the vertical movement. Run your code again. Hopefully now your shapes are contained within the panel.

When you're ready, go and change some things. Try changing the delay to 5, then to 20. See what happens. In the move method, comment out the line which increments y. Can you predict what will happen? In the Shape class, set the value of data field moveX to 5. Some challenges.

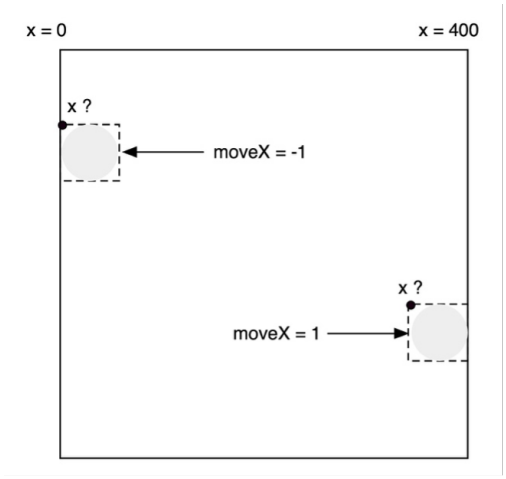


Figure 5.2: What happens to objects at the edge?

9. Make shapes which are wider than 15 pixels travel straight up and down, while all other shapes travel sideways.
10. Get the shapes which are initially drawn in the lower half of the screen to start their travel in an upwards rather than downwards direction. (There is a very simple solution to this one – it happens in the Constructor.)

Your mission: Get your shapes to change as they move according to some criteria of your own design.

Lesson 6

Abstract classes and Hierarchies

Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

6.1 ** Preparation

1. (a) Finish the Car and Boat constructors so that all data fields are correctly filled.

   (b) A car's fuel consumption is calculated by mutiplying litres per kilometre by trip length, and a boat's fuel consumption is calculated by mutiplying litres per hour by trip length - write the missing (required) methods.

```
public abstract class Vehicle{ protected String name;

    protected String ctryOfOrigin;

    public abstract int fuelConsumption(int tripLength);

    public Vehicle(String name, String ctryOfOrigin){
        this.ctryOfOrigin = ctryOfOrigin;
        this.name = name;
    }
}
```

```
public class Car extends Vehicle{
    private int numAirBags;
    private int litresPerKm;

    public Car(String name, String ctryOfOrigin, int lpK, int airBags){


    }

    public                                                    {


    }
```

```
    public String toString(){
        return "Car with " + numAirBags + " air bags made in " + ctryOfOrigin;
    }
}
```

```
public class Boat extends Vehicle{
    private int litresPerHr;
    private int numBerths;

    public Boat(String name, String ctryOfOrigin, int lpH, int brths){


    }

    public                                                          {


    }

    public String toString(){
        return "Boat with " + numBerths + " berths made in " + ctryOfOrigin;
    }
}
```

(c) In the main method of the application class (below), write a foreach loop which prints out what is returned by the toString method of every element in the vehicles array.

```
public class VehicleApp{
    public static void main(String[]args){
        Vehicle[] vehicles = {new Car("Toyota Starlet", "Japan", 25, 0),
            new Car("Volvo XC40", "Sweden", 20, 2),
            new Boat("Markline 700 Sport", "New Zealand", 40, 2)};


    }
}
```

2. (a) Which two of the classes in 6.1 could legally call a method named test even if you removed the method definition from that class?

(b) Which one of the classes in 6.1 can not be instantiated?

(c) Which two classes' test method could legally contain the statement super.test() ?

(d) Which two classes must each legally contain a method called test?

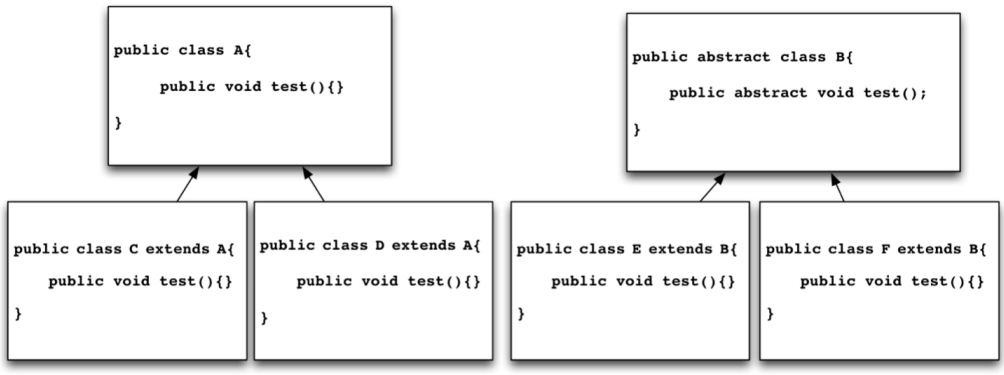


Figure 6.1: A class hierarchy with an abstract class

6.2 ** Lab Work

In this lab, we are going to explore some of the possibilities arising when our Shape class is made abstract, and our Shape can be something more than just a circle. With Shape as an abstract class we can have any number of classes which extend Shape, each of which can represent a different sort of Shape (e.g. a square, an oval, or a more complicated image).

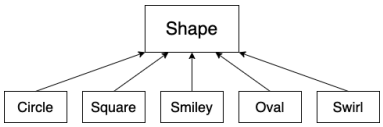


Figure 6.2: Planned Shape hierarchy

6.2.1 Shape to abstract

We can quickly adapt the finished Lab 5 code to the abstract format.

To divide the Shape class into its abstract and non-abstract components, we need to think about the things all Shape objects will want. Since all Shape objects will want to be drawn somewhere, and move, they will all want the data fields x, y, width, height, shade, moveX and moveY.
The constructor method must remain with the abstract class. The move method is also common to all Shape objects, and can stay with the abstract class.

The method which needs to be different for each of our yet-to-be-written classes is the display method. At present, it draws a circle.

1. Make a copy of your Lab05 folder and rename it Lab06. Open the files in JGrasp.
2. Make Shape an abstract class. It can not now be instantiated.
3. Make a new class called Circle which extends Shape and put it in the shapes package. This class will also need to import java.awt.* in order to easily access Graphics methods and Color. (Remember: Importing is not an inherited characteristic - it just saves you from typing the full name of the class each time you use it e.g. java.util.Scanner scan = new java.util.Scanner(System.in))
4. Copy the existing display method from Shape into the new Circle class.
5. Turn the display method of Shape into an abstract method (with no method body). This forces all classes that extend Shape to supply a display method.

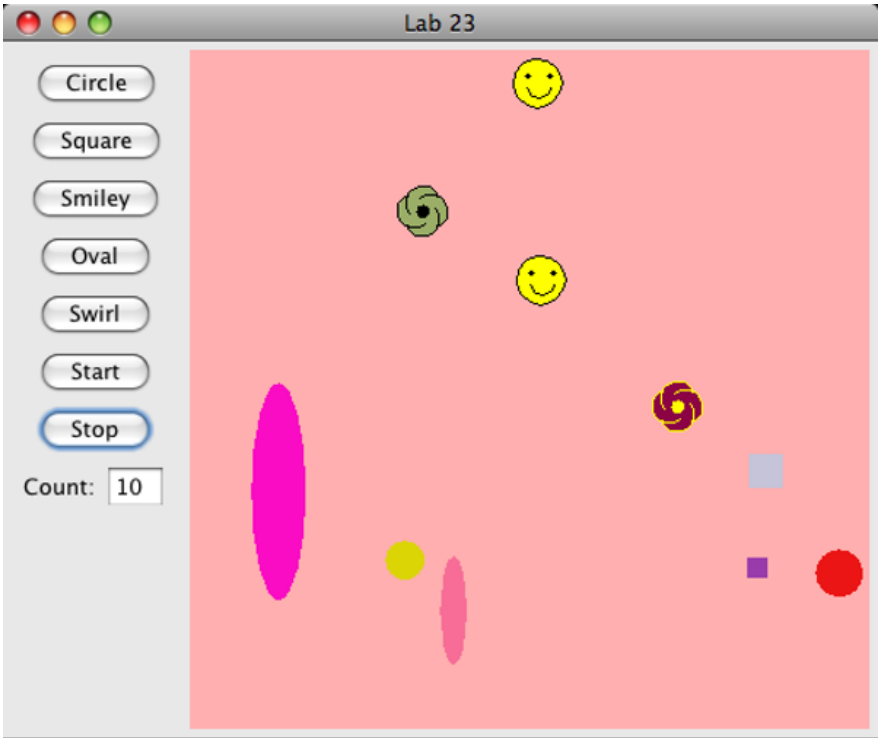


Figure 6.3: Shape window after Lab 5

6. Compile. If your data fields are private, you will get an error message. Circle needs to access the inherited data fields of Shape (x, y, height etc.), so they will need to have protected rather than private visibility. (This would show as # visibility rather than + or – on UML diagrams)

7. Compile. Another error message: the abstract class Shape cannot be instantiated. In ShapePanel, instead of adding a new Shape when the "Add Shape" button is pressed, add a new Circle. There is no need to change the ArrayList declaration - it still holds references to Shape objects. Each Circle is also a Shape because of the type hierarchy. Neat!

8. Compile, fix any other errors, and run. Now you have code which will exhibit exactly the same behaviour as before, but the scene has been set for growth.

9. Write another class called Square which also extends Shape. Write a display method in Square which draws a square rather than a circle. This is all very well, but you haven't got the button structure set up yet to be able to draw a Square.

