SIT232

Object Oriented Development

Learning Summary Report

Peter Stacey
219011171

Self-Assessment Details

The following checklists provide an overview of my self-assessment for this unit.

	Pass (D)	Credit (C)	Distinction (B)	High Distinction (A)
Self-Assessment				✓

Self-Assessment Statement

Declaration

I declare that this portfolio is my individual work. I have not copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part of this submission been written for me by another person.

Signature: **Peter Stacey**

Portfolio Overview

This portfolio includes work that demonstrates that I have achieve all Unit Learning Outcomes for SIT102 Unit Title to a **High Distinction** level.

Before the start of the Trimester I had never as much as even read a line of C# code that I can recall. While I had done some casual small snippets in Java previously, I hadn't had any exposure to object-oriented development in any formal sense.

In that regard, I found this subject to be both interesting and extremely educational. Particularly with tasks like 5.3D and 5.4HD, implementing object-oriented design patterns, incorporating the C# unit test tooling into my testing, and especially through helping others, I feel I have gained an excellent baseline knowledge of object-oriented programming, C# and parts of .NET Core. In addition, I've gained a deeper understanding of both Visual Studio and Visual Studio Code, as tools to assist with development.

Through my videos, I've demonstrated both knowledge of the subject, and the ability to develop effective visuals to help explain the concepts. In doing so, I completed 100% of all tasks set in the subject and produced videos for all the tasks that required them:

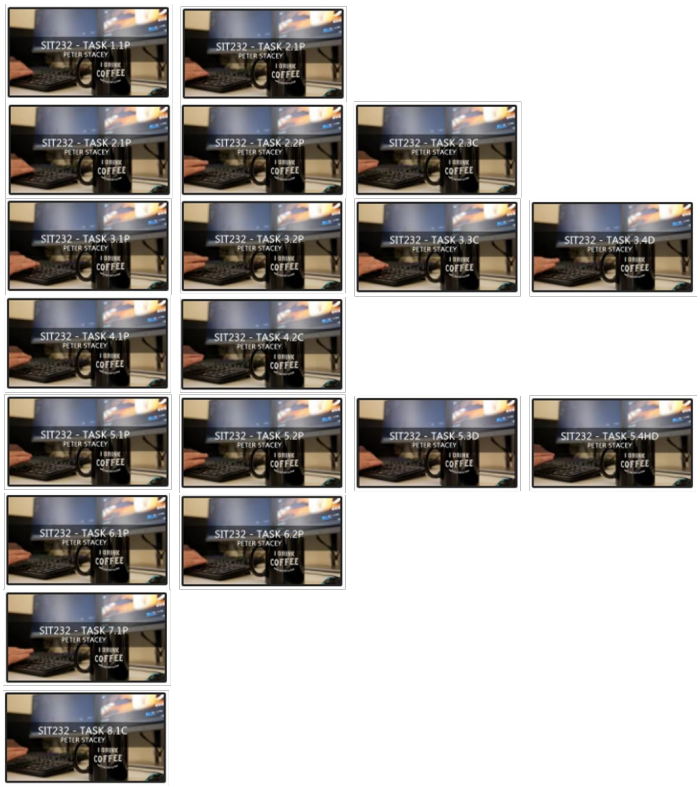


Figure 1: Thumbnails of videos produced to support my work

Particularly through Task 1.3D and offering assistance to other students, I feel I've gained a deeper understanding of the subject, beyond the material provided and also extended myself in task completion.

For Task 5.3D for example, I implemented the solution using both the object-oriented State Pattern, and and Mealy and Moore Finite State Machine approach, which is contained in my video submission.

Reflections

The most important things I learnt:

Abstraction, encapsulation, polymorphism and inheritance as core principles of OOP stand out as key learning, together with the value of composition and its flexibility.

I also feel as though the tasks involving UML diagrams also provided a greater understanding of the benefit of designing before programming and the productivity benefits in doing so, as well as the ease with which designs can be communicated through a shared understanding og UML.

I feel I learnt these topics, concepts, and/or tools really well:

I'm now confident about the principles of OOP, reading and producing basic UML class diagrams, being able to read documentation, given the excellent documentation that Microsoft provide for C#.

I also feel I have a good understanding of basic data types, iteration, flow-control and data structures, especially arrays and lists.

I also feel pretty comfortable using Visual Studio now, much more so than at the start of the Trimester.

I found the following topics particularly challenging:

The biggest challenge was working through the existing code of task 5.3D and figuring out how to integrate my design into that. Once it became clear though, working with an existing codebase became straight forward.

I found the following topics particularly interesting:

Design patterns were by far the most interesting aspect for me. Not only because they involve all of the principles we covered in the subject, but because they also provide templates for solving a problem that I can apply in real-world situations.

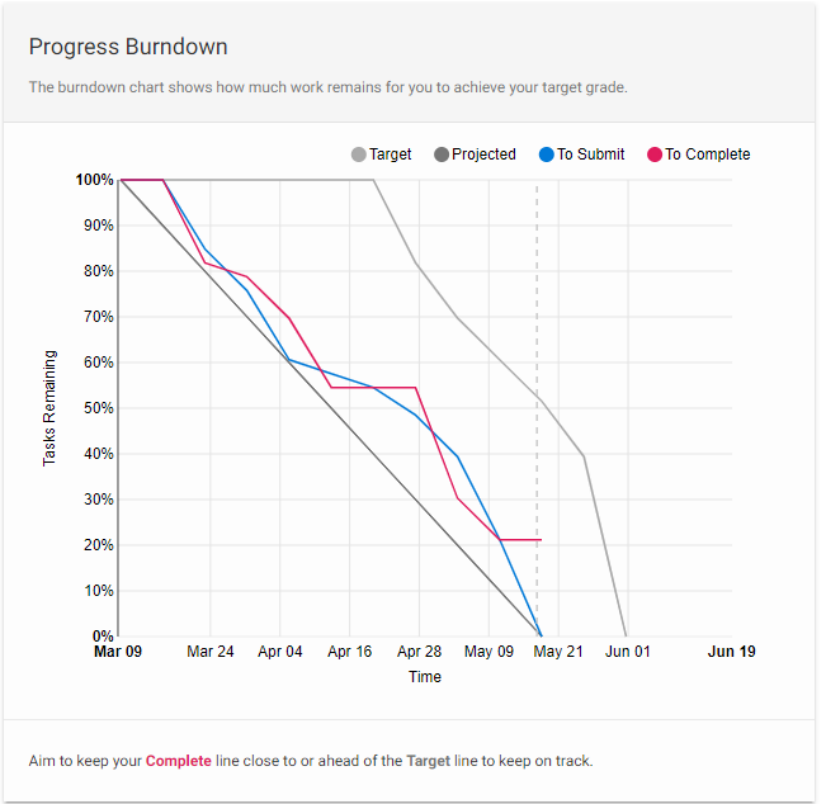
I still need to work on the following areas:

I think my next step is to keep taking on small problems and continuing to practice the skills learnt in this subject. I feel this is a good platform to go deeper into algorithms and to produce and release small projects as open-source codebases.

The things that helped me most were:

Sergey and the tutors were by far the most helpful resource. Therse were followed by the Learning Videos and Workbook. Together, these provided access to all of the knowledge that the weeks and lectures then provided structure around, in order to develop skills gradually.

My progress in this unit was …:



I feel this burndown chart demonstrates consist and early completion of required work and consistent effort throughout the semester.

If I did this unit again, I would do the following things differently:

At this point, I can't think of anything I would necessarily do differently myself. Perhaps to help others that have never used the tooling in the subject before, I might produce a video or two, to help them become familiar with Visual Studio, and particularly to understand the features they can ignore or filter out, in order to progress through the subject.

I might also consider helping others earlier to read and understand error messages. Just as with SIT102, there appears to be difficulty in reading errors and then relating those back to programs, in order to effectively debug problems.

Deakin University

Object Oriented Development

Peter Stacey

Portfolio Submission

Submitted By:
Peter Stacey
pstacey

Tutor:
Sergey Polyakovskiy

May 17, 2020

Contents

2 Overall Task Status

Task	Status	Times Assessed
C# Essentials: Selection and Casting	Complete	1
C# Essentials: Repetition	Complete	1
Helping Your Peers	Ready to Mark	0
C# Essentials: Classes and Objects	Complete	2
The Account Class	Complete	1
The MyTime Class	Complete	2
C# Essentials: Arrays and Lists	Complete	2
Validating Accounts	Complete	1
The MyPolynomial class	Complete	1
Bucket Sort	Complete	1
Exceptions and Error Handling	Complete	1
BuggySoft: Program Design and Class Composition	Complete	1
C# Essentials: Inheritance	Complete	1
Bank Transactions	Complete	2
A Simple Reaction-Timer Controller	Complete	1
An Enhanced Reaction-Timer Controller	Discuss	1
C# Essentials: Polymorphism	Complete	1
Multiple Bank Accounts	Complete	1
Documenting the Banking System	Complete	1
Abstract Transactions	Complete	2

3 Learning Outcomes

3.1 Evaluate Code

Evaluate simple program code for correct use of coding conventions, and use code tracing and debugging techniques to identify and correct issues.

Task	Rating	Status	Times Assessed
C# Essentials: Selection and Casting	♦♦♦◊◊	Complete	1
C# Essentials: Repetition	♦◊◊◊◊	Complete	1
Helping Your Peers	♦♦◊◊◊	Ready to Mark	0
C# Essentials: Classes and Objects	♦♦♦◊◊	Complete	2
C# Essentials: Arrays and Lists	♦♦♦◊◊	Complete	2
The MyTime Class	♦♦♦◊◊	Complete	2
The Account Class	♦◊◊◊◊	Complete	1
The MyPolynomial class	♦♦♦◊◊	Complete	1
Validating Accounts	♦♦◊◊◊	Complete	1
Bank Transactions	♦♦♦◊◊	Complete	2
Bucket Sort	♦♦♦◊◊	Complete	1
Exceptions and Error Handling	♦♦♦◊◊	Complete	1
C# Essentials: Inheritance	♦◊◊◊◊	Complete	1
Multiple Bank Accounts	♦♦◊◊◊	Complete	1
BuggySoft: Program Design and Class Composition	♦♦♦♦♦	Complete	1
A Simple Reaction-Timer Controller	♦♦♦♦◊	Complete	1
An Enhanced Reaction-Timer Controller	♦♦◊◊◊	Discuss	1
C# Essentials: Polymorphism	♦♦◊◊◊	Complete	1
Abstract Transactions	♦♦♦◊◊	Complete	2
Documenting the Banking System	♦♦◊◊◊	Complete	1

3.2 Principles

Apply and explain the principles of object oriented programming including abstraction, encapsulation, inheritance and polymorphism.

Task	Rating	Status	Times Assessed
C# Essentials: Selection and Casting	♦♦♦◊◊	Complete	1
C# Essentials: Repetition	♦♦♦◊◊	Complete	1
Helping Your Peers	♦♦♦♦◊	Ready to Mark	0
C# Essentials: Classes and Objects	♦♦♦◊◊	Complete	2
C# Essentials: Arrays and Lists	♦♦♦◊◊	Complete	2
The MyTime Class	♦♦♦◊◊	Complete	2
The Account Class	♦◊◊◊◊	Complete	1
The MyPolynomial class	♦♦♦◊◊	Complete	1
Validating Accounts	♦♦◊◊◊	Complete	1
Bank Transactions	♦♦♦◊◊	Complete	2
Bucket Sort	♦♦♦◊◊	Complete	1
Exceptions and Error Handling	♦♦♦♦◊	Complete	1
C# Essentials: Inheritance	♦♦♦◊◊	Complete	1
Multiple Bank Accounts	♦♦♦◊◊	Complete	1
BuggySoft: Program Design and Class Composition	♦♦♦♦◊	Complete	1
A Simple Reaction-Timer Controller	♦♦♦◊◊	Complete	1
An Enhanced Reaction-Timer Controller	♦♦♦♦♦	Discuss	1
C# Essentials: Polymorphism	♦♦♦♦◊	Complete	1
Abstract Transactions	♦♦♦◊◊	Complete	2
Documenting the Banking System	♦♦♦◊◊	Complete	1

3.3 Build Programs

Implement, and test small object oriented programs that conform to planned system structures and requirements

Task	Rating	Status	Times Assessed
C# Essentials: Selection and Casting	♦♦♦♦◊	Complete	1
C# Essentials: Repetition	♦♦♦♦◊	Complete	1
Helping Your Peers	♦♦◊◊◊	Ready to Mark	0
C# Essentials: Classes and Objects	♦♦♦◊◊	Complete	2
C# Essentials: Arrays and Lists	♦♦♦◊◊	Complete	2
The MyTime Class	♦♦♦◊◊	Complete	2
The Account Class	♦♦◊◊◊	Complete	1
The MyPolynomial class	♦♦♦♦◊	Complete	1
Validating Accounts	♦♦♦◊◊	Complete	1
Bank Transactions	♦♦♦◊◊	Complete	2
Bucket Sort	♦♦♦♦◊	Complete	1
Exceptions and Error Handling	♦♦♦◊◊	Complete	1
C# Essentials: Inheritance	♦♦♦◊◊	Complete	1
Multiple Bank Accounts	♦♦◊◊◊	Complete	1
BuggySoft: Program Design and Class Composition	♦♦♦♦◊	Complete	1
A Simple Reaction-Timer Controller	♦♦♦♦◊	Complete	1
An Enhanced Reaction-Timer Controller	♦♦♦♦♦	Discuss	1
C# Essentials: Polymorphism	♦♦◊◊◊	Complete	1
Abstract Transactions	♦♦◊◊◊	Complete	2
Documenting the Banking System	♦◊◊◊◊	Complete	1

3.4 Design

Design, communicate, and evaluate solution structures using appropriate diagrams and textual descriptions.

Task	Rating	Status	Times Assessed
C# Essentials: Selection and Casting	♦♦◊◊◊	Complete	1
C# Essentials: Repetition	♦♦◊◊◊	Complete	1
Helping Your Peers	♦♦◊◊◊	Ready to Mark	0
C# Essentials: Classes and Objects	♦◊◊◊◊	Complete	2
C# Essentials: Arrays and Lists	♦♦◊◊◊	Complete	2
The MyTime Class	♦♦♦◊◊	Complete	2
The Account Class	♦♦◊◊◊	Complete	1
The MyPolynomial class	♦♦♦◊◊	Complete	1
Validating Accounts	♦♦♦◊◊	Complete	1
Bank Transactions	♦♦♦◊◊	Complete	2
Bucket Sort	♦♦♦◊◊	Complete	1
Exceptions and Error Handling	♦♦◊◊◊	Complete	1
C# Essentials: Inheritance	♦♦◊◊◊	Complete	1
Multiple Bank Accounts	♦♦♦◊◊	Complete	1
BuggySoft: Program Design and Class Composition	♦♦♦♦◊	Complete	1
A Simple Reaction-Timer Controller	♦♦♦♦♦	Complete	1
An Enhanced Reaction-Timer Controller	♦♦♦♦♦	Discuss	1
C# Essentials: Polymorphism	♦◊◊◊◊	Complete	1
Abstract Transactions	♦♦◊◊◊	Complete	2
Documenting the Banking System	♦♦♦♦♦	Complete	1

3.5 Justify

Justify meeting specified outcomes through providing relevant evidence and critiquing the quality of that evidence against given criteria.

Task	Rating	Status	Times Assessed
C# Essentials: Selection and Casting	♦♦♦◊◊	Complete	1
C# Essentials: Repetition	♦♦♦◊◊	Complete	1
Helping Your Peers	♦♦◊◊◊	Ready to Mark	0
C# Essentials: Classes and Objects	♦♦♦◊◊	Complete	2
C# Essentials: Arrays and Lists	♦♦♦◊◊	Complete	2
The MyTime Class	♦♦♦◊◊	Complete	2
The Account Class	♦◊◊◊◊	Complete	1
The MyPolynomial class	♦♦♦◊◊	Complete	1
Validating Accounts	♦♦♦◊◊	Complete	1
Bank Transactions	♦♦♦◊◊	Complete	2
Bucket Sort	♦♦♦♦◊	Complete	1
Exceptions and Error Handling	♦♦♦♦♦	Complete	1
C# Essentials: Inheritance	♦♦♦◊◊	Complete	1
Multiple Bank Accounts	♦♦◊◊◊	Complete	1
BuggySoft: Program Design and Class Composition	♦♦♦◊◊	Complete	1
A Simple Reaction-Timer Controller	♦♦♦◊◊	Complete	1
An Enhanced Reaction-Timer Controller	♦♦♦♦♦	Discuss	1
C# Essentials: Polymorphism	♦♦♦◊◊	Complete	1
Abstract Transactions	♦♦♦◊◊	Complete	2
Documenting the Banking System	♦◊◊◊◊	Complete	1

4 C# Essentials: Selection and Casting

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦♦◊◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

Outcome	Weight
Principles	♦♦♦◊◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

Outcome	Weight
Build Programs	♦♦♦♦◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

Outcome	Weight
Design	♦♦◊◊◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

Outcome	Weight
Justify	♦♦♦◊◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

Date	Author	Comment
2020/03/17 11:13	Peter Stacey	Ready to Mark
2020/03/17 14:51	Peter Stacey	I will link my video here this evening
2020/03/18 08:08	Peter Stacey	Ready to Mark
2020/03/18 08:08	Peter Stacey	Just added some access modifiers to methods that I didn't include initially
2020/03/18 11:58	Peter Stacey	Video link: https://youtu.be/DcyAP9SbTfw
2020/03/18 15:58	Dipto Pratyaksa	Complete
2020/03/18 15:58	Dipto Pratyaksa	Very well done

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Selection and Casting

Submitted By:
Peter Stacey
pstacey
2020/03/18 08:08

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦♦◊
Design	♦♦◊◊◊
Justify	♦♦♦◊◊

The task involves evaluating provided code that contain errors, to identify problems and then correct them before coding. Additionally, the task involves working in an object oriented way through the use of C# and there are multiple problems to be solved, each with an individual program. While there is some design, all of the problems can be solved in single file programs and with simple design.

March 18, 2020

```
using System;

namespace Task_1._1P
{
    class IfStatement
    {
        static void Main(string[] args)
        {
            int number = 0;

            Console.WriteLine("Enter the number (as an integer): ");
            try
            {
                number = Convert.ToInt32(Console.ReadLine());
            }
            catch (FormatException) // Thrown for non-integer input
            {
                Console.WriteLine("ERROR: Input was not a number");
                System.Environment.Exit(1);
            }

            if (number == 1)
            {
                Console.WriteLine("ONE");
            }
            else if (number == 2)
            {
                Console.WriteLine("TWO");
            }
            else if (number == 3)
            {
                Console.WriteLine("THREE");
            }
            else if (number == 4)
            {
                Console.WriteLine("FOUR");
            }
            else if (number == 5)
            {
                Console.WriteLine("FIVE");
            }
            else if (number == 6)
            {
                Console.WriteLine("SIX");
            }
            else if (number == 7)
            {
                Console.WriteLine("SEVEN");
            }
            else if (number == 8)
            {
                Console.WriteLine("EIGHT");
            }
```

```
54              else if (number == 9)
55              {
56                  Console.WriteLine("NINE");
57              }
58              else
59              {
60                  Console.WriteLine("ERROR: Number must be from 1-9");
61              }
62          }
63      }
64  }
```

```cs
using System;

namespace Program_2
{
    class SwitchStatement
    {
        static void Main(string[] args)
        {
            int number = 0;

            Console.WriteLine("Enter a number (as an integer): ");
            try
            {
                number = Convert.ToInt32(Console.ReadLine());
            }
            catch (FormatException)
            {
                Console.WriteLine("ERROR: Input was not an integer");
                System.Environment.Exit(1);
            }
            switch (number)
            {
                case 1: Console.WriteLine("One"); break;
                case 2: Console.WriteLine("Two"); break;
                case 3: Console.WriteLine("Three"); break;
                case 4: Console.WriteLine("Four"); break;
                case 5: Console.WriteLine("Five"); break;
                case 6: Console.WriteLine("Six"); break;
                case 7: Console.WriteLine("Seven"); break;
                case 8: Console.WriteLine("Eight"); break;
                case 9: Console.WriteLine("Nine"); break;
                default: Console.WriteLine("ERROR: Number must be from 1-9"); break;
            }

            Console.ReadLine();
        }
    }
}
```

```cs
using System;

namespace Program_5
{
    /// <summary>
    /// Calculates recommended cooking time in a microwave
    /// </summary>
    class Microwave
    {
        // Reads String input in the console
        /// <summary>
        /// Reads String input in the console
        /// </summary>
        /// <returns>
        /// The String input of the user
        /// </returns>
        /// <param name="prompt">The String prompt for the user</param>
        public String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The String prompt for the user</param>
        public int ReadInteger(String prompt)
        {
            int number = 0;
            String numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

        // Reads double input in the console
        /// <summary>
        /// Reads double input in the console
        /// </summary>
        /// <returns>
        /// The input of the user as a double
        /// </returns>
        /// <param name="prompt">The String prompt for the user</param>
        public double ReadDouble(String prompt)
        {
```

```
54              double number = 0.0;
55              String numberInput = ReadString(prompt);
56              while (!(double.TryParse(numberInput, out number)))
57              {
58                  Console.WriteLine("Please enter a number");
59                  numberInput = ReadString(prompt);
60              }
61              return Convert.ToDouble(numberInput);
62          }
63
64          // Returns the number of items to cook
65          /// <summary>
66          /// Returns the number of items to cook
67          /// </summary>
68          /// <returns>
69          /// An integer of the number of items
70          /// </returns>
71          public int NumberOfItems()
72          {
73              String prompt = "Enter the number of items";
74              int items = ReadInteger(prompt);
75              while (items < 1)
76              {
77                  Console.WriteLine("Please enter at least 1 item");
78                  items = ReadInteger(prompt);
79              }
80              return items;
81          }
82
83          // Returns the time in minutes, to cook a single item
84          /// <summary>
85          /// Returns the time in minutes, to cook a single item
86          /// </summary>
87          /// <returns>
88          /// A double of the cooking time for one item
89          /// </returns>
90          public double SingleCookingTime()
91          {
92              String prompt = "Enter the time for one item (minutes)";
93              double time = ReadDouble(prompt);
94              while (time <= 0.0)
95              {
96                  Console.WriteLine("Please enter a time more than 0.0");
97                  time = ReadDouble(prompt);
98              }
99              return time;
100         }
101
102         // Returns the recommended cooking time for the number of items
103         /// <summary>
104         /// Returns the recommended cooking time for the number of items
105         /// </summary>
106         /// <returns>
```

```
107         /// A double of the cooking time for the number of items, or -1
108         /// if more than 3 items are to be cooked
109         /// </returns>
110         public double CookingTime(int items, double singleTime)
111         {
112             switch (items)
113             {
114                 case 1: return singleTime;
115                 case 2: return singleTime * 1.5;
116                 case 3: return singleTime * 2;
117                 default: return -1; // number of items not recommended
118             }
119         }
120 
121         // Outputs recommended cooking time for the number of items
122         /// <summary>
123         /// Outputs the recommended cooking time for the number of items
124         /// </summary>
125         public void RecommendedCookingTime()
126         {
127             int items = NumberOfItems();
128             double singleTime = SingleCookingTime();
129             double cookingTime = CookingTime(items, singleTime);
130             if (cookingTime > -1)
131             {
132                 String output = String.Format("Recommended cooking time: {0}
                    ↪  minutes", cookingTime);
133                 Console.WriteLine(output);
134             }
135             else
136             {
137                 Console.WriteLine("Maximum of 3 items recommended");
138             }
139         }
140 
141         static void Main(String[] args)
142         {
143             Microwave microwave = new Microwave();
144             microwave.RecommendedCookingTime();
145 
146             Console.ReadLine();
147         }
148     }
149 }
```

```cs
using System;

namespace Program_6
{
    class DoCasting
    {
        static void Main(string[] args)
        {
            int sum = 17;
            int count = 5;

            int intAverage = sum / count;
            Console.WriteLine(intAverage); // average is integer division and not precise

            double doubleAverage = 0.0;
            doubleAverage = sum / count;
            Console.WriteLine(doubleAverage); // still integer division and not precise answer

            doubleAverage = (double)sum / count;
            Console.WriteLine(doubleAverage); // Now more precise as the division now uses doubles
        }
    }
}
```

5 C# Essentials: Repetition

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦◊◊◊◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

Outcome	**Weight**
Principles	♦♦♦◊◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

Outcome	**Weight**
Build Programs	♦♦♦♦◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

Outcome	**Weight**
Design	♦♦◊◊◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

Outcome	**Weight**
Justify	♦♦♦◊◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

Date	Author	Comment
2020/03/17 14:51	Peter Stacey	Ready to Mark
2020/03/17 14:52	Peter Stacey	I will link my video here this evening
2020/03/18 08:09	Peter Stacey	Ready to Mark
2020/03/18 08:09	Peter Stacey	Just added some access modifiers that I didn't include originally
2020/03/18 13:40	Peter Stacey	Video link: https://youtu.be/A9neQu8zVD8
2020/03/18 16:12	Dipto Pratyaksa	why do you need access modifier? We haven't dealt with OO program as yet
2020/03/18 16:12	Dipto Pratyaksa	Discuss
2020/03/18 17:02	Peter Stacey	I didn't really need it, however since I included the ReadString, ReadInteger and ReadDouble methods in the DividsibleFour and GuessingNumber classes, I added access modifiers as I'd normally have them as public static functions if I was implementing them elsewhere. There wouldn't normally be in those classes at all. The code runs fine without the access modifiers, so either way would have been fine and I added them more for preference (eg. if so some reason I needed to use them elsewhere, then I could use GuessingNumber and access them in another class).
2020/03/22 00:11	Dipto Pratyaksa	Complete
2020/03/22 00:11	Dipto Pratyaksa	ok move on to next task

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Repetition

Submitted By:
Peter Stacey
pstacey
2020/03/18 08:09

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦◊◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦♦◊
Design	♦♦◊◊◊
Justify	♦♦♦◊◊

Like the first task, this task is broken into multiple pieces, with each requiring code to be analysed and/or small programs to be designed and programmed. There are a range of loops used to reinforce different approaches to repetition and logical operators used to for flow control and decision making.

March 18, 2020

```
using System;

namespace Program_1
{
    class Repetition
    {
        static void Main(string[] args)
        {
            int sum = 0;
            double average;
            int upperbound = 100;

            for (int number = 1; number <= upperbound; number++)
            {
                sum += number;
                //Console.WriteLine("Current number: " + number + " the sum is " +
                ↪  sum);
            }

            average = (double)sum / upperbound;

            Console.WriteLine("The sum is " + sum);
            Console.WriteLine("The average is " + average);

            sum = 0; // reset back to 0 for while loop approach
            average = 0.0; // reset back to 0
            int num = 1;
            while (num <= upperbound)
            {
                sum += num;
                //Console.WriteLine("Current number: " + number + " the sum is " +
                ↪  sum);
                num++;
            }

            average = (double)sum / upperbound;

            Console.WriteLine("The sum is " + sum);
            Console.WriteLine("The average is " + average);

            num = 1;
            sum = 0;
            average = 0.0;
            do
            {
                sum += num;
                num++;
            } while (num <= upperbound);

            average = (double)sum / upperbound;

            Console.WriteLine("The sum is " + sum);
            Console.WriteLine("The average is " + average);
```

```
52            }
53        }
54    }
```

```cs
using System;

namespace Program_4
{
    class GuessingNumber
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

        // Reads integer input in the console between two numbers
        /// <summary>
        /// Reads integer input in the console between two numbers
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        /// <param name="minimum">The minimum number allowed</param>
        /// <param name="maximum">The maximum number allowed</param>
        public static int ReadInteger(String prompt, int minimum, int maximum)
        {
            int number = ReadInteger(prompt);
```

```
54              while (number < minimum || number > maximum)
55              {
56                  Console.WriteLine("Please enter a whole number from " +
57                                    minimum + " to " + maximum);
58                  number = ReadInteger(prompt);
59              }
60              return number;
61          }
62
63          static void Main(string[] args)
64          {
65              int minimum = 1;
66              int maximum = 100;
67              int user1Number = ReadInteger(
68                  "USER1 Enter the number you are thinking of between " +
69                  minimum + " and " + maximum);
70              string prompt = "USER2 Enter a guess";
71              int user2Guess = ReadInteger(prompt, minimum, maximum);
72              while (user2Guess != user1Number)
73              {
74                  Console.WriteLine("You missed it. Guess again");
75                  user2Guess = ReadInteger(prompt);
76              }
77              Console.WriteLine("YOU GUESSED IT NOSTRADAMUS");
78          }
79      }
80  }
```

```cs
using System;

namespace Program_5
{
    class DivisibleFour
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

        // Reads integer input in the console between two numbers
        /// <summary>
        /// Reads integer input in the console between two numbers
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        /// <param name="minimum">The minimum number allowed</param>
        /// <param name="maximum">The maximum number allowed</param>
        public static int ReadInteger(String prompt, int minimum, int maximum)
        {
            int number = ReadInteger(prompt);
```

```
54              while (number < minimum || number > maximum)
55              {
56                  Console.WriteLine("Please enter a whole number from " +
57                                    minimum + " to " + maximum);
58                  number = ReadInteger(prompt);
59              }
60              return number;
61          }
62          static void Main(string[] args)
63          {
64              int n = ReadInteger("Enter a number larger than 1");
65              for (int i = 1; i <= n; i++)
66              {
67                  if ((i % 4 == 0) && (i % 5 != 0))
68                  {
69                      Console.WriteLine(i);
70                  }
71              }
72          }
73      }
74  }
```

6 C# Essentials: Classes and Objects

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦♦◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

Outcome	Weight
Principles	♦♦♦◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

Outcome	Weight
Build Programs	♦♦♦◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

Outcome	Weight
Design	♦◊◊◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

Outcome	Weight
Justify	♦♦♦◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

Date	Author	Comment
2020/03/17 23:42	Peter Stacey	Ready to Mark
2020/03/17 23:43	Peter Stacey	I will record video now and paste the link as soon as the edit is complete (might be in the morning)
2020/03/18 17:11	Dipto Pratyaksa	Discuss
2020/03/18 17:11	Dipto Pratyaksa	Where is it?
2020/03/19 09:58	Peter Stacey	Video link: https://youtu.be/Mh9iEmVSyBM
2020/03/22 00:16	Dipto Pratyaksa	Fantastic explanation. You sound and look like a pro!
2020/03/22 00:17	Dipto Pratyaksa	Complete
2020/03/22 00:17	Dipto Pratyaksa	What recording / editing program do you use to make the vidoe?
2020/03/22 09:50	Peter Stacey	I just record using OBS Studio and then process the audio in Adobe Audition and finalise editing of the video in Premiere Pro.

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Classes and Objects

Submitted By:
Peter Stacey
pstacey
2020/03/19 10:00

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦◊◊◊◊
Justify	♦♦♦◊◊

The move into more object oriented aspects of development and the creation of classes, and separating code into separate files involves evaluating requirements, designing a solution and coding the end result into working programs that are tested.