6.3 ** Array of JButtons

It's time to set up a few extra buttons on the controlPanel, so that you can draw your Square as well as some other shapes when you have written some more classes. There will be at least 7 buttons (see Figure 6.3 on the previous page), so let's consolidate the code by dealing with them in an array. Repeated tasks can be performed using a loop.

1. In ShapePanel, use an initialiser list to declare and fill an array of JButtons that will hold 7 new instances of JButton labelled Circle, Square, Oval, Smiley, Swirl, Start and Stop. This array is a data

field: private JButton [] buttons = new JButton(“Circle”), new JButton(“Square”),. . .;

2. In the ShapePanel constructor, write a foreach loop which accesses each element in the array and
   - adds a listener
   - adds the button to the control panel.
3. Remove all references to the addShape, start and stop JButtons from the constructor.

Where the actionPerformed method before was using (hopefully) meaningful names for the buttons, it would now need to be referring to something like buttons[0], buttons[1] etc. Matching the right element of the button array to the right event will be prone to error. A neater solution would be to access the (meaningful) text on the button itself. The timer (which is not a button) needs to be dealt with separately.

1. Make the if . . timer block the first block in the actionPerformed method.
2. Follow it with an else block which includes all the remaining code in the method EXCEPT the call to repaint(). This else block will deal just with button presses, so in it you can safely cast the object source into a JButton variable.
   `JButton button = (JButton) e.getSource();`
   Now each if statement for which button was pressed can access the text on the button itself e.g.
   `if(button.getText().equals("Circle")){`
   It is now much easier to match the event to the right process than it would have been with the code
   `if(e.getSource() == buttons[0]){`
3. In the actionPerformed method, alter the Circle, Start and Stop buttons to the pattern described above. Add the required number of if..else statements to cope with all the new buttons. Two of the classes to be called have been written already (Circle and Square), so test your code now using these two buttons. We need to make more classes to extend Shape . . .

6.4 ** More Shapes

1. Class Oval will look very similar to Circle, except it needs a constructor which sets height to 4 * width. This will muck up the y location, which may draw the oval initially below the bottom edge of the panel, so the Oval constructor should also calculate a new random value for y using the new value of height. Note: Oval can use the ”random” method of its parent (Shape) class because the method has public visibility.
2. To make Smiley easier to draw, it is always going to be a standard size - 30 by 30 pixels. A Smiley constructor can set height and width to 30, and recalculate both x and y.
3. In Smiley’ s display method, draw a filled yellow circle, an unfilled black circle, 2 black eyes and an arc mouth. To save you time try these parameters:
   - draw the left eye at x + 7, y + 8, 4pixels across.
   - draw the right eye at x + 20, y + 8
   - try the arc at x + 8, y + 10, 15, 13, 190, 160
4. A file called Swirl.java is on Blackboard (along with many others made by students over the years.)
5. Link all your button presses to calls to the correct class constructors, and off you go. Your various shapes (Circle, Square, Smiley, Oval, Swirl) can look different, but are behaving in the same way because they are all extensions of the Shape class.

If you want to be really clever, see if you can make Smiley smile on the way up and frown on the way down.
If you have still got some size changing going on from Lab 22, this may not be appropriate for Swirl and Smiley. There are three ways you could deal with this:

- remove it
- make move an abstract method in Shape and have different versions of it in each child class, or
- in the Shape class, use the expression
  `if (!(this instanceof Swirl) && !(this instanceof Smiley)){ //leave out` smiley & swirl from the size changing

Lesson 7

Interfaces

An interface is not a class, but a set of requirements. We use them to define a role that other classes can play, regardless of where in the inheritance tree, or in which tree, they exist.

For this exercise we will need a simple class that has an array of int as a data field and a couple of constructors and a toString method.

ArrayFun
- integers:int[]
+ ArrayFun() + ArrayFun(lowLength:int, highLength:int, lowNum:int, highNum:int) + getArrayLength(): int + toString():String

The default constructor should initialise the array's length to be a randomly generated number between 10 and 100. It should then initialise each element to be a randomly generated number between 0 and the length of the array.

Remember: ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

7.1 ** A Simple Interface

Write an interface called **`ArrayMethodsInterface`** that specifies the following methods:

1. A method called **`getNumElements`** that returns the number of elements in the array as an int.
2. A method called **`getArrayCopy`** that returns a *copy* of the array.
3. A method called **`getArrayTotal`** that returns the sum of the elements in the array as an int.
4. A method called **`getArrayAverage`** that returns the average of the array elements as a double.
5. A method called **`getArrayReversed`** that returns a *copy* of the array in reverse order.
6. A method called **`getSmallestElement`** that returns the smallest number in the array as an int.
7. A method called **`getLargestElement`** that returns the largest number in the array as an int.

8. A method called **`getLargestElementIndex`** that returns the index of the largest number in the array as an int.
9. A method called **`getSmallestElementIndex`** that returns the index of the smallest number in the array as an int.

Get a demonstrator to check your interface before you continue.

7.1.1 Implement your interface

Now that you have got a complete interface modify **`ArrayFun`** so that it implements the ArrayFunInterface. Remember you will need to implement all the methods.

7.1.2 Write an Application Class

Once you have all the methods implemented create an application class **`ArrayFunApp`** that creates an three instances of the ArrayFun class and tests the methods.

7.2 ** Implement the Comparable Interface

Add to your **`ArrayFun`** class so that it also implements the **`Comparable`** interface Java API Comparable This interface doesn't tell you how objects should be ordered, that is up to you. List below four ways (at least) that the arrays could be ordered:

1. -
2. -
3. -
4. -

Get a demonstrator to check your interface before you continue.

Lesson 8

Algorithms

Arrays are *the* fundamental data structure of many programming languages including Java. A clear, complete and deep understanding of their strengths and weaknesses is a prerequisite to becoming an effective programmer.

Back in Lecture 2, you were introduced to an algorithm for bubble sort. The algorithm was expressed using pseudocode but first the algorithm was explored and represented visually with diagrams of the different states of the array.

Your task in this lab is to devise algorithms to achieve the tasks described below and then code them. Many of them involve arrays, others may use ArrayLists.

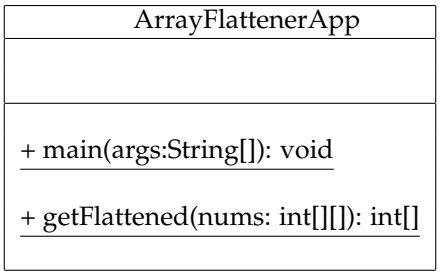
You are welcome to ask demonstrators for help but they will first want to see evidence of your planning - this may involve sketches, pseudocode or possibly interpretive dance. **Remember:** ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

8.1 Before you begin to code, remember:

- It is often useful to start with a sketch.
- It is often useful to start by solving the problem manually, perhaps using pen and paper to keep track of what you need to know at each stage.
- Sensible identifiers save a world of pain.
- Talk with a demonstrator if you need to clarify the task.
- Demonstrators will want to see evidence of planning and discuss your pseudocode before any help can be given with your java code.

8.2 ** Flatten a 2-dimensional Array

Write a class called ArrayFlattenerApp according to the UML diagram below:



8.2.1 In main:

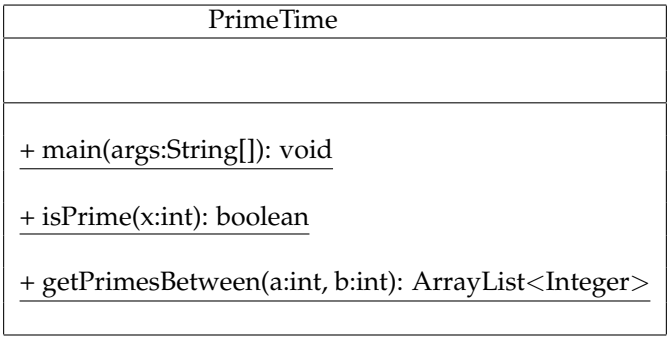
1. Create and populate a 2-dimensional array of ints by reading in numbers from a file. You have been provided with numbers1.txt and numbers2.txt on Blackboard.
   (a) You can be sure that the file contains only integers which are separated by spaces and span multiple lines.
   (b) Each line of numbers should form a complete inner array.
2. Print the contents of the 2-dimensional array that has been created from the file.
3. Call the getFlattened method, passing in a reference to the 2-dimensional array you created.
4. Print out the contents of the returned array.

8.2.2 In getFlattened:

- Create and return a 1-dimensional array that contains all the ints from the parameter array in the same order.

8.3 ** Prime Numbers

Remember: a prime number is a number that is divisible only by itself and 1. (1 is not a prime number.)
Write a class called PrimeTime according to the UML diagram below:



8.3.1 In main:

- Test your isPrime and getPrimesBetween methods thoroughly.

8.3.2 In isPrime:

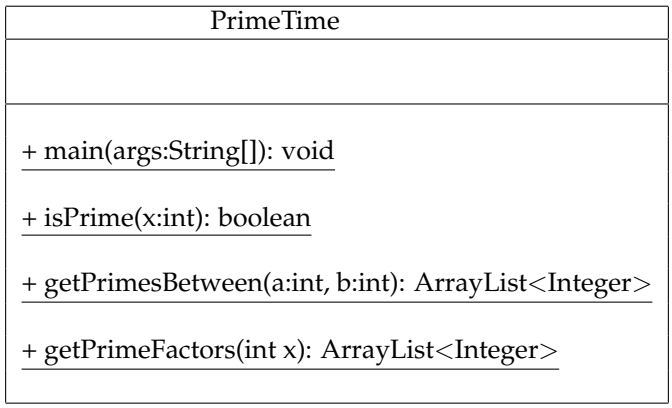
- Return true if the parameter is a prime number and false if it isn't.

8.3.3 In getPrimesBetween:

- Return an ArrayList containing all the Integer objects with values between a and b inclusive which are prime.

8.4 Prime Factors:

Add to your PrimeTime class from the previous task according to the UML diagram below:



8.4.1 In getPrimeFactors:

- Return an ArrayList containing all the factors of x which are prime.
  E.g If the number 12 is passed in, the method should return an ArrayList containing 2, 2, 3. If the number 70 is passed in, the method should return an ArrayList containing 2, 5, 7.

8.5 Square-free numbers:

From Wikipedia, the free encyclopedia

> In mathematics, a square-free integer (or square-free integer) is an integer which is divisible by no perfect square other than 1. That is, its prime factorization has exactly one factor for each prime that appears in it.
>
> For example, 10 = 2 * 5 is square-free, but 18 = 2 * 3 * 3 is not, because 18 is divisible by 9 = 32.
>
> The smallest positive square-free numbers are
>
> 1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, 19, 21, 22, 23, 26, 29, 30, 31, 33, 34, 35, 37, 38, 39, ...

Add a method to **PrimeTime** called **getSquareFreeNums** which takes an int, x, and returns all the square-free numbers between 1 and x inclusive.

8.5.1 Remember:

- It is often useful to break a problem down and write smaller methods that solve aspects of the bigger problem (**isPrime** for example.)

8.6 Memory card Game

You have been provided with working code for a GUI that provides the basis for a memory game, traditionally played with deck of cards. The idea is that the cards are shuffled and laid face down. Each player takes turns to turn 2 cards face up. If they are a matching pair the player takes them. The player with the most pairs when the game ends is the winner.

Your task is to develop your own algorithm to 'shuffle' the cards and then use it to write an actionPerformed method that rearranges the cards in response to the shuffle button. (No sneaky using the shuffle method from the Collections class!)

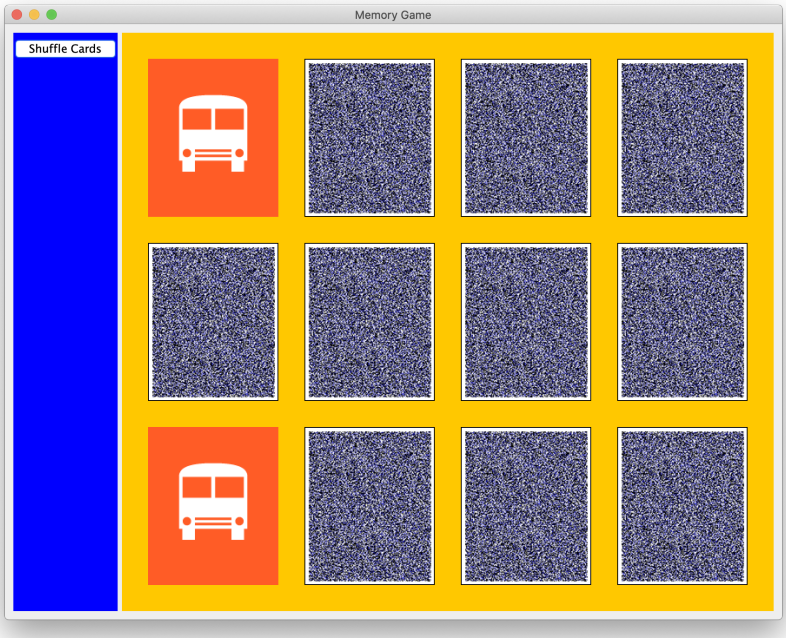
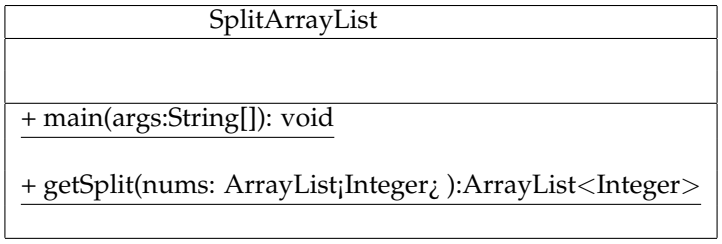


Figure 8.1: The 'game' when two matching cards have been found

8.7 ** Split an ArrayList

Write a class called SplitArrayList according to the UML diagram below:



8.7.1 In main:

- Instantiate an ArrayList of Integer objects
- Add between 20 and 30 Integer instances – the actual number should be randomly generated. Each of these values should be randomly generated betwen 1 and 100.

8.7.2 In getSplit

- Return a reference to a new ArrayList which contains all the numbers from the parameter, **nums**, except that any even numbers are now at the beginning followed by the odd numbers.
- Apart from the split between evens and odds the numbers should all occur in the same order that they do in **nums**.

8.7.3 Challenges

- Achieve the 'split' ArrayList with only one iteration of nums.
- Adapt **getSplit** so that it also takes an int parameter – instead of evens and odds, multiples of the parameter go to the beginning of the returned ArrayList followed by non-multiples.

8.7.4 Remember:

- Autoboxing!

8.8 ** Transpose a Matrix

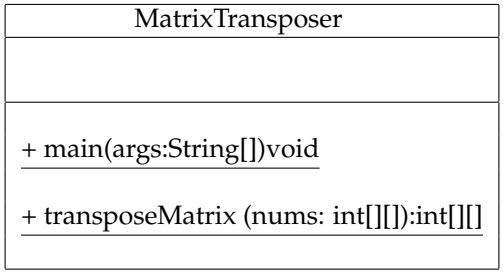
From Wikipedia, the free encyclopedia

> In mathematics, a matrix (plural matrices) is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns

Also:

> In linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by AT .

We typically represent a matrix as a 2-dimensional array. Write a class called MatrixTransposer according the UML below:



8.8.1 In main:

1. Create and populate a 2-dimensional array of random ints
   (a) The length of the array should be a random number between 3 and 7
   (b) The length of the inner arrays should also be a random number between 3 and 7. (They are all the same length.)
   (c) The array should be filled by numbers between 1 and 50 inclusive.
2. Print the contents of the 2-dimensional array that has been created.
3. Call the transpose method, passing in a reference to the 2-dimensional array you created, printing out the contents of the returned array.

8.8.2 In transpose:

- Create and return an array in which is the transposition of **nums**.

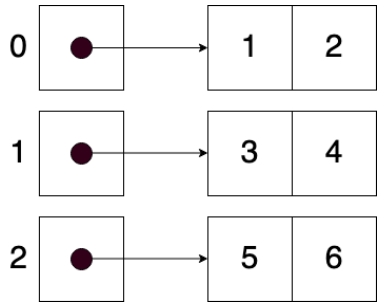


Figure 8.2: The 'matrix' before it is transposed

8.9 More Transposition

A ragged array cannot be referred to as a matrix but write a method to transpose one anyway.

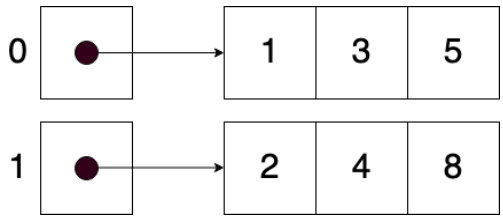
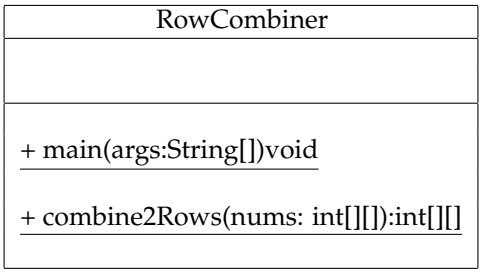


Figure 8.3: The 'matrix' after it is transposed

8.10 ** Combine 2 rows

Write a class called RowCombiner according the UML below:



8.10.1 In main:

1. Create and populate a 2-dimensional array of ints by reading in numbers from a file. You have been provided with numbers1.txt and numbers2.txt on Blackboard.
   (a) You can be sure that the file contains only integers which are separated by spaces and span multiple lines.
   (b) Each line of numbers should form a complete inner array.
2. Print the contents of the 2-dimensional array that has been created from the file.
3. Call the combine2Rows method, passing in a reference to the 2-dimensional array you created, printing out the contents of the returned array.

8.10.2 In combine2Rows:

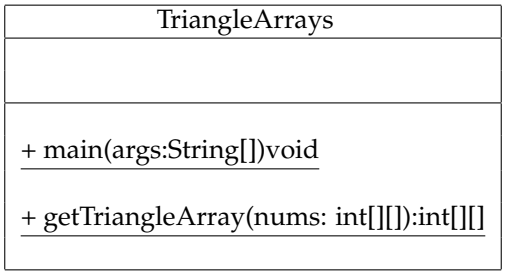
1. Create and return an array in which every 2 inner arrays (or 'rows') of nums has been combined to form one row. Row 0 of the returned array should consist of the contents of row 0 of nums followed by the contents of row 1 of nums and so on.
2. If the number of rows in the original array is odd, the final row of the returned array should just consist of that last row in nums.

8.11 Combine X Rows

In your RowCombiner class from above, write a method called combineXRows where the number of rows that are combined to form one is specified by an int parameter.

8.12 Triangular Arrays

Write a class called TriangleArrays according the UML below:



8.12.1 In main:

1. Create and populate a 2-dimensional array of random ints
   (a) The length of the array should be a random number between 3 and 7
   (b) The length of the inner arrays should also be a random number between 3 and 7. (The inner arrays are all the same length.)
   (c) The array should be filled by numbers between 1 and 50 inclusive.
2. Print the contents of the 2-dimensional array that has been created.
3. Call the getTriangleArray method, passing in a reference to the 2-dimensional array you created, printing out the contents of the returned array.