March 19, 2020

```
using System;

namespace Program_1
{
    class MobileProgram
    {
        static void Main(string[] args)
        {
            Mobile jimMobile = new Mobile("Monthly", "Samsung Galaxy S6",
            ↪  "07712223344");

            Console.WriteLine("Account Type: " + jimMobile.getAccType() +
                "\nMobile Number: " + jimMobile.getNumber() +
                "\nDevice: " + jimMobile.getDevice() +
                "\nBalance: " + jimMobile.getBalance());


            Console.WriteLine();

            jimMobile.setAccType("PAYG");
            jimMobile.setDevice("iPhone 6S");
            jimMobile.setNumber("07713334466");
            jimMobile.setBalance(15.50);

            Console.WriteLine("Account Type: " + jimMobile.getAccType() +
                "\nMobile Number: " + jimMobile.getNumber() +
                "\nDevice: " + jimMobile.getDevice() +
                "\nBalance: " + jimMobile.getBalance());

            Console.WriteLine();

            jimMobile.addCredit(10.0);
            jimMobile.makeCall(5);
            jimMobile.sendText(2);

            // Create additional mobile account and test
            Console.WriteLine("\nCreating new mobile account for Peter\n");

            Mobile peterMobile = new Mobile("Monthly", "Samsung Galaxy S9+",
            ↪  "0412324124");
            peterMobile.addCredit(50.00);
            peterMobile.makeCall(25);
            peterMobile.sendText(20);

            Console.ReadLine();
        }
    }
}
```

```cs
using System;

namespace Program_1
{
    /// <summary>
    /// The mobile class defines attributes and methods on mobile
    /// phone accounts
    /// </summary>
    class Mobile
    {
        // Instance variables
        private String accType, device, number;
        private double balance;

        // VARIABLES
        private const double CALL_COST = 0.245;
        private const double TEXT_COST = 0.078;


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="accType">The account type</param>
        /// <param name="device">The mobile phone make and model</param>
        /// <param name="number">The mobile phone number</param>
        public Mobile(String accType, String device, String number)
        {
            this.accType = accType;
            this.device = device;
            this.number = number;
            this.balance = 0.0;
        }

        /// <summary>
        /// Returns the account type
        /// </summary>
        /// <returns>
        /// The account type
        /// </returns>
        public String getAccType()
        {
            return this.accType;
        }

        /// <summary>
        /// Returns the device details
        /// </summary>
        /// <returns>
        /// The device make and model
        /// </returns>
        public String getDevice()
        {
            return this.device;
```

```
54          }
55
56          /// <summary>
57          /// Returns the mobile phone number
58          /// </summary>
59          /// <returns>
60          /// The the mobile phone number
61          /// </returns>
62          public String getNumber()
63          {
64              return this.number;
65          }
66
67          /// <summary>
68          /// Returns the account credit balance
69          /// </summary>
70          /// <returns>
71          /// The account credit balance in currency format
72          /// </returns>
73          public String getBalance()
74          {
75              return this.balance.ToString("C");
76          }
77
78          /// <summary>
79          /// Sets the account type
80          /// </summary>
81          /// <param name="accType">The new account type</param>
82          public void setAccType(String accType)
83          {
84              this.accType = accType;
85          }
86
87          /// <summary>
88          /// Sets the device details
89          /// </summary>
90          /// <param name="device">The device make and model</param>
91          public void setDevice(String device)
92          {
93              this.device = device;
94          }
95
96          /// <summary>
97          /// Sets the mobile phone number
98          /// </summary>
99          /// <param name="number">The new mobile phone number</param>
100         public void setNumber(String number)
101         {
102             this.number = number;
103         }
104
105         /// <summary>
106         /// Sets the account type
```

```
107         /// </summary>
108         /// <param name="balance">The new balance to set</param>
109         public void setBalance(double balance)
110         {
111             this.balance = balance;
112         }
113 
114         /// <summary>
115         /// Adds credit to the account balance
116         /// </summary>
117         /// <param name="amount">The amount to credit the account</param>
118         public void addCredit(double amount)
119         {
120             this.balance += amount;
121             Console.WriteLine("Credit added successfully. New balance " +
                ↪  getBalance());
122         }
123 
124         /// <summary>
125         /// Calculates the cost of a call by minutes talking and updates
126         /// the balance
127         /// </summary>
128         /// <param name="minutes">The time of the call(s) in minutes</param>
129         public void makeCall(int minutes)
130         {
131             double cost = minutes * CALL_COST;
132             this.balance -= cost;
133             Console.WriteLine("Call made. New balanace " + getBalance());
134         }
135 
136         /// <summary>
137         /// Calculates the cost of text sent by the number of texts and
138         /// updates the balance
139         /// </summary>
140         /// <param name="numTexts">The number of texts sent</param>
141         public void sendText(int numTexts)
142         {
143             double cost = numTexts * TEXT_COST;
144             this.balance -= cost;
145             Console.WriteLine("Text Sent. New balance " + getBalance());
146         }
147     }
148 }
```

```cs
using System;

namespace Program_2
{
    class EmployeeProgram
    {
        static void Main(string[] args)
        {
            // Create two employees with different salaries
            Employee andrew = new Employee("Andrew Cain", 180000);
            Employee jane = new Employee("Jane Doe", 45000);

            // Test getting the name and salary
            Console.WriteLine("Employee Name: " + andrew.getName() +
                ", Salary: " + andrew.getSalary());
            Console.WriteLine("Employee Name: " + jane.getName() +
                ", Salary: " + jane.getSalary());

            // Test increasing the salary
            andrew.raiseSalary(5.0);  // expect $189000
            jane.raiseSalary(15.0);   // expect $51750

            Console.WriteLine();

            // Create additional employee in lowest tax bracket
            Employee trev = new Employee("Trev", 12300);
            Console.WriteLine("Employee Name: " + trev.getName() +
                ", Salary: " + trev.getSalary());

            // Test tax calculates correctly
            Console.WriteLine("Employee Name: " + andrew.getName() +
                ", Tax Burden: " + andrew.Tax()); // expect $58146
            Console.WriteLine("Employee Name: " + jane.getName() +
                ", Tax Burden: " + jane.Tax()); // expect $8365.75
            Console.WriteLine("Employee Name: " + trev.getName() +
                ", Tax Burden: " + trev.Tax()); // expect Nil tax

        }
    }
}
```

```cs
using System;

namespace Program_2
{
    /// <summary>
    /// Class for employee details and salary
    /// </summary>
    class Employee
    {
        // Instance variables
        private String name;
        private double salary;

        /// <summary>
        /// The class constructor
        /// </summary>
        /// <param name="employeeName">The name of the employee</param>
        /// <param name="currentSalary">The current salary</param>
        public Employee(string employeeName, double currentSalary)
        {
            this.name = employeeName;
            this.salary = currentSalary;
        }

        /// <summary>
        /// Returns the name of the employee
        /// </summary>
        /// <returns>
        /// The name of the employee
        /// </returns>
        public String getName()
        {
            return this.name;
        }

        /// <summary>
        /// Returns the current salary of the employee
        /// </summary>
        /// <returns>
        /// The current salary of the employee as a string
        /// </returns>
        public String getSalary()
        {
            return this.salary.ToString("C");
        }

        /// <summary>
        /// Raises the current salary by a percentage
        /// </summary>
        /// <param name="percentRaise">The percent amount to add to the
        ↪  salary</param>
        public void raiseSalary(double percentRaise)
        {
```

```
53              this.salary = this.salary * (1.0 + (percentRaise / 100));
54              Console.WriteLine("Current salary for " + getName() + " now " +
                ↪  getSalary());
55          }
56
57          /// <summary>
58          /// Calculates the amount of tax deducted annually from the salary
59          /// </summary>
60          /// <returns>
61          /// The annual tax burden as a double
62          /// </returns>
63          public String Tax()
64          {
65              if (this.salary >= 180000)
66              {
67                  double tax = 54096 + (0.45 * (this.salary - 180000));
68                  return tax.ToString("C");
69              }
70              else if (this.salary > 90000)
71              {
72                  double tax = 20797 + (0.37 * (this.salary - 90000));
73                  return tax.ToString("C");
74              }
75              else if (this.salary > 37000)
76              {
77                  double tax = 3572 + (0.325 * (this.salary - 37000));
78                  return tax.ToString("C");
79              }
80              else if (this.salary > 18200)
81              {
82                  double tax = 0.18 * (this.salary - 18200);
83                  return tax.ToString("C");
84              }
85              else
86              {
87                  return "Nil";
88              }
89          }
90      }
91  }
```

```cs
using System;

namespace Program_3
{
    class CarProgram
    {
        static void Main(string[] args)
        {
            // Create a myCar object
            Car myCar = new Car(14.5, 65.5);

            // Test setting total miles, getting total miles and fuel
            myCar.setTotalMiles(100);
            Console.WriteLine("Current mileage: " + myCar.getTotalMiles()
                + ", Current fuel: " + myCar.getFuel());

            // Test adding fuel and printing the fuel cost
            myCar.addFuel(22);
            Console.WriteLine("Current fuel cost per litre: " +
            ↪  myCar.printFuelCost());

            // Test driving
            myCar.drive(60);
            Console.WriteLine("Current mileage: " + myCar.getTotalMiles()
                + ", Current fuel: " + myCar.getFuel());

            // Test driving again to check mileage accumulates correctly
            myCar.drive(30);
            Console.WriteLine("Current mileage: " + myCar.getTotalMiles()
                + ", Current fuel: " + myCar.getFuel());

            Console.ReadLine();
        }
    }
}
```

```
using System;

namespace Program_3
{
    /// <summary>
    /// Defines properties and methods to track car mileage, fuel and cost
    /// </summary>
    class Car
    {
        // Instance variables
        private double fuelEfficiency;
        private double fuelLevel;
        private int mileage;

        // VARIABLES
        double FUEL_COST = 1.385;
        double GALLONS_TO_LITRES = 4.546;

        /// <summary>
        /// Class constructor
        /// </summary>
        public Car(double efficiency, double fuel)
        {
            this.fuelEfficiency = efficiency;
            this.fuelLevel = fuel;
            this.mileage = 0;
        }

        /// <summary>
        /// Returns the current fuel in litres
        /// </summary>
        /// <returns>
        /// The current fuel in litres
        /// </returns>
        public double getFuel()
        {
            return this.fuelLevel;
        }

        /// <summary>
        /// Returns the total mileage
        /// </summary>
        /// <returns>
        /// The current mileage of the car
        /// </returns>
        public int getTotalMiles()
        {
            return this.mileage;
        }

        /// <summary>
        /// Sets the total mileage
        /// </summary>
```

```
54          /// <param name="miles">The total miles to set</param>
55          public void setTotalMiles(int miles)
56          {
57              this.mileage = miles;
58          }
59
60          /// <summary>
61          /// Returns the cost of fuel in currency format
62          /// </summary>
63          /// <returns>
64          /// The current cost of fuel
65          /// </returns>
66          public String printFuelCost()
67          {
68              return this.FUEL_COST.ToString("C");
69          }
70
71          /// <summary>
72          /// Returns the total cost of fuel use
73          /// </summary>
74          /// <returns>
75          /// The total cost of using an amount of fuel
76          /// </returns>
77          /// <param name="fuelLitres">The litres of fuel used</param>
78          public double calcCost(double fuelLitres)
79          {
80              return fuelLitres * this.FUEL_COST;
81          }
82
83          /// <summary>
84          /// Adds fuel to the fuel tank
85          /// </summary>
86          /// <param name="fuelLitres">Volume of fuel in litres</param>
87          public void addFuel(double fuelLitres)
88          {
89              this.fuelLevel += fuelLitres;
90              double fillCost = calcCost(fuelLitres);
91              Console.WriteLine("Cost of fill: "
92                  + calcCost(fuelLitres).ToString("C"));
93          }
94
95          /// <summary>
96          /// Converts fuel volume from gallons to litres
97          /// </summary>
98          /// <returns>
99          /// The volume of fuel in litres
100         /// </returns>
101         /// <param name="gallons">The gallons of fuel to convert</param>
102         public double convertToLitres(double gallons)
103         {
104             return gallons * this.GALLONS_TO_LITRES;
105         }
106
```

```
107         /// <summary>
108         /// Calculates and outputs the cost of a trip and updates car
109         /// properties
110         /// </summary>
111         /// <param name="milesTravelled">The total miles travelled</param>
112         public void drive(int milesTravelled)
113         {
114             this.mileage += milesTravelled; // accumulate mileage
115             double gallonsUsed = milesTravelled / this.fuelEfficiency;
116             double litresUsed = convertToLitres(gallonsUsed);
117             this.fuelLevel -= litresUsed; // remove fuel from the tank
118             double tripCost = calcCost(litresUsed);
119             Console.WriteLine("Total cost of travelling "
120                 + milesTravelled + " miles = "
121                 + tripCost.ToString("C"));
122         }
123     }
124 }
```

7 The Account Class

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦◊◊◊◊

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

Outcome	**Weight**
Principles	♦◊◊◊◊

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

Outcome	**Weight**
Build Programs	♦♦◊◊◊

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

Outcome	**Weight**
Design	♦♦◊◊◊

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

Outcome	**Weight**
Justify	♦◊◊◊◊

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

Date	Author	Comment
2020/03/19 07:17	Peter Stacey	Ready to Mark
2020/03/19 12:09	Peter Stacey	Video link: https://youtu.be/RcVudQIwLM0
2020/03/22 00:21	Dipto Pratyaksa	I love your coding and video production skills
2020/03/22 00:21	Dipto Pratyaksa	Discuss
2020/03/22 00:25	Dipto Pratyaksa	Question for you... After you created an object , how long will the object's data get stored in the computer?
2020/03/22 09:47	Peter Stacey	It will be stored on the heap of the virtual address space of the .NET runtime for as long as you have a reference to that object in the running program. After an object is no longer referenced, then it will still remain there until the garbage collector removes it. That can occur if the remaining physical memory gets low, if the reserved and committed space on the managed heap reaches a threshold that the .NET runtime controls, and if GC.Collect is called manually.So in general, the object will remain while ever it is needed and then afterwards, it will still remain until garbage collected. The exact time will vary depending on how frequently the garbage collector needs to run on a given system or for a given program, and if called manually.
2020/03/23 11:22	Dipto Pratyaksa	perfect
2020/03/23 11:22	Dipto Pratyaksa	Complete
2020/03/23 11:22	Dipto Pratyaksa	Keep up your high energy in crushing all the upcoming tasks

Deakin University

Object Oriented Development

OnTrack Submission

The Account Class

Submitted By:
Peter Stacey
pstacey
2020/03/19 07:17

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦◇◇◇◇
Principles	♦◇◇◇◇
Build Programs	♦♦◇◇◇
Design	♦♦◇◇◇
Justify	♦◇◇◇◇

At this point, being a task that we will continue to build on throughout the semester, the program didn't require much design, although it currently has some significant limitations (eg. no protection against overdrawing the account) that need to be fixed in the future. It was more, the beginning of a larger program and straight forward to implement.

March 19, 2020

```
using System;

namespace Task_2._2P
{
    class TestAccount
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n**************");
            Console.WriteLine("TESTING START");
            Console.WriteLine("**************");

            Console.WriteLine("\n----------------------");
            Console.WriteLine("GOOD ACCOUNT BEHAVIOUR");
            Console.WriteLine("----------------------");

            Account okAccount = new Account("Mrs Good", 0);

            okAccount.Print(); // Expect balance to be $0.00
            okAccount.Deposit(500);
            okAccount.Withdraw(100);
            okAccount.Print(); // Expect balance to be $400.00
            Console.WriteLine("Account name: " + okAccount.Name);

            Console.WriteLine("\n----------------------");
            Console.WriteLine("BAD ACCOUNT BEHAVIOUR");
            Console.WriteLine("----------------------");

            Account badAccount = new Account("Mr Bad", -100); // Allows a negative
            ↪  balance

            badAccount.Print(); // Expect balance to be -$100.00
            badAccount.Deposit(100);
            badAccount.Print(); // Expect balance to be $0.00
            badAccount.Withdraw(1000000000); // Expect $1 billion overdrawn
            badAccount.Print();
            // badAccount.Name = "I'm really ok"; // Confirm read-only

            Console.WriteLine("\n--------------------");
            Console.WriteLine("ATTEMPTED BEHAVIOUR");
            Console.WriteLine("--------------------");

            Account terribleAccount = new Account("okAccount.Withdraw(1000);", 0);
            ↪  // Attempting to affect ok account
            terribleAccount.Print();
            okAccount.Print(); // Expect $400.00

            Console.WriteLine("\n************");
            Console.WriteLine("TESTING END");
            Console.WriteLine("************\n");

            Console.ReadLine();
        }
```

```
52      }
53  }
```

```cs
using System;

namespace Task_2._2P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get { return _name; } }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance)
        {
            _name = name;
            _balance = balance;  // !Allows negative initial balance
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public void Deposit(decimal amount)
        {
            _balance += amount;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <param name="amount">The amount to subtract from the balance</param>
        public void Withdraw(decimal amount)
        {
            _balance -= amount; // !Allows unlimited overdraw
        }

        /// <summary>
        /// Outputs the account name and current balance as a string
        /// </summary>
        public void Print()
        {
            Console.WriteLine("Account Name: {0}, Balance: {1}",
                _name, _balance.ToString("C"));
```

```
54            }
55        }
56    }
```

8 The MyTime Class

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

Outcome	**Weight**
Principles	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

Outcome	**Weight**
Build Programs	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

Outcome	**Weight**
Design	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

Outcome	**Weight**
Justify	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

Date	Author	Comment
2020/03/21 07:38	Peter Stacey	Quick question I am hoping you can help me with. The instructions ask us to validate the method inputs and to throw an IllegalArgumentException if needed. From the C# reference, that isn't a built in derived exception from ArgumentException (https://docs.microsoft.com/en-us/dotnet/api/system.argumentexception?view=netframework-4.8) but ArgumentOutOfRangeException is, and also seems to fit what we are trying to achieve. Is this OK to use, or do we specifically need to derive a new subclass of Exception and define IllegalArgumentException ourselves?
2020/03/21 07:55	Peter Stacey	FYI, I also asked this over on the forum just in case anyone else has already resolved this: https://d2l.deakin.edu.au/d2l/common/popup/popup.d2l?ou=882365&queryString=ou%3D882365%26postId%3D5690198%26topicId%3D634273%26isPopup%3D1%26actId%3D0%26viewIsPoppedOut%3DTrue&footerMsg=&popBodySrc=/d2l/lms/discussions/messageLists/message_preview.d2l&width=900&height=855&hasStatusBar=false&hasAutoScroll=true&hasHiddenHeader=false&p=d2l_cntl_e885ef042f7040fa82d5f3f52db3e316_1
2020/03/21 23:46	Dipto Pratyaksa	See now you can supply the constructors and methods with the right data type
2020/03/21 23:52	Dipto Pratyaksa	then if it is not matching, you throw ArgumentExceptionPlease read through: https://www.dotnetperls.com/argumentexception
2020/03/22 09:28	Peter Stacey	Ready to Mark
2020/03/22 20:33	Peter Stacey	Video link to come
2020/03/23 11:31	Dipto Pratyaksa	awaiting video ....
2020/03/23 11:31	Dipto Pratyaksa	Discuss
2020/03/23 14:18	Peter Stacey	Addition of one further test
2020/03/23 14:18	Peter Stacey	Video link: https://youtu.be/bavD2ylwOUg
2020/03/30 00:11	Peter Stacey	Is there anything further I need to do for this?
2020/04/01 12:23	Dipto Pratyaksa	Great work so far! see you in next task
2020/04/01 12:23	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

The MyTime Class

Submitted By:
Peter Stacey
pstacey
2020/03/23 14:18

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦♦♦◊◊
Justify	♦♦♦◊◊

This task involved broader reading of the C# reference and evaluation of different approaches to achieve an efficient implementation. Additionally, the task provided a set of specifications without providing any code, so the design needed to be developed as part of the task and then implemented in the time class and the associated testing. Defining a class for the time and implementing it in testing aligns well with the object oriented principles of the subject.

March 23, 2020

```
using System;

namespace Task_2._3C
{
    class TestMyTime
    {
        static void Main(string[] args)
        {
            int thrown = 0;

            Console.WriteLine("\n**************");
            Console.WriteLine("TESTING START");
            Console.WriteLine("**************");

            // ----------------------------------------------------------------
            // Class instantiation
            // ----------------------------------------------------------------
            MyTime time1 = new MyTime(); // Empty constructor
            MyTime time2 = new MyTime(10, 36, 45); // hour, min, sec

            try
            {
                MyTime time3 = new MyTime(100, 100, 100);
            }
            catch (ArgumentOutOfRangeException ex)
            {
                thrown++;
                Console.WriteLine("ERROR {0}: Constructor: {1}",
                    thrown, ex.Message);  // 1, expect to be thrown
            }

            /// ----------------------------------------------------------------
            // Property getters
            // ----------------------------------------------------------------
            Console.WriteLine("Property get time1 hour: {0}, min: {1}, sec {2}",
                time1.Hour, time1.Minute, time1.Second); // expect 0, 0 ,0
            Console.WriteLine("Property get time2 hour: {0}, min: {1}, sec {2}",
                time2.Hour, time2.Minute, time2.Second); // expect 10, 36, 45

            // ----------------------------------------------------------------
            // Property setters
            // ----------------------------------------------------------------

            // Reasonable values, no error expected
            try
            {
                time1.Hour = 10;
                time1.Minute = 10;
                time1.Second = 10;
            }
            catch (ArgumentOutOfRangeException ex)
            {
                thrown++;
```

```
54                  Console.WriteLine("Property Setters: Should not be thrown, {0}:
                    ↪ {1}",
55                      thrown, ex.Message);
56              }
57
58              // Invalid hour
59              try
60              {
61                  time1.Hour = 30;
62              }
63              catch (ArgumentOutOfRangeException ex)
64              {
65                  thrown++;
66                  Console.WriteLine("ERROR {0}: Hour Property: {1}",
67                      thrown, ex.Message); // 2, expect to be thrown
68              }
69
70              // Invalid minute
71              try
72              {
73                  time1.Minute = -15;
74              }
75              catch (ArgumentOutOfRangeException ex)
76              {
77                  thrown++;
78                  Console.WriteLine("ERROR {0}: Minute Property: {1}",
79                      thrown, ex.Message); // 3, expect to be thrown
80              }
81
82              // Invalid second
83              try
84              {
85                  // time1.Second = "thirty";  // syntax error
86                  // time1.Second = 13.5;      // syntax error
87                  time1.Second = 60;
88              }
89              catch (ArgumentOutOfRangeException ex)
90              {
91                  thrown++;
92                  Console.WriteLine("ERROR {0}: Second Property: {1}",
93                      thrown, ex.Message); // 4, expect to be thrown
94              }
95
96              // ------------------------------------------------------------------
97              // SetTime
98              // ------------------------------------------------------------------
99
100             // SetTime (reasonable values, no error expected)
101             try
102             {
103                 time2.SetTime(20, 15, 30);
104             }
105             catch (ArgumentOutOfRangeException ex)
```

```
106             {
107                 thrown++;
108                 Console.WriteLine("SetTime: Should not be thrown {0}: {1}",
109                     thrown, ex.Message);
110             }
111 
112             // SetTime (bad hour)
113             try
114             {
115                 time2.SetTime(24, 15, 30);
116             }
117             catch (ArgumentOutOfRangeException ex)
118             {
119                 thrown++;
120                 Console.WriteLine("ERROR {0}: SetTime: {1}", thrown, ex.Message);
                    ↪ // 5, expect to be thrown
121             }
122 
123             // SetTime (bad minute)
124             try
125             {
126                 time2.SetTime(20, 60, 30);
127             }
128             catch (ArgumentOutOfRangeException ex)
129             {
130                 thrown++;
131                 Console.WriteLine("ERROR {0}: SetTime: {1}", thrown, ex.Message);
                    ↪ // 6, expect to be thrown
132             }
133 
134             // SetTime (bad second)
135             try
136             {
137                 time2.SetTime(20, 15, 100);
138             }
139             catch (ArgumentOutOfRangeException ex)
140             {
141                 thrown++;
142                 Console.WriteLine("ERROR {0}: SetTime: {1}", thrown, ex.Message);
                    ↪ // 7, expect to be thrown
143             }
144 
145             // ------------------------------------------------------------------
146             // SetHour
147             // ------------------------------------------------------------------
148 
149             // Reasonable value, no error expected
150             try
151             {
152                 time1.SetHour(20);
153             }
154             catch (ArgumentOutOfRangeException ex)
155             {
```

```
156                 thrown++;
157                 Console.WriteLine("SetHour: Should not be thrown {0}: {1}",
158                     thrown, ex.Message);
159             }
160 
161             // SetTime (bad hour)
162             try
163             {
164                 time1.SetHour(70);
165             }
166             catch (ArgumentOutOfRangeException ex)
167             {
168                 thrown++;
169                 Console.WriteLine("ERROR {0}: SetHour: {1}", thrown, ex.Message);
                    ↪ // 8, expect to be thrown
170             }
171 
172             // ------------------------------------------------------------
173             // SetMinute
174             // ------------------------------------------------------------
175 
176             // Reasonable value, no error expected
177             try
178             {
179                 time1.SetMinute(20);
180             }
181             catch (ArgumentOutOfRangeException ex)
182             {
183                 thrown++;
184                 Console.WriteLine("SetMinute: Should not be thrown {0}: {1}",
185                     thrown, ex.Message);
186             }
187 
188             // SetTime (bad minute)
189             try
190             {
191                 time1.SetMinute(70);
192             }
193             catch (ArgumentOutOfRangeException ex)
194             {
195                 thrown++;
196                 Console.WriteLine("ERROR {0}: SetMinute: {1}", thrown, ex.Message);
                    ↪ // 9, expect to be thrown
197             }
198 
199             // ------------------------------------------------------------
200             // SetSecond
201             // ------------------------------------------------------------
202 
203             // Reasonable value, no error expected
204             try
205             {
206                 time1.SetSecond(20);
```

```
207             }
208             catch (ArgumentOutOfRangeException ex)
209             {
210                 thrown++;
211                 Console.WriteLine("SetSecond: Should not be thrown {0}: {1}",
212                     thrown, ex.Message);
213             }
214 
215             // SetTime (bad second)
216             try
217             {
218                 time1.SetSecond(70);
219             }
220             catch (ArgumentOutOfRangeException ex)
221             {
222                 thrown++;
223                 Console.WriteLine("ERROR {0}: SetSecond: {1}", thrown, ex.Message);
                    // 10, expect to be thrown
224             }
225 
226             // ------------------------------------------------------------
227             // GetHour, GetMinute, GetSecond
228             // ------------------------------------------------------------
229             time2.SetTime(10, 20, 30);
230 
231             int hour = time2.GetHour();
232             int minute = time2.GetMinute();
233             int second = time2.GetSecond();
234 
235             Console.WriteLine("GetHour, GetMinute, GetSecond: Expect 10:20:30 -
                {0}:{1}:{2}",
236                 hour, minute, second);
237 
238             // ------------------------------------------------------------
239             // ToString
240             // Also tests Format
241             // ------------------------------------------------------------
242             time1.SetTime(1, 2, 3);
243             Console.WriteLine("ToString: Expect 01:02:03 - Result {0}",
                time1.ToString());
244 
245             time2.SetTime(10, 5, 35);
246             Console.WriteLine("ToString: Expect 10:05:35 - Result {0}",
                time2.ToString());
247 
248             // ------------------------------------------------------------
249             // NextSecond, NextMinute, NextHour
250             // Also tests Format
251             // ------------------------------------------------------------
252             time1.SetTime(23, 59, 55);
253 
254             Console.WriteLine();
255             for (int i = 0; i < 10; i++)
```

```
256             {
257                 time1.NextSecond();
258                 Console.WriteLine(time1.ToString());
259             }
260 
261             // --------------------------------------------------------------
262             // PreviousSecond, PreviousMinute, PreviousHour
263             // Also tests Format
264             // --------------------------------------------------------------
265             Console.WriteLine();
266             for (int i = 0; i < 10; i++)
267             {
268                 time1.PreviousSecond();
269                 Console.WriteLine(time1.ToString());
270             }
271 
272             Console.WriteLine("\n***********");
273             Console.WriteLine("TESTING END");
274             Console.WriteLine("***********\n");
275         }
276     }
277 }
```

```cs
using System;

namespace Task_2._3C
{
    class MyTime
    {
        // Instance variables
        private int _hour;
        private int _minute;
        private int _second;

        /// Reference for this approach, from:
        /// https://stackoverflow.com/questions/56197825
        public int Hour
        {
            get => _hour;
            set => _hour = (value >= 0) && (value <= 23)
                ? value
                : throw new ArgumentOutOfRangeException("Invalid hour. Must be
                ↪  0-23");
        }

        public int Minute
        {
            get => _minute;
            set => _minute = (value >= 0) && (value <= 59)
                ? value
                : throw new ArgumentOutOfRangeException("Invalid minute. Must be
                ↪  0-59");
        }

        public int Second
        {
            get => _second;
            set => _second = (value >= 0) && (value <= 59)
                ? value
                : throw new ArgumentOutOfRangeException("Invalid second. Must be
                ↪  0-59");
        }

        /// <summary>
        /// Constructor to create new time with 0 values
        /// </summary>
        public MyTime() { }

        /// <summary>
        /// Constructor to create a time with hour, minute and second
        /// </summary>
        /// <param name="hour">Hour in the range 0-23</param>
        /// <param name="minute">Minute in the range 0-59</param>
        /// <param name="second">Second in the range 0-59</param>
        /// <exception cref="System.ArgumentOutOfRangeException">Thrown
        /// when one of the parameters is outside the range</exception>
```

```
51          public MyTime(int hour, int minute, int second)
52          {
53              Hour = hour;
54              Minute = minute;
55              Second = second;
56          }
57
58          /// <summary>
59          /// Sets a time with hour, minute and second
60          /// </summary>
61          /// <param name="hour">Hour in the range 0-23</param>
62          /// <param name="minute">Minute in the range 0-59</param>
63          /// <param name="second">Second in the range 0-59</param>
64          /// <exception cref="System.ArgumentOutOfRangeException">Thrown
65          /// when one of the parameters is outside the range</exception>
66          public void SetTime(int hour, int minute, int second)
67          {
68              Hour = hour;
69              Minute = minute;
70              Second = second;
71          }
72
73          /// <summary>
74          /// Sets the hour of a time
75          /// </summary>
76          /// <param name="hour">Hour in the range 0-23</param>
77          /// <exception cref="System.ArgumentOutOfRangeException">Thrown
78          /// when hour is outside the range 0-23</exception>
79          public void SetHour(int hour)
80          {
81              Hour = hour;
82          }
83
84          /// <summary>
85          /// CSets the minute of a time
86          /// </summary>
87          /// <param name="minute">Minute in the range 0-59</param>
88          /// <exception cref="System.ArgumentOutOfRangeException">Thrown
89          /// when minute is outside the range 0-59</exception>
90          public void SetMinute(int minute)
91          {
92              Minute = minute;
93          }
94
95          /// <summary>
96          /// Sets the second of a time
97          /// </summary>
98          /// <param name="second">Second in the range 0-59</param>
99          /// <exception cref="System.ArgumentOutOfRangeException">Thrown
100         /// when second is outside the range 0-59</exception>
101         public void SetSecond(int second)
102         {
103             Second = second;
```

```
104         }
105
106         /// <summary>
107         /// Gets the hour of a time
108         /// </summary>
109         /// <returns>
110         /// The hour of the time
111         /// </returns>
112         public int GetHour()
113         {
114             return _hour;
115         }
116
117         /// <summary>
118         /// Gets the minute of a time
119         /// </summary>
120         /// <returns>
121         /// The minute of the time
122         /// </returns>
123         public int GetMinute()
124         {
125             return _minute;
126         }
127
128         /// <summary>
129         /// Gets the second of a time
130         /// </summary>
131         /// <returns>
132         /// The second of the time
133         /// </returns>
134         public int GetSecond()
135         {
136             return _second;
137         }
138
139         /// <summary>
140         /// Formats a number to two digits with leading 0 if needed
141         /// </summary>
142         /// <returns>
143         /// The formatted string
144         /// </returns>
145         /// <param name="value">The integer to format as a 2-digit string</param>
146         private String Format(int value)
147         {
148             return value < 10 ? "0" + value : value.ToString();
149         }
150
151         public override String ToString()
152         {
153             return Format(_hour) + ":" + Format(_minute) + ":" + Format(_second);
154         }
155
156         /// <summary>
```

```
157         /// Advances a time by 1 second
158         /// </summary>
159         public MyTime NextSecond()
160         {
161             try
162             {
163                 Second += 1;
164             }
165             catch (ArgumentOutOfRangeException)
166             {
167                 Second = 0;
168                 NextMinute();
169             }
170             return this;
171         }
172 
173         /// <summary>
174         /// Advances a time by 1 minute
175         /// </summary>
176         public MyTime NextMinute()
177         {
178             try
179             {
180                 Minute += 1;
181             }
182             catch (ArgumentOutOfRangeException)
183             {
184                 Minute = 0;
185                 NextHour();
186             }
187             return this;
188         }
189 
190         /// <summary>
191         /// Advances a time by 1 hour
192         /// </summary>
193         public MyTime NextHour()
194         {
195             try
196             {
197                 Hour += 1;
198             }
199             catch (ArgumentOutOfRangeException)
200             {
201                 Hour = 0;
202             }
203             return this;
204         }
205 
206         /// <summary>
207         /// Reduces a time by 1 second
208         /// </summary>
209         public MyTime PreviousSecond()
```

```
210         {
211             try
212             {
213                 Second -= 1;
214             }
215             catch (ArgumentOutOfRangeException)
216             {
217                 Second = 59;
218                 PreviousMinute();
219             }
220             return this;
221         }
222 
223         /// <summary>
224         /// Reduces a time by 1 minute
225         /// </summary>
226         public MyTime PreviousMinute()
227         {
228             try
229             {
230                 Minute -= 1;
231             }
232             catch (ArgumentOutOfRangeException)
233             {
234                 Minute = 59;
235                 PrevioustHour();
236             }
237             return this;
238         }
239 
240         /// <summary>
241         /// Reduces a time by 1 hour
242         /// </summary>
243         public MyTime PrevioustHour()
244         {
245             try
246             {
247                 Hour -= 1;
248             }
249             catch (ArgumentOutOfRangeException)
250             {
251                 Hour = 23;
252             }
253             return this;
254         }
255     }
256 }
```