8.12.2 In getTriangleArray:

1. Create and return an array which contains all the numbers from nums in the same order. In the returned array, the length of each inner array should be one more than the previous one. E.g If the original array contains 3 rows of 4, the returned array will have a row of 1 followed by a row of 2 followed by a row of 3 and finally a row of 4.
2. The last row of the triangular array should be of the necessary length to complete the triangle. If there are not enough remaining elements from the original array to fill the last row, the default value should be used.

8.12.3 Extension:

Instead of creating a grid array in main create a ragged array and ensure that **getTriangleArray** is coded robustly enough to still work.

This section should to be shown to a demonstrator before you move on.

Lesson 9

Recursion

Recursion is one of the key concepts in computer science – the idea that methods (recursive functions) or data (recursive data structures) can be described in terms of themselves. The most important thing to remember is that these definitions must not be circular or they cannot be resolved. The definitions must always have:

- a simple base case or cases (non-recursive), and
- rules that unambiguously reduce all other cases towards the base case.

For instance, in mathematics, a function on the natural numbers might be defined in terms of a given value at 0 (the base case), and some rules for computing its value at a natural number n in terms of its values at smaller numbers. This would be more-or-less the opposite of the Countdown example in lecture 7.

In this lab you will first gain some experience with basic recursive functions, and then explore a simple recursive data structure. **Remember:** ** Any exercise marked with a "**" is required to be completed and marked as satisfactory by a demonstrator before you can sit your Mastery Test for this level.

9.1 ** Countdown

Copy the Countdown program from lectures and verify that it works.

```
public class Countdown {
    public static void main(String[] args) {
        countdown(3);
    }

    public static void countdown(int i) {
        if (i <= 0) {
            return;                              // Base case
        } else {
            System.out.println(i);
            countdown(--i);              // Recursive case -- Line 11
        }
    }
}
```

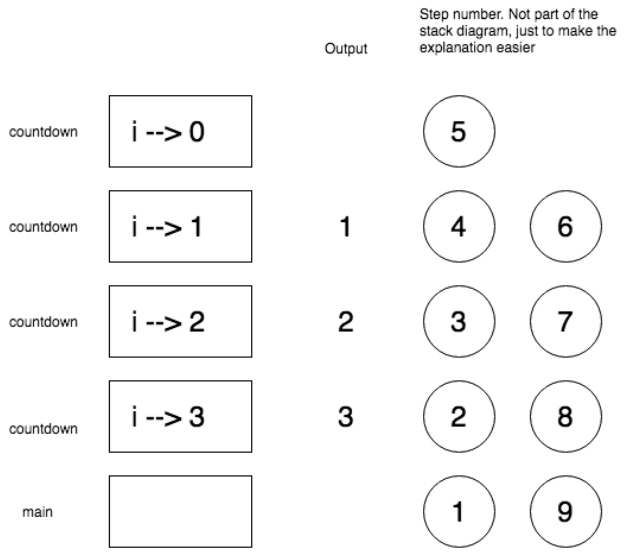


Figure 9.1: The Stack Diagram for Countdown.java

To understand how it works it may be easier to think of how Java performs these recursive calls. As noted in Lecture 7:
Whenever a method calls another one (or itself), its state is stored, execution is temporarily suspended, and the new method begins execution. But what does that mean?

Trace through the code and at the same time see how the stack diagram 9.1 relates to it. The numbers in the diagram relate to the listed description.

(1) The main method starts. The main method calls the countdown method with the value 3. This causes the execution of main to be suspended and a new frame to be placed on the stack.

(2) In this call to the countdown method the parameter **`i`** has a value of 3. The code in the countdown method is run. **`i`** is not equal to or less than 0 so the method executes the code in the else clause and prints out the value of **`i`** and then... calls the countdown method with **`i-1`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(3) In this call to the countdown method the parameter **`i`** has a value of 2. The code in the countdown method is run. **`i`** is not equal to or less than 0 so the method executes the code in the else clause and prints out the value of **`i`** and then... calls the countdown method with **`i-1`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(4) In this call to the countdown method the parameter **`i`** has a value of 1. The code in the countdown method is run. **`i`** is not equal to or less than 0 so the method executes the code in the else clause and prints out the value of **`i`** and then... calls the countdown method with **`i-1`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(5) In this call to the countdown method the parameter **`i`** has a value of 0. The code in the countdown method is run. **`i`** is not equal to or less than 0 so the method executes the code in the **`else`** clause and prints out the value of **`i`** and then... calls the countdown method with **`i-1`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(6) This method resumes where it left off. But if you recall this method was suspended at the line 11 **`(countdown(--i))`**. There isn't another statement after that so the method terminates and its frame

is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(7) This method resumes where it left off. But if you recall this method was suspended at the line 11 **`(countdown(--i))`**. There isn't another statement after that so the method terminates and its frame is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(8) This method resumes where it left off. But if you recall this method was suspended at the line 11 **`(countdown(--i))`**. There isn't another statement after that so the method terminates and its frame is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(9) This method (the main method) resumes where it left off. this method only had the one line, which has now been completed and so the program finishes.

9.1.1 Countdown - extended

Now you will modify it so that not only does it print out the countdown but it also specifies if each number is odd or even. The expected output for countdown(10) is:

```
10 is an Even Number
9 is an Odd Number
8 is an Even Number
7 is an Odd Number
6 is an Even Number
5 is an Odd Number
4 is an Even Number
3 is an Odd Number
2 is an Even Number
1 is an Odd Number
```

The base case remains the same, however the recursive case now has two options, one for if the number is odd and one for if it is even. So all you need is a selection statement **`if`** with the two cases.

(1) Add code to Countdown.java so that is produces the correct output.

9.2 ** Multiplier

In the Countdown exercises the recursive calls produced output as they were called via the **`println`** statements. In this exercise the recursive calls will not produce output as they are called but at the end as the stack undoes.

A recursive method to multiply two positive integers without using the **`*`** operator; **`mutliply(base, limit)`**.

```
public class Multiplier{
   public static void main(String[] args){
      System.out.println( multiply(5, 4));
      }

   public static int multiply(int base, int limit){
      if(limit == 1){
```

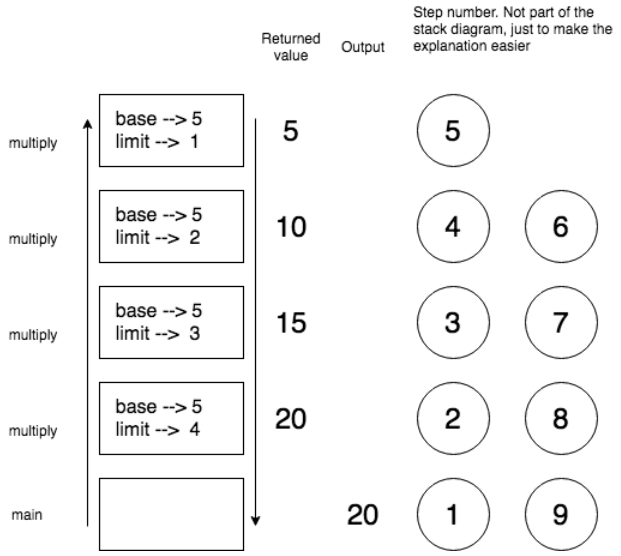


Figure 9.2: The Stack Diagram for Multiplier.java

```
            return base;
        }
        return base + multiply(base, limit-1); //line 10
    }
}
```

(1) The main method starts. The main method calls the **`multiply`** method with the values **`base=5, limit=4`**. This causes the execution of main to be suspended and a new frame to be placed on the stack.

(2) In this call to the multiply method the parameter **`base`** has a value of 5 and **`limit`** has the value 4. The code in the **`multiply`** method is run. **`limit`** is not equal to 1 so the method executes the code in the **`else`** clause and calls the **`multiply`** method with the values **`base=5, limit=3`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(3) In this call to the multiply method the parameter **`base`** has a value of 5 and **`limit`** has the value 3. The code in the **`multiply`** method is run. **`limit`** is not equal to 1 so the method executes the code in the **`else`** clause and calls the **`multiply`** method with the values **`base=5, limit=2`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(4) In this call to the multiply method the parameter **`base`** has a value of 5 and **`limit`** has the value 2. The code in the **`multiply`** method is run. **`limit`** is not equal to 1 so the method executes the code in the **`else`** clause and calls the **`multiply`** method with the values **`base=5, limit=1`**. This causes the execution of the current method to be paused, its state is saved and another frame is placed on the stack.

(5) In this call to the multiply method the parameter **`base`** has a value of 5 and **`limit`** has the value 1. The code in the **`multiply`** method is run. **`limit`** *is* equal to 1 so the method executes the code in the primary clause and this returns the value of **`base`** (which is 5) and then terminates. The frame is removed from the stack and control is passed to the calling method.

(6) This method resumes where it left off. But if you recall this method was suspended at the line 10 **`return base + multiply(base, limit-1)`** and so it returns 5+ the value it received from the previous method (which was 5). There isn't another statement after that so the method terminates and its frame is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(7) This method resumes where it left off. But if you recall this method was suspended at the line 10 **`return base + multiply(base, limit-1)`** and so it returns 5+ the value it received from the previous method (which was 10). There isn't another statement after that so the method terminates and its frame is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(8) This method resumes where it left off. But if you recall this method was suspended at the line 10 **`return base + multiply(base, limit-1)`** and so it returns 5+ the value it received from the previous method (which was 15). There isn't another statement after that so the method terminates and its frame is removed from the stack and the calling method resumes at whatever point it was at when it was suspended.

(9) This method (the main method) resumes where it left off. The method call in the **`println`** statement is replaced with the resolved value (20) and this is printed. There are no more statements so the **`main`** method terminates and the program is complete.

9.3 ** Practice Problems 1%

There are plenty of practice problems at https://codingbat.com/java/Recursion-1. To get your 1% for this lab you should complete at least 6 of the problems and discuss your solutions with a demonstrator.

Lesson 10

Catch-up

No new material in this lab.

Lesson 11

Arrays - Young tableaux

In this lab we will work with some special objects called Young tableaux, which are naturally represented as doubly indexed arrays of integers. These are very important objects in advanced algebra and discrete mathematics, but that will not concern us – we're just using the concept as a convenient foundation for some exercises on array manipulation.

A *Young tableau* is shown below:

1	4	5	10	11
2	6	8		
3	9	12		
7				

The key features of a Young tableau are as follows:

(1) it consists of cells which are filled with integers, and arranged in left-justified rows,

(2) no row is longer than a preceding row,

(3) from left to right in *any* row, and down *any* column the integers are increasing,

(4) the set of integers used is $\{1, 2, \ldots, n\}$ where n is the number of cells and every number is used exactly once, i.e. there are no missing or duplicate numbers.

In Java, an obvious way to represent a tableau is as a doubly indexed array of integers, i.e, `int[][]`. The tableau above might be represented as:

```
int[][] tableau = {{1, 4, 5, 10, 11}, {2, 6, 8}, {3, 9, 12}, {7}};
```

A basic problem is that doubly indexed arrays of integers need have none of the properties that define tableaux. One of the objects of this lab is to write code that will check when such an array really is a tableau.

11.1 Problem description

The main objective of this lab is to write a function `isTableau` which takes an `int[][]` as input and returns a `boolean` as output, where the result is `true` if the array represents a tableau, and `false` otherwise. Because the defining properties of a tableau separate naturally into various individual properties, this is a good opportunity to practice bottom up design, and some form of test driven development – building and testing your code a piece at a time.

The conditions that a tableau must satisfy are listed above. The first is represented automatically by the representation as a doubly indexed array of integers, so the main issue is to test the remaining three conditions, noting that the third of the four conditions splits into two parts.

11.2 Part one

There is a skeleton class `TableauApp.java` provided in the Lab Files directory on Blackboard. This includes a basic utility function for `String` representations of doubly indexed arrays of integers as if they were tableaux. There is also a file called `YoungTableauxTestData.java` that contains some test data for you to check your methods against.

Add a couple of static functions to the skeleton code class which implement condition two and the first part of condition three:

`rowLengthsDecrease(int[][] t)` A method that returns `true` if no row is longer than a preceding row, otherwise `false`.

`rowValuesIncrease(int[][] t)` A method that returns `true` if from left to right in any row, the integers are increasing, otherwise `false`.

Implement each one separately, testing as you go. Your methods need not work sensibly if passed a **`null`** argument, but should work properly on the empty tableau, i.e.,

```
int[][] t = {};
```

which mathematicians insist is a proper tableau (with $n = 0$).

This section should to be shown to a demonstrator before you move on.

11.3 Part two

Add two more static functions to the skeleton code class which implement the second part of condition three, and condition four:

`columnValuesIncrease(int[][] t)` A method that returns `true` if from top to bottom in any column, the integers are increasing, otherwise `false`.

`isSetOf1toN(int[][] t)` A method that returns `true` if the set of integers used is $\{1, 2, \ldots, n\}$ where n is the number of cells, otherwise `false`. So, for example, if there are 10 cells the numbers would be $1, 2, \ldots, 10$ with no missing or duplicate numbers.

Finally, use the separate methods above to complete the **`isTableau`** method for determining whether an **`int[][]`** represents a tableau.

This section should to be shown to a demonstrator before you move on.

11.4 Reflection and extension

- What is the complexity of your method for checking that the entries of the tableau form the set $\{1, 2, \ldots, n\}$? One natural implementation is of quadratic (i.e., $O(n^2)$) but not linear ($O(n)$) complexity. Can you think of a linear one – or convince yourself that yours is?
- How could the `tableauToString` function be improved so that the result is nicely aligned even if some of the numbers are quite large? That is, it should dynamically take into account the size of the entries in the doubly indexed array.
- Your mathematician friend who wants to work with tableaux only now gets around to telling you that the tableaux she is interested in are generated by a sequence of additions of cells. Her plan is to test some conjectures about tableaux which will involve building up millions or possibly billions of tableaux in this way. Why is the tableau representation we have been using here *not* a good idea for this project?
- If you look at a tableau on its side (i.e., read each column from top to bottom as a row from left to right) you get another tableau called its *conjugate*. How would you write a method that, given a tableau, returned the conjugate tableau?

Lesson 12

Linked List Introduction

A linked list is a dynamic data structure. We can change the structure of a linked list by changing where links point to. Linked list are particularly useful in situations where we want to insert (or delete) items other than to the end of a list. In the Ant Traveler example earlier an Array made sense as a data structure because we knew in advance how many elements each list would contain and we never needed to insert new elements anywhere other than to the end (tail) of the list. In Java if you want to add a new element to a list it is quite a expensive operation computationally because an Array is of a fixed size. So to add a new element (to an Array) you need to actually create a whole new Array and completely re-build it.

Linked lists are composed of *nodes*. Each node is an object that has some data fields. In a singly linked list one of these data fields will hold a reference to another node, the next node in the list.

12.1 An Address Book as a Singly Linked List

In this exercise you will create a linked list that keeps track of an address book. Each node of the collection will contain a contact's first name, last name, phone number and email address. Other information (address, other phone number etc.) are omitted for brevity.

12.1.1 The AddressBookEntry class

1. Begin by writing and testing the AddressBookEntry class, the UML is below. Developing the application class will be much easier if this class works properly. :-)

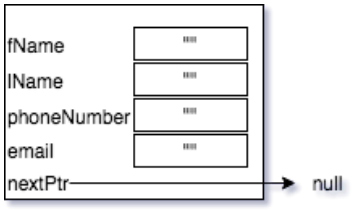


Figure 12.1: A single AddressBook node with default values.

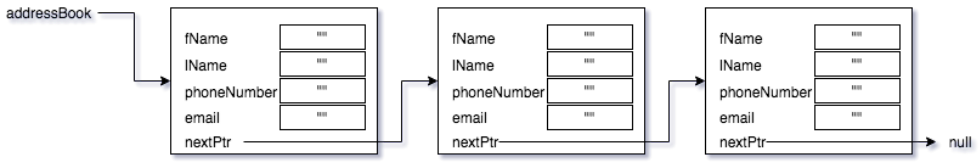


Figure 12.2: An AddressBook with three enteries

AddressBookEntry
- fName:String - lName:String - phoneNumber:String - email: String + nextPtr: AddressBookEntry
+ AddressBookEntry() + AddressBookEntry(fName:String, lName:String, phoneNumber:String, address: String) + setFName(fName:String):void + setLName(lName:String):void + setPhoneNumber(phoneNumber:String):void + setEmail(email:String):void + getFName():String + getLName():String + getPhoneNumber():String + getEmail():String + toString():String

This section should to be shown to a demonstrator before you move on.

12.1.2 The Address Book Application class

We will also need an application class to manage the list, the UML for that is shown below.

AddressBookApp
- addressBook:AddressBookEntry
+ addNewContact(contact:AddressBookEntry):void + removeContactByName(fName:String, lName:String):void + getContactByLName(lName:String):AddressBookEntry + getContactByPhoneNumber(phoneNumber:String):AddressBookEntry + toString():String

Now to start implementing the methods in the application class.

The `addNewContact` method needs to:

1. make a new instance of the `AddressBookEntry` class and
2. add it to the list.

To begin with you can just add the entry to the end of the list.

The `removeContactByName` method needs to:

1. search the list for the correct entry and
2. remove it from the list.

You need to think about how you are going to add the new entries. For the time being you can just add them to the front (head) of the list.

The `getContactByLName` method needs to:

1. search the list for the correct entry and
2. return the appropriate `AddressBookEntry`.

The `getContactByPhoneNumber` method needs to:

1. search the list for the correct entry and
2. return the appropriate`AddressBookEntry`.

The `toString` method needs to:

1. return a String representation of the entire AddressBook

This section should to be shown to a demonstrator before you move on.

12.1.3 Making the Address Book Store Contact Alphabetically

Adding new contacts to the beginning of the list isn't how address books are usually organised.

1. Modify `addNewContact` so that the addresses are organised by `lName`

Currently the `nextPtr` data field in the `AddressBookEntry` class needs to be `public` so that the methods in the `AddressBookApp` class can access it. This is far from ideal and is one of the reasons a private inner class was used in the lecture example.

1. Modify the `AddressBookEntry` class so that it is a private inner class of the `AddressBookApp` class.

This section should to be shown to a demonstrator before you move on.

12.2 A Bus Route Information System Implemented as a Doubly Linked List

In this section you will implement a doubly linked list that represents a bus route. Each node represents a single bus stop. It has an id, and information about how long it take the bus to get to the next stop East and West; all busses run from West to East and then back again. In addition there are pointer data fields that point to the next node to the East and West. Below is a single BusStop node 12.3 and and (very) short bus route, 12.4.

In the Lab Files section of Blackboard you will find `BusRoute.java` and `BusRoute.java`. You should complete the bodies of the empty methods.

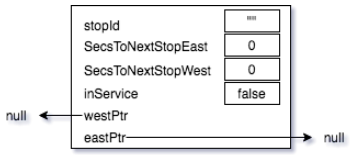


Figure 12.3: A single BusStop node with default values.