9 C# Essentials: Arrays and Lists

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦♦◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

Outcome	Weight
Principles	♦♦♦◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

Outcome	Weight
Build Programs	♦♦♦◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

Outcome	Weight
Design	♦♦◊◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

Outcome	Weight
Justify	♦♦♦◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

Date	Author	Comment
2020/03/24 17:46	Peter Stacey	Ready to Mark
2020/03/24 17:47	Peter Stacey	Video link to come
2020/03/24 17:48	Peter Stacey	I am not sure that I have solved question 9 correctly, so will check in the forum (the second sentence of the description doesn't quite make sense to me). If I need to change it, I'll upload a modified version.
2020/03/25 11:28	Dipto Pratyaksa	It's about mapping 2-dimenstional array into one-dimensional array... think of it like your multiplication table... except this one is filtered by a criteria.. and that criteria is factor of 3.., in other words, you only need to store integers that are divisible by 3
2020/03/25 11:29	Dipto Pratyaksa	"Multiple 3" does not make sense to me, but I'm pretty sure it meant factor of 3
2020/03/25 11:29	Dipto Pratyaksa	Discuss
2020/03/25 11:32	Dipto Pratyaksa	Multiples of 3 includes 3 .. 6... 9... 12.. and so on.. so you only to transfer these numbers into your one-dimensional array
2020/03/25 12:41	Peter Stacey	yeah thanks. I'll modify it to just store the numbers that are multiples of 3, instead of all as it does currently
2020/03/26 08:29	Peter Stacey	Fixed up code for question 9, to only add in multiples of 3 and to also search the columns before the rows (ie. move down a column and then across, not across a row and then down)
2020/03/26 18:10	Peter Stacey	Video link: https://youtu.be/JxpZBAssewo
2020/03/26 18:59	Dipto Pratyaksa	Excellent effort. Well done
2020/03/26 18:59	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Arrays and Lists

Submitted By:
Peter Stacey
pstacey
2020/03/26 08:29

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦♦◊◊◊
Justify	♦♦♦◊◊

This task has a large number of subtasks all related to arrays and lists, with several problems to work through, with their use. This includes a number of pieces of code to evaluate and also to write up using correct conventions. Additionally, there are multiple programs to solve and code, resulting in a significant amount of programming to be completed. The task contains designed pieces of code and my video has further diagrams and descriptions. Between the submitted csharp files and video, evidence is provided against each of the criteria of the task.

March 26, 2020

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Task_3._1P
{
    class Program
    {
        static void Main(string[] args)
        {
            // ***************************************************************
            // PART 1: STEP 1
            // ***************************************************************
            // declares an array of type double with 10 elements
            double[] myArray = new double[10];

            // assigning the first element of the array
            myArray[0] = 1.0;

            // assigning the second element of the array
            myArray[1] = 1.1;

            // assigning the third element of the array
            myArray[2] = 1.2;

            // assigning the fourth element of the array
            myArray[3] = 1.3;

            // assigning the fifth element of the array
            myArray[4] = 1.4;

            // assigning the sixth element of the array
            myArray[5] = 1.5;

            // assigning the seventh element of the array
            myArray[6] = 1.6;

            // assigning the eighth element of the array
            myArray[7] = 1.7;

            // assigning the ninth element of the array
            myArray[8] = 1.8;

            // assigning the tenth element of the array
            myArray[9] = 1.9;

            // ***************************************************************
            // PART 1: STEP 2
            // ***************************************************************
            Console.WriteLine("The element at index 0 in the array is " +
            ↪  myArray[0]);
            Console.WriteLine("The element at index 1 in the array is " +
            ↪  myArray[1]);
```

```
52              Console.WriteLine("The element at index 2 in the array is " +
                ↪  myArray[2]);
53              Console.WriteLine("The element at index 3 in the array is " +
                ↪  myArray[3]);
54              Console.WriteLine("The element at index 4 in the array is " +
                ↪  myArray[4]);
55              Console.WriteLine("The element at index 5 in the array is " +
                ↪  myArray[5]);
56              Console.WriteLine("The element at index 6 in the array is " +
                ↪  myArray[6]);
57              Console.WriteLine("The element at index 7 in the array is " +
                ↪  myArray[7]);
58              Console.WriteLine("The element at index 8 in the array is " +
                ↪  myArray[8]);
59              Console.WriteLine("The element at index 9 in the array is " +
                ↪  myArray[9]);
60
61              // ********************************************************************
62              // PART 2: STEP 1
63              // ********************************************************************
64              int[] myIntArray = new int[10];
65
66              for (int i = 0; i < myIntArray.Length; i++)
67              {
68                  myIntArray[i] = i;
69              }
70
71              // ********************************************************************
72              // PART 2: STEP 2
73              // ********************************************************************
74              for (int i = 0; i < myIntArray.Length; i++)
75              {
76                  Console.WriteLine("The element at position {0} is {1}",
77                      i, myIntArray[i]);
78              }
79
80              // ********************************************************************
81              // PART 3
82              // ********************************************************************
83              int[] studentArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
84              int total = 0;
85
86              for (int i = 0; i < studentArray.Length; i++)
87              {
88                  total += studentArray[i];
89              }
90
91              Console.WriteLine("The total marks for the studen is " + total);
92              Console.WriteLine("This consists of " + studentArray.Length + " marks");
93              Console.WriteLine("Therefore the average mark is "
94                  + (total / studentArray.Length));
95
96              // ********************************************************************
```

```
97              // PART 4
98              // ***************************************************************
99              String[] studentNames = new String[6];
100
101             for (int i = 0; i < studentNames.Length; i++)
102             {
103                 Console.Write("Student {0} name: ", i + 1);
104                 studentNames[i] = Console.ReadLine();
105             }
106
107             for (int i = 0; i < studentNames.Length; i++)
108             {
109                 Console.WriteLine("Student {0}: {1}", i + 1, studentNames[i]);
110             }
111
112             // ***************************************************************
113             // PART 5
114             // ***************************************************************
115             double[] values = new double[10];
116             double currentLargest, currentSmallest;
117
118             for (int i = 0; i < values.Length; i++)
119             {
120                 Console.Write("Enter a double for position {0}: ", i);
121                 String input = Console.ReadLine();
122                 // Note - no error checking. Expects valid input only
123                 values[i] = Convert.ToDouble(input);
124             }
125
126             currentLargest = values[0];
127
128             for (int i = 0; i < values.Length; i++)
129             {
130                 if (values[i] > currentLargest)
131                     currentLargest = values[i];
132                 Console.WriteLine(values[i]);
133             }
134
135             Console.WriteLine("The largest value is " + currentLargest);
136
137             currentSmallest = values[0];
138
139             for (int i = 0; i < values.Length; i++)
140             {
141                 if (values[i] < currentSmallest)
142                     currentSmallest = values[i];
143             }
144
145             Console.WriteLine("The smallest value is " + currentSmallest);
146
147             // ***************************************************************
148             // PART 6
149             // ***************************************************************
```

```
150             int[,] myMultiArray = new int[3, 4] { { 1, 2, 3, 4 }, { 1, 1, 1, 1 }, {
                ↪  2, 2, 2, 2 } };
151
152             for (int i = 0; i < myMultiArray.GetLength(0); i++)
153             {
154                 for (int j = 0; j < myMultiArray.GetLength(1); j++)
155                 {
156                     Console.Write(myMultiArray[i, j] + "\t");
157                 }
158                 Console.WriteLine();
159             }
160
161             List<String> myStudentList = new List<string>();
162
163             Random randomValue = new Random();
164             int randomNumber = randomValue.Next(1, 12);
165
166             Console.WriteLine("You now need to add all " + randomNumber
167                 + " students to your class list");
168
169             for (int i = 0; i < randomNumber; i++)
170             {
171                 Console.Write("Please enter the name of Student " + (i + 1) + ": ");
172                 myStudentList.Add(Console.ReadLine());
173                 Console.WriteLine();
174             }
175
176             // ********************************************************************
177             // PART 7
178             // ********************************************************************
179             int FuncOne(int[] values)
180             {
181                 if (values.Length <= 10)
182                 {
183                     return GetOddProduct(values);
184                 }
185                 else
186                 {
187                     return NumberOfEvens(values);
188                 }
189             }
190
191             int GetOddProduct(int[] values)
192             {
193                 int oddProduct = 1;
194                 for (int i = 0; i < values.Length; i++)
195                 {
196                     if (values[i] % 2 == 1)
197                         oddProduct *= values[i];
198                 }
199                 return oddProduct;
200             }
201
```

```
202             int NumberOfEvens(int[] values)
203             {
204                 int numberOfEvens = 0;
205                 for (int i = 0; i < values.Length; i++)
206                 {
207                     if (values[i] % 2 == 0)
208                         numberOfEvens++;
209                 }
210                 return numberOfEvens;
211             }
212
213
214             int[] numArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
215
216             // Expect 9 elements, odd product = 1*3*5*7*9=945
217             Console.WriteLine("Result from FuncOne for array of {0} elements: {1}",
218                 numArray.Length, FuncOne(numArray));
219
220             int[] numArray2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
221
222             // Expect 12 elements, number of evens = 2,4,6,8,10,12 = 6
223             Console.WriteLine("Result from FuncOne for array of {0} elements: {1}",
224                 numArray2.Length, FuncOne(numArray2));
225
226             // ****************************************************************
227             // PART 8
228             // ****************************************************************
229             void FuncTwo(List<double> values)  // list is passed in by reference,
                ↪  so we can modify it
230             {
231                 double sum = values.Sum();
232                 double average = sum / values.Count;
233                 for (int i = 0; i < values.Count; i++)
234                 {
235                     values[i] -= average;
236                 }
237                 return;
238             }
239
240             List<double> myList = new List<double>() { 1.0, 2.0, 3.0, 4.0, 5.0 };
241
242             for (int i = 0; i < myList.Count; i++)
243             {
244                 Console.Write(myList[i] + "\t");
245             }
246             Console.WriteLine();
247
248             FuncTwo(myList);
249
250             for (int i = 0; i < myList.Count; i++)
251             {
252                 Console.Write(myList[i] + "\t");
253             }
```

```
254             Console.WriteLine();
255
256             // ****************************************************************
257             // PART 9
258             // ****************************************************************
259             int[] FuncThree(int[,] values)
260             {
261                 List<int> result = new List<int>();
262                 for (int j = 0; j < values.GetLength(1); j++)
263                 {
264                     for (int i = 0; i < values.GetLength(0); i++)
265                     {
266                         if (values[i, j] % 3 == 0)
267                             result.Add(values[i, j]);
268                     }
269                 }
270                 return result.ToArray();
271             }
272
273             int[,] myMulti = { { 1,  2,  3,  4 , 5 },
274                                { 6,  7,  8,  9, 10 },
275                                {11, 12, 13, 14, 15 } };
276
277             int[] mySingle = FuncThree(myMulti);
278
279             for (int i = 0; i < mySingle.Length; i++)
280             {
281                 // Expected result 6, 12, 3, 9, 15
282                 Console.WriteLine(mySingle[i]);
283             }
284
285             // ****************************************************************
286             // PART 10
287             // ****************************************************************
288             int[,] FuncFour(int[] values)
289             {
290                 int[,] result = new int[values.Length, 10];
291                 for (int i = 0; i < values.Length; i++)
292                 {
293                     for (int j = 1; j <= 10; j++)
294                     {
295                         result[i, j - 1] = values[i] * j;
296                     }
297                 }
298                 return result;
299             }
300
301             int[] intArray = { 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10 };
302             int[,] result = FuncFour(intArray);
303
304             Console.WriteLine("  \t|1\t2\t3\t4\t5\t6\t7\t8\t9\t10");
305             Console.WriteLine("-------------------------------------------------
                 ↪  --------------------------------");
```

```
306             for (int i = 0; i < result.GetLength(0); i++)
307             {
308                 Console.Write(intArray[i] + "\t|");
309                 for (int j = 0; j < 10; j++)
310                 {
311                     Console.Write(result[i, j] + "\t");
312                 }
313                 Console.WriteLine();
314             }
315             Console.WriteLine();
316         }
317     }
318 }
```

10 Validating Accounts

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

Outcome	**Weight**
Principles	♦♦◊◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

Outcome	**Weight**
Build Programs	♦♦♦◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

Outcome	**Weight**
Design	♦♦♦◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

Outcome	**Weight**
Justify	♦♦♦◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To

implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

Date	Author	Comment
2020/03/21 08:40	Peter Stacey	Ready to Mark
2020/03/21 08:40	Peter Stacey	Video link to come
2020/03/25 11:33	Dipto Pratyaksa	Take your time, you are 2 weeks ahead of most people
2020/03/29 19:43	Peter Stacey	video link: https://youtu.be/HL19a_Di6Dw
2020/04/01 12:27	Dipto Pratyaksa	:+1: impressive work, you are still ahead
2020/04/01 12:27	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Validating Accounts

Submitted By:
Peter Stacey
pstacey
2020/03/21 08:40

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦◊◊◊
Principles	♦♦◊◊◊
Build Programs	♦♦♦◊◊
Design	♦♦♦◊◊
Justify	♦♦♦◊◊

The task involves evaluating both the existing code and the new requirements, to design additional features and changes to the existing design from task 2.2. Additionally, it involves applying the principles of object oriented programming, and specifically encapsulation - by hiding the implementation of the underlying types in private variables with a public interface for users of the program. To implement the changes, there is a reasonable amount of new code, in addition to a small number of code changes and between the submitted code and the video I will link, this aligns well with meeting the outcomes of the subject.

March 21, 2020

```
using System;

namespace Task_3._2P
{
    enum MenuOption
    {
        Withdraw,
        Deposit,
        Print,
        Quit
    }

    class BankSystem
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

        // Reads integer input in the console between two numbers
        /// <summary>
        /// Reads integer input in the console between two numbers
        /// </summary>
        /// <returns>
```

```
54          /// The input of the user as an integer
55          /// </returns>
56          /// <param name="prompt">The string prompt for the user</param>
57          /// <param name="minimum">The minimum number allowed</param>
58          /// <param name="maximum">The maximum number allowed</param>
59          public static int ReadInteger(String prompt, int minimum, int maximum)
60          {
61              int number = ReadInteger(prompt);
62              while (number < minimum || number > maximum)
63              {
64                  Console.WriteLine("Please enter a whole number from " +
65                                    minimum + " to " + maximum);
66                  number = ReadInteger(prompt);
67              }
68              return number;
69          }
70
71          // Reads decimal input in the console
72          /// <summary>
73          /// Reads decimal input in the console
74          /// </summary>
75          /// <returns>
76          /// The input of the user as a decimal
77          /// </returns>
78          /// <param name="prompt">The string prompt for the user</param>
79          public static decimal ReadDecimal(String prompt)
80          {
81              decimal number = 0;
82              string numberInput = ReadString(prompt);
83              while (!(decimal.TryParse(numberInput, out number)))
84              {
85                  Console.WriteLine("Please enter a decimal number");
86                  numberInput = ReadString(prompt);
87              }
88              return Convert.ToDecimal(numberInput);
89          }
90
91          /// <summary>
92          /// Displays a menu of possible actions for the user to choose
93          /// </summary>
94          private static void DisplayMenu()
95          {
96              Console.WriteLine("\n**********************");
97              Console.WriteLine("*       Menu         *");
98              Console.WriteLine("**********************");
99              Console.WriteLine("*  1. Withdraw       *");
100             Console.WriteLine("*  2. Deposit        *");
101             Console.WriteLine("*  3. Print          *");
102             Console.WriteLine("*  4. Quit           *");
103             Console.WriteLine("**********************");
104         }
105
106         private static void DisplayResult(MenuOption action, Boolean result)
```

```
107         {
108             String output = action + " "
109                 + (result == true ? "succeeded" : "failed. Invalid amount.");
110             Console.WriteLine(output);
111         }
112 
113         /// <summary>
114         /// Returns a menu option chosen by the user
115         /// </summary>
116         /// <returns>
117         /// MenuOption chosen by the user
118         /// </returns>
119         static MenuOption ReadUserOption()
120         {
121             DisplayMenu();
122             int option = ReadInteger("Choose an option", 1,
123                 Enum.GetNames(typeof(MenuOption)).Length);
124             return (MenuOption)(option - 1);
125         }
126 
127         /// <summary>
128         /// Attempts to deposit funds into an account
129         /// </summary>
130         /// <param name="account">The account to deposit into</param>
131         static void DoDeposit(Account account)
132         {
133             decimal amount = ReadDecimal("Enter the amount");
134             bool result = account.Deposit(amount);
135             DisplayResult(MenuOption.Deposit, result);
136         }
137 
138         /// <summary>
139         /// Attempts to withdraw funds from an account
140         /// </summary>
141         /// <param name="account">The account to withdraw from</param>
142         static void DoWithdraw(Account account)
143         {
144             decimal amount = ReadDecimal("Enter the amount");
145             Boolean result = account.Withdraw(amount);
146             DisplayResult(MenuOption.Withdraw, result);
147         }
148 
149         /// <summary>
150         /// Outputs the account name and balance
151         /// </summary>
152         /// <param name="account">The account to print</param>
153         static void DoPrint(Account account)
154         {
155             account.Print();
156         }
157 
158         static void Main(string[] args)
159         {
```

```
160             Account acc = new Account("Peter Stacey", -100);
161             acc.Print(); // confirm balance not set to negative
162 
163             do
164             {
165                 MenuOption chosen = ReadUserOption();
166                 switch (chosen)
167                 {
168                     case MenuOption.Withdraw:
169                         DoWithdraw(acc); break;
170                     case MenuOption.Deposit:
171                         DoDeposit(acc); break;
172                     case MenuOption.Print:
173                         DoPrint(acc); break;
174                     case MenuOption.Quit:
175                     default:
176                         Console.WriteLine("Goodbye");
177                         System.Environment.Exit(0); // terminates the program
178                         break; // unreachable
179                 }
180             } while (true);
181         }
182     }
183 }
```

```cs
using System;

namespace Task_3._2P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get { return _name; } }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance <= 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if (amount <= 0)
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The amount to subtract from the balance</param>
```

```
54          public Boolean Withdraw(decimal amount)
55          {
56              if ((amount <= 0) || (amount > _balance))
57                  return false;
58
59              _balance -= amount;
60              return true;
61          }
62
63          /// <summary>
64          /// Outputs the account name and current balance as a string
65          /// </summary>
66          public void Print()
67          {
68              Console.WriteLine("Account Name: {0}, Balance: {1}",
69                  _name, _balance.ToString("C"));
70          }
71      }
72  }
```

11 The MyPolynomial class

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

Outcome	**Weight**
Principles	♦♦♦◊◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

Outcome	**Weight**
Build Programs	♦♦♦♦◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

Outcome	**Weight**
Design	♦♦♦◊◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

Outcome	**Weight**
Justify	♦♦♦◊◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

Date	Author	Comment
2020/03/30 00:05	Peter Stacey	Ready to Mark
2020/03/30 00:05	Peter Stacey	video link to come
2020/03/30 00:07	Peter Stacey	This was a pretty challenging task, especially trying to work through the ToString method and ensure it covered all possibilities. Felt more like a distinction task than a credit task
2020/03/31 11:02	Peter Stacey	Ready to Mark
2020/03/31 11:02	Peter Stacey	updated testing and resolved one issue I identified through more testing. Video now being recorded
2020/03/31 14:42	Peter Stacey	Video link: https://youtu.be/2ILND7SROjk
2020/04/01 12:31	Dipto Pratyaksa	You are a head for about a month... very good effort!I like the way you format the string.. looks very complicated.You could break it up into a few variables of strings that you concatenate at the end
2020/04/01 12:31	Dipto Pratyaksa	Discuss
2020/04/25 19:13	Peter Stacey	Do I need to do anything more to have this task marked off, or is my current submission ok?
2020/04/26 15:41	Sergey Polyakovskiy	Looks good.
2020/04/26 15:41	Sergey Polyakovskiy	Complete

Deakin University

Object Oriented Development

OnTrack Submission

The MyPolynomial class

Submitted By:
Peter Stacey
pstacey
2020/03/31 11:02

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦♦◊
Design	♦♦♦◊◊
Justify	♦♦♦◊◊

This task is far more complex than all earlier tasks, requiring close attention to the use of correct conventions to be able to read and understand the code, significant time (2-3 hours) to design the solution before writing any code, the research of additional concepts to translate them into code (eg. adding and multiplying polynomials), and the addition of comments that add explanation for some coding choices. Overall, a really challenging and good task

March 31, 2020

```
using System;

namespace Task_3._3C
{
    class TestMyPolynomial
    {
        public static String CoeffsToString(MyPolynomial poly)
        {
            String result = "[";
            for (int i = 0; i < poly.GetDegree(); i++)
            {
                result += poly.CoeffAt(i) + ", ";
            }
            result += poly.CoeffAt(poly.GetDegree()) + "]:  ";
            return result;
        }

        public static MyPolynomial RandomPolynomial(Random rnd, int min, int max)
        {
            double[] coeffs = new double[rnd.Next(1, max)];
            int num = 0;
            for (int j = 0; j < coeffs.Length; j++)
            {
                num = rnd.Next(min, max);
                if (num % 7 == 0)
                    num = 0;
                coeffs[j] = Convert.ToDouble(num);
            }
            if (coeffs[coeffs.Length - 1] == 0)
                coeffs[coeffs.Length - 1] = 7;
            return new MyPolynomial(coeffs);
        }
        public static void Main(string[] args)
        {
            int numberOfTests = 50;

            // Test ToString by creating 100 random sets of arrays and
            // creating polynomials for each one and printing via
            // ToString
            Console.WriteLine("\n\nTesting ToString:\n");
            Random rnd = new Random();

            for (int i = 0; i < numberOfTests; i++)
            {
                MyPolynomial poly = RandomPolynomial(rnd, -20, 20);
                Console.WriteLine("\n" + CoeffsToString(poly));
                Console.WriteLine(poly.ToString());
            }

            // Test evaluate
            Console.WriteLine("\n\nTesting evaluating a polynomial");

            for (int i = 0; i < numberOfTests; i++)
```

```
54              {
55
56                  double x = rnd.Next(-10, 10);
57                  MyPolynomial poly2 = RandomPolynomial(rnd, -10, 10);
58                  double result = poly2.Evaluate(x);
59                  Console.WriteLine("\n({0}), {1} = {2}", poly2.ToString(), x,
                    ↪  result);
60              }
61
62              // Test adding polynomials
63              Console.WriteLine("\n\nTesting adding polynomials:\n");
64
65              for (int i = 0; i < numberOfTests; i++)
66              {
67                  MyPolynomial poly5 = RandomPolynomial(rnd, -5, 5);
68                  MyPolynomial poly6 = RandomPolynomial(rnd, -5, 5);
69                  Console.Write("\n({0}) + ({1}) = ", poly5.ToString(),
                    ↪  poly6.ToString());
70                  poly5.Add(poly6);
71                  Console.WriteLine(poly5.ToString());
72              }
73
74              // Test multiplying polynomials
75              Console.WriteLine("\n\nTesting multiplying polynomials:\n");
76
77              for (int i = 0; i < numberOfTests; i++)
78              {
79                  MyPolynomial poly5 = RandomPolynomial(rnd, -5, 5);
80                  MyPolynomial poly6 = RandomPolynomial(rnd, -5, 5);
81                  Console.Write("\n({0}) x ({1}) = ", poly5.ToString(),
                    ↪  poly6.ToString());
82                  poly5.Multiply(poly6);
83                  Console.WriteLine(poly5.ToString());
84              }
85          }
86      }
87  }
```

```cs
using System;

namespace Task_3._3C
{
    class MyPolynomial
    {
        // Instance variables
        private double[] _coeffs;

        /// <summary>
        /// Constructor for a polynomial
        /// </summary>
        /// <param name="coeffs">Double array of coefficients</summary>
        public MyPolynomial(double[] coeffs)
        {
            _coeffs = coeffs;
        }

        /// <summary>
        /// Returns the degree of the polynomial
        /// </summary>
        /// <returns>
        /// The degree of the polynomial as an integer
        /// </returns>
        public int GetDegree()
        {
            return _coeffs.Length - 1;
        }

        /// <summary>
        /// Returns a common string representation of a polynomial
        /// </summary>
        /// <returns>
        /// String representation of an expanded polynomial
        /// </returns>
        public override String ToString()
        {
            String result = "";
            String op, exp;
            double num;
            for (int i = _coeffs.Length - 1; i >= 0; i--)
            {
                num = _coeffs[i];
                // If num is less than 0, add " - " to the string, otherwise
                // add " + " as long as this isn't the start of the string
                // Example:
                // - 4x^2 + 2x - 2 (print minus sign at start)
                // 4x^2 + 2x - 2 (don't print plus sign at start)
                op = num < 0 ? "- " : (i < _coeffs.Length - 1 && num > 0) ? "+ " :
                ↪  "";
                num = Math.Abs(num);
                exp = (i == 0 && num != 0) ? num.ToString()             // If
                ↪  constant and not 0, then concatenate to string
```

```
52                      : (i == 1 && num == 1) ? "x "                          // if 1x^1,
                        ↪ don't print 1 or ^1, just x
53                      : (i == 1 && num != 0) ? num.ToString() + "x "         // if ax^1,
                        ↪ don't print ^1, just ax (a not 0)
54                      : (i > 1 && num == 1) ? $"x^{i} "                       // don't
                        ↪ print 1 before any number, but include coefficient if > 1
55                      : (i > 1 && num != 0) ? num.ToString() + $"x^{i} "  //
                        ↪ otherwise if not 0, print ax^coeff
56                      : "";                                                  // if 0,
                        ↪ add nothing to the string
57                  result += (op + exp);                                      // add the
                    ↪ operator and the exponent part
58              }
59              return result;
60          }
61
62          /// <summary>
63          /// Calculates the value of f(x) for a given value of x
64          /// </summary>
65          /// <returns>
66          /// Double value of substituting x into the polynomial
67          /// </returns>
68          public double Evaluate(double x)
69          {
70              double result = _coeffs[0];
71              for (int i = 1; i < _coeffs.Length; i++)
72              {
73                  result += _coeffs[i] * Math.Pow(x, i);
74              }
75              return result;
76          }
77
78          /// <summary>
79          /// Returns the value of the coefficent at the given index position
80          /// </summary>
81          /// <returns>
82          /// Double value of the coefficient at the given index
83          /// </returns>
84          /// <exception cref="System.IndexOutOfRangeException">Thrown when index
85          /// is larger than the size of the coefficients array</exception>
86          public double CoeffAt(int index)
87          {
88              try
89              {
90                  return _coeffs[index];
91              }
92              catch (IndexOutOfRangeException)
93              {
94                  throw new IndexOutOfRangeException(
95                          "Index " + index
96                          + " not valid for a polynomial of length "
97                          + _coeffs.Length);
98              }
```

```
99          }
100
101         /// <summary>
102         /// Adds another polynomial to this
103         /// </summary>
104         /// <returns>
105         /// this polynomial
106         /// </returns>
107         /// <param name="other">The polynomial to add</param>
108         public MyPolynomial Add(MyPolynomial other)
109         {
110             double[] result;
111             bool longer = this.GetDegree() >= other.GetDegree();
112             if (longer)
113                 result = new double[_coeffs.Length];
114             else
115                 result = new double[other.GetDegree() + 1];
116             for (int i = 0; i < result.Length; i++)
117             {
118                 try
119                 {
120                     result[i] = _coeffs[i] += other.CoeffAt(i);
121                 }
122                 catch (IndexOutOfRangeException)
123                 {
124                     if (longer)
125                         result[i] = _coeffs[i];
126                     else
127                         result[i] = other.CoeffAt(i);
128                 }
129             }
130             _coeffs = result;
131             return this;
132         }
133
134         /// <summary>
135         /// Multiplies this by another polynomial
136         /// </summary>
137         /// <returns>
138         /// this polynomial
139         /// </returns>
140         /// <param name="other">The polynomial to multiply by</param>
141         public MyPolynomial Multiply(MyPolynomial other)
142         {
143             double[] product = new double[_coeffs.Length + other.GetDegree()];
144             for (int i = 0; i < _coeffs.Length; i++)
145             {
146                 for (int j = 0; j < other.GetDegree() + 1; j++)
147                 {
148                     product[i + j] += _coeffs[i] * other.CoeffAt(j);
149                 }
150             }
151             _coeffs = product;
```