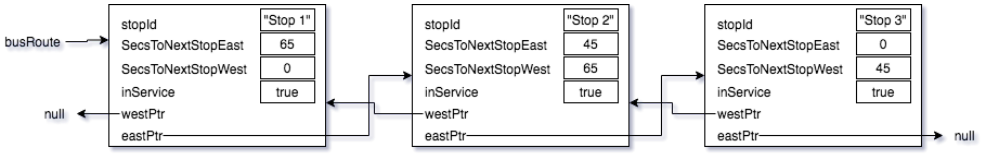


Figure 12.4: A short bus route.

This section should to be shown to a demonstrator before you move on.

Lesson 13

Linked Lists

In this lab we are going to be revisiting the Young tableaux that we looked at a few labs ago.

1	4	5	10	11
2	6	8		
3	9	12		
7				

See 13.2 for a link list version of the same tableau.

The key features of a Young tableau are as follows:

- it consists of cells which are filled with integers, and arranged in left-justified rows,
- no row is longer than a preceding row,
- from left to right in any row, and down any column the integers are increasing,
- the set of integers used is $\{1, 2, \ldots, n\}$ where n is the number of cells.

Previously we represented tableaux using `int[][]`. A disadvantage of that approach is that arrays are static data structures and so it was not convenient to represent tableaux that changed, or to make structural modifications to a tableau. In this lab we'll use a different linked data structure that provides us with the dynamic behaviour required for such operations.

At first glance you might think that items could be added to a tableau using something like **`addValue(x, y, value)`** where x and y are the indexes of where you want the *value* to go. However, your mathematician friend comes back and says "That's not really how I want to add values to tableaux." She goes on to explain that a tableau is usually built up from a sequence of positive integers (frequently a permutation) by a process called *bumping*. This works as follows: to add a new value, v, to a tableau we first look in the first row of the tableau. If v can be added to the end of the row (because it's at least as big as anything in the row), then we do so and are done. Otherwise, we find the first element of the row, say w, that is bigger than v. We replace the cell containing w by one containing v ("v bumps w out of the row") and then proceed to try and insert w into the second row by the same means. We always add to the end of a row if we can (in which case we're done), or bump something into the next row. If absolutely necessary, we can add an extra row containing the final bumped element.

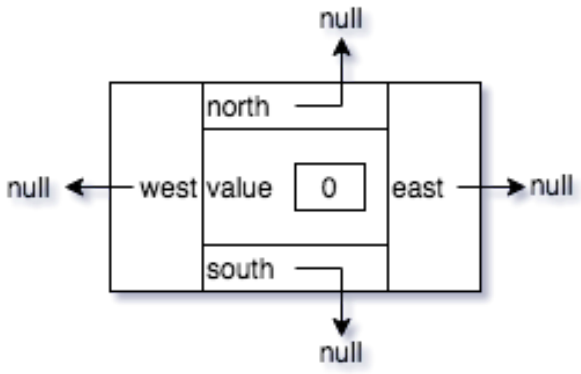


Figure 13.1: A single Cell initialised with default values.

13.1 Problem description

Your task is to complete a `Tableau` class which implements the bumping algorithm described above. There will be no need for a global check like the `isTableau` method of Lab 10 because you will ensure that all the methods that actually change the structure of a tableau never create a structure that is not a tableau (e.g., by having cells in non-increasing order, or by having later rows longer than earlier ones). You can assume that the numbers given as input to your program will be a valid set of integers and don't need to perform any checking.

The basic idea is that a `Tableau` will be built up of objects of type `Cell`, see 13.1. (`Cell` will be an inner class of `Tableau`). A `Cell` contains an `int` which is its value, together with references to four other `Cell` objects, its north, east, south, and west neighbours in the tableau (some of these might be `null`). Ensuring consistency among these references (if `a.west` is a cell `b` (not `null`) then `b.east` has to be `a` etc.) is one of the key things to take care of – and the way to do this is to make sure that all modifications of neighbour references are handled by methods that maintain the consistency.

13.2 Task

A skeleton (but compilable) version of the `Tableau` class is provided in Lab Files on Blackboard. You need to complete the methods (testing as you go of course) that remain to be implemented.

`addToRow(Cell curr, int value)` This method takes a **`cell`** as a starting point and follows *east* pointing links until it finds a value which is greater than the given value or until the east pointing link is **`null`**. If it finds a bigger value then it replaces it with the given value and returns the previous value. If it comes to the end of the row it adds a new cell with the given value and returns **`null`**.

`addValue(Integer value)` This method takes care of the case where the tableau is empty, you must complete the implementation by adding the other cases. You can call the **`addToRow`** method to add the value to the first row. If **`addToRow`** returns **`null`** there is nothing more to do. If it returns a value then that value must be inserted into the row below. If the row below is empty then a new cell should be added as the only item in that row, otherwise just call **`addToRow`** again to add the returned value to the row below.

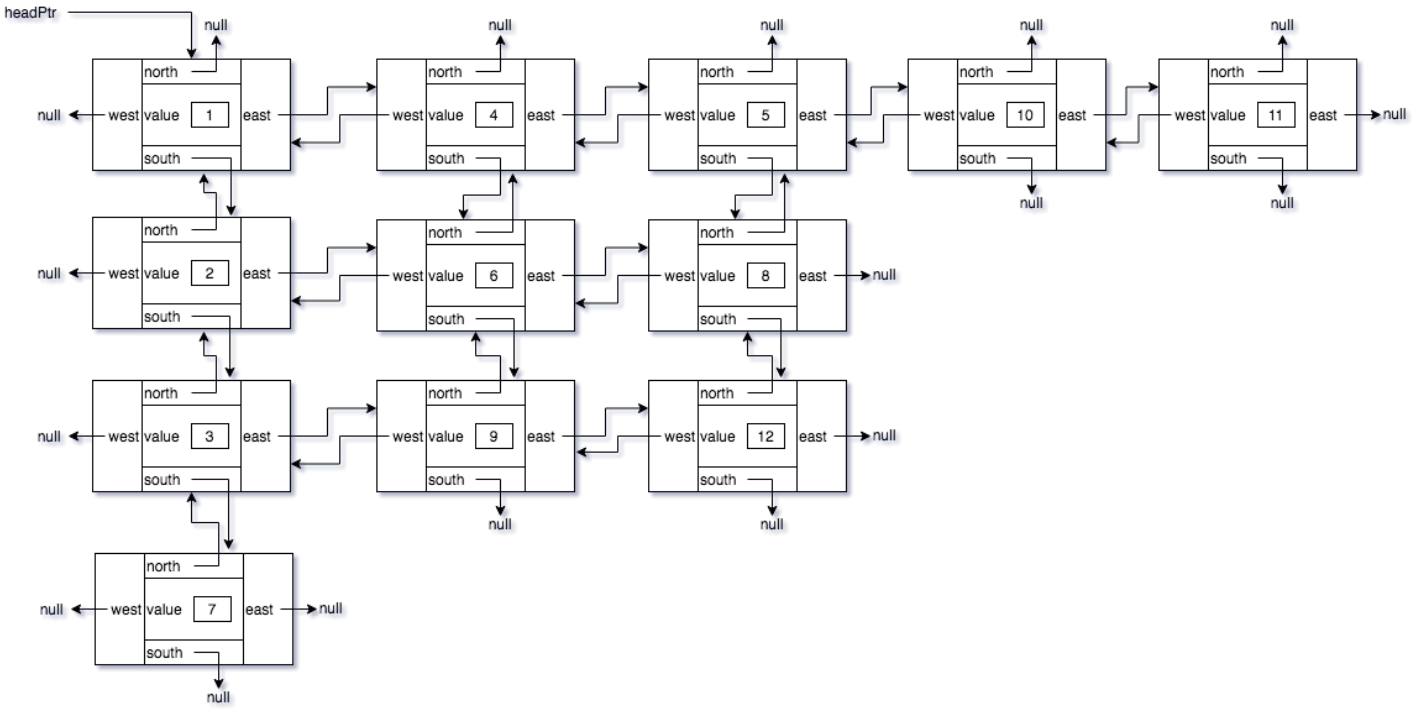


Figure 13.2: A representation of a Young Tableau as a linked list.

13.3 Reflection and extension

- Remember that in the conjugate of a tableau, rows become columns, and columns become rows. How could you convert a tableau into its own conjugate? Some care is necessary ...
- If you were to add a *delete* operation to your tableau, how could you 'fix up' the tableau after performing a deletion so that the first three properties are maintained?

Lesson 14

Stacks and Reverse Polish notation

We are all familiar with the concept of a stack; we create stacks by placing one thing on top of another. Stacks are defined by two key features:

1. Items can only be added to the top of the stack.
2. Items can only be removed from the top of the stack.

Stacks are often described as FILO (First In Last Out) structures and in that respect are in contrast to queues which are FIFO (First In First Out). Stacks can only be accessed from one point, the top of the stack, this again is in contrast to queues in which items are added to one end of the queue (the tail) and removed from the other end (the head of the queue).

14.1 Implement a stack class

Implement a stack - what data structure should you use? Array, ArrayList, LinkedList? + Stack() + push() + pop() + peek() + empty() + search()

14.2 Using a Stack to implement a RPN calculator

Reverse Polish notation is a method of mathematical notation that precludes the need for parentheses. See Wikipedia for a more detailed explanation.

Lesson 15

Catch-up

Catch-up lab. No new material.

Lesson 16

Queues

In this lab we are going to be using one data structure (a stack) to implement a different data structure (a queue).