```
152                 return this;
153             }
154         }
155 }
```

12 Bucket Sort

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦♦◊◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

Outcome	Weight
Principles	♦♦♦◊◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

Outcome	Weight
Build Programs	♦♦♦♦◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

Outcome	Weight
Design	♦♦♦◊◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

Outcome	Weight
Justify	♦♦♦♦◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

Date	**Author**	**Comment**
2020/04/04 19:04	Peter Stacey	Ready to Mark
2020/04/04 19:04	Peter Stacey	Video link to follow
2020/04/05 11:22	Peter Stacey	Ready to Mark
2020/04/05 11:23	Peter Stacey	Changed the testing slightly and the maximum balance method
2020/04/05 14:39	Peter Stacey	Video Link: https://youtu.be/voWpmEtdJCg
2020/04/09 23:12	Dipto Pratyaksa	impressive work
2020/04/09 23:12	Dipto Pratyaksa	well done for your great effort
2020/04/09 23:12	Dipto Pratyaksa	Demonstrate
2020/04/09 23:12	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Bucket Sort

Submitted By:
Peter Stacey
pstacey
2020/04/05 11:22

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦♦◊
Design	♦♦♦◊◊
Justify	♦♦♦♦◊

The task, while providing the basic pseudocode, doesn't provide a lot of guidance and a lot of room is left to evaluate the needs and design a solution. As the task sheet indicates we can use List<T>.Sort (as an example) to complete the final sorting after distributing the accounts into buckets, the task encourages going beyond the learning, to read the Microsoft documentation for C# and the .NET framework. This aligns well with writing code that complies with conventions in the language, especially with the ability to use Linq features that are relevant to the .NET Framework and C# syntax and semantics. My video additionally provides further evidence and critiquing of the quality of the code and result.

April 5, 2020

```
using System;
using System.Collections.Generic;

namespace Task_3._4D
{
    class Program
    {

        static void PrintAccountArray(Account[] accounts)
        {
            foreach (Account account in accounts)
                account.Print();
        }

        public static void Main(string[] args)
        {
            Console.WriteLine("\n*************************************************");
            Console.WriteLine("** TESTING START");
            Console.WriteLine("*************************************************\n");

            Random random = new Random();
            int numberOfAccounts = random.Next(15, 50);

            // Testing REASONABLE Arguments

            Account[] accountsArray = new Account[numberOfAccounts];
            for (int i = 0; i < accountsArray.Length; i++)
            {
                accountsArray[i] = new Account("Jane Doe",
                    Convert.ToDecimal(random.Next(10, 5000)));
            }

            Console.WriteLine("\n*************************************************");
            Console.WriteLine("** Array Order before beginning to sort:");
            Console.WriteLine("*************************************************\n");

            PrintAccountArray(accountsArray);
            AccountSorter.Sort(accountsArray, 5);

            Console.WriteLine("\n*************************************************");
            Console.WriteLine("** Array Order After sorting:");
            Console.WriteLine("*************************************************\n");

            PrintAccountArray(accountsArray);

            List<Account> accountsList = new List<Account>();
            for (int i = 0; i < numberOfAccounts; i++)
```

```
47              {
48                  accountsList.Add(new Account("Jane Doe",
                    ↪  Convert.ToDecimal(random.Next(10, 5000))));
49              }
50
51              Console.WriteLine("\n*************************************************"
                ↪  );
52              Console.WriteLine("** List Order before beginning to sort:");
53              Console.WriteLine("*************************************************\n"
                ↪  );
54
55              PrintAccountArray(accountsList.ToArray());
56              AccountSorter.Sort(accountsList, 5);
57
58              Console.WriteLine("\n*************************************************"
                ↪  );
59              Console.WriteLine("** List Order After sorting:");
60              Console.WriteLine("*************************************************\n"
                ↪  );
61
62              PrintAccountArray(accountsList.ToArray());
63
64
65              // Testing BAD Arguments
66
67              Console.WriteLine("\n\n*************************************************
                ↪  *");
68              Console.WriteLine("** Testing Bad Arguments:");
69              Console.WriteLine("*************************************************\n"
                ↪  );
70
71
72              Account[] badArray = null;
73
74              try
75              {
76                  AccountSorter.Sort(badArray, 5);  // Null array
77              }
78              catch (NullReferenceException ex)
79              {
80                  Console.WriteLine(ex.Message);
81              }
82
83              try
84              {
85                  AccountSorter.Sort(accountsArray, 0); // 0 buckets
86              }
87              catch (ArgumentOutOfRangeException ex)
88              {
89                  Console.WriteLine(ex.Message);
90              }
91
92              List<Account> badList = null;
```

```
93
94              try
95              {
96                  AccountSorter.Sort(badList, 5); // Null list
97              }
98              catch (NullReferenceException ex)
99              {
100                 Console.WriteLine(ex.Message);
101             }
102
103             try
104             {
105                 AccountSorter.Sort(accountsList, 0); // 0 buckets
106             }
107             catch (ArgumentOutOfRangeException ex)
108             {
109                 Console.WriteLine(ex.Message);
110             }
111
112             Console.WriteLine("\n\n**********************************************
                ↪  *");
113             Console.WriteLine("** TESTING END");
114             Console.WriteLine("**********************************************");
115         }
116     }
117 }
```

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace Task_3._4D
{
    static class AccountSorter
    {
        /// <summary>
        /// Returns the maximum account balance from an array of accounts
        /// </summary>
        /// <returns>
        /// The maximum account balance as a decimal
        /// </returns>
        /// <param name="accounts">The array of accounts</param>
        private static decimal MaximumBalance(Account[] accounts)
        {
            return accounts.Max(a => a.Balance);
        }

        /// <summary>
        /// Creates and initializes required list of buckets
        /// </summary>
        /// <returns>
        /// Array of buckets containing a list to store accounts
        /// </returns>
        /// <param name="b">The number of buckets required</param>
        private static List<Account>[] CreateBuckets(int b)
        {
            List<Account>[] buckets = new List<Account>[b];
            for (int i = 0; i < buckets.Length; i++)
            {
                buckets[i] = new List<Account>();
            }
            return buckets;
        }

        /// <summary>
        /// Distributes accounts into buckets from the array of accounts
        /// </summary>
        /// <param name="accounts">The array of accounts to distribute</param>
        /// <param name="buckets">The array of buckets to distribute into</param>
        private static void DistributeAccounts(Account[] accounts, List<Account>[]
        ↪ buckets)
        {
            decimal maximum = MaximumBalance(accounts);
            foreach (Account account in accounts)
            {
                int bucket = (int)(Math.Floor(buckets.Length * account.Balance /
                ↪ maximum));
                if (bucket == buckets.Length)
                    bucket -= 1;
                buckets[bucket].Add(account);
```

```
52              }
53          }
54
55          /// <summary>
56          /// Sorts the accounts in each bucket by account balance
57          /// </summary>
58          /// <param name="buckets">The buckets holding accounts</param>
59          private static void SortBuckets(List<Account>[] buckets)
60          {
61              for (int i = 0; i < buckets.Length; i++)
62              {
63                  buckets[i] = buckets[i].OrderBy(a => a.Balance).ToList();
64              }
65          }
66
67          /// <summary>
68          /// Sorts an array of accounts by their account balance from
69          /// smallest to largest
70          /// </summary>
71          /// <param name="accounts">The array of accounts to sort</param>
72          /// <param name="b">The number of buckets to use</param>
73          /// <exception cref="System.NullReferenceException">Thrown
74          /// if the accounts array is null</exception>
75          /// <exception cref="System.ArgumentOutOfRangeException">Thrown
76          /// if the number of buckets is 0 or less</exception>
77          public static void Sort(Account[] accounts, int b)
78          {
79              if (accounts == null)
80              {
81                  throw new NullReferenceException("Accounts cannot be null");
82              }
83
84              if (b <= 1)
85              {
86                  throw new ArgumentOutOfRangeException("At least 2 buckets needed");
87              }
88
89              List<Account>[] buckets = CreateBuckets(b);
90              DistributeAccounts(accounts, buckets);
91              SortBuckets(buckets);
92
93              // Write the accounts in the buckets back into the original
94              // accounts array. Idx tracks the position in the
95              // original accounts array to write to
96              int idx = 0;
97              for (int i = 0; i < buckets.Length; i++)
98              {
99                  foreach (Account account in buckets[i])
100                 {
101                     accounts[idx] = account;
102                     idx++;
103                 }
104             }
```

```
105         }
106
107         /// <summary>
108         /// Sorts a list of accounts by their account balance from
109         /// smallest to largest
110         /// </summary>
111         /// <param name="accounts">The list of accounts to sort</param>
112         /// <param name="b">The number of buckets to use</param>
113         /// <exception cref="System.NullReferenceException">Thrown
114         /// if the accounts list is null</exception>
115         /// <exception cref="System.ArgumentOutOfRangeException">Thrown
116         /// if the number of buckets is 0 or less</exception>
117         public static void Sort(List<Account> accounts, int b)
118         {
119             if (accounts == null)
120             {
121                 throw new NullReferenceException("Accounts cannot be null");
122             }
123
124             Account[] accountsArray = accounts.ToArray();
125             Sort(accountsArray, b);
126
127             // Write the accountsArray back into the accounts list.
128             // Cannot simply call .ToList() as order is not guaranteed.
129             for (int i = 0; i < accounts.Count; i++)
130             {
131                 accounts[i] = accountsArray[i];
132             }
133         }
134     }
135 }
```

```cs
using System;

namespace Task_3._4D
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get => _name; }
        public decimal Balance { get => _balance; }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance <= 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if (amount <= 0)
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
```

```
54          /// <param name="amount">The amount to subtract from the balance</param>
55          public Boolean Withdraw(decimal amount)
56          {
57              if ((amount <= 0) || (amount > _balance))
58                  return false;
59
60              _balance -= amount;
61              return true;
62          }
63
64          /// <summary>
65          /// Outputs the account name and current balance as a string
66          /// </summary>
67          public void Print()
68          {
69              Console.WriteLine("Account Name: {0}, Balance: {1}",
70                  _name, _balance.ToString("C"));
71          }
72      }
73  }
```

13 Exceptions and Error Handling

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦♦◊◊

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

Outcome	Weight
Principles	♦♦♦♦◊

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

Outcome	Weight
Build Programs	♦♦♦◊◊

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

Outcome	Weight
Design	♦♦◊◊◊

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

Outcome	Weight
Justify	♦♦♦♦♦

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

Date	Author	Comment
2020/04/14 14:58	Peter Stacey	Ready to Mark
2020/04/14 14:58	Peter Stacey	Final video editing now underway. Will be linked this evening.
2020/04/20 15:02	Peter Stacey	Welcome back to the second half
2020/04/20 15:02	Peter Stacey	Video Link: https://youtu.be/ft2O8HdsqxU
2020/05/02 22:31	Dipto Pratyaksa	Good job well done
2020/05/02 22:31	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Exceptions and Error Handling

Submitted By:
Peter Stacey
pstacey
2020/04/14 14:58

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦♦◊
Build Programs	♦♦♦◊◊
Design	♦♦◊◊◊
Justify	♦♦♦♦♦

This task, which involves both implementing code to specifically thrown exceptions, and researching those exceptions to identify their uses, who should thrown them, whether they can be caught, what information to provide and how to avoid them, is a good example through the evidence in the report, of justifying findings with references and evidence. Additionally, the program, which implements the exceptions also demonstrates the ability to follow conventions to a specific outcome, in this case, a set of exceptions that are handled wherever possible.

April 14, 2020

SIT232 – Object Oriented Development Task 4.1P- Report on Exceptions in C#

Student Name: Peter Stacey

Student ID: 219011171

Introduction to Exceptions

Exception handling is an important aspect of developing resilient programs that meet user needs. It helps to identify and respond to exceptional circumstances during the runtime of a program, and allows these exceptional circumstances to be handled gracefully, ideally with no interruption visible to the end-user.

In this task we look at a sub-set of the large number of exceptions that derive from the Exception class (refer Figure 1).

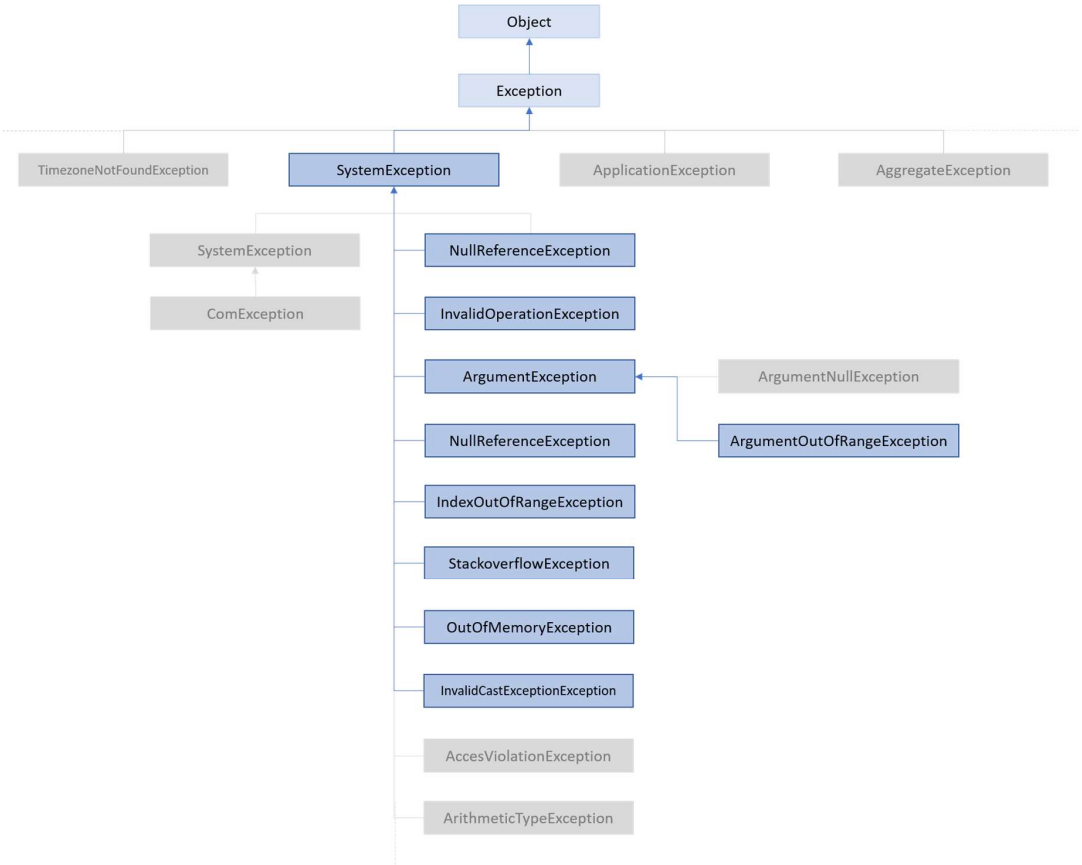


Figure 1: Hierarchy of Exceptions investigated

NullReferenceException

Possible situation that leads to the exception	Occurs when trying to access a member on a type whose value is null
Who is in charge of throwing the exception?	The called code, where it relies on arguments not being null. If it is internally used code, then the exception should be eliminated, but where arguments or types are passed to a piece of code, that code should call the exception if it relies on the data not being null. Null is unusual in that it is common for types to be null at different times, in different programs. Some programmers intentionally use and pass around null references for different uses. For example, in C# database tables can have nullable fields to indicate there is no value stored. If code then accesses that database and wants to operate on the data, checking for null types is important, to handle the exception locally in the code, or pass the exception to the calling point.
What details would you provide to the callers when throwing this exception?	• The type of the exception • Which type or argument is null • Inform them they may have forgotten to initialize the type
Can the exception be generally caught?	Yes, this can be generally caught when it is thrown
Should you catch this exception type or pass to the user?	If the exception results inside my own code, I would eliminate the risk of it all together by properly initializing types that I use. If the type is passed by a caller into a method, then I would pass it back to the caller.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, in my own code, I would always want to avoid this error by ensuring that all variables and objects are properly initialized before using them. However, since I can't control the values passed by users of my code, I would ensure that I can catch it where it needs to be and pass the exception to the caller.

References

https://docs.microsoft.com/en-us/dotnet/api/system.nullreferenceexception?view=netframework-4.8

https://stackoverflow.com/questions/4660142/what-is-a-nullreferenceexception-and-how-do-i-fix-it

IndexOutOfRangeException

Possible situation that leads to the exception	When attempting to access an element of an array or collection with an index that is outside its bounds. That is, either an index that is below the lowest index, or an index greater than the maximum index.
Who is in charge of throwing the exception?	The code that tries to access an index that is not in the available range.
What details would you provide to the callers when throwing this exception?	• The type of the exception • If there is an acceptable range, the range of values that can be used for the index or method • If there isn't a known acceptable range (eg. In C++, ranging over an array requires the length of the array to be passed in as there is no equivalent to C# Length property in C++), then I would let them know that the index is outside the range
Can the exception be generally caught?	Yes, this can be generally caught when it is thrown
Should you catch this exception type or pass to the user?	The called code should catch and handle this wherever possible, so that there is no further action required by the calling point.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, this can in most cases be avoided by using methods and properties that return the length of an array or collection, and then the values can be used. However, in cases such as the MyTime class we created for Task 2.3C, we would want to explicitly call the exception when an argument outside the allowed range is supplied.

References

https://docs.microsoft.com/en-us/dotnet/api/system.indexoutofrangeexception?view=netframework-4.8

https://stackoverflow.com/questions/20940979/what-is-an-indexoutofrangeexception-argumentoutofrangeexception-and-how-do-i-f

StackOverflowException

Possible situation that leads to the exception	Thrown when the execution stack overflows because it contains too many nested method calls.
Who is in charge of throwing the exception	The Common Language Runtime or System throws this exception.
What details would you provide to the callers when throwing this exception	Standard output is a “Stack overflow” message to the standard output.
Can the exception be generally caught	No. Starting in .NET Framework 2.0, the exception cannot be caught with a try/catch block and the process terminates by default.
Should you catch this exception type or pass to the user	There is no option. The process will terminate by default. According to the C# documentation, even applying HandleProcessCorruptedStateExceptionsAttribute will have no effect when this exception occurs.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes. Code should be written to detect and prevent possible stack overflow, since the effect is termination of the process. This is particularly relevant for situations that involve recursion. It is possible in the CLR to specify that the CLR should unload the application domain that caused the exception and let the process continue, however it is better to prevent the possibility from occurring. As per the Stackoverflow link below, which contains a code example, there are a couple of approaches: • Write code that checks the xsl for infinite recursion and notifies the user prior to applying a transform. • Load the XslTransform code into a separate process.

References

https://docs.microsoft.com/en-us/dotnet/api/system.stackoverflowexception?view=netframework-4.8

https://stackoverflow.com/questions/206820/how-do-i-prevent-and-or-handle-a-stackoverflowexception

OutOfMemoryException

Possible situation that leads to the exception	In general, the exception is thrown when there is not enough memory to continue execution of a program. There are two major causes: 1. Attempting to expand a StringBuilder object beyond the maximum capacity defined by the MaxCapacity property 2. When the CLR cannot allocate enough contiguous memory to full support an operation Four examples of the second case include: • When a disk is physically full and/or signficiantly fragmented, so that there are few contiguous blocks of memory left, • Running a program on a 32-bit operating system, which is limited to 2 GB memory, but expecting to be able to use more, • Working in a Big Data environment with a large volume of data that the system cannot manage effectively, or • Memory leaks.
Who is in charge of throwing the exception?	The system in terms of the CLR will throw the exception.
What details would you provide to the callers when throwing this exception?	• The exception type • The process that was running when the exception occurred • Hint to check physical disks and defrag their environment
Can the exception be generally caught?	Yes, this can be generally caught, and a strategy developed to handle it gracefully.
Should you catch this exception type or pass to the user?	Catch and handle if possible. Otherwise if it cannot be handled, pass it to the caller.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, if possible. It is good to avoid (eg. Use generators to limit the memory use of large datasets), but it may not be possible (eg. System physical memory is nearly full).

References

https://docs.microsoft.com/en-us/dotnet/api/system.outofmemoryexception?view=netframework-4.8

https://stackoverflow.com/questions/8563933/c-sharp-out-of-memory-exception

InvalidCastException

Possible situation that leads to the exception	When trying to cast a type to another type that it cannot be cast to. For example, casting a boolean value to a char type, or casting an object to a inbuilt type (eg. int)
Who is in charge of throwing the exception	The CLR will throw this if it cannot cast one type to another and the programmer pass this back to the caller, as there is no real strategy to deal with it effectively.
What details would you provide to the callers when throwing this exception	• The type of the exception • Which cast failed • The type of the input and the type trying to cast to
Can the exception be generally caught	Yes, this can be caught, although handling it gracefully without passing it to the user is difficult.
Should you catch this exception type or pass to the user	Pass this to the user, so they can correct the issue before calling the method that causes the exception, or develop a different strategy in their software.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, since dealing with it after the exception is thrown is more difficult. Ideally, check all variables that need to be cast, to ensure they can be cast to relevant types and/or avoiding the need to cast wherever possible.

References

https://docs.microsoft.com/en-us/dotnet/api/system.invalidcastexception?view=netframework-4.8

https://stackoverflow.com/questions/14327071/how-do-i-solve-an-invalidcastexception/14327121

https://stackoverflow.com/questions/39891504/casting-object-to-int-throws-invalidcastexception-in-c-sharp

DivideByZeroException

Possible situation that leads to the exception	Whenever there is an attempt to divide an integral or decimal by zero.
Who is in charge of throwing the exception	The programmer, should check before division whether the denominator is zero and throw the exception.
What details would you provide to the callers when throwing this exception	• The type of the exception • The values attempted to be divided • The method name
Can the exception be generally caught	Yes this can be caught and if the input is validated before division, the exception avoided.
Should you catch this exception type or pass to the user	Catch and handle where it relates to internal code, but otherwise passed to the user.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, this can generally be avoided by guarding against a 0 denominator.

References

https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception?view=netframework-4.8

https://www.dotnetperls.com/dividebyzeroexception

https://stackoverflow.com/questions/2601350/is-there-any-reason-to-throw-a-dividebyzeroexception

ArgumentException

Possible situation that leads to the exception	When arguments provided to a method are not valid.
Who is in charge of throwing the exception	The program should check the arguments and throw it where necessary. In general, it might be thrown by the CLR and indicate developer error.
What details would you provide to the callers when throwing this exception	• The type of the exception • The expected types of valid arguments • Details of the invalid argument
Can the exception be generally caught	Yes this can be generally caught and handled, although handling it may involve passing the exception to the caller. According to the C# documentation, one of the derived classes should be used instead of Argument Exception wherever possible, in order to provide more specific information. Alternatively a new class can be derived from Argument Exception to provide that additional detail by default.
Should you catch this exception type or pass to the user	Passed to the user where it is not possible to handle it gracefully without loss of information.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Yes, this is a case where avoiding the error is generally a good approach. Thorough and effective testing of code to the full extent of its intended use will assist to ensure that no Argument Exception can occur.

References

https://docs.microsoft.com/en-us/dotnet/api/system.argumentexception?view=netframework-4.8

https://stackoverflow.com/questions/774104/what-exceptions-should-be-thrown-for-invalid-or-unexpected-parameters-in-net

ArgumentOutOfRangeException

Possible situation that leads to the exception	As a derived class of ArgumentException, this is thrown when an argument to a method is outside the allowed range of values defined by the method.
Who is in charge of throwing the exception	The method that places the limit on the range of values.
What details would you provide to the callers when throwing this exception	• The type of the exception • The argument that is outside the range • Acceptable range for the argument
Can the exception be generally caught	Yes, this can be generally caught and handled. This can be specifically thrown after checking the arguments provided to a method call.
Should you catch this exception type or pass to the user	Catch and handle if possible. Otherwise pass to the caller.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Not necessarily. It may be perfectly acceptable for this exception to occur and to trigger a specific approach to handling it. For example, in Task 2.3C, this was used in the NextHour, NextMinute and NextSecond methods to know when the upper limit of the range had been exceeded and a new minute, hour or day was reached.

References

https://docs.microsoft.com/en-us/dotnet/api/system.argumentoutofrangeexception?view=netframework-4.8

http://www1.cs.columbia.edu/~lok/csharp/refdocs/System/types/ArgumentOutOfRangeException.html

https://stackoverflow.com/questions/9900481/system-argumentoutofrangeexception-argument-is-out-of-range-error-in-a-shortes

SystemException

Possible situation that leads to the exception	This class derives directly from Exception and forms the base class of the other exceptions investigated in this task. As such, the polymorphic behavior of the subclasses often indicates a subclass error, even when throwing a SystemException. However, in general this is reserved for the CLR and may be thrown when there is a system level exception, and allows differentiation of exceptions with with ApplicationException and classes derived from it.
Who is in charge of throwing the exception	Normally, the CLR. As an important note in the C# documentation: Because SystemException serves as the base class of a variety of exception types, your code should not throw a SystemException exception, nor should it handle a SystemException exception unless you intend to re-throw the original exception.
What details would you provide to the callers when throwing this exception	• As per the above, re-throw the original exception when not using a more specific, derived class and this exception occurs.
Can the exception be generally caught	Yes, however although my program specifically throws this exception, it was always a subclass that then was thrown and caught.
Should you catch this exception type or pass to the user	No. It should not be caught, unless the intend is to rethrow the original exception.
Is the exception a case when you want to avoid the exception to occur? If so, what would be your actions as a programmer to avoid it?	Ideally, yes but since this is not an exception normally being thrown by a programmer, it may be difficult to guard against, as it will be thrown primarily by an interruption elsewhere in the system, outside the program.

References

https://docs.microsoft.com/en-us/dotnet/api/system.systemexception?view=netframework-4.8

http://etutorials.org/Programming/programming+microsoft+visual+c+sharp+2005/Part+III+More+C+Language/Chapter+9+Exception+Handling/System.Exception/

```
using System;
using System.Text;

namespace Task_4._1P
{

    /// <summary>
    /// Helper class for the exception implementations. It models
    /// a simple account class for an account holder and their balance.
    /// </summary>
    class Account
    {
        public string FirstName { get; private set; }
        public string LastName { get; private set; }
        public int Balance { get; private set; }

        /// <summary>
        /// Constructor for an account
        /// </summary>
        /// <param name="firstName">Account holder's first name</param>
        /// <param name="lastName">Account holder's last name</param>
        /// <param name="balance">Balance of the account as an int</param>
        public Account(string firstName, string lastName, int balance)
        {
            FirstName = firstName;
            LastName = lastName;
            Balance = balance;
        }

        /// <summary>
        /// Attempts to withdraw funds from the account if sufficient
        /// funds are available
        /// </summary>
        /// <param name="amount">The amount to attempt to withdraw</param>
        /// <exception cref="System.InvalidOperationException">Thrown
        /// when the amount to withdraw is more than the available
        /// funds</exception>
        public void Withdraw(int amount)
        {
            if (amount > Balance)
            {
                throw new InvalidOperationException("Insufficient funds");
            }
            Balance = Balance - amount;
        }
    }

    /// <summary>
    /// Empty class definition to assist with the ArgumentException
    /// example
    /// </summary>
    public class MyClass { }

```

```
54      /// <summary>
55      /// Helper class for the ArgumentOutOfRangeException, which we
56      /// encountered in Task 2.3C
57      /// </summary>
58      class MyTime
59      {
60          // Instance variables
61          private int _hour;
62          private int _minute;
63          private int _second;
64
65          /// Reference for this approach, from:
66          /// https://stackoverflow.com/questions/56197825
67          public int Hour
68          {
69              get => _hour;
70              set => _hour = (value >= 0) && (value <= 23)
71                  ? value
72                  : throw new ArgumentOutOfRangeException("Invalid hour. Must be
                    ↪  0-23");
73          }
74
75          public int Minute
76          {
77              get => _minute;
78              set => _minute = (value >= 0) && (value <= 59)
79                  ? value
80                  : throw new ArgumentOutOfRangeException("Invalid minute. Must be
                    ↪  0-59");
81          }
82
83          public int Second
84          {
85              get => _second;
86              set => _second = (value >= 0) && (value <= 59)
87                  ? value
88                  : throw new ArgumentOutOfRangeException("Invalid second. Must be
                    ↪  0-59");
89          }
90
91          public MyTime(int hour, int minute, int second)
92          {
93              Hour = hour;
94              Minute = minute;
95              Second = second;
96          }
97      }
98
99      class Program
100     {
101         // Theoretically sets a limit on the number of recursive
102         // calls in the execute method, but this is bypassed in
103         // the implementation
```

```
104         const int MAX_RECURSIVE_CALLS = 1000;
105
106         /// <summary>
107         /// Helper method for the StackOverflowException
108         /// </summary>
109         /// <param name="counter"></param>
110         static void Execute(int counter)
111         {
112             counter++;
113
114             if (counter <= MAX_RECURSIVE_CALLS)
115                 counter--;
116
117             Execute(counter);
118         }
119
120         public static void Main(string[] args)
121         {
122             // NullReference Example
123             int[] values = null;
124
125             try
126             {
127                 // NullReferenceException occurs here because the
128                 // loop attempts to set a value within the array
129                 // but the size of the array has not been set
130                 // so no position can be addressed, as it is null
131                 for (int i = 0; i < 10; i++)
132                     values[i] = i * 2;
133
134                 foreach (var value in values)
135                     Console.WriteLine(value);
136             }
137             catch (NullReferenceException exception)
138             {
139                 Console.WriteLine("The following error detected: "
140                     + exception.GetType().ToString()
141                     + " with message \"" + exception.Message + "\"");
142             }
143
144             // IndexOutOfRange Example
145             try
146             {
147                 values = new int[10];
148
149                 // IndexOutOfRangeException occurs here because the loop
150                 // attempts to set a value for an index equal to the
151                 // length of the array, which is one more than the last
152                 // available index value (ie. values[10] is beyond the
153                 // end of the array)
154                 for (int i = 0; i <= values.Length; i++)
155                 {
156                     values[i] = i * 2;
```

```
157                 }
158
159                 foreach (var value in values)
160                     Console.WriteLine(value);
161
162             }
163             catch (IndexOutOfRangeException exception)
164             {
165                 Console.WriteLine("The following error detected: "
166                     + exception.GetType().ToString()
167                     + " with message \"" + exception.Message + "\"");
168             }
169
170             // StackOverflow Example
171             try
172             {
173                 // Will cause recursive filling of the stack with
174                 // increment and decrement steps
175                 Execute(0);
176             }
177             catch (StackOverflowException exception)
178             {
179                 // This catch block will never be executed as a
180                 // stack overflow always terminates the program
181                 Console.WriteLine("The following error detected: "
182                     + exception.GetType().ToString()
183                     + " with message \"" + exception.Message + "\"");
184             }
185
186             // OutOfMemory Example
187             try
188             {
189                 // OutOfMemory occurs because the capacity and length
190                 // are set smaller than the amount of memory required
191                 // to create the initial string and then insert
192                 // the second string into the first at index 0
193                 StringBuilder sb = new StringBuilder(15, 15);
194                 sb.Append("Substring #1 ");
195                 sb.Insert(0, "Substring #2 ", 1);
196             }
197             catch (OutOfMemoryException exception)
198             {
199                 Console.WriteLine("The following error detected: "
200                     + exception.GetType().ToString()
201                     + " with message \"" + exception.Message + "\"");
202             }
203
204             // InvalidCast Example
205             try
206             {
207                 bool flag = true;
208                 char ch = Convert.ToChar(flag); // bool cannot cast to char
209             }
```

```
210             catch (InvalidCastException exception)
211             {
212                 Console.WriteLine("The following error detected: "
213                     + exception.GetType().ToString()
214                     + " with message \"" + exception.Message + "\"");
215             }
216 
217             // DivideByZeroException Example
218             try
219             {
220                 int x = 1000;
221                 int y = 0;
222 
223                 // This is a simple and explicitly coded 0, however this
224                 // is more relevant where a method involves some division
225                 // involving arguments and/or where user or sensor input
226                 // is involved
227                 Console.WriteLine(x / y);
228             }
229             catch (DivideByZeroException exception)
230             {
231                 Console.WriteLine("The following error detected: "
232                     + exception.GetType().ToString()
233                     + " with message \"" + exception.Message + "\"");
234             }
235 
236             // ArgumentException Example
237             try
238             {
239                 MyClass my = new MyClass();
240                 string s = "test text";
241                 int i = s.CompareTo(my); // comparing object to string
242             }
243             catch (ArgumentException exception)
244             {
245                 Console.WriteLine("The following error detected: "
246                     + exception.GetType().ToString()
247                     + " with message \"" + exception.Message + "\"");
248             }
249 
250             // ArgumentOutOfRangeException Example
251             try
252             {
253                 MyTime t = new MyTime(24, 0, 0); // 24 is larger than the allowed
                    ↪ range
254             }
255             catch (ArgumentOutOfRangeException exception)
256             {
257                 Console.WriteLine("The following error detected: "
258                     + exception.GetType().ToString()
259                     + " with message \"" + exception.Message + "\"");
260             }
261 
```

```
262             // SystemException Example
263             try
264             {
265                 // This is the base class for other exceptions and
266                 // is normally reserved for runtime errors in the
267                 // CLR and a more specific exception type should be derived
268                 // or thrown
269                 int[] array = new int[5];
270                 array[10] = 25; // This is also an IndexOutOfRangeException
271             }
272             catch (SystemException exception)  // This is normally bad
273             {
274                 Console.WriteLine("The following error detected: "
275                     + exception.GetType().ToString()
276                     + " with message \"" + exception.Message + "\"");
277             }
278 
279             // Original code supplied in the task for an
280             // InvalidOperationException
281             try
282             {
283                 Account account = new Account("Sergey", "P", 100);
284                 account.Withdraw(1000);
285             }
286             catch (InvalidOperationException exception)
287             {
288                 Console.WriteLine("The following error detected: "
289                     + exception.GetType().ToString()
290                     + " with message \"" + exception.Message + "\"");
291             }
292         }
293     }
294 }
```

14 BuggySoft: Program Design and Class Composition

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦♦♦

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

Outcome	**Weight**
Principles	♦♦♦♦◊

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

Outcome	**Weight**
Build Programs	♦♦♦♦◊

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

Outcome	**Weight**
Design	♦♦♦♦◊

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

Outcome	**Weight**
Justify	♦♦♦◊◊

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

Date	Author	Comment
2020/04/22 14:46	Peter Stacey	Ready to Mark
2020/04/22 14:46	Peter Stacey	I have some refactoring to do after not looking at this for a few weeks and realising how much I can improve it. Will work on that now and then record my video.
2020/04/25 19:05	Peter Stacey	Video link: https://youtu.be/lH6tID7GKio
2020/04/25 19:07	Peter Stacey	Ready to Mark
2020/04/25 20:51	Peter Stacey	Just a quick note. The final code is a bit longer than it would be just to do the basic aspects asked for in the task sheet, but I really enjoyed this task, so added a couple of additional features
2020/05/02 22:39	Dipto Pratyaksa	It;s all good, don't get too carried away, we still have plenty to do
2020/05/02 22:39	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

BuggySoft: Program Design and Class Composition

Submitted By:
Peter Stacey
pstacey
2020/04/25 19:07

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦♦♦♦
Principles	♦♦♦♦◊
Build Programs	♦♦♦♦◊
Design	♦♦♦♦◊
Justify	♦♦♦◊◊

Taking an existing codebase and being able to reason it, directly related to outcome 1, and this task also extends beyond that by requiring us to maintain and refactor the code to a more robust solution to the original problem. This goes straight to outcomes 3 and 4 and then the final part of the task, requiring the addition of functionality, directly relates even further, to outcomes 2, 3 and 4. Together with the submitted code and my video, the evidence aligns with outcome 5.