The basic idea is that a queue can be represented by a *pair of stacks* called (say) **left** and **right**. To insert into the queue they represent, you push onto the left stack. To remove from the queue, you pop from the right stack *unless the right stack is empty* in which case you first transfer, by pop and push operations, the entire left stack into the right stack.

You'll need to think for a moment before you are convinced that this method does produce the First-In-First-Out behaviour which characterises a queue.

16.1 ** What you need to do

When you have done this, create a class called `TwoStackQueue` which implements the queue interface and uses the technique described above to do its work. You don't need to implement your own stack but can use the `java.util.Stack` class. If you want to save yourself a bit of typing you can start your `TwoStackQueue.java` from a copy of `Queue.java` since it will have all of those method declarations.

You should create an application class called `QueueApp` which reads lines of input from *stdin* and handles it based on the choices given in the table below. Your application class should check that the queue is not empty before calling any methods which could throw an `EmptyQueueException`. When you implement your queue you need to include a `debug()` method which concatenates the results of calling the `toString()` methods of your left and right stacks, and returns that string. This will verify that your stacks are being used correctly.

Input	Action
`a` item, item...	*Add the given item(s) to the back of your queue (in left to right order)*
`c`	*Clear the queue*
`d`	*Print out the result of calling the `debug()` method*
`g`	*Get and print the item at the front of the queue (if there is one)*
`p`	*Print out the result of calling the `toString()` method*
`r`	*Remove (and print) the item from the front of the queue (if there is one)*
`s`	*Print out the size of the queue*

When printing out your queue with the `toString` method the items should be printed out in order, from front to back of the queue, as a comma separated list surrounded by brackets like this: `[1, 2, 3, 4, 5, 6]`

When printing out your queue with the `debug` method the two stacks which make up your queue should be printed in order, from bottom to top of each stack, as two comma separated lists surrounded by brackets like this: `[4, 5, 6][3, 2, 1]`

You can easily do this by just concatenating the two stacks:

```
left.toString() + right.toString()
```

16.2 Example

Some sample input to your program could be:

```
a one two 3
d
p
r
d
a 4 5 6
d
p
```

While the corresponding output would be:

```
[one, two, 3][]
[one, two, 3]
one
[][3, two]
[4, 5, 6][3, two]
[two, 3, 4, 5, 6]
```

This section should to be shown to a demonstrator before you move on.

Lesson 17

Sorting with Bucket Sort

See Bucket (Bin) Sort

Bucket sort works by distributing the values to a number of buckets. The basic algorithm for bucket sort is:

1. START
2. Set up an array of initially empty "buckets".
3. Scatter: Go over the original array, putting each object in its bucket.
4. Sort each non-empty bucket.
5. Gather: Visit the buckets in order and put all elements back into the original array.
6. STOP

See 17.1 for an example of bucket sort where the numbers have been distributed to the buckets, but the buckets haven't themselves been sorted. In the example the numbers to be sorted fall between 0 and 24, inclusive.

There are a couple of points worth noting about this example:

1. The bucket boundaries were chosen to be equally sized. Buckets[0] contains values between 0-4 inclusive, Buckets[1] contains values between 5-9 inclusive, etc. There are better ways to choose the bucket boundaries.
2. The method used to sort the buckets (Step 4 of the algorithm above) can be any sorting method including bucket sort.

This section should to be shown to a demonstrator before you move on.

17.0.1 Implement Bucket Sort

There is some started code on Blackboard that generates an array of random integers. Implement the `bucketSort` method. A demonstrator will tell you which secondary sorting method to use:

1. Selection Sort
2. Insertion Sort
3. Bubble Sort

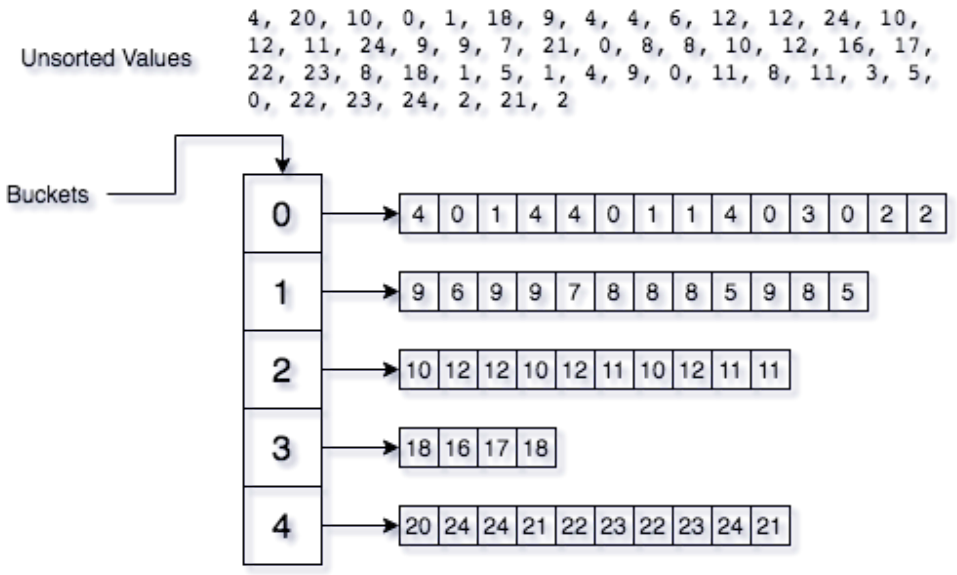


Figure 17.1: Bucket sort after the numbers have been distributed to the buckets.

17.0.2 Timing program execution

17.1 Extension

You might find it interesting to compare the efficiency of bucket sort when different secondary sorting methods are used

Lesson 18

Project Euler

Project Euler is a great resource for programmers looking to find a challenge.

Browse Project Euler's archives and choose a couple to solve. Your selections should be challenging enough to require some planning.

Lesson 19

R programming 1

In this lesson we will cover the basics of R and the RStudio IDE.

19.1 R Basics

Open RStudio. You'll find the icon on the dock. The interface will look like 19.1.

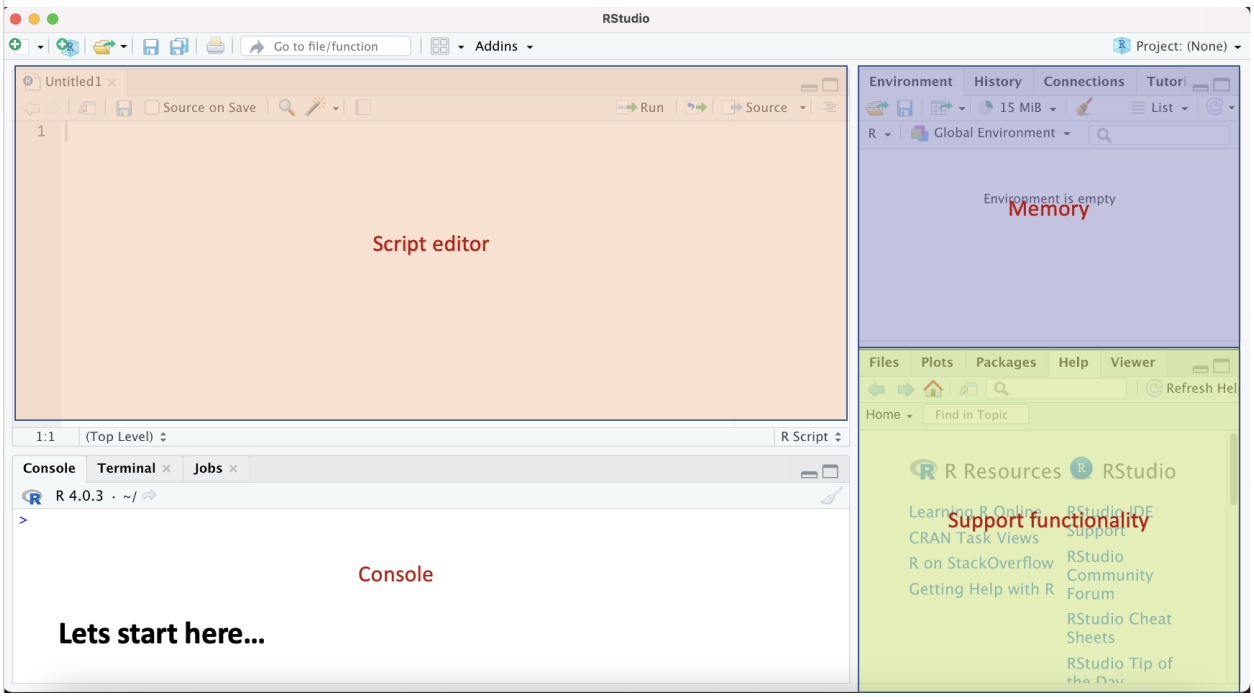


Figure 19.1: RStudio interface

We will begin by entering commands directly into the console.

1. Enter the following command, followed by pressing enter:

```
1+1
```

The console should have returned `[1] 2`. The commands were executed in real-time. Unlike Java, we did not first compile our code. This is because R is a dynamic programming language.