April 25, 2020

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace DuplicateCode
{
    class RevisedCode
    {
        /// <summary>
        /// Outputs a prompt for input and returns the input string
        /// <summary>
        /// <returns>
        /// The input of the user as a string
        /// </summary>
        public static string ReadString(string prompt)
        {
            Console.WriteLine(prompt);
            Console.Write(">> ");
            return Console.ReadLine();
        }

        /// <summary>
        /// Returns the length of the longest list in the dictionary
        /// </summary>
        /// <returns>
        /// Length of the longest list as an integer
        /// </returns>
        /// <exception cref="System.ArgumentNullException">Thrown if the
        /// dictionary or any list is null
        /// </exception>
        static int MaximumLength(Dictionary<string, List<string>> tasks)
        {
            return tasks.Values.Max(list => list.Count);
        }

        /// <summary>
        /// Prints the current list of tasks by category, out to the
        /// console
        /// </summary>
        static void PrintTasks(Dictionary<string, List<string>> tasks)
        {
            int max = MaximumLength(tasks);
            Console.ForegroundColor = ConsoleColor.Blue;
            Console.WriteLine(new string(' ', 12) + "CATEGORIES");
            Console.WriteLine(new string(' ', 10) + new string('-', 94));
            Console.Write("{0,10}|", "Item #");
            foreach (var category in tasks.Keys)
            {
                Console.Write("{0,30}|", category);
            }
            Console.WriteLine();
            Console.WriteLine(new string(' ', 10) + new string('-', 94));
            for(int i = 0; i < max; i++)
```

```
54              {
55                  Console.Write("{0,10}|", i+1);
56                  foreach(var list in tasks.Values)
57                  {
58                      if (list.Count > i)
59                          Console.Write("{0,30}|", list[i]);
60                      else
61                          Console.Write("{0,30}|", "N/A");
62                  }
63                  Console.WriteLine();
64              }
65              Console.ResetColor();
66          }
67
68          static void Main(string[] args)
69          {
70              var tasks = new Dictionary<string, List<string>>();
71              tasks["Personal"] = new List<string>();
72              tasks["Work"] = new List<string>();
73              tasks["Family"] = new List<string>();
74
75              string category;
76              string task;
77
78              while (true)
79              {
80                  Console.Clear();
81                  PrintTasks(tasks);
82
83                  category = ReadString("\nWhich category do you want to place " +
84                      "a new task? Type \'Personal\', \'Work\', \'Family\', or
                        ↪  \'Quit\'").ToLower();
85                  if (category.ToLower() == "quit")
86                      break;
87
88                  task = ReadString("Describe your task below (max. 30 symbols).");
89                  if (task.Length > 30)
90                  {
91                      task = task.Substring(0, 30);
92                  }
93
94                  try
95                  {
96                      tasks[category].Add(task);
97                  }
98                  catch (ArgumentException)
99                  {
100                     continue;  // if category is not present, add no task
101                 }
102             }
103         }
104     }
105 }
```

```cs
using System;
using System.Linq;
using System.Collections.Generic;

namespace DuplicateCode
{
    /// <summary>
    /// List of available options for actions in the program
    /// </summary>
    public enum MenuOption
    {
        AddCategory,
        DeleteCategory,
        AddTask,
        DeleteTask,
        CompleteTask,
        ChangeImportance,
        SetDueDate,
        ChangePosition,
        MoveTask,
        Quit
    }

    /// <summary>
    /// Level of task mportance. All overdue tasks print white on red
    /// background, regardless of the importance level.
    /// </summary>
    public enum TaskImportance
    {
        Low,
        Medium,
        High,
        Complete
    }

    /// <summary>
    /// Static class to provide methods for console input
    /// </summary>
    public static class ConsoleInput
    {
        /// <summary>
        /// Outputs a prompt for input and returns the input string
        /// <summary>
        /// <returns>
        /// The input of the user as a string
        /// </summary>
        /// <param name="prompt">The string to prompt the user with</param>
        public static string ReadString(string prompt)
        {
            Console.WriteLine(prompt);
            Console.Write(">> ");
            return Console.ReadLine();
        }
```

```
54
55          /// <summary>
56          /// Outputs a prompt for input and returns the input as an integer
57          /// <summary>
58          /// <returns>
59          /// The input of the user as an integer
60          /// </summary>
61          /// <param name="prompt">The string to prompt the user with</param>
62          public static int ReadInteger(string prompt)
63          {
64              string input = ReadString(prompt);
65              int output;
66              while (!int.TryParse(input, out output))
67              {
68                  Console.WriteLine("Enter a whole number only");
69                  input = ReadString(prompt);
70              }
71              return Convert.ToInt32(input);
72          }
73
74          /// <summary>
75          /// Outputs a prompt for input and returns the input as an integer
76          /// within a range from min to max
77          /// <summary>
78          /// <returns>
79          /// The input of the user as an integer
80          /// </summary>
81          /// <param name="prompt">The string to prompt the user with</param>
82          /// <param name="min">The minimum number allowed</param>
83          /// <param name="max">The maximum number allowed</param>
84          public static int ReadInteger(string prompt, int min, int max)
85          {
86              int input = ReadInteger(prompt);
87              while(input < min || input > max)
88              {
89                  Console.WriteLine("Enter a number between {0} to {1}", min, max);
90                  input = ReadInteger(prompt);
91              }
92              return input;
93          }
94      }
95
96      /// <summary>
97      /// Model for an individual task
98      /// </summary>
99      public class TaskModel
100     {
101         // Public properties
102         public string Description { get; set;}
103         public TaskImportance Importance { get; set; }
104         public DateTime DueDate { get; set; }
105
106         /// <summary>
```

```
107         /// Creates a new task object
108         /// </summary>
109         /// <param name="description">Description for the task</param>
110         /// <param name="importance">Level of task importance</param>
111         public TaskModel(
112             string description,
113             DateTime dueDate,
114             TaskImportance importance = TaskImportance.Medium)
115         {
116             Description = description;
117             Importance = importance;
118             DueDate = dueDate;
119         }
120 
121         /// <summary>
122         /// Returns the console color for the corresponding task
123         /// importance
124         /// </summary>
125         /// <returns>
126         /// ConsoleColor relevant to the importance
127         /// </returns>
128         public ConsoleColor GetColor()
129         {
130             switch (Importance)
131             {
132                 case TaskImportance.Low:
133                     return ConsoleColor.Green;
134                 case TaskImportance.Medium:
135                     return ConsoleColor.Blue;
136                 case TaskImportance.High:
137                     return ConsoleColor.Red;
138                 case TaskImportance.Complete:
139                 default:
140                     return ConsoleColor.DarkGray;
141             }
142         }
143     }
144 
145     /// <summary>
146     /// Repository for a collection of tasks within a category
147     /// </summary>
148     public class TaskList
149     {
150         // Public properties
151         private List<TaskModel> _tasks;
152 
153         /// <summary>
154         /// Creates a new task list
155         /// </summary>
156         public TaskList()
157         {
158             _tasks = new List<TaskModel>();
159         }
```

```
160
161         /// <summary>
162         /// Adds a task to the list of tasks
163         /// </summary>
164         /// <param name="task">The TaskModel to add to the list</param>
165         public void AddTask(TaskModel task)
166         {
167             _tasks.Add(task);
168         }
169
170         /// <summary>
171         /// Removes a task from the list of tasks
172         /// </summary>
173         /// <param name="task">The TaskModel to remove from the list</param>
174         public void DeleteTask(TaskModel task)
175         {
176             _tasks.Remove(task);
177         }
178
179         /// <summary>
180         /// Changes the position of a task within the task list
181         /// </summary>
182         /// <param name="currentIndex">Current index of the task</param>
183         /// <param name="newIndex">New index for the task</param>
184         public void ChangePriority(int currentIndex, int newIndex)
185         {
186             TaskModel task = new TaskModel(
187                 _tasks[currentIndex].Description,
188                 _tasks[currentIndex].DueDate,
189                 _tasks[currentIndex].Importance
190             );
191             _tasks.Remove(_tasks[currentIndex]);
192             _tasks.Insert(newIndex, task);
193         }
194
195         /// <summary>
196         /// Returns the count of tasks in the tasklist
197         /// </summary>
198         /// <returns>Integer of the number of tasks</returns>
199         public int GetCount()
200         {
201             return _tasks.Count;
202         }
203
204         /// <summary>
205         /// Returns the task at a specific index in range
206         /// </summary>
207         /// <returns>
208         /// The TaskModel at a specific index
209         /// </returns>
210         /// <param name="index">The index position of the task to return</param>
211         public TaskModel TaskAt(int index)
212         {
```

```
213             try
214             {
215                 return _tasks[index];
216             }
217             catch (IndexOutOfRangeException)
218             {
219                 throw;
220             }
221         }
222     }
223 
224     public class TaskListRepository
225     {
226         // Instance variables
227         private Dictionary<string, TaskList> _repo;
228 
229         /// <summary>
230         /// Creates a new TaskListRepository
231         /// </summary>
232         public TaskListRepository()
233         {
234             _repo = new Dictionary<string, TaskList>();
235         }
236 
237         /// <summary>
238         /// Adds a new category of tasks to the controller
239         /// </summary>
240         /// <param name="category"></param>
241         public void AddCategory(string category)
242         {
243             TaskList tasks = new TaskList();
244             _repo.Add(category, tasks);
245         }
246 
247         /// <summary>
248         /// Removes a category and associated tasks
249         /// </summary>
250         /// <param name="category"></param>
251         public void DeleteCategory(string category)
252         {
253             _repo.Remove(category);
254         }
255 
256         /// <summary>
257         /// Moves a task from one category to another
258         /// </summary>
259         /// <param name="task">The task to move</param>
260         /// <param name="current">Name of the current category</param>
261         /// <param name="updated">Name of the new category</param>
262         public void MoveTask(TaskModel task, string current, string updated)
263         {
264             _repo[current].DeleteTask(task);
265             _repo[updated].AddTask(task);
```

```
266         }
267
268         /// <summary>
269         /// Returns the length of the longest list in the dictionary
270         /// </summary>
271         /// <returns>
272         /// Length of the longest list as an integer
273         /// </returns>
274         public int MaximumLength()
275         {
276             return _repo.Values.Max(list => list.GetCount());
277         }
278
279         /// <summary>
280         /// Returns the task list for the given category
281         /// </summary>
282         /// <param name="category"></param>
283         /// <returns></returns>
284         public TaskList GetTaskList(string category)
285         {
286             return _repo[category];
287         }
288
289         /// <summary>
290         /// Returns whether a specific key exists in the repository
291         /// </summary>
292         /// <param name="key">The key to check for</param>
293         /// <returns>
294         /// Boolean whether the key exists or not
295         /// </returns>
296         public bool ContainsKey(string category)
297         {
298             return _repo.ContainsKey(category);
299         }
300
301         /// <summary>
302         /// Prints the current list of tasks by category, out to the
303         /// console
304         /// </summary>
305         public void Print()
306         {
307             Console.ForegroundColor = ConsoleColor.Blue;
308             Console.WriteLine(new string(' ', 12) + "CATEGORIES");
309             Console.WriteLine(new string(' ', 10)
310                 + new string('-', 51 * _repo.Count));
311             Console.Write("{0,10}|", "Item #");
312             foreach (var category in _repo.Keys) // Print the category names
313             {
314                 Console.Write("{0,-50}|", category);
315             }
316             Console.WriteLine();
317             Console.WriteLine(new string(' ', 10)
318                 + new string('-', 51 * _repo.Count));
```

```
            for(int i = 0; i < MaximumLength(); i++)
            {
                Console.Write("{0,10}|", i+1);
                foreach(var list in _repo.Values) // Print the list of tasks
                {
                    if (list.GetCount() > i)
                    {
                        Console.ForegroundColor = list.TaskAt(i).GetColor();
                        // Print white on red background if not complete and due
                        // today or overdue
                        if (list.TaskAt(i).DueDate <= DateTime.Today
                            && list.TaskAt(i).Importance != TaskImportance.Complete)
                        {
                            Console.BackgroundColor = ConsoleColor.Red;
                            Console.ForegroundColor = ConsoleColor.White;
                        }
                        Console.Write("{0,-30}{1,20}", list.TaskAt(i).Description,
                            list.TaskAt(i).DueDate.Date.ToString("d"));
                        Console.ResetColor();
                        Console.ForegroundColor = ConsoleColor.Blue;
                        Console.Write("|");
                    }
                    else
                        Console.Write("{0,-50}|", "N/A");
                }
                Console.WriteLine();
            }
            Console.ResetColor();
        }
    }

    /// <summary>
    /// Manages a collection of task lists
    /// </summary>
    public class TaskListController
    {
        // Instance variables
        private TaskListRepository _taskRepo;
        private MenuOption _action;

        /// <summary>
        /// Creates a new task list controller
        /// </summary>
        public TaskListController()
        {
            _taskRepo = new TaskListRepository();
        }

        /// <summary>
        /// Prints the menu options to the console
        /// </summary>
        static void PrintMenu()
        {
```

```
372             Console.WriteLine("\n" + new string('-', 30));
373             Console.WriteLine("| {0,-26} |", "MENU:");
374             Console.WriteLine(new string('-', 30));
375             Console.WriteLine("| {0,-26} |", " 1. Add Category");
376             Console.WriteLine("| {0,-26} |", " 2. Delete Category");
377             Console.WriteLine("| {0,-26} |", " 3. Add Task");
378             Console.WriteLine("| {0,-26} |", " 4. Delete Task");
379             Console.WriteLine("| {0,-26} |", " 5. Mark Task Complete");
380             Console.WriteLine("| {0,-26} |", " 6. Set Task Importance");
381             Console.WriteLine("| {0,-26} |", " 7. Set Task Due Date");
382             Console.WriteLine("| {0,-26} |", " 8. Change Task Position");
383             Console.WriteLine("| {0,-26} |", " 9. Move Task Category");
384             Console.WriteLine("| {0,-26} |", "10. Quit");
385             Console.WriteLine(new string('-', 30));
386         }
387 
388         /// <summary>
389         /// Selectas an action to perform from the menu
390         /// </summary>
391         /// <param name="repo">The TaskListController to tak action on</param>
392         /// <returns>MenuOption of the required option</returns>
393         static MenuOption ReadUserOption()
394         {
395             int action = ConsoleInput.ReadInteger("\nChoose a menu option:",
396                 1, Enum.GetNames(typeof(MenuOption)).Length);
397             return (MenuOption)action - 1;
398         }
399 
400         /// <summary>
401         /// Selects a valid category in the current repository of tasks
402         /// </summary>
403         /// <returns>The selected tasklist if the category exists</returns>
404         public TaskList SelectTaskList()
405         {
406             string category = ConsoleInput.ReadString("Enter name of category");
407             while(!_taskRepo.ContainsKey(category))
408             {
409                 Console.WriteLine("That is an invalid key");
410                 category = ConsoleInput.ReadString("Enter name of category");
411             }
412             return _taskRepo.GetTaskList(category);
413         }
414 
415         /// <summary>
416         /// Selects a task from a tasklist
417         /// </summary>
418         /// <param name="tasks"></param>
419         /// <returns></returns>
420         public TaskModel SelectTask(TaskList tasklist)
421         {
422             int selected = ConsoleInput.ReadInteger(
423                 "Task item #", 1, tasklist.GetCount());
424             return tasklist.TaskAt(selected - 1);
```

```
425         }
426
427         /// <summary>
428         /// Provides continuous looping of the controller, to maintain
429         /// and manage a collection of task lists while the user
430         /// continues to use the program.
431         /// </summary>
432         public void Run()
433         {
434             // Add initial default categories
435             _taskRepo.AddCategory("work");
436             _taskRepo.AddCategory("family");
437             _taskRepo.AddCategory("personal");
438
439             do
440             {
441                 Console.Clear();
442                 _taskRepo.Print();
443                 PrintMenu();
444
445                 _action = ReadUserOption();
446
447                 switch (_action)
448                 {
449                     case MenuOption.AddCategory:
450                         string category = ConsoleInput.ReadString(
451                             "Name of the category");
452                         if (!_taskRepo.ContainsKey(category))
453                             _taskRepo.AddCategory(category);
454                         break;
455
456                     case MenuOption.DeleteCategory:
457                         category = ConsoleInput.ReadString(
458                             "Name of the category");
459                         if (_taskRepo.ContainsKey(category))
460                             _taskRepo.DeleteCategory(category);
461                         break;
462
463                     case MenuOption.AddTask:
464                         TaskList tasklist = SelectTaskList();
465                         string description = ConsoleInput.ReadString(
466                             "Describe your task below (max. 30 symbols).");
467                         if (description.Length > 30)
468                         {
469                             description = description.Substring(0, 30);
470                         }
471                         DateTime dueDate = DateTime.Now;
472                         tasklist.AddTask(new TaskModel(description,
473                             dueDate.AddDays(1))); // Default due in 1 day
474                         break;
475
476                     case MenuOption.CompleteTask:
477                         tasklist = SelectTaskList();
```

```
478                         TaskModel task = SelectTask(tasklist);
479                         task.Importance = TaskImportance.Complete;
480                         break;
481
482                     case MenuOption.DeleteTask:
483                         tasklist = SelectTaskList();
484                         if (tasklist.GetCount() != 0)
485                         {
486                             task = SelectTask(tasklist);
487                             tasklist.DeleteTask(task);
488                         }
489                         break;
490
491                     case MenuOption.ChangeImportance:
492                         tasklist = SelectTaskList();
493                         task = SelectTask(tasklist);
494                         string importance = ConsoleInput.ReadString(
495                                 "Enter new importance");
496                         switch (importance.ToLower())
497                         {
498                             case "low":
499                                 task.Importance = TaskImportance.Low;
500                                 break;
501                             case "medium":
502                                 task.Importance = TaskImportance.Medium;
503                                 break;
504                             case "high":
505                                 task.Importance = TaskImportance.High;
506                                 break;
507                             case "complete":
508                                 task.Importance = TaskImportance.Complete;
509                                 break;
510                             default:
511                                 break;
512                         }
513                         break;
514
515                     case MenuOption.SetDueDate:
516                         tasklist = SelectTaskList();
517                         task = SelectTask(tasklist);
518                         string date = ConsoleInput.ReadString(
519                             "Enter due date \'YYYY-MM-DD\'");
520                         task.DueDate = DateTime.Parse(date);
521                         break;
522
523                     case MenuOption.ChangePosition:
524                         tasklist = SelectTaskList();
525                         int c = ConsoleInput.ReadInteger(
526                             "Enter task current item #");
527                         int n = ConsoleInput.ReadInteger(
528                             "Enter the new item #");
529                         tasklist.ChangePriority(c - 1, n - 1);
530                         break;
```

```
531
532                     case MenuOption.MoveTask:
533                         string current = ConsoleInput.ReadString(
534                             "Name of the current category");
535                         tasklist = _taskRepo.GetTaskList(current);
536                         task = SelectTask(tasklist);
537                         string updated = ConsoleInput.ReadString(
538                             "Name of the new category");
539                         _taskRepo.MoveTask(task, current, updated);
540                         break;
541
542                     case MenuOption.Quit:
543                     default:
544                         break;
545                 }
546             } while (_action != MenuOption.Quit);
547         }
548     }
549
550     /// <summary>
551     /// Final code for Task 4.2P
552     /// </summary>
553     class FinalCode
554     {
555         static void Main(string[] args)
556         {
557             var controller = new TaskListController();
558             controller.Run();
559         }
560     }
561 }
```

15 C# Essentials: Inheritance

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦◊◊◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

Outcome	Weight
Principles	♦♦♦◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

Outcome	Weight
Build Programs	♦♦♦◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

Outcome	Weight
Design	♦♦◊◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

Outcome	Weight
Justify	♦♦♦◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

Date	Author	Comment
2020/04/06 13:00	Peter Stacey	Ready to Mark
2020/04/06 13:01	Peter Stacey	Video link to come after I upload the tasks 4.1 and 4.2
2020/04/09 23:13	Dipto Pratyaksa	ok
2020/04/09 23:13	Dipto Pratyaksa	Discuss
2020/04/26 15:00	Peter Stacey	Video Link: https://youtu.be/P73qtWAb56s
2020/05/02 22:44	Dipto Pratyaksa	Good job well done
2020/05/02 22:44	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Inheritance

Submitted By:
Peter Stacey
pstacey
2020/04/06 13:00

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦◊◊◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦♦◊◊◊
Justify	♦♦♦◊◊

This task introduced inheritance, one of the key principles that help define object oriented programming. The task also begins to introduce the concept of polymorphism, although it primarily sticks with inheritence as the major theme. The task involves implementing a set design and evaluating our implementation for correct use of conventions.

April 6, 2020

```
using System;

namespace Task01
{
    class ZooPark
    {
        static void Main(string[] args)
        {
            Animal williamWolf = new Animal("William the Wolf", "Meat", "Dog
            ↪  Village", 50.6, 9, "Grey");
            Animal tonyTiger = new Animal("Tony the Tiger", "Meat", "Cat Land",
            ↪  110, 6, "Orange and White");
            Animal edgarEagle = new Animal("Edgar the Eagle", "Fish", "Bird Mania",
            ↪  20, 15, "Black");
        }
    }
}
```

```
using System;

namespace Task01
{
    class Animal
    {
        // Instance variables
        private String _name;
        private String _diet;
        private String _location;
        private double _weight;
        private int _age;
        private String _colour;

        /// <summary>
        /// Constructor for an animal
        /// </summary>
        /// <param name="name">The personal name of the animal</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the animal is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the animal in years</param>
        /// <param name="colour">The dominant color(s)</param>
        public Animal(String name, String diet, String location,
            double weight, int age, String colour)
        {
            _name = name;
            _diet = diet;
            _location = location;
            _weight = weight;
            _age = age;
            _colour = colour;
        }

        /// <summary>
        /// Method to make the animal eat food
        /// </summary>
        public void eat()
        {
            // Code for the animal to eat
            Console.WriteLine("The animal eats food");
        }

        /// <summary>
        /// Method to make the animal sleep
        /// </summary>
        public void sleep()
        {
            // Code for the animal to sleep
            Console.WriteLine("The animal sleeps");
        }

        /// <summary>
```

```
54          /// Method to make the animal make a noise
55          /// </summary>
56          public void makeNoise()
57          {
58              // Code for the animal to make a noise
59              Console.WriteLine("The animal makes a noise");
60          }
61
62          /// <summary>
63          /// Method to make any animal sound like a lion
64          /// </summary>
65          public void makeLionNoise()
66          {
67              // Code for the animal to make a noise
68              Console.WriteLine("The Lion makes a noise");
69          }
70
71          /// <summary>
72          /// Method to make any animal sound like an eagle
73          /// </summary>
74          public void makeEagleNoise()
75          {
76              // Code for the animal to make a noise
77              Console.WriteLine("The eagle makes a noise");
78          }
79
80          /// <summary>
81          /// Method to make any animal sound like a wolf
82          /// </summary>
83          public void makeWolfNoise()
84          {
85              // Code for the animal to make a noise
86              Console.WriteLine("The wold makes a noise");
87          }
88
89          /// <summary>
90          /// Method to make an animal eat meat
91          /// </summary>
92          public void eatMeat()
93          {
94              // Code for the animal to make a noise
95              Console.WriteLine("The animal eats meat");
96          }
97
98          /// <summary>
99          /// Method to make an animal eat berries
100         /// </summary>
101         public void eatBerries()
102         {
103             // Code for the animal to make a noise
104             Console.WriteLine("The animal eats berries");
105         }
106
```

```
107         /// <summary>
108         /// Method to make an animal eat fish
109         /// </summary>
110         public void eatFish()
111         {
112             // Code for the animal to make a noise
113             Console.WriteLine("The animal eats fish");
114         }
115     }
116 }
```

```
using System;

namespace Task02
{
    /// <summary>
    /// Base class for all animals
    /// </summary>
    class Animal
    {
        // Instance variables
        private String _name;
        private String _diet;
        private String _location;
        private double _weight;
        private int _age;
        private String _colour;

        // Public properties
        public String Name { get => _name; }

        /// <summary>
        /// Constructor for a base animal instance
        /// </summary>
        /// <param name="name">The personal name of the animal</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the animal is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the animal in years</param>
        /// <param name="colour">The dominant color(s)</param>
        public Animal(String name, String diet, String location,
            double weight, int age, String colour)
        {
            _name = name;
            _diet = diet;
            _location = location;
            _weight = weight;
            _age = age;
            _colour = colour;
        }

        /// <summary>
        /// Method to make the animal eat food
        /// </summary>
        public virtual void eat()
        {
            // code for animal to eat
            Console.WriteLine("An animal eats");
        }

        /// <summary>
        /// Puts the animal to sleep
        /// </summary>
        public virtual void sleep()
```

```
54          {
55              // code for animal to sleep
56              Console.WriteLine("An animal sleeps");
57          }
58
59          /// <summary>
60          /// Allows the animal to speak or make noise
61          /// </summary>
62          public virtual void makeNoise()
63          {
64              // code for animal to make a noise
65              Console.WriteLine("An animal makes a noise");
66          }
67
68          /// <summary>
69          /// Allows the animal to construct it's home within the display
70          /// </summary>
71          public virtual void buildHome()
72          {
73              Console.WriteLine("An animal builds a home");
74          }
75      }
76  }
```

```
using System;

namespace Task02
{
    class ZooPark
    {
        static void Main(string[] args)
        {
            //Animal williamWolf = new Animal("William the Wolf", "Meat", "Dog Village", 50.6, 9, "Grey");
            //Animal tonyTiger = new Animal("Tony the Tiger", "Meat", "Cat Land", 110, 6, "Orange and White");
            //Animal edgarEagle = new Animal("Edgar the Eagle", "Fish", "Bird Mania", 20, 15, "Black");

            Tiger tonyTiger = new Tiger("Tony the Tiger", "Meat", "Cat Land", 110, 6,
                "Orange and White", "Siberian", "White");
            Wolf williamWolf = new Wolf("William the Wolf", "Meat", "Dog Village", 50.6, 9, "Grey");
            Eagle edgarEagle = new Eagle("Edgar the Eagle", "Fish", "Bird Mania", 20, 15,
                "Black", "Harpy", 98.5);

            Animal baseAnimal = new Animal("Animal Name", "Animal Diet", "Animal Location",
                0.0, 0, "Animal Colour");

            baseAnimal.eat();
            tonyTiger.eat();
            williamWolf.eat();
            edgarEagle.eat();

            baseAnimal.sleep();
            tonyTiger.sleep();
            williamWolf.sleep();
            edgarEagle.sleep();

            baseAnimal.makeNoise();
            tonyTiger.makeNoise();
            williamWolf.makeNoise();
            edgarEagle.makeNoise();

            baseAnimal.buildHome();
            tonyTiger.buildHome();
            williamWolf.buildHome();
            edgarEagle.buildHome();

            edgarEagle.layEgg();
            edgarEagle.fly();

            Lion leoLion = new Lion("Leo the Lion", "Meat", "Lion's Pride", 145, 3, "Sandy", "African");
```

```
46              Penguin percyPenguin = new Penguin("Percy the Penguin", "Fish",
                ↪  "Antarctic Experience",
47                  12, 2, "Black and White", "Emperor", 20);
48
49              leoLion.eat();
50              leoLion.makeNoise();
51              leoLion.buildHome();
52              leoLion.sleep();
53
54              percyPenguin.eat();
55              percyPenguin.buildHome();
56              percyPenguin.layEgg();
57              percyPenguin.makeNoise();
58              percyPenguin.fly();
59
60              Wolf walterWolf = new Wolf("Walter the Wolf", "Meat", "Dog Village",
                ↪  45.5, 5, "Brown");
61
62              williamWolf.makeNoise();
63              walterWolf.makeNoise();
64              williamWolf.buildHome();
65              walterWolf.sleep();
66          }
67      }
68  }
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Prototype for a tiger as a type of feline
    /// </summary>
    class Tiger : Feline
    {
        private String _colourStripes;

        /// <summary>
        /// Constructor for a tiger as a type of feline
        /// </summary>
        /// <param name="name">The personal name of the tiger</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the tiger is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the tiger in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of tiger</param>
        public Tiger(String name, String diet, String location,
            double weight, int age, String colour, String species,
            String colourStripes)
            : base(name, diet, location, weight, age, colour, species)
        {
            _colourStripes = colourStripes;
        }