2. Now type: `30:100`. Your output should resemble:

```
 [1]  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
[19]  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
[37]  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83
[55]  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

We just created a sequence of values from 30 to 100, in increments of 1. The value `[1]` refers to the index of the value immediately to its right in the sequence. That is, `30` is the `[1]`st value in the sequence, while `48` is the `[19]`th value in the sequence value.

3. These index values will change if you resize the console window, and run the command again. Try this now.

4. The command `seq(from, to)` is a more powerful sequence function than colon operator. Enter the following command:

```
seq(1, 12, length = 5)
```

5. We just saw our first function in R. We can learn more about the sequence function by typing `?seq`. Try this now.

6. We create variables in R using the following syntax:

```
my_age <- 21
```

Variable names may consist of letters [a-z, A-Z], numbers [0-9], the period [.], and underscore [_]. Variable names must begin with a letter or period. Variable names are case-sensitive.

7. R is an array-based programming language. As covered in L24, it enables the application of operations to an entire collection of values through a concise and transparent syntax. We will now explore this behaviour.

8. Create a vector using the following:

```
die <- 1:6
```

Next, divide all values in die with:

```
die/2
```

Your output should look like:

```
 [1] 0.5 1.0 1.5 2.0 2.5 3.0
```

19.2 Practical exercises 1

Lets practice what we've learned.

1. Create a sequence of even numbers from 0 to 10, saving it to a variable called `my_seq`. Tip: remember to use `?seq` to read the help page.
2. Multiply all elements in `my_seq` by 3, then square them. Your output should be:

```
[1]   0   9  36  81 144 225 324 441 576 729 900
```

3. In a single one-line command, do the following:
   (a) Choose any number and add 2 to it.
   (b) Multiply the result by 3.
   (c) Subtract 6 from the answer.
   (d) Divide what you get by 3.

   What number did you get?
4. Create a sequence of numbers from 10 to 20, storing it in `my_nums`. Sum the elements in `my_nums` using the function `sum`. You should get the value 165.
5. Take the mean of values stored in `my_nums`. You should get the value 15.

19.3 Vectors and data types

R has several native data structures for storing data, the two most important being vectors and data frames. In this section we will work with vectors. A vector is a collection of ordered, homogeneous elements. Vectors store data in a 1-dimensional array, and can only store one type of data.

1. Create a vector `c`ontaining six sequential values:

```
die <- c(1,2,3,4,5,6)
```

2. The `c` function stands for combine. You can read more about it with `?c`.
3. The concatenate function `c()` can be used to concatenate single elements, or entire vectors. Enter the following:

```
x <- c(3, -90, 4)
c(1:6, 9, -4, x, 2)
```

   Were you surprised by the output?
4. You can check that `die` is of type vector with the following command:

```
is.vector(die)
```

5. Notice that R returned the value `TRUE`. Vectors can store a range of data types, the most of important of which include:

(a) Doubles
(b) Characters
(c) Logicals

The value 'TRUE' is a *logical*, which we call Booleans in Java.

6. Characters store text data. Type the following into the console:

```
letters
```

R comes packaged with a range of convenient in-memory data sets and variables for you to use.

7. Create a vector using `c()` which contains the data: 10, 20, 30. Next, add the value "a" to your vector, again using `c()`. Your output should look like:

```
[1] "10"  "20"  "30"  "a"
```

Look closely at the contents. What happened to your numeric values? Why might this have happened? Recall that vectors can only store homogeneous data. That is, they must all be of the same type.

8. R is a weakly typed language, and allows for the dynamic conversion of data types. If a character string is present in an atomic vector, R will *coerce* everything else in the vector to character strings. R's coercion system is visualised in 19.2.

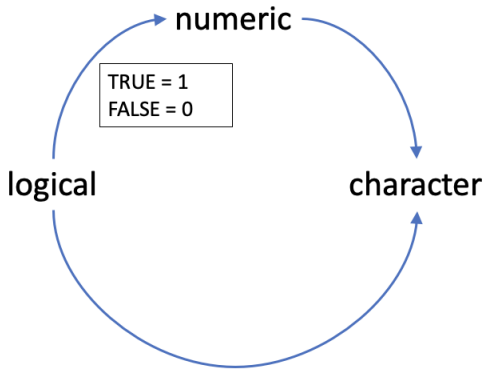


Figure 19.2: Coercion in R

9. Logicals play an important role in R. Create a vector of logicals with the following:

```
c(TRUE, FALSE, FALSE,TRUE)
```

19.4 Practical exercises 2

1. Create a sequence `a` which contains values 1 to 10. Create a second vector `b` which contains the values 11 through 30. Combine them into a new vector called `c`.

2. Create a vector `data` which combines the values: 10, "a", TRUE. What data type will your vector hold? Confirm your suspicion with one of the following three commands:

```
is.character(data)
is.logical(data)
is.numeric(data)
```

3. Run the command `set.seed(3)`. Using the `sample` function, generate 4 random values, from the set of values 1:10. Your output must exactly match:

```
[1] 5 7 4 2
```

4. Check the `length()` of the `letters` vector.

5. Calculate the length of the letters vector modulus 6. In R, the mod command is `%%`. You should get the value `2`.

Lesson 20

R programming 2

In this lesson we will meet the `data.frame`, and explore some of the more powerful features of array programming, including subsetting and querying with boolean operators.

20.1 Data frames

Vectors are a powerful structure for organising data. But they are limited to 1 dimension, and can only store elements of the same type. Data frames solve both of these issues. They are 2D, and can store mixed types. Think of them as a spreadsheet. Mixed types of data is very common in data analysis.

Data frames group column vectors together into a two-dimensional table. Each vector becomes a column in the table. Each column of a data frame can contain a different type of data. Within a column, every cell must be the same type of data. The data frame illustrated in 20.1 has 4 rows and 3 columns. Its dimensions are 4x3.

Data frames often contain column names. These are not part of the data or the frame's dimensions. Rather, they provide a reminder as to what data each column stores.

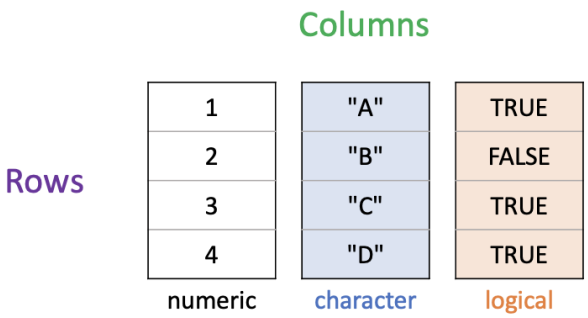


Figure 20.1: A 4x3 data frame

1. Type `mtcars` into the console.
2. Read more about the data set with `?mtcars`
3. Check its dimensions with the `dim` command.

4. Get a list of `mtcars` column names with `colnames`.
5. To select a particular column of data, use the `frame$colname` syntax. Select the mpg column with: `mtcars$mpg`.

20.2 Subsetting

We can access values in vectors and dataframes in a similar fashion to arrays in Java. To do so, we use `vector[indices]` and `dataframe[rows,cols]` syntax. E.g., `my_data[1:5]`.

1. Get the first 10 elements of the vector `letters`.
2. Use the `c()` function to return the 3rd, 7-12th, and 15th elements from `letters`.
3. Select rows 1-5, columns 2 and 4 from `mtcars`. Your results should match:

```
                  cyl  hp
Mazda RX4           6 110
Mazda RX4 Wag       6 110
Datsun 710          4  93
Hornet 4 Drive      6 110
Hornet Sportabout   8 175
```

4. Omitting the row or column index is shorthand for "all elements". Select the first 3 rows, all columns of `mtcars`. Your results should match:

```
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

20.3 Changing values

We modify values within an R object with a two step process:

- Describe the value (or values) you wish to modify.
- Use the assignment operator ← to overwrite those values.

For example, die[c(1,3)] ← c(10,20).

1. Create a sequence from 0 to 50, in steps of 5, assigning it in d.
2. Create a sequence from 100 to 130, in steps of 10, assign it to $nums$.
3. Overwrite values in d at indices 2 through 4, with those in nums. Your results should match:

```
> d
 [1]   0 100 110 120 130  25  30  35  40  45  50
```

4. Take the values in d at indices 2 through 4, and add 5 to them. Your results should now match:

```
> d
[1]   0 200 210 220 230  25  30  35  40  45  50
```

20.4 Logical and subsetting

Logicals are helpful when performing relational operations. For example:

```
> d <- 1:10
> d > 5
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

We can also use logicals to subset values in a vector or data frame.

1. Create a series of 10 logicals, all FALSE, using the $logical()$ function, storing in variable idx. Tip: check the help if your stuck with ?.

2. Set the first three elements of idx to TRUE. You could do this the laborious way with $c(TRUE, TRUE, TRUE)$, but you'll use R's recycling rule as we strive for concise syntax. See L25 for an example. Your idx should now match:

```
> idx
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

3. Lets use idx to extract the first three values in d with $d[idx]$. Your output should match:

```
# Extract values in d at indices 1:3
> d[idx]
[1] 1 2 3
```

   Notice that only values at index positions with $TRUE$ were returned; those with $FALSE$ were omitted in the output.

This was a tedious process that could've been achieved far more quickly with `[1:3]`. So why bother? Logical subsetting becomes powerful when combined with relational and boolean operators:

1. Overwrite d with values from 1 to 10.

2. Identify values in d greater than 5, storing in idx.

3. Extract elements in d using idx. Your results should match 6 through 10.

4. Overwrite elements in d at indices in idx with -1. Your output should match:

```
> d
[1]  1  2  3  4  5 -1 -1 -1 -1 -1
```

20.5 Querying with logicals and boolean operators

R has Boolean operators similar to Java. The most common being `and` and `or`, performed with & and | respectively. See L25 for a complete list. Lets bring together what we've learned to perform querying on the `mtcars` data frame.

1. Identify cars in `mtcars` with mpg greater than 25.
2. Logicals take on the vales 1 and 0 for $TRUE$ and $FALSE$ respectively. Use this information, along with the `sum` function to count how many cars have an mpg over 25. There should be 6.
3. Lets identify some lemon vehicles in our `mtcars` data set; those with high gas consumption but lower power. Using logical and boolean operators, identify vehicles with mpg over 25, and horsepower below 70. Your results should match:

```
                mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1
```