        /// <summary>
        /// The tiger eats meat
        /// </summary>
        public override void eat()
        {
            Console.WriteLine("{0}, eats 20lbs of meat", Name);
        }

        /// <summary>
        /// The tiger roars
        /// </summary>
        public override void makeNoise()
        {
            Console.WriteLine("ROARRRRRRRRRR");
        }

        /// <summary>
        /// The tiger makes it's home in the display
        /// </summary>
        public override void buildHome()
        {
            Console.WriteLine("{0} builds a lair", Name);
        }
    }
```

```
54 }
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Prototype for an eagle as a type of bird
    /// </summary>
    class Eagle : Bird
    {

        /// <summary>
        /// Constructor for an eagle as a type of bird
        /// </summary>
        /// <param name="name">The personal name of the eagle</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the eagle is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the eagle in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of bird</param>
        /// <param name="wingspan">The wingspan in centimetres</param>
        public Eagle(String name, String diet, String location,
            double weight, int age, String colour, String species,
            double wingSpan)
            : base(name, diet, location, weight, age, colour, species, wingSpan)
        { }

        /// <summary>
        /// Allows the eagle to roost in it's nest by laying an egg
        /// </summary>
        public void layEgg()
        {
            // code to allow eagles to lay eggs
            Console.WriteLine("{0} lays an egg. That's a feat of evolution", Name);
        }

        /// <summary>
        /// The eagle flies
        /// </summary>
        public override void fly()
        {
            // code to allow eagles to fly
            Console.WriteLine("{0} spreads his wings and flies", Name);
        }

        /// <summary>
        /// The eagle eats food
        /// </summary>
        public override void eat()
        {
            Console.WriteLine("{0} eats 1lb of fish", Name);
        }

```

```
54          /// <summary>
55          /// The eagle sleeps in it's nest
56          /// </summary>
57          public override void sleep()
58          {
59              Console.WriteLine("{0} rests in his nest, asleep", Name);
60          }
61
62          /// <summary>
63          /// The eagle squarks
64          /// </summary>
65          public override void makeNoise()
66          {
67              Console.WriteLine("{0} squarks", Name);
68          }
69
70          /// <summary>
71          /// The eagle builds it's nest
72          /// </summary>
73          public override void buildHome()
74          {
75              Console.WriteLine("{0} builds a nest", Name);
76          }
77      }
78  }
```

```cs
using System;

namespace Task02
{
    class Wolf : Animal
    {
        /// <summary>
        /// Constructor for a wolf as a type of animal
        /// </summary>
        /// <param name="name">The personal name of the wolf</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the wolf is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the wolf in years</param>
        /// <param name="colour">The dominant color(s)</param>
        public Wolf(String name, String diet, String location,
            double weight, int age, String colour)
            : base(name, diet, location, weight, age, colour)
        { }

        /// <summary>
        /// The wolf eats meat
        /// </summary>
        public override void eat()
        {
            Console.WriteLine("{0} eats 10lbs of meat", Name);
        }

        /// <summary>
        /// The wolf sleeps
        /// </summary>
        public override void sleep()
        {
            Console.WriteLine("{0} settles down in his den and sleeps", Name);
        }

        /// <summary>
        /// The wolf howls
        /// </summary>
        public override void makeNoise()
        {
            Console.WriteLine("{0} howls", Name);
        }

        /// <summary>
        /// The wolf makes it's den
        /// </summary>
        public override void buildHome()
        {
            Console.WriteLine("{0} builds a den", Name);
        }
    }
}
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Super class for all cats as a type of animal
    class Feline : Animal
    {
        private String _species;

        /// <summary>
        /// Constructor for an eagle as a type of feline
        /// </summary>
        /// <param name="name">The personal name of the feline</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the feline is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the feline in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of feline</param>
        public Feline(String name, String diet, String location,
            double weight, int age, String colour, String species)
            : base(name, diet, location, weight, age, colour)
        {
            _species = species;
        }

        /// <summary>
        /// Allows a cat to sleep
        /// </summary>
        public override void sleep()
        {
            Console.WriteLine("{0} lays down and goes to sleep", Name);
        }
    }
}
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Prototype for a lion as a type of feline
    /// </summary>
    class Lion : Feline
    {
        /// <summary>
        /// Constructor for a lion as a type of feline
        /// </summary>
        /// <param name="name">The personal name of the lion</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the lion is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the lion in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of lion</param>
        public Lion(String name, String diet, String location,
            double weight, int age, String colour, String species)
            : base(name, diet, location, weight, age, colour, species)
        { }

        /// <summary>
        /// The lion eats
        /// </summary>
        public override void eat()
        {
            Console.WriteLine("{0} eats 50lbs of meat", Name);
        }

        /// <summary>
        /// The lion roars bigly
        /// </summary>
        public override void makeNoise()
        {
            Console.WriteLine("BIIIIGGGG ROARRRRRRRRRR");
        }

        /// <summary>
        /// The lion builds the location for it's pride in the display
        /// </summary>
        public override void buildHome()
        {
            Console.WriteLine("{0} builds a den", Name);
        }
    }
}
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Prototype for a bird as type of animal
    /// </summary>
    class Bird : Animal
    {
        // Instance variables
        private String _species;
        private double _wingSpan;

        /// <summary>
        /// Constructor for a bird instance
        /// </summary>
        /// <param name="name">The personal name of the bird</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the bird is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the bird in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of bird</param>
        /// <param name="wingspan">The wingspan in centimetres</param>
        public Bird(String name, String diet, String location,
            double weight, int age, String colour, String species, double wingSpan)
            : base(name, diet, location, weight, age, colour)
        {
            _species = species;
            _wingSpan = wingSpan;
        }

        /// <summary>
        /// Allows the bird to sleep
        /// </summary>
        public override void sleep()
        {
            Console.WriteLine("{0} lays down and goes to sleep", Name);
        }

        /// <summary>
        /// Message posted when the bird tries to fly
        /// </summary>
        public virtual void fly()
        {
            // code to allow eagles to fly
            Console.WriteLine("{0} thinks about flying", Name);
        }
    }
}
```

```cs
using System;

namespace Task02
{
    /// <summary>
    /// Prototype for a penguin as a type of bird
    /// </summary>
    class Penguin : Bird
    {
        /// <summary>
        /// Constructor for a penguin as a type of bird
        /// </summary>
        /// <param name="name">The personal name of the penguin</param>
        /// <param name="diet">The primary type of food eaten</param>
        /// <param name="location">The exhibition the penguin is in</param>
        /// <param name="weight">Weight in pounds</param>
        /// <param name="age">Age of the penguin in years</param>
        /// <param name="colour">The dominant color(s)</param>
        /// <param name="species">The species of bird</param>
        /// <param name="wingspan">The wingspan in centimetres</param>
        public Penguin(String name, String diet, String location,
            double weight, int age, String colour, String species,
            double wingSpan)
            : base(name, diet, location, weight, age, colour, species, wingSpan)
        { }

        /// <summary>
        /// The penguin lays an egg
        /// </summary>
        public void layEgg()
        {
            // code to allow penguins to lay eggs
            Console.WriteLine("{0} lays an egg in the ice.", Name);
        }

        /// <summary>
        /// The penguin eats fish
        /// </summary>
        public override void eat()
        {
            Console.WriteLine("{0} eats 0.5lb of fish", Name);
        }

        /// <summary>
        /// The penguin sleeps
        /// </summary>
        public override void sleep()
        {
            Console.WriteLine("{0} rests in his nest, asleep", Name);
        }

        /// <summary>
        /// The penguin goes nowhere in the air
```

```
54          /// </summary>
55          public override void fly()
56          {
57              Console.WriteLine("{0} flaps his little wings and goes nowhere", Name);
58          }
59
60          /// <summary>
61          /// The penguin makes a penguin noise
62          /// </summary>
63          public override void makeNoise()
64          {
65              Console.WriteLine("{0} sneezes", Name);
66          }
67
68          /// <summary>
69          /// The penguin makes it's home
70          /// </summary>
71          public override void buildHome()
72          {
73              Console.WriteLine("{0} builds a rookery", Name);
74          }
75      }
76  }
```

```cs
using System;

namespace Program_3
{
    class Overloading
    {
        public static void methodToBeOverloaded(String name)
        {
            Console.WriteLine("Name: " + name);
        }

        public static void methodToBeOverloaded(String name, int age)
        {
            Console.WriteLine("Name: " + name + "\nAge: " + age);
        }

        static void Main(string[] args)
        {
            methodToBeOverloaded("John Doe");
            methodToBeOverloaded("Jane Doe", 24);
        }
    }
}
```

16 Bank Transactions

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

Outcome	**Weight**
Principles	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

Outcome	**Weight**
Build Programs	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

Outcome	**Weight**
Design	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

Outcome	**Weight**
Justify	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

Date	Author	Comment
2020/04/07 22:08	Peter Stacey	Ready to Mark
2020/04/07 22:09	Peter Stacey	My task 4 exercises will go up this weekend, and then the video for this will follow (I might refine some of the code a bit before then. It works as expected, but I think I can improve some of it with a bit more thought)
2020/04/09 23:14	Dipto Pratyaksa	when the going gets tough, the tough gets going
2020/04/09 23:14	Dipto Pratyaksa	Discuss
2020/04/27 14:23	Peter Stacey	Updated testing to use System.Diagnostics and Debug.Assert, to assist with testing, and made some minor changes to the transaction classes.
2020/04/28 11:47	Peter Stacey	Video link: https://youtu.be/D_ghX7zGfyA
2020/05/02 22:49	Dipto Pratyaksa	Great work, well done
2020/05/02 22:49	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Bank Transactions

Submitted By:
Peter Stacey
pstacey
2020/04/27 14:23

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦♦♦◊◊
Justify	♦♦♦◊◊

This exercise significantly develops the banking application and contains quite a lot of new code and business rules. The task provides the basic guidance and design, however all of the implementation approach needs to be developed and coded, across multiple files. The task involves using many of the elements of object oriented development we have learnt to date, as well as a number of the features of C# such as exceptions, flow-control and iteration, that are relevant in program development.

April 27, 2020

```
using System;

namespace Task_4._2P
{
    /// <summary>
    /// Prototype for a Withdraw transaction
    /// </summary>
    class WithdrawTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a WithdrawTransaction
        /// </summary>
        /// <param name="account">Account to withdraw from</param>
        /// <param name="amount">Amount to withdraw</param>
        public WithdrawTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Withdrawal amount must be >
                ↪  $0.00");
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of the withdrawal
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "WITHDRAW AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!_executed)
```

```
52              {
53                  Console.Write("{0, 20}|", "Pending");
54              }
55              else if (_reversed)
56              {
57                  Console.Write("{0, 20}|", "Withdraw reversed");
58              }
59              else if (_success)
60              {
61                  Console.Write("{0, 20}|", "Withdraw complete");
62              }
63              else if (!_success)
64              {
65                  Console.Write("{0, 20}|", "Insufficient funds");
66              }
67              Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
68              Console.WriteLine(new String('-', 85));
69          }
70
71          /// <summary>
72          /// Executes the withdrawal
73          /// </summary>
74          /// <exception cref="System.InvalidOperationException">Thrown
75          /// when the withdraw is already complete or insufficient funds</exception>
76          public void Execute()
77          {
78              if (_executed && _success)
79              {
80                  throw new InvalidOperationException("Withdraw previously executed");
81              }
82              _executed = true;
83
84              _success = _account.Withdraw(_amount);
85              if (!_success)
86              {
87                  throw new InvalidOperationException("Insufficient funds");
88              }
89          }
90
91          /// <summary>
92          /// Reverses the withdraw if previously executed successfully
93          /// </summary>
94          /// <exception cref="System.InvalidOperationException">Thrown
95          /// if already rolled back or if there are insufficient
96          /// funds to complete the rollback</exception>
97          public void Rollback()
98          {
99              if (_reversed)
100             {
101                 throw new InvalidOperationException("Transaction already reversed");
102             }
103             else if (!_success)
104             {
```

```
105                 throw new InvalidOperationException(
106                     "Withdraw not successfully executed. Nothing to rollback.");
107             }
108             _reversed = _account.Deposit(_amount); // Deposit returns boolean
109             if (!_reversed) // Deposit didn't occur
110             {
111                 throw new InvalidOperationException("Invalid amount");
112             }
113             _reversed = true;
114         }
115     }
116 }
```

```
using System;

namespace Task_4._2P
{
    /// <summary>
    /// Prototype for a deposit transaction
    /// </summary>
    class DepositTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a deposit transaction object
        /// </summary>
        /// <param name="account">Account to deposit into</param>
        /// <param name="amount">Amount to deposit</param>
        public DepositTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException(
                    "Deposit amount invalid: {0}", amount.ToString("C"));
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of a deposit
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "DEPOSIT AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!_executed)
```

```
53              {
54                  Console.Write("{0, 20}|", "Pending");
55              }
56              else if (_reversed)
57              {
58                  Console.Write("{0, 20}|", "Deposit reversed");
59              }
60              else if (_success)
61              {
62                  Console.Write("{0, 20}|", "Deposit complete");
63              }
64              else if (!_success)
65              {
66                  Console.Write("{0, 20}|", "Invalid deposit");
67              }
68              Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
69              Console.WriteLine(new String('-', 85));
70          }
71
72          /// <summary>
73          /// Executes a deposit transaction
74          /// </summary>
75          public void Execute()
76          {
77              if (_executed && _success)
78              {
79                  throw new InvalidOperationException("Deposit previously executed");
80              }
81              _executed = true;
82
83              _success = _account.Deposit(_amount);
84              if (!_success)
85              {
86                  _executed = false;
87                  throw new InvalidOperationException("Deposit amount invalid");
88              }
89          }
90
91          /// <summary>
92          /// Reverses a deposit if previously executed successfully
93          /// </summary>
94          public void Rollback()
95          {
96              if (_reversed)
97              {
98                  throw new InvalidOperationException("Transaction already reversed");
99              }
100             else if (!_success)
101             {
102                 throw new InvalidOperationException(
103                     "Deposit not successfully executed. Nothing to rollback.");
104             }
105             _reversed = _account.Withdraw(_amount); // Withdraw returns boolean
```

```
106             if (!_reversed) // Withdraw didn't occur
107             {
108                 throw new InvalidOperationException("Insufficient funds to
                    ↪ rollback");
109             }
110             _reversed = true;
111         }
112     }
113 }
```

```
using System;

namespace Task_4._2P
{
    /// <summary>
    /// Prototype for a transfer transaction
    /// </summary>
    class TransferTransaction
    {
        // Instance variables
        private Account _fromAccount;
        private Account _toAccount;
        private decimal _amount;
        private DepositTransaction _deposit;
        private WithdrawTransaction _withdraw;
        private bool _executed;
        private bool _reversed;

        // Properties
        public bool Executed { get => _executed; }
        public bool Reversed { get => _reversed; }
        public bool Success { get => (_deposit.Success && _withdraw.Success); }

        /// <summary>
        /// Constructor for a transfer transaction
        /// </summary>
        /// <param name="fromAccount">The account to transfer from</param>
        /// <param name="toAccount">The account to transfer to</param>
        /// <param name="amount">The amount to transfer</param>
        /// <exception cref="System.ArgumentOutOfRangeException">Thrown
        /// when the amount is negative</exception>
        public TransferTransaction(Account fromAccount, Account toAccount, decimal
        ↪  amount)
        {
            _fromAccount = fromAccount;
            _toAccount = toAccount;
            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("Negative transfer amount");
                ↪  // THIS IS NOT GOOD. NEED TO ADJUST THIS
            }
            _amount = amount;

            _withdraw = new WithdrawTransaction(_fromAccount, _amount);
            _deposit = new DepositTransaction(_toAccount, _amount);
        }

        /// <summary>
        /// Prints the details of the transfer
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
```

```
52              Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
53                  "FROM ACCOUNT", "To ACCOUNT", "TRANSFER AMOUNT", "STATUS");
54              Console.WriteLine(new String('-', 85));
55              Console.Write("|{0, -20}|{1, 20}|{2, 20}|", _fromAccount.Name,
                ↪  _toAccount.Name, _amount.ToString("C"));
56              if (!_executed)
57              {
58                  Console.WriteLine("{0, 20}|", "Pending");
59              }
60              else if (_reversed)
61              {
62                  Console.WriteLine("{0, 20}|", "Transfer reversed");
63              }
64              else if (Success)
65              {
66                  Console.WriteLine("{0, 20}|", "Transfer complete");
67              }
68              else if (!Success)
69              {
70                  Console.WriteLine("{0, 20}|", "Transfer failed");
71              }
72              Console.WriteLine(new String('-', 85));
73          }
74
75          /// <summary>
76          /// Executes the transfer
77          /// </summary>
78          /// <exception cref="System.InvalidOperationException">Thrown
79          /// when previously executed or deposit or withdraw fail</exception>
80          public void Execute()
81          {
82              if (_executed)
83              {
84                  throw new InvalidOperationException("Transfer previously executed");
85              }
86              _executed = true;
87
88              try
89              {
90                  _withdraw.Execute();
91              }
92              catch (InvalidOperationException exception)
93              {
94                  Console.WriteLine("Transfer failed with reason: " +
                    ↪  exception.Message);
95                  _withdraw.Print();
96              }
97
98              if (_withdraw.Success)
99              {
100                 try
101                 {
102                     _deposit.Execute();
```

```
103                 }
104                 catch (InvalidOperationException exception)
105                 {
106                     Console.WriteLine("Transfer failed with reason: " +
                        ↪ exception.Message);
107                     _deposit.Print();
108                     try
109                     {
110                         Rollback();
111                     }
112                     catch (InvalidOperationException e)
113                     {
114                         Console.WriteLine("Withdraw could not be reversed with
                            ↪ reason: " + e.Message);
115                         _withdraw.Print();
116                     }
117                 }
118             }
119             Print();
120             _deposit.Print();
121             _withdraw.Print();
122         }
123 
124         /// <summary>
125         /// Rolls the transfer back
126         /// </summary>
127         /// <exception cref="System.InvalidOperationException">Thrown
128         /// when the rollback has already been executed or it fails</exception>
129         public void Rollback()
130         {
131             if (!_executed)
132             {
133                 throw new InvalidOperationException("Transfer not executed. Nothing
                    ↪ to rollback.");
134             }
135 
136             if (_reversed)
137             {
138                 throw new InvalidOperationException("Transfer already rolled back");
139             }
140 
141             if (this.Success)
142             {
143                 try
144                 {
145                     _deposit.Rollback();
146                 }
147                 catch (InvalidOperationException exception)
148                 {
149                     Console.WriteLine("Failed to rollback deposit: "
150                         + exception.Message);
151                     return;
152                 }
```

```
153
154                 try
155                 {
156                     _withdraw.Rollback();
157                 }
158                 catch (InvalidOperationException exception)
159                 {
160                     Console.WriteLine("Failed to rollback withdraw: "
161                         + exception.Message);
162                     return;
163                 }
164             }
165             _reversed = true;
166         }
167     }
168 }
```

```
using System;
using System.Diagnostics;

namespace Task_4._2P
{
    enum MenuOption
    {
        Withdraw,
        Deposit,
        Transfer,
        Print,
        Quit
    }

    /// <summary>
    /// BankSystem implements a banking system to operate on accounts
    /// </summary>
    class BankSystem
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }

```

```
54          // Reads integer input in the console between two numbers
55          /// <summary>
56          /// Reads integer input in the console between two numbers
57          /// </summary>
58          /// <returns>
59          /// The input of the user as an integer
60          /// </returns>
61          /// <param name="prompt">The string prompt for the user</param>
62          /// <param name="minimum">The minimum number allowed</param>
63          /// <param name="maximum">The maximum number allowed</param>
64          public static int ReadInteger(String prompt, int minimum, int maximum)
65          {
66              int number = ReadInteger(prompt);
67              while (number < minimum || number > maximum)
68              {
69                  Console.WriteLine("Please enter a whole number from " +
70                                    minimum + " to " + maximum);
71                  number = ReadInteger(prompt);
72              }
73              return number;
74          }
75
76          // Reads decimal input in the console
77          /// <summary>
78          /// Reads decimal input in the console
79          /// </summary>
80          /// <returns>
81          /// The input of the user as a decimal
82          /// </returns>
83          /// <param name="prompt">The string prompt for the user</param>
84          public static decimal ReadDecimal(String prompt)
85          {
86              decimal number = 0;
87              string numberInput = ReadString(prompt);
88              while (!(decimal.TryParse(numberInput, out number)) || number <= 0)
89              {
90                  Console.WriteLine("Please enter a decimal number greater than
                    ↪  $0.00");
91                  numberInput = ReadString(prompt);
92              }
93              return Convert.ToDecimal(numberInput);
94          }
95
96          /// <summary>
97          /// Displays a menu of possible actions for the user to choose
98          /// </summary>
99          private static void DisplayMenu()
100         {
101             Console.WriteLine("\n**********************");
102             Console.WriteLine("*       Menu         *");
103             Console.WriteLine("**********************");
104             Console.WriteLine("*  1. Withdraw       *");
105             Console.WriteLine("*  2. Deposit        *");
```

```
106             Console.WriteLine("*  3. Transfer     *");
107             Console.WriteLine("*  4. Print        *");
108             Console.WriteLine("*  5. Quit         *");
109             Console.WriteLine("********************");
110         }
111 
112         /// <summary>
113         /// Returns a menu option chosen by the user
114         /// </summary>
115         /// <returns>
116         /// MenuOption chosen by the user
117         /// </returns>
118         static MenuOption ReadUserOption()
119         {
120             DisplayMenu();
121             int option = ReadInteger("Choose an option", 1,
122                 Enum.GetNames(typeof(MenuOption)).Length);
123             return (MenuOption)(option - 1);
124         }
125 
126         /// <summary>
127         /// Attempts to deposit funds into an account
128         /// </summary>
129         /// <param name="account">The account to deposit into</param>
130         static void DoDeposit(Account account)
131         {
132             decimal amount = ReadDecimal("Enter the amount");
133             DepositTransaction transaction = new DepositTransaction(account,
                ↪ amount);
134             try
135             {
136                 transaction.Execute();
137             }
138             catch (InvalidOperationException)
139             {
140                 transaction.Print();
141                 return;
142             }
143             transaction.Print();
144         }
145 
146         /// <summary>
147         /// Attempts to withdraw funds from an account
148         /// </summary>
149         /// <param name="account">The account to withdraw from</param>
150         static void DoWithdraw(Account account)
151         {
152             decimal amount = ReadDecimal("Enter the amount");
153             WithdrawTransaction transaction = new WithdrawTransaction(account,
                ↪ amount);
154             try
155             {
156                 transaction.Execute();
```

```
157             }
158             catch (InvalidOperationException)
159             {
160                 transaction.Print();
161                 return;
162             }
163             transaction.Print();
164         }
165 
166         /// <summary>
167         /// Attempts to transfer funds between accounts
168         /// </summary>
169         /// <param name="account">The account to withdraw from</param>
170         static void DoTransfer(Account fromAccount, Account toAccount)  // this is
            ↪  temporary until we add multiple accounts in task 6.2
171         {
172             decimal amount = ReadDecimal("Enter the amount");
173             try
174             {
175                 TransferTransaction transfer = new TransferTransaction(fromAccount,
                    ↪  toAccount, amount);
176                 transfer.Execute();
177             }
178             catch (Exception)
179             {
180                 // Currently this is handled in the TransferTransaction. This will
                    ↪  be changed
181             }
182         }
183 
184         /// <summary>
185         /// Outputs the account name and balance
186         /// </summary>
187         /// <param name="account">The account to print</param>
188         static void DoPrint(Account account)
189         {
190             account.Print();
191         }
192 
193         static void Main(string[] args)
194         {
195             /*****************************************************
196              *  TESTS
197              *****************************************************/
198             Account acc = new Account("Peter Stacey");
199             Account acc1 = new Account("Jane Doe", 100);
200             Account acc2 = new Account("John Doe", -500);
201 
202             Debug.Assert(acc.Balance == 0);
203             Debug.Assert(acc1.Balance == 100);
204             Debug.Assert(acc2.Balance == 0);
205 
206             // Test deposit success and rollback
```

```
207             DepositTransaction dep = new DepositTransaction(acc, 500);
208
209             dep.Print();
210             dep.Execute();
211             Debug.Assert(acc.Balance == 500);
212             Debug.Assert(dep.Executed == true);
213             Debug.Assert(dep.Success == true);
214             dep.Print();
215
216             dep.Rollback();
217             Debug.Assert(acc.Balance == 0);
218             Debug.Assert(dep.Reversed == true);
219             dep.Print();
220
221             Console.WriteLine("\n\n");
222
223             // Test withdraw success and rollback
224             WithdrawTransaction with = new WithdrawTransaction(acc1, 50);
225
226             with.Print();
227             with.Execute();
228             Debug.Assert(acc1.Balance == 50);
229             Debug.Assert(with.Executed == true);
230             Debug.Assert(with.Success == true);
231             with.Print();
232
233             with.Rollback();
234             Debug.Assert(acc1.Balance == 100);
235             Debug.Assert(with.Reversed == true);
236             with.Print();
237
238             Console.WriteLine("\n\n");
239
240             // Test transfer success and rollback
241             TransferTransaction tran = new TransferTransaction(acc1, acc, 50);
242
243             tran.Print();
244             tran.Execute();
245             Debug.Assert(acc.Balance == 50);
246             Debug.Assert(acc1.Balance == 50);
247             Debug.Assert(tran.Executed == true);
248             Debug.Assert(tran.Success == true);
249
250             tran.Rollback();
251             Debug.Assert(acc.Balance == 0);
252             Debug.Assert(acc1.Balance == 100);
253             Debug.Assert(tran.Reversed == true);
254             tran.Print();
255
256             Console.WriteLine("\n\n");
257
258             // Test withdraw failure when there is insufficient funds to complete
                ↪  the transaction
```

```
259             // followed by repeating the withdraw after funds are deposited
260             WithdrawTransaction with2 = new WithdrawTransaction(acc, 100);
261
262             with2.Print();
263             try
264             {
265                 with2.Execute();
266             }
267             catch (InvalidOperationException exception)
268             {
269                 Console.WriteLine(exception.Message);
270             }
271
272             Debug.Assert(acc.Balance == 0);
273             Debug.Assert(with2.Success == false);
274             Debug.Assert(with2.Executed == true);
275             with2.Print();
276
277             DepositTransaction dep2 = new DepositTransaction(acc, 500);
278
279             dep2.Execute();
280             dep2.Print();
281
282             try
283             {
284                 with2.Execute();
285             }
286             catch (InvalidOperationException exception)
287             {
288                 Console.WriteLine(exception.Message);
289             }
290
291             Debug.Assert(acc.Balance == 400);
292             Debug.Assert(with2.Success == true);
293             Debug.Assert(with2.Executed == true);
294             with2.Print();
295
296             Console.WriteLine("\n\n");
297
298             // Test fail to rollback before deposit or withdraw are
299             // complete
300             DepositTransaction dep3 = new DepositTransaction(acc, 500);
301             WithdrawTransaction with3 = new WithdrawTransaction(acc, 500);
302             TransferTransaction tran2 = new TransferTransaction(acc, acc1, 200);
303
304             try
305             {
306                 dep3.Rollback();
307             }
308             catch (InvalidOperationException exception)
309             {
310                 Console.WriteLine(exception.Message);
311             }
```

```
312
313             try
314             {
315                 with3.Rollback();
316             }
317             catch (InvalidOperationException exception)
318             {
319                 Console.WriteLine(exception.Message);
320             }
321
322             try
323             {
324                 tran2.Rollback();
325             }
326             catch (InvalidOperationException exception)
327             {
328                 Console.WriteLine(exception.Message);
329             }
330
331             Console.WriteLine("\n\n");
332
333             // Try to rollback deposit from account with insufficient funds
334             DepositTransaction dep4 = new DepositTransaction(acc2, 100);
335             WithdrawTransaction with4 = new WithdrawTransaction(acc2, 100);
336
337             dep4.Execute();
338             with4.Execute();
339             try
340             {
341                 dep4.Rollback();
342             }
343             catch (Exception exception)
344             {
345                 Console.WriteLine(exception.Message);
346             }
347
348             Console.WriteLine("\n\n");
349
350             /*********************************************************
351              *  CLI
352              *********************************************************/
353             do
354             {
355                 MenuOption chosen = ReadUserOption();
356                 switch (chosen)
357                 {
358                     case MenuOption.Withdraw:
359                         DoWithdraw(acc); break;
360                     case MenuOption.Deposit:
361                         DoDeposit(acc); break;
362                     case MenuOption.Transfer:
363                         DoTransfer(acc, acc1); break;
364                     case MenuOption.Print:
```

```
365                         DoPrint(acc); break;
366                     case MenuOption.Quit:
367                     default:
368                         Console.WriteLine("Goodbye");
369                         System.Environment.Exit(0); // terminates the program
370                         break; // unreachable
371                 }
372             } while (true);
373         }
374     }
375 }
```

```
using System;

namespace Task_4._2P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get => _name; }
        public decimal Balance { get => _balance; }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance <= 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if ((amount < 0) || (amount == decimal.MaxValue))
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
```

```
54          /// <param name="amount">The amount to subtract from the balance</param>
55          public Boolean Withdraw(decimal amount)
56          {
57              if ((amount < 0) || (amount > _balance))
58                  return false;
59
60              _balance -= amount;
61              return true;
62          }
63
64          /// <summary>
65          /// Outputs the account name and current balance as a string
66          /// </summary>
67          public void Print()
68          {
69              Console.WriteLine("Account Name: {0}, Balance: {1}",
70                  _name, _balance.ToString("C"));
71          }
72      }
73  }
```

17 C# Essentials: Polymorphism

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦◊◊◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

Outcome	Weight
Principles	♦♦♦♦◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

Outcome	Weight
Build Programs	♦♦◊◊◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

Outcome	Weight
Design	♦◊◊◊◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

Outcome	Weight
Justify	♦♦♦◊◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

Date	Author	Comment
2020/04/28 08:48	Peter Stacey	Ready to Mark
2020/04/30 12:06	Peter Stacey	Video Likn: https://youtu.be/mqnJkiKIwbQ
2020/05/03 16:46	Dipto Pratyaksa	Great work demonstrating the parent class will have default implementation, while allowing inheritance to have their own implementation unique to their character. Great OOP in action!
2020/05/03 16:46	Dipto Pratyaksa	Discuss
2020/05/03 16:46	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

C# Essentials: Polymorphism

Submitted By:
Peter Stacey
pstacey
2020/04/28 08:48

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦♦◊
Build Programs	♦♦◊◊◊
Design	♦◊◊◊◊
Justify	♦♦♦◊◊

The task explores polymorphism, which is one of the code principles of OOP and directly related to the principles outcome. There is no code design in this task, however there is evaluation of code and understanding what the output will be.The task implements a small set of classes, which aligns with the code outcome and we develop a UML diagram, which relates both the design and to evaluating code and representing it using standard conventions.

April 28, 2020

```cs
using System;

namespace Task_6_1P
{
    /// <summary>
    /// Prototype for a Bird and base class for more specific bird classes
    /// </summary>
    class Bird
    {
        // Instance variables
        public string Name { get; set; }

        /// <summary>
        /// Creates a new Bird
        /// </summary>
        public Bird()
        {

        }

        /// <summary>
        /// Allows the bird to fly
        /// </summary>
        public virtual void fly()
        {
            Console.WriteLine("Flap, Flap, Flap");
        }

        /// <summary>
        /// Returns a string representation of a bird
        /// </summary>
        /// <returns>
        /// String representation of a Bird
        /// </returns>
        public override string ToString()
        {
            return "A bird named " + Name;
        }
    }
}
```

```cs
using System;

namespace Task_6_1P
{
    /// <summary>
    /// Prototype for a Penguin as a type of Bird
    /// </summary>
    class Penguin : Bird
    {
        /// <summary>
        /// Prevents a Penguin from flying
        /// </summary>
        public override void fly()
        {
            Console.WriteLine("Penguins cannot fly");
        }

        /// <summary>
        /// Returns a string representation of a Penguin
        /// </summary>
        /// <returns>
        /// String representation of a Penguin
        /// </returns>
        public override string ToString()
        {
            return "A penguin named " + Name;
        }
    }
}
```

```cs
using System;

namespace Task_6_1P
{
    /// <summary>
    /// Prototype for a Duck as a type of Bird
    /// </summary>
    class Duck : Bird
    {
        // Instance variables
        public double Size { get; set; }
        public string Kind { get; set; }

        /// <summary>
        /// Returns a string representation of a Duck
        /// </summary>
        /// <returns>
        /// String representation of a Duck
        /// </returns>
        public override string ToString()
        {
            return "A duck named " + Name + " is a " + Size + " inch " + Kind;
        }
    }
}
```

```
using System;
using System.Collections.Generic;

namespace Task_6_1P
{
    class Program
    {
        static void Main(string[] args)
        {
            // Step 2 - Part 1
            Bird bird1 = new Bird();
            Bird bird2 = new Bird();

            bird1.Name = "Feathers";
            bird2.Name = "Polly";

            Console.WriteLine(bird1.ToString());
            bird1.fly();

            Console.WriteLine(bird2.ToString());
            bird2.fly();

            // Step 2 - Part 2
            Penguin penguin1 = new Penguin();
            Penguin penguin2 = new Penguin();

            penguin1.Name = "Happy Feet";
            penguin2.Name = "Gloria";

            Console.WriteLine(penguin1.ToString());
            penguin1.fly();

            Console.WriteLine(penguin2.ToString());
            penguin2.fly();

            Duck duck1 = new Duck();
            Duck duck2 = new Duck();

            duck1.Name = "Daffy";
            duck1.Size = 15;
            duck1.Kind = "Mallard";

            duck2.Name = "Donald";
            duck2.Size = 20;
            duck2.Kind = "Decoy";

            Console.WriteLine(duck1.ToString());
            Console.WriteLine(duck2.ToString());

            // Step 2 - Part 3
            List<Bird> birds = new List<Bird>();
            Bird bird3 = new Bird();
            bird3.Name = "Feathers";
```

```
54              Bird bird4 = new Bird();
55              bird4.Name = "Polly";
56
57              Penguin penguin3 = new Penguin();
58              penguin3.Name = "Happy Feet";
59              Penguin penguin4 = new Penguin();
60              penguin4.Name = "Gloria";
61
62              Duck duck3 = new Duck();
63              duck3.Name = "Daffy";
64              duck3.Size = 15;
65              duck3.Kind = "Mallard";
66
67              Duck duck4 = new Duck();
68              duck4.Name = "Donald";
69              duck4.Size = 20;
70              duck4.Kind = "Decoy";
71
72              birds.Add(bird3);
73              birds.Add(bird4);
74              birds.Add(penguin3);
75              birds.Add(penguin4);
76              birds.Add(duck3);
77              birds.Add(duck4);
78
79              birds.Add(new Bird { Name = "Birdy" });
80
81              foreach (Bird bird in birds)
82              {
83                  Console.WriteLine(bird);
84              }
85
86              // Part 3
87              Duck duck5 = new Duck();
88              duck5.Name = "Daffy";
89              duck5.Size = 15;
90              duck5.Kind = "Mallard";
91
92              Duck duck6 = new Duck();
93              duck6.Name = "Donald";
94              duck6.Size = 20;
95              duck6.Kind = "Decoy";
96
97              List<Duck> ducksToAdd = new List<Duck>()
98              {
99                  duck5,
100                 duck6
101             };
102
103             IEnumerable<Bird> upcastDucks = ducksToAdd;
104
105             List<Bird> birds2 = new List<Bird>();
106             birds2.Add(new Bird() { Name = "Feather" });
```

```
107
108             birds2.AddRange(upcastDucks);
109
110             foreach (Bird bird in birds2)
111             {
112                 Console.WriteLine(bird);
113             }
114         }
115     }
116 }
```

SIT232 – Object Oriented Development Task 6.1P- Report on Polymorphism

Student Name: Peter Stacey

Student ID: 219011171

UML Diagram

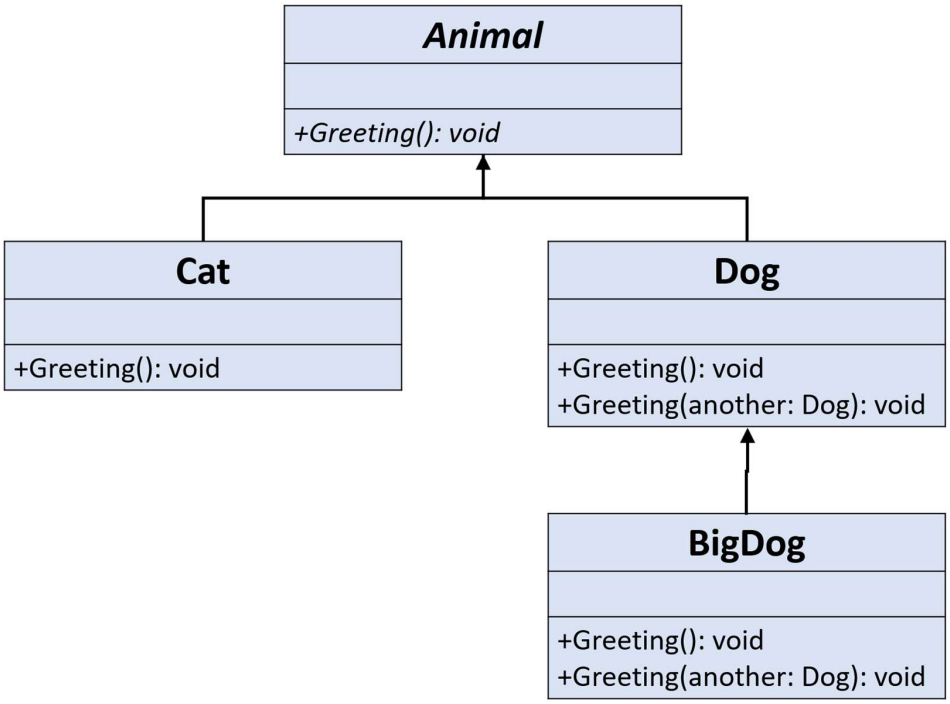


Figure 1: UML Diagram of Inheritance Hierarchy

Explanation of outputs

Ignoring the syntax error, with the classes defining methods Greeting (capitalized) and the TestAnimal class calling greeting (all lower-case), the following outputs and or errors occur:

Lines	Output/Error
`Cat cat1 = new Cat();` `cat1.Greeting();`	Cat: meow
`Dog dog1 = new Dog();` `dog1.Greeting();`	Dog: woof
`BigDog bigDog1 = new BigDog();` `bigDog1.Greeting();`	BigDog: Woow
`Animal animal1 = new Cat();` `animal1.Greeting();`	Cat: meow
`Animal animal2 = new Dog();` `animal2.Greeting();`	Dog: woof
`Animal animal3 = new BigDog();` `animal3.Greeting();`	BigDog: Woow
`Animal animal4 = new Animal();`	Error: Abstract classes cannot be instantiated

`Dog dog2 = (Dog)animal2;`
`BigDog bigDog2 = (BigDog)animal3;`
`Dog dog3 = (Dog)animal3;`
	No output from these lines, but also no problems. These are all valid casts.

`Cat cat2 = (Cat)animal2;`	Error: This is an invalid cast, as animal2 has already been cast to a Dog, which is not in the same branch of the inheritance tree.

`dog2.Greeting(dog3);`
`dog3.Greeting(dog2);`
`dog2.Greeting(bigDog2);`
`bigDog2.Greeting(dog2);`
`bigDog2.Greeting(bigDog1);`
	Dog: wooooooffffff
	Dog: wooooooffffff
	Dog: wooooooffffff
	Wooooooooowwwww
	Wooooooooowwwww
	
	In all cases, the calls are to the overloaded Greeting that accepts another Dog as a parameter, so the output matches the body of those methods.

18 Multiple Bank Accounts

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

Outcome	**Weight**
Principles	♦♦♦◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

Outcome	**Weight**
Build Programs	♦♦◊◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

Outcome	**Weight**
Design	♦♦♦◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

Outcome	**Weight**
Justify	♦♦◊◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

Date	Author	Comment
2020/04/28 07:15	Peter Stacey	Ready to Mark
2020/05/01 10:56	Peter Stacey	Ready to Mark
2020/05/01 10:56	Peter Stacey	Realised a small oversight in my code when I was finishing the video this morning - I hadn't updated the DoDeposit, DoWithdraw and DoTransfer methods to actually use the passed in Bank parameter. That has been fixed.
2020/05/01 11:06	Peter Stacey	Video Likn: https://youtu.be/U8Z1yslwFWE
2020/05/03 16:53	Dipto Pratyaksa	Thank you and well done in submitting advanced task early.Efficient FindAccount method, but your DoTransfer need to have 2 parameters, one for source, one for destination, otherwise you will end-up transferring to the same account!
2020/05/03 16:53	Dipto Pratyaksa	Fix and Resubmit
2020/05/03 17:02	Dipto Pratyaksa	Can you double check or explain to me how work out this bit?
2020/05/03 17:03	Dipto Pratyaksa	Demonstrate
2020/05/03 20:30	Peter Stacey	Looking at my submission, my DoTransfer does have 2 accounts. In 6.2 we switch the methods to taking a Bank parameter (that is for all of the methods - DoDeposit, DoWithdraw, DoTransfer and DoPrint).That is explained in Step 4 (on page 2) of the Task Sheet.So my DoTransfer now takes a Bank as the parameter and then once the accounts are returned using the FindAccount method, it sets up a TransferTransaction that passes the from and to accounts, and the amount.Looking at Page 4 of my uploaded submission, it all looks ok
2020/05/03 20:33	Peter Stacey	Step 4 and for DoTransfer, Step 5 of the Task sheet. Apoology. Part 4 only discusses updating the DoDeposit method. Step 5 covers updating the others.
2020/05/03 20:38	Peter Stacey	image comment
2020/05/03 20:39	Peter Stacey	quick explanation with comments on top of the method as it is in my submission
2020/05/03 20:45	Peter Stacey	image comment
2020/05/03 20:45	Peter Stacey	This is what I based that approach on
2020/05/03 23:23	Dipto Pratyaksa	Good work, you have been going through all requirements very thoroughly and demonstrated you understood the concept well. I like your demo video, with nice implementation of visuals (red boxes) you could really develop and market what you created. Rare combo in IT industry
2020/05/03 23:23	Dipto Pratyaksa	Complete
2020/05/04 07:26	Peter Stacey	Thanks for the feedback. When we had to put videos together this semester I imagined that for you, they would be pretty boring and it would be a real downer to have to watch lots of videos because making videos isn't something that comes naturally. So I just decided to try to make them into something that isn't so boring for you. Glad that is working out ok.

Deakin University

Object Oriented Development

OnTrack Submission

Multiple Bank Accounts

Submitted By:
Peter Stacey
pstacey
2020/05/01 10:56

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦◊◊◊
Design	♦♦♦◊◊
Justify	♦♦◊◊◊

As a continuation of the banking application, overall the app is growing into a larger design and work of coding, however this individual task, while important, was relatively small. It involved evaluating our existing code in order to extend it with new features, and the addition of a new class, which involves new code writing. My video will include additional evidence to align with the learning outcomes.

May 1, 2020

```
using System;
using System.Diagnostics;

namespace Task_6._2P
{
    enum MenuOption
    {
        CreateAccount,
        Withdraw,
        Deposit,
        Transfer,
        Print,
        Quit
    }

    /// <summary>
    /// BankSystem implements a banking system to operate on accounts
    /// </summary>
    class BankSystem
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }
```

```
54
55          // Reads integer input in the console between two numbers
56          /// <summary>
57          /// Reads integer input in the console between two numbers
58          /// </summary>
59          /// <returns>
60          /// The input of the user as an integer
61          /// </returns>
62          /// <param name="prompt">The string prompt for the user</param>
63          /// <param name="minimum">The minimum number allowed</param>
64          /// <param name="maximum">The maximum number allowed</param>
65          public static int ReadInteger(String prompt, int minimum, int maximum)
66          {
67              int number = ReadInteger(prompt);
68              while (number < minimum || number > maximum)
69              {
70                  Console.WriteLine("Please enter a whole number from " +
71                                    minimum + " to " + maximum);
72                  number = ReadInteger(prompt);
73              }
74              return number;
75          }
76
77          // Reads decimal input in the console
78          /// <summary>
79          /// Reads decimal input in the console
80          /// </summary>
81          /// <returns>
82          /// The input of the user as a decimal
83          /// </returns>
84          /// <param name="prompt">The string prompt for the user</param>
85          public static decimal ReadDecimal(String prompt)
86          {
87              decimal number = 0;
88              string numberInput = ReadString(prompt);
89              while (!(decimal.TryParse(numberInput, out number)) || number < 0)
90              {
91                  Console.WriteLine("Please enter a decimal number, $0.00 or
                    ↪  greater");
92                  numberInput = ReadString(prompt);
93              }
94              return Convert.ToDecimal(numberInput);
95          }
96
97          /// <summary>
98          /// Displays a menu of possible actions for the user to choose
99          /// </summary>
100         private static void DisplayMenu()
101         {
102             Console.WriteLine("\n**********************");
103             Console.WriteLine("*       Menu       *");
104             Console.WriteLine("**********************");
105             Console.WriteLine("*  1. New Account  *");
```

```
106             Console.WriteLine("*  2. Withdraw     *");
107             Console.WriteLine("*  3. Deposit      *");
108             Console.WriteLine("*  4. Transfer     *");
109             Console.WriteLine("*  5. Print        *");
110             Console.WriteLine("*  6. Quit         *");
111             Console.WriteLine("********************");
112         }
113
114         /// <summary>
115         /// Returns a menu option chosen by the user
116         /// </summary>
117         /// <returns>
118         /// MenuOption chosen by the user
119         /// </returns>
120         static MenuOption ReadUserOption()
121         {
122             DisplayMenu();
123             int option = ReadInteger("Choose an option", 1,
124                 Enum.GetNames(typeof(MenuOption)).Length);
125             return (MenuOption)(option - 1);
126         }
127
128         /// <summary>
129         /// Attempts to deposit funds into an account at a bank
130         /// </summary>
131         /// <param name="bank">The bank holding the account to deposit into</param>
132         static void DoDeposit(Bank bank)
133         {
134             Account account = FindAccount(bank);
135             if (account != null)
136             {
137                 decimal amount = ReadDecimal("Enter the amount");
138                 DepositTransaction transaction = new DepositTransaction(account,
                    ↪ amount);
139                 try
140                 {
141                     bank.ExecuteTransaction(transaction);
142                 }
143                 catch (InvalidOperationException)
144                 {
145                     transaction.Print();
146                     return;
147                 }
148                 transaction.Print();
149             }
150         }
151
152         /// <summary>
153         /// Attempts to withdraw funds from an account at a bank
154         /// </summary>
155         /// <param name="bank">The bank holding account to withdraw from</param>
156         static void DoWithdraw(Bank bank)
157         {
```

```
158             Account account = FindAccount(bank);
159             if (account != null)
160             {
161                 decimal amount = ReadDecimal("Enter the amount");
162                 WithdrawTransaction transaction = new WithdrawTransaction(account,
                    ↪ amount);
163                 try
164                 {
165                     bank.ExecuteTransaction(transaction);
166                 }
167                 catch (InvalidOperationException)
168                 {
169                     transaction.Print();
170                     return;
171                 }
172                 transaction.Print();
173             }
174         }
175 
176         /// <summary>
177         /// Attempts to transfer funds between accounts
178         /// </summary>
179         /// <param name="bank">The bank holding the account
180         /// to transfer between</param>
181         static void DoTransfer(Bank bank)
182         {
183             Console.WriteLine("Transfer from:");
184             Account from = FindAccount(bank);
185             Console.WriteLine("Transfer to:");
186             Account to = FindAccount(bank);
187             if (from != null && to != null)
188             {
189                 decimal amount = ReadDecimal("Enter the amount");
190                 try
191                 {
192                     TransferTransaction transaction = new TransferTransaction(from,
                        ↪ to, amount);
193                     bank.ExecuteTransaction(transaction);
194                     transaction.Print();
195                 }
196                 catch (Exception)
197                 {
198                     // Currently this is handled in the TransferTransaction. This
                        ↪ will be changed
199                 }
200             }
201         }
202 
203         /// <summary>
204         /// Outputs the account name and balance
205         /// </summary>
206         /// <param name="account">The account to print</param>
207         static void DoPrint(Bank bank)
```

```
208         {
209             Account account = FindAccount(bank);
210             if (account != null)
211             {
212                 account.Print();
213             }
214         }
215 
216         /// <summary>
217         /// Creates a new account and adds it to the Bank
218         /// </summary>
219         /// <param name="bank">The bank to create the account in</param>
220         static void CreateAccount(Bank bank)
221         {
222             string name = ReadString("Enter account name");
223             decimal balance = ReadDecimal("Enter the opening balance");
224             bank.AddAccount(new Account(name, balance));
225         }
226 
227         private static Account FindAccount(Bank bank)
228         {
229             Account account = null;
230             string name = ReadString("Enter the account name");
231             account = bank.GetAccount(name);
232             if (account == null)
233             {
234                 Console.WriteLine("That account name does not exist at this bank");
235             }
236             return account;
237         }
238 
239         static void Main(string[] args)
240         {
241             Bank bank = new Bank();
242 
243             do
244             {
245                 MenuOption chosen = ReadUserOption();
246                 switch (chosen)
247                 {
248                     case MenuOption.CreateAccount:
249                         CreateAccount(bank); break;
250 
251                     case MenuOption.Withdraw:
252                         DoWithdraw(bank); break;
253 
254                     case MenuOption.Deposit:
255                         DoDeposit(bank); break;
256 
257                     case MenuOption.Transfer:
258                         DoTransfer(bank); break;
259 
260                     case MenuOption.Print:
```

```
261                         DoPrint(bank); break;
262
263                     case MenuOption.Quit:
264                     default:
265                         Console.WriteLine("Goodbye");
266                         System.Environment.Exit(0); // terminates the program
267                         break; // unreachable
268                 }
269             } while (true);
270         }
271     }
272 }
```

```
using System;
using System.Collections.Generic;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a bank to hold accounts
    /// </summary>
    class Bank
    {
        // Instance variables
        private List<Account> _accounts;

        /// <summary>
        /// Creates an empty bank object with a list for accounts
        /// </summary>
        public Bank()
        {
            _accounts = new List<Account>();
        }

        /// <summary>
        /// Adds an account to the Bank accounts register
        /// </summary>
        /// <param name="account"></param>
        public void AddAccount(Account account)
        {
            _accounts.Add(account);
        }

        /// <summary>
        /// Returns the first Account corresponding to the name, or
        /// null if there is no account matching the criteria
        /// </summary>
        /// <param name="name"></param>
        /// <returns>
        /// Account matching the provided name, or null
        /// </returns>
        public Account GetAccount(string name)
        {
            foreach (Account account in _accounts)
            {
                if (account.Name == name)
                {
                    return account;
                }
            }
            return null;
        }

        /// <summary>
        /// Executes a deposit into an account
        /// </summary>
```

```
54          /// <param name="transaction">DepositTransaction to execute</param>
55          public void ExecuteTransaction(DepositTransaction transaction)
56          {
57              try
58              {
59                  transaction.Execute();
60              }
61              catch (InvalidOperationException exception)
62              {
63                  Console.WriteLine("An error occurred in executing the transaction");
64                  Console.WriteLine("The error was: " + exception.Message);
65              }
66          }
67
68          /// <summary>
69          /// Executes a WithdrawTransaction on an account
70          /// </summary>
71          /// <param name="transaction">WithdrawTransaction to execute</param>
72          public void ExecuteTransaction(WithdrawTransaction transaction)
73          {
74              try
75              {
76                  transaction.Execute();
77              }
78              catch (InvalidOperationException exception)
79              {
80                  Console.WriteLine("An error occurred in executing the transaction");
81                  Console.WriteLine("The error was: " + exception.Message);
82              }
83          }
84
85          /// <summary>
86          /// Transfers funds between accounts held by the bank
87          /// </summary>
88          /// <param name="transaction">TransferTransaction to execute</param>
89          public void ExecuteTransaction(TransferTransaction transaction)
90          {
91              try
92              {
93                  transaction.Execute();
94              }
95              catch (InvalidOperationException exception)
96              {
97                  Console.WriteLine("An error occurred in executing the transaction");
98                  Console.WriteLine("The error was: " + exception.Message);
99              }
100         }
101     }
102 }
```

```
using System;

namespace Task_6._2P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get => _name; }
        public decimal Balance { get => _balance; }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance < 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if ((amount < 0) || (amount == decimal.MaxValue))
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
```

```
54          /// <param name="amount">The amount to subtract from the balance</param>
55          public Boolean Withdraw(decimal amount)
56          {
57              if ((amount < 0) || (amount > _balance))
58                  return false;
59
60              _balance -= amount;
61              return true;
62          }
63
64          /// <summary>
65          /// Outputs the account name and current balance as a string
66          /// </summary>
67          public void Print()
68          {
69              Console.WriteLine("Account Name: {0}, Balance: {1}",
70                  _name, _balance.ToString("C"));
71          }
72      }
73  }
```

```
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a Withdraw transaction
    /// </summary>
    class WithdrawTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a WithdrawTransaction
        /// </summary>
        /// <param name="account">Account to withdraw from</param>
        /// <param name="amount">Amount to withdraw</param>
        public WithdrawTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Withdrawal amount must be >
                ↪  $0.00");
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of the withdrawal
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "WITHDRAW AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!_executed)
```

```
52              {
53                  Console.Write("{0, 20}|", "Pending");
54              }
55              else if (_reversed)
56              {
57                  Console.Write("{0, 20}|", "Withdraw reversed");
58              }
59              else if (_success)
60              {
61                  Console.Write("{0, 20}|", "Withdraw complete");
62              }
63              else if (!_success)
64              {
65                  Console.Write("{0, 20}|", "Insufficient funds");
66              }
67              Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
68              Console.WriteLine(new String('-', 85));
69          }
70
71          /// <summary>
72          /// Executes the withdrawal
73          /// </summary>
74          /// <exception cref="System.InvalidOperationException">Thrown
75          /// when the withdraw is already complete or insufficient funds</exception>
76          public void Execute()
77          {
78              if (_executed && _success)
79              {
80                  throw new InvalidOperationException("Withdraw previously executed");
81              }
82              _executed = true;
83
84              _success = _account.Withdraw(_amount);
85              if (!_success)
86              {
87                  throw new InvalidOperationException("Insufficient funds");
88              }
89          }
90
91          /// <summary>
92          /// Reverses the withdraw if previously executed successfully
93          /// </summary>
94          /// <exception cref="System.InvalidOperationException">Thrown
95          /// if already rolled back or if there are insufficient
96          /// funds to complete the rollback</exception>
97          public void Rollback()
98          {
99              if (_reversed)
100             {
101                 throw new InvalidOperationException("Transaction already reversed");
102             }
103             else if (!_success)
104             {
```

```
105                 throw new InvalidOperationException(
106                     "Withdraw not successfully executed. Nothing to rollback.");
107             }
108             _reversed = _account.Deposit(_amount); // Deposit returns boolean
109             if (!_reversed) // Deposit didn't occur
110             {
111                 throw new InvalidOperationException("Invalid amount");
112             }
113             _reversed = true;
114         }
115     }
116 }
```

```
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a deposit transaction
    /// </summary>
    class DepositTransaction
    {
        // Instance variables
        private Account _account;
        private decimal _amount;
        private Boolean _executed;
        private Boolean _success;
        private Boolean _reversed;

        // Properties
        public Boolean Executed { get => _executed; }
        public Boolean Success { get => _success; }
        public Boolean Reversed { get => _reversed; }

        /// <summary>
        /// Constructs a deposit transaction object
        /// </summary>
        /// <param name="account">Account to deposit into</param>
        /// <param name="amount">Amount to deposit</param>
        public DepositTransaction(Account account, decimal amount)
        {
            _account = account;
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                throw new ArgumentOutOfRangeException(
                    "Deposit amount invalid: {0}", amount.ToString("C"));
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Prints the details and status of a deposit
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "DEPOSIT AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!_executed)
```

```
            {
                Console.Write("{0, 20}|", "Pending");
            }
            else if (_reversed)
            {
                Console.Write("{0, 20}|", "Deposit reversed");
            }
            else if (_success)
            {
                Console.Write("{0, 20}|", "Deposit complete");
            }
            else if (!_success)
            {
                Console.Write("{0, 20}|", "Invalid deposit");
            }
            Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
            Console.WriteLine(new String('-', 85));
        }

        /// <summary>
        /// Executes a deposit transaction
        /// </summary>
        public void Execute()
        {
            if (_executed && _success)
            {
                throw new InvalidOperationException("Deposit previously executed");
            }
            _executed = true;

            _success = _account.Deposit(_amount);
            if (!_success)
            {
                _executed = false;
                throw new InvalidOperationException("Deposit amount invalid");
            }
        }

        /// <summary>
        /// Reverses a deposit if previously executed successfully
        /// </summary>
        public void Rollback()
        {
            if (_reversed)
            {
                throw new InvalidOperationException("Transaction already reversed");
            }
            else if (!_success)
            {
                throw new InvalidOperationException(
                    "Deposit not successfully executed. Nothing to rollback.");
            }
            _reversed = _account.Withdraw(_amount); // Withdraw returns boolean
```

```
106             if (!_reversed) // Withdraw didn't occur
107             {
108                 throw new InvalidOperationException("Insufficient funds to
                    ↪ rollback");
109             }
110             _reversed = true;
111         }
112     }
113 }
```

```cs
using System;

namespace Task_6._2P
{
    /// <summary>
    /// Prototype for a transfer transaction
    /// </summary>
    class TransferTransaction
    {
        // Instance variables
        private Account _fromAccount;
        private Account _toAccount;
        private decimal _amount;
        private DepositTransaction _deposit;
        private WithdrawTransaction _withdraw;
        private bool _executed;
        private bool _reversed;

        // Properties
        public bool Executed { get => _executed; }
        public bool Reversed { get => _reversed; }
        public bool Success { get => (_deposit.Success && _withdraw.Success); }

        /// <summary>
        /// Constructor for a transfer transaction
        /// </summary>
        /// <param name="fromAccount">The account to transfer from</param>
        /// <param name="toAccount">The account to transfer to</param>
        /// <param name="amount">The amount to transfer</param>
        /// <exception cref="System.ArgumentOutOfRangeException">Thrown
        /// when the amount is negative</exception>
        public TransferTransaction(Account fromAccount, Account toAccount, decimal
        ↪ amount)
        {
            _fromAccount = fromAccount;
            _toAccount = toAccount;
            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("Negative transfer amount");
            }
            _amount = amount;

            _withdraw = new WithdrawTransaction(_fromAccount, _amount);
            _deposit = new DepositTransaction(_toAccount, _amount);
        }

        /// <summary>
        /// Prints the details of the transfer
        /// </summary>
        public void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
```

```
53                  "FROM ACCOUNT", "To ACCOUNT", "TRANSFER AMOUNT", "STATUS");
54              Console.WriteLine(new String('-', 85));
55              Console.Write("|{0, -20}|{1, 20}|{2, 20}|", _fromAccount.Name,
                ↪  _toAccount.Name, _amount.ToString("C"));
56              if (!_executed)
57              {
58                  Console.WriteLine("{0, 20}|", "Pending");
59              }
60              else if (_reversed)
61              {
62                  Console.WriteLine("{0, 20}|", "Transfer reversed");
63              }
64              else if (Success)
65              {
66                  Console.WriteLine("{0, 20}|", "Transfer complete");
67              }
68              else if (!Success)
69              {
70                  Console.WriteLine("{0, 20}|", "Transfer failed");
71              }
72              Console.WriteLine(new String('-', 85));
73          }
74
75          /// <summary>
76          /// Executes the transfer
77          /// </summary>
78          /// <exception cref="System.InvalidOperationException">Thrown
79          /// when previously executed or deposit or withdraw fail</exception>
80          public void Execute()
81          {
82              if (_executed)
83              {
84                  throw new InvalidOperationException("Transfer previously executed");
85              }
86              _executed = true;
87
88              try
89              {
90                  _withdraw.Execute();
91              }
92              catch (InvalidOperationException exception)
93              {
94                  Console.WriteLine("Transfer failed with reason: " +
                    ↪  exception.Message);
95                  _withdraw.Print();
96              }
97
98              if (_withdraw.Success)
99              {
100                 try
101                 {
102                     _deposit.Execute();
103                 }
```

```
104                 catch (InvalidOperationException exception)
105                 {
106                     Console.WriteLine("Transfer failed with reason: " +
                        ↪  exception.Message);
107                     _deposit.Print();
108                     try
109                     {
110                         _withdraw.Rollback();
111                     }
112                     catch (InvalidOperationException e)
113                     {
114                         Console.WriteLine("Withdraw could not be reversed with
                            ↪  reason: " + e.Message);
115                         _withdraw.Print();
116                     }
117                 }
118             }
119         }
120 
121         /// <summary>
122         /// Rolls the transfer back
123         /// </summary>
124         /// <exception cref="System.InvalidOperationException">Thrown
125         /// when the rollback has already been executed or it fails</exception>
126         public void Rollback()
127         {
128             if (!_executed)
129             {
130                 throw new InvalidOperationException("Transfer not executed. Nothing
                    ↪  to rollback.");
131             }
132 
133             if (_reversed)
134             {
135                 throw new InvalidOperationException("Transfer already rolled back");
136             }
137 
138             if (this.Success)
139             {
140                 try
141                 {
142                     _deposit.Rollback();
143                 }
144                 catch (InvalidOperationException exception)
145                 {
146                     Console.WriteLine("Failed to rollback deposit: "
147                         + exception.Message);
148                     return;
149                 }
150 
151                 try
152                 {
153                     _withdraw.Rollback();
```

```
154                 }
155                 catch (InvalidOperationException exception)
156                 {
157                     Console.WriteLine("Failed to rollback withdraw: "
158                         + exception.Message);
159                     return;
160                 }
161             }
162             _reversed = true;
163         }
164     }
165 }
```

19 A Simple Reaction-Timer Controller

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦♦◊

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

Outcome	**Weight**
Principles	♦♦♦◊◊

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

Outcome	**Weight**
Build Programs	♦♦♦♦◊

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

Outcome	**Weight**
Design	♦♦♦♦♦

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

Outcome	**Weight**
Justify	♦♦♦◊◊

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

Date	Author	Comment
2020/05/10 18:12	Peter Stacey	Ready to Mark
2020/05/10 18:12	Peter Stacey	Just doing final edits on the video now and I've also included both the OOP State Pattern and then at the end, a Mealy and Moore version as well (just because I wanted to compare the two approaches). Will link the video this evening.
2020/05/10 19:58	Peter Stacey	Video Link: https://youtu.be/MsehHjetM5I
2020/05/10 22:12	Dipto Pratyaksa	Nice work, well read, good demo, I think you got this soft-spot for architecture and coding, which will make bosses job so much easier when they employ you
2020/05/10 22:12	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

A Simple Reaction-Timer Controller

Submitted By:
Peter Stacey
pstacey
2020/05/10 18:12

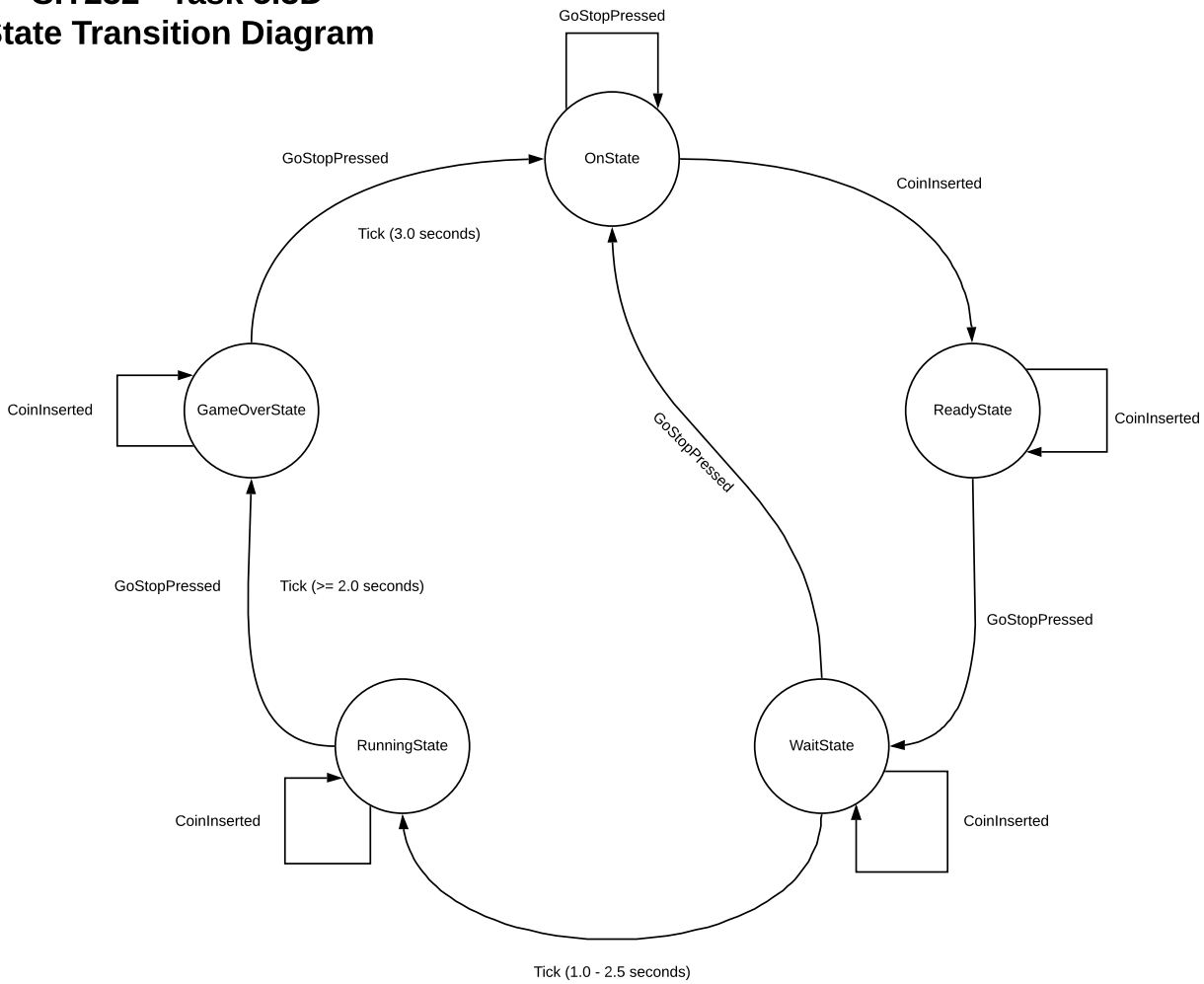
Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦♦◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦♦◊
Design	♦♦♦♦♦
Justify	♦♦♦◊◊

By providing the base files, this task involved evaluating an existing program in order to understand how to integrate our additional file into it. It also involved applying the principle of OOP, to encapsulate data, abstract the design our to a recognised OOP design pattern, to implement the design through coding the required file using a State pattern and supporting our design with a diagram of the states and transitions. With the transition diagram and file, we have evidence of our effort and this is further supported by my video.

May 10, 2020

SIT232 - Task 5.3D
State Transition Diagram



```cs
using SimpleReactionMachine;
using System;
using System.Data;

namespace SimpleReactionMachine
{
    public class SimpleReactionController : IController
    {
        // Settings for the game times
        private const int MIN_WAIT_TIME = 100; // Minimum wait time, 1 sec in ticks
        private const int MAX_WAIT_TIME = 250; // Maximum wait time, 2.5 sec in
        ↪  ticks
        private const int MAX_GAME_TIME = 200; // Maximum of 2 sec to react, in
        ↪  ticks
        private const int GAMEOVER_TIME = 300; // Display result for 3 sec, in ticks
        private const double TICKS_PER_SECOND = 100.0; // Based on 10ms ticks

        // Instance variables and properties
        private State _state;
        private IGui Gui { get; set; }
        private IRandom Rng { get; set; }
        private int Ticks { get; set; }

        /// <summary>
        /// Connects the controller to the Gui and Random Number Generator
        /// </summary>
        /// <param name="gui">IGui concrete implementation</param>
        /// <param name="rng">IRandom concreate implementation</param>
        public void Connect(IGui gui, IRandom rng)
        {
            Gui = gui;
            Rng = rng;
            Init();
        }

        /// <summary>
        /// Initialises the state of the controller at the start of the program
        /// </summary>
        public void Init()
        {
            _state = new OnState(this);
        }

        /// <summary>
        /// Coin inserted event handler
        /// </summary>
        public void CoinInserted()
        {
            _state.CoinInserted();
        }

        /// <summary>
        /// Go/Stop pressed event handler
```

```
52          /// </summary>
53          public void GoStopPressed()
54          {
55              _state.GoStopPressed();
56          }
57
58          /// <summary>
59          /// Tick event handler
60          /// </summary>
61          public void Tick()
62          {
63              _state.Tick();
64          }
65
66          /// <summary>
67          /// Sets the state of the controller to the desired state
68          /// </summary>
69          /// <param name="state">The new state to transition to</param>
70          private void SetState(State state)
71          {
72              _state = state;
73          }
74
75          /// <summary>
76          /// Base class for concrete State classes
77          /// </summary>
78          private abstract class State
79          {
80              protected SimpleReactionController _controller;
81
82              public State(SimpleReactionController controller)
83              {
84                  _controller = controller;
85              }
86
87              public abstract void CoinInserted();
88              public abstract void GoStopPressed();
89              public abstract void Tick();
90          }
91
92          /// <summary>
93          /// State of the game when it is waiting for a coin to be inserted
94          /// </summary>
95          private class OnState : State
96          {
97              public OnState(SimpleReactionController controller) : base(controller)
98              {
99                  _controller.Gui.SetDisplay("Insert coin");
100             }
101
102             public override void CoinInserted()
103             {
104                 _controller.SetState(new ReadyState(_controller));
```

```
105             }
106             public override void GoStopPressed() { }
107             public override void Tick() { }
108         }
109 
110         /// <summary>
111         /// State of the game when a coin has been inserted, but the game is not yet
112         /// started
113         /// </summary>
114         private class ReadyState : State
115         {
116             public ReadyState(SimpleReactionController controller) :
                ↪ base(controller)
117             {
118                 _controller.Gui.SetDisplay("Press Go!");
119             }
120 
121             public override void CoinInserted() { }
122             public override void GoStopPressed()
123             {
124                 _controller.SetState(new WaitState(_controller));
125             }
126             public override void Tick() { }
127         }
128 
129         /// <summary>
130         /// State of the game when the game has started and it is waiting for the
131         /// random time
132         /// </summary>
133         private class WaitState : State
134         {
135             private int _waitTime;
136             public WaitState(SimpleReactionController controller) : base(controller)
137             {
138                 _controller.Gui.SetDisplay("Wait...");
139                 _controller.Ticks = 0;
140                 _waitTime = _controller.Rng.GetRandom(MIN_WAIT_TIME, MAX_WAIT_TIME);
141             }
142 
143             public override void CoinInserted() { }
144             public override void GoStopPressed()
145             {
146                 _controller.SetState(new OnState(_controller));
147             }
148             public override void Tick()
149             {
150                 _controller.Ticks++;
151                 if(_controller.Ticks == _waitTime)
152                 {
153                     _controller.SetState(new RunningState(_controller));
154                 }
155             }
156         }
```

```
157
158         /// <summary>
159         /// State of the game when the timer is counting and it is waiting for the
160         /// user to react by pressing the Go/Stop button
161         /// </summary>
162         private class RunningState : State
163         {
164             public RunningState(SimpleReactionController controller) :
                ↪ base(controller)
165             {
166                 _controller.Gui.SetDisplay("0.00");
167                 _controller.Ticks = 0;
168             }
169
170             public override void CoinInserted() { }
171             public override void GoStopPressed()
172             {
173                 _controller.SetState(new GameOverState(_controller));
174             }
175
176             public override void Tick()
177             {
178                 _controller.Ticks++;
179                 _controller.Gui.SetDisplay(
180                     (_controller.Ticks / TICKS_PER_SECOND).ToString("0.00"));
181                 if(_controller.Ticks == MAX_GAME_TIME)
182                 {
183                     _controller.SetState(new GameOverState(_controller));
184                 }
185             }
186         }
187
188         /// <summary>
189         /// State of the game when the time has expired, or the user reacted.
190         /// </summary>
191         private class GameOverState : State
192         {
193             public GameOverState(SimpleReactionController controller) :
                ↪ base(controller)
194             {
195                 _controller.Ticks = 0;
196             }
197
198             public override void CoinInserted() { }
199             public override void GoStopPressed()
200             {
201                 _controller.SetState(new OnState(_controller));
202             }
203             public override void Tick()
204             {
205                 _controller.Ticks++;
206                 if(_controller.Ticks == GAMEOVER_TIME)
207                 {
```

```
208                         _controller.SetState(new OnState(_controller));
209                     }
210                 }
211             }
212         }
213 }
```

20 An Enhanced Reaction-Timer Controller

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

Outcome	**Weight**
Principles	♦♦♦♦♦

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

Outcome	**Weight**
Build Programs	♦♦♦♦♦

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

Outcome	**Weight**
Design	♦♦♦♦♦

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

Outcome	**Weight**
Justify	♦♦♦♦♦

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure

the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

Date	Author	Comment
2020/05/14 20:02	Peter Stacey	Ready to Mark
2020/05/14 20:02	Peter Stacey	My video is edited. I'll upload to YouTube and link it
2020/05/14 22:03	Peter Stacey	Ready to Mark
2020/05/14 22:03	Peter Stacey	Uploaded correctly named version.
2020/05/15 13:44	Dipto Pratyaksa	ok keenly waiting for it
2020/05/15 13:44	Dipto Pratyaksa	Discuss
2020/05/16 22:17	Peter Stacey	Video Link: https://youtu.be/wboh30mWTVc

Deakin University

Object Oriented Development

OnTrack Submission

An Enhanced Reaction-Timer Controller

Submitted By:
Peter Stacey
pstacey
2020/05/14 22:03

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦♦♦
Build Programs	♦♦♦♦♦
Design	♦♦♦♦♦
Justify	♦♦♦♦♦

As an extension of 5.3D, this task involves evaluating both the requirements of the task and our existing code in order to design the additional features. The design component involves adding to our existing design and producing a new state transition diagram as evidence of the design updates.This also involves writing addition code to implement the design and developing a suite of tests to ensure the design is correct. For this task, I have used the C# unit testing framework and developed a set of tests that are demonstrated in my video.My video and the attached files provide evidence of meeting the outcomes, which aligns with the last outcome.

May 14, 2020

```
using System.Windows.Forms;

namespace SimpleReactionMachine
{
    public class EnhancedReactionController : IController
    {
        // Settings for the game times
        private const int MAX_READY_TIME = 1000; // Maximum time in ready without
        ↪  pressing GoStop
        private const int MIN_WAIT_TIME = 100;   // Minimum wait time, 1 sec in
        ↪  ticks
        private const int MAX_WAIT_TIME = 250;   // Maximum wait time, 2.5 sec in
        ↪  ticks
        private const int MAX_GAME_TIME = 200;   // Maximum of 2 sec to react, in
        ↪  ticks
        private const int GAMEOVER_TIME = 300;   // Display result for 3 sec, in
        ↪  ticks
        private const int RESULTS_TIME = 500;    // Display average time for 5 sec,
        ↪  in ticks
        private const double TICKS_PER_SECOND = 100.0; // Based on 10ms ticks

        // Instance variables and properties
        private State _state;
        private IGui Gui { get; set; }
        private IRandom Rng { get; set; }
        private int Ticks { get; set; }
        private int Games { get; set; }
        private int TotalReactionTime { get; set; }

        /// <summary>
        /// Connects the controller to the Gui and Random Number Generator
        /// </summary>
        /// <param name="gui">IGui concrete implementation</param>
        /// <param name="rng">IRandom concreate implementation</param>
        public void Connect(IGui gui, IRandom rng)
        {
            Gui = gui;
            Rng = rng;
            Init();
        }

        /// <summary>
        /// Initialises the state of the controller at the start of the program
        /// </summary>
        public void Init() => _state = new OnState(this);

        /// <summary>
        /// Coin inserted event handler
        /// </summary>
        public void CoinInserted() => _state.CoinInserted();

        /// <summary>
        /// Go/Stop pressed event handler
```

```
48          /// </summary>
49          public void GoStopPressed() => _state.GoStopPressed();
50
51          /// <summary>
52          /// Tick event handler
53          /// </summary>
54          public void Tick() => _state.Tick();
55
56          /// <summary>
57          /// Sets the state of the controller to the desired state
58          /// </summary>
59          /// <param name="state">The new state to transition to</param>
60          void SetState(State state) => _state = state;
61
62          /// <summary>
63          /// Base class for concrete State classes
64          /// </summary>
65          abstract class State
66          {
67              protected EnhancedReactionController controller;
68              public State(EnhancedReactionController con) => controller = con;
69              public abstract void CoinInserted();
70              public abstract void GoStopPressed();
71              public abstract void Tick();
72          }
73
74          /// <summary>
75          /// State of the game when it is waiting for a coin to be inserted
76          /// </summary>
77          class OnState : State
78          {
79              public OnState(EnhancedReactionController con) : base(con)
80              {
81                  controller.Games = 0;
82                  controller.TotalReactionTime = 0;
83                  controller.Gui.SetDisplay("Insert coin");
84              }
85              public override void CoinInserted() => controller.SetState(new
                    ↪ ReadyState(controller));
86              public override void GoStopPressed() { }
87              public override void Tick() { }
88          }
89
90          /// <summary>
91          /// State of the game when a coin has been inserted, but the game is not yet
92          /// started
93          /// </summary>
94          class ReadyState : State
95          {
96              public ReadyState(EnhancedReactionController con) : base(con)
97              {
98                  controller.Gui.SetDisplay("Press Go!");
99                  controller.Ticks = 0;
```

```
100             }
101             public override void CoinInserted() { }
102             public override void GoStopPressed()
103             {
104                 controller.SetState(new WaitState(controller));
105             }
106             public override void Tick()
107             {
108                 controller.Ticks++;
109                 if (controller.Ticks == MAX_READY_TIME)
110                     controller.SetState(new OnState(controller));
111             }
112         }
113 
114         /// <summary>
115         /// State of the game when the game has started and it is waiting for the
116         /// random time
117         /// </summary>
118         class WaitState : State
119         {
120             private int _waitTime;
121             public WaitState(EnhancedReactionController con) : base(con)
122             {
123                 controller.Gui.SetDisplay("Wait...");
124                 controller.Ticks = 0;
125                 _waitTime = controller.Rng.GetRandom(MIN_WAIT_TIME, MAX_WAIT_TIME);
126             }
127             public override void CoinInserted() { }
128             public override void GoStopPressed() => controller.SetState(new
                ↪ OnState(controller));
129             public override void Tick()
130             {
131                 controller.Ticks++;
132                 if (controller.Ticks == _waitTime)
133                 {
134                     controller.Games++;
135                     controller.SetState(new RunningState(controller));
136                 }
137             }
138         }
139 
140         /// <summary>
141         /// State of the game when the timer is counting and it is waiting for the
142         /// user to react by pressing the Go/Stop button
143         /// </summary>
144         class RunningState : State
145         {
146             public RunningState(EnhancedReactionController con) : base(con)
147             {
148                 controller.Gui.SetDisplay("0.00");
149                 controller.Ticks = 0;
150             }
151             public override void CoinInserted() { }
```

```
152             public override void GoStopPressed()
153             {
154                 controller.TotalReactionTime += controller.Ticks;
155                 controller.SetState(new GameOverState(controller));
156             }
157             public override void Tick()
158             {
159                 controller.Ticks++;
160                 controller.Gui.SetDisplay(
161                     (controller.Ticks / TICKS_PER_SECOND).ToString("0.00"));
162                 if (controller.Ticks == MAX_GAME_TIME)
163                     controller.SetState(new GameOverState(controller));
164             }
165         }
166 
167         /// <summary>
168         /// State of the game when the time has expired, or the user reacted.
169         /// If 3 games not yet played, sets the state to Wait, otherwise to
170         /// Results
171         /// </summary>
172         class GameOverState : State
173         {
174             public GameOverState(EnhancedReactionController con) : base(con)
175             {
176                 controller.Ticks = 0;
177             }
178             public override void CoinInserted() { }
179             public override void GoStopPressed() => CheckGames();
180             public override void Tick()
181             {
182                 controller.Ticks++;
183                 if (controller.Ticks == GAMEOVER_TIME)
184                     CheckGames();
185             }
186             private void CheckGames()
187             {
188                 if (controller.Games == 3)
189                 {
190                     controller.SetState(new ResultsState(controller));
191                     return;
192                 }
193                 controller.SetState(new WaitState(controller));
194             }
195         }
196 
197         /// <summary>
198         /// Shows the average reaction time for the 3 games played, for
199         /// 5 seconds, or until GoStop is pressed
200         /// </summary>
201         class ResultsState : State
202         {
203             public ResultsState(EnhancedReactionController con) : base(con)
204             {
```

```
205                 controller.Gui.SetDisplay("Average: "
206                     + ((double)controller.TotalReactionTime / controller.Games *
                        ↪ 0.01)
207                     .ToString("0.00"));
208                 controller.Ticks = 0;
209             }
210             public override void CoinInserted() { }
211             public override void GoStopPressed() => controller.SetState(new
                ↪ OnState(controller));
212             public override void Tick()
213             {
214                 controller.Ticks++;
215                 if (controller.Ticks == RESULTS_TIME)
216                     controller.SetState(new OnState(controller));
217             }
218         }
219     }
220 }
```

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SimpleReactionMachine;
using System;

namespace EnhancedSimpleReactionControllerTests
{
    [TestClass]
    public class EnhancedReactionControllerTests
    {
        private static IController controller;
        private static IGui gui;
        private static IRandom rng;
        private static string displayText;
        private static int RandomNumber { get; set; }

        [TestMethod]
        public void Create_Controller()
        {
            // Tests that the controller can be created and is not null
            // after creation
            controller = new EnhancedReactionController();
            Assert.IsNotNull(controller);
        }

        [TestMethod]
        public void Connect_And_Initialise_Controller()
        {
            controller = new EnhancedReactionController();
            gui = new DummyGui();
            gui.Connect(controller);
            controller.Connect(gui, new RndGenerator());

            // Controller Init() sets initial state to OnState
            // and display should be "Insert coin"
            controller.Init();
            Assert.AreEqual("Insert coin", displayText);
        }

        [TestMethod]
        public void Test_OnState_GoStopPressed()
        {
            controller = new EnhancedReactionController();
            gui = new DummyGui();
            rng = new RndGenerator();
            InitialiseToOnState(controller, gui, rng);

            // GoStopPressed has no effect in OnState
            Assert.AreEqual("Insert coin", displayText);
            controller.GoStopPressed();
            Assert.AreEqual("Insert coin", displayText);
        }

        [TestMethod]
```

```
54          public void Test_OnState_Tick()
55          {
56              controller = new EnhancedReactionController();
57              gui = new DummyGui();
58              rng = new RndGenerator();
59              InitialiseToOnState(controller, gui, rng);
60
61              // Tick has no effect in OnState
62              Assert.AreEqual("Insert coin", displayText);
63              controller.Tick();
64              Assert.AreEqual("Insert coin", displayText);
65          }
66
67          [TestMethod]
68          public void Test_OnState_CoinInserted()
69          {
70              controller = new EnhancedReactionController();
71              gui = new DummyGui();
72              rng = new RndGenerator();
73              InitialiseToOnState(controller, gui, rng);
74
75              // Inserting a coin sets the state to ReadyState
76              // Display should then be "Press Go!"
77              Assert.AreEqual("Insert coin", displayText);
78              controller.CoinInserted();
79              Assert.AreEqual("Press Go!", displayText);
80          }
81
82          [TestMethod]
83          public void Test_ReadyState_CoinInserted()
84          {
85              controller = new EnhancedReactionController();
86              gui = new DummyGui();
87              rng = new RndGenerator();
88              InitialiseToReadyState(controller, gui, rng);
89
90              // Inserting a coin has no effect in ReadyState
91              Assert.AreEqual("Press Go!", displayText);
92              controller.CoinInserted();
93              Assert.AreEqual("Press Go!", displayText);
94          }
95
96          [TestMethod]
97          public void Test_ReadyState_Tick()
98          {
99              controller = new EnhancedReactionController();
100             gui = new DummyGui();
101             rng = new RndGenerator();
102             InitialiseToReadyState(controller, gui, rng);
103
104             // Tick has no effect in ReadyState
105             Assert.AreEqual("Press Go!", displayText);
106             controller.Tick();
```

```
107             Assert.AreEqual("Press Go!", displayText);
108         }
109 
110         [TestMethod]
111         public void Test_ReadyState_GoStopPressed()
112         {
113             controller = new EnhancedReactionController();
114             gui = new DummyGui();
115             rng = new RndGenerator();
116             InitialiseToReadyState(controller, gui, rng);
117 
118             // Pressing Go/Stop sets the state to WaitState
119             // Display should then be "Wait..."
120             Assert.AreEqual("Press Go!", displayText);
121             controller.GoStopPressed();
122             Assert.AreEqual("Wait...", displayText);
123         }
124 
125         [TestMethod]
126         public void Test_ReadyState_Too_Long()
127         {
128             controller = new EnhancedReactionController();
129             gui = new DummyGui();
130             rng = new RndGenerator();
131             InitialiseToReadyState(controller, gui, rng);
132 
133             // Waiting for 10 seconds in WaitState resets the
134             // controller back to OnState
135             // Display should then be "Insert coin"
136             for (int t = 0; t < 999; t++) controller.Tick();
137             Assert.AreEqual("Press Go!", displayText);
138             controller.Tick();
139             Assert.AreEqual("Insert coin", displayText);
140         }
141 
142         [TestMethod]
143         public void Test_WaitState_CoinInserted()
144         {
145             controller = new EnhancedReactionController();
146             gui = new DummyGui();
147             rng = new RndGenerator();
148             InitialiseToWaitState(controller, gui, rng);
149 
150             // Inserting a coin has no effect in WaitState
151             Assert.AreEqual("Wait...", displayText);
152             controller.CoinInserted();
153             Assert.AreEqual("Wait...", displayText);
154         }
155 
156         [TestMethod]
157         public void Test_WaitState_GoStopPressed()
158         {
159             controller = new EnhancedReactionController();
```

```
160             gui = new DummyGui();
161             rng = new RndGenerator();
162             InitialiseToWaitState(controller, gui, rng);
163 
164             // GoStopPressed in the WaitState is considered
165             // cheating and it sets the game back to the OnState
166             // Display should then by "Insert coin"
167             Assert.AreEqual("Wait...", displayText);
168             controller.GoStopPressed();
169             Assert.AreEqual("Insert coin", displayText);
170         }
171 
172         [TestMethod]
173         public void Test_WaitState_Tick()
174         {
175             controller = new EnhancedReactionController();
176             gui = new DummyGui();
177             rng = new RndGenerator();
178             InitialiseToWaitState(controller, gui, rng);
179 
180             // After the rendom wait time, the controller should
181             // be set to the RunningState
182             // Display should then be "0.00"
183             for (int t = 0; t < RandomNumber - 1; t++) controller.Tick();
184             Assert.AreEqual(displayText, "Wait...");
185             controller.Tick();
186             Assert.AreEqual(displayText, "0.00");
187         }
188 
189         [TestMethod]
190         public void Test_RunningState_CoinInserted()
191         {
192             controller = new EnhancedReactionController();
193             gui = new DummyGui();
194             rng = new RndGenerator();
195             InitialiseToRunningState(controller, gui, rng);
196 
197             // CoinInserted has no effect in the RunningState
198             Assert.AreEqual("0.00", displayText);
199             controller.CoinInserted();
200             Assert.AreEqual("0.00", displayText);
201         }
202 
203         [TestMethod]
204         public void Test_RunningState_Tick()
205         {
206             controller = new EnhancedReactionController();
207             gui = new DummyGui();
208             rng = new RndGenerator();
209             InitialiseToRunningState(controller, gui, rng);
210 
211             // Ticks advance the time display in the RunningState
212             Assert.AreEqual("0.00", displayText);
```

```
213             controller.Tick();
214             Assert.AreEqual("0.01", displayText);
215 
216             for (int t = 0; t < 10; t++) controller.Tick();
217             Assert.AreEqual("0.11", displayText);
218 
219             for (int t = 0; t < 100; t++) controller.Tick();
220             Assert.AreEqual("1.11", displayText);
221 
222             // GoStopPressed should advance to the GameOverState
223             // and no further update to the display
224             controller.GoStopPressed();
225             Assert.AreEqual("1.11", displayText);
226         }
227 
228         [TestMethod]
229         public void Test_RunningState_GoStopPressed()
230         {
231             controller = new EnhancedReactionController();
232             gui = new DummyGui();
233             rng = new RndGenerator();
234             InitialiseToRunningState(controller, gui, rng);
235 
236             // GoStopPressed records the reaction time in the RunningState
237             // and advances the controller to the GameOverState
238             // Display should be the same as the reaction time when
239             // GoStop is pressed
240             for (int t = 0; t < 164; t++) controller.Tick();
241             Assert.AreEqual("1.64", displayText);
242             controller.GoStopPressed();
243             Assert.AreEqual("1.64", displayText);
244         }
245 
246         [TestMethod]
247         public void Test_RunningState_Tick_Two_Seconds()
248         {
249             controller = new EnhancedReactionController();
250             gui = new DummyGui();
251             rng = new RndGenerator();
252             InitialiseToRunningState(controller, gui, rng);
253 
254             // Not reacting in 2 seconds automatically ends the game
255             // Display should show 2.00 seconds
256             for (int t = 0; t < 199; t++) controller.Tick();
257             Assert.AreEqual("1.99", displayText);
258             controller.Tick();
259             Assert.AreEqual("2.00", displayText);
260             controller.Tick();
261             Assert.AreEqual("2.00", displayText);
262         }
263 
264         [TestMethod]
265         public void Test_GameOverState_CoinInserted()
```

```
266         {
267             controller = new EnhancedReactionController();
268             gui = new DummyGui();
269             rng = new RndGenerator();
270             InitialiseToRunningState(controller, gui, rng);
271 
272             // Inserting a coin has no effect in GameOverState
273             for (int t = 0; t < 22; t++) controller.Tick();
274             Assert.AreEqual("0.22", displayText);
275             controller.CoinInserted();
276             Assert.AreEqual("0.22", displayText);
277         }
278 
279         [TestMethod]
280         public void Test_GameOverState_Tick()
281         {
282             controller = new EnhancedReactionController();
283             gui = new DummyGui();
284             rng = new RndGenerator();
285             InitialiseToRunningState(controller, gui, rng);
286 
287             // Tick shows the reaction time and then sets the
288             // controller to the WaitState
289             // NOTE: This test does not test the transition
290             // to the ResultState. That is tested in
291             // Test_Play_Three_Games_And_Wait_Ticks
292             for (int t = 0; t < 50; t++) controller.Tick();
293             controller.GoStopPressed();
294             Assert.AreEqual("0.50", displayText);
295             for (int t = 0; t < 299; t++) controller.Tick();
296             Assert.AreEqual("0.50", displayText);
297             controller.Tick();
298             Assert.AreEqual("Wait...", displayText);
299         }
300 
301         [TestMethod]
302         public void Test_GameOver_GoStopPressed()
303         {
304             controller = new EnhancedReactionController();
305             gui = new DummyGui();
306             rng = new RndGenerator();
307             InitialiseToRunningState(controller, gui, rng);
308 
309             // GoStopPressed immediately ends the GameOverState
310             // and sets the state to WaitState
311             // NOTE: This does not test the state moving
312             // to the ResultState after 3 games. That is tested in
313             // Test_Play_Three_Games_And_GoStopPressed
314             for (int t = 0; t < 56; t++) controller.Tick();
315             controller.GoStopPressed();
316             Assert.AreEqual("0.56", displayText);
317             controller.GoStopPressed();
318             Assert.AreEqual("Wait...", displayText);
```

```
319         }
320
321         [TestMethod]
322         public void Test_Play_Three_Games_And_Wait_Ticks()
323         {
324             controller = new EnhancedReactionController();
325             gui = new DummyGui();
326             rng = new RndGenerator();
327             InitialiseToRunningState(controller, gui, rng);
328
329             // Run three games and then wait the final 3 seconds
330             // State should advance to ResultState
331             // Display should then show the average reaction time
332             for (int t = 0; t < 20; t++) controller.Tick();
333             controller.GoStopPressed();
334             Assert.AreEqual("0.20", displayText);
335             for (int t = 0; t < 299; t++) controller.Tick();
336             Assert.AreEqual("0.20", displayText);
337             controller.Tick();
338             Assert.AreEqual("Wait...", displayText);
339
340             for (int t = 0; t < RandomNumber + 30; t++) controller.Tick();
341             controller.GoStopPressed();
342             Assert.AreEqual("0.30", displayText);
343             for (int t = 0; t < 299; t++) controller.Tick();
344             Assert.AreEqual("0.30", displayText);
345             controller.Tick();
346             Assert.AreEqual("Wait...", displayText);
347
348             for (int t = 0; t < RandomNumber + 40; t++) controller.Tick();
349             controller.GoStopPressed();
350             Assert.AreEqual("0.40", displayText);
351             for (int t = 0; t < 299; t++) controller.Tick();
352             controller.Tick();
353             Assert.AreEqual("Average: 0.30", displayText);
354
355         }
356         [TestMethod]
357         public void Test_Play_Three_Games_And_GoStopPressed()
358         {
359             controller = new EnhancedReactionController();
360             gui = new DummyGui();
361             rng = new RndGenerator();
362             InitialiseToRunningState(controller, gui, rng);
363
364             // Run three games and then press GoStop
365             // State should advance to ResultState immediately
366             // Display should then show the average reaction time
367             for (int t = 0; t < 155; t++) controller.Tick();
368             controller.GoStopPressed();
369             Assert.AreEqual("1.55", displayText);
370             controller.GoStopPressed();
371             Assert.AreEqual("Wait...", displayText);
```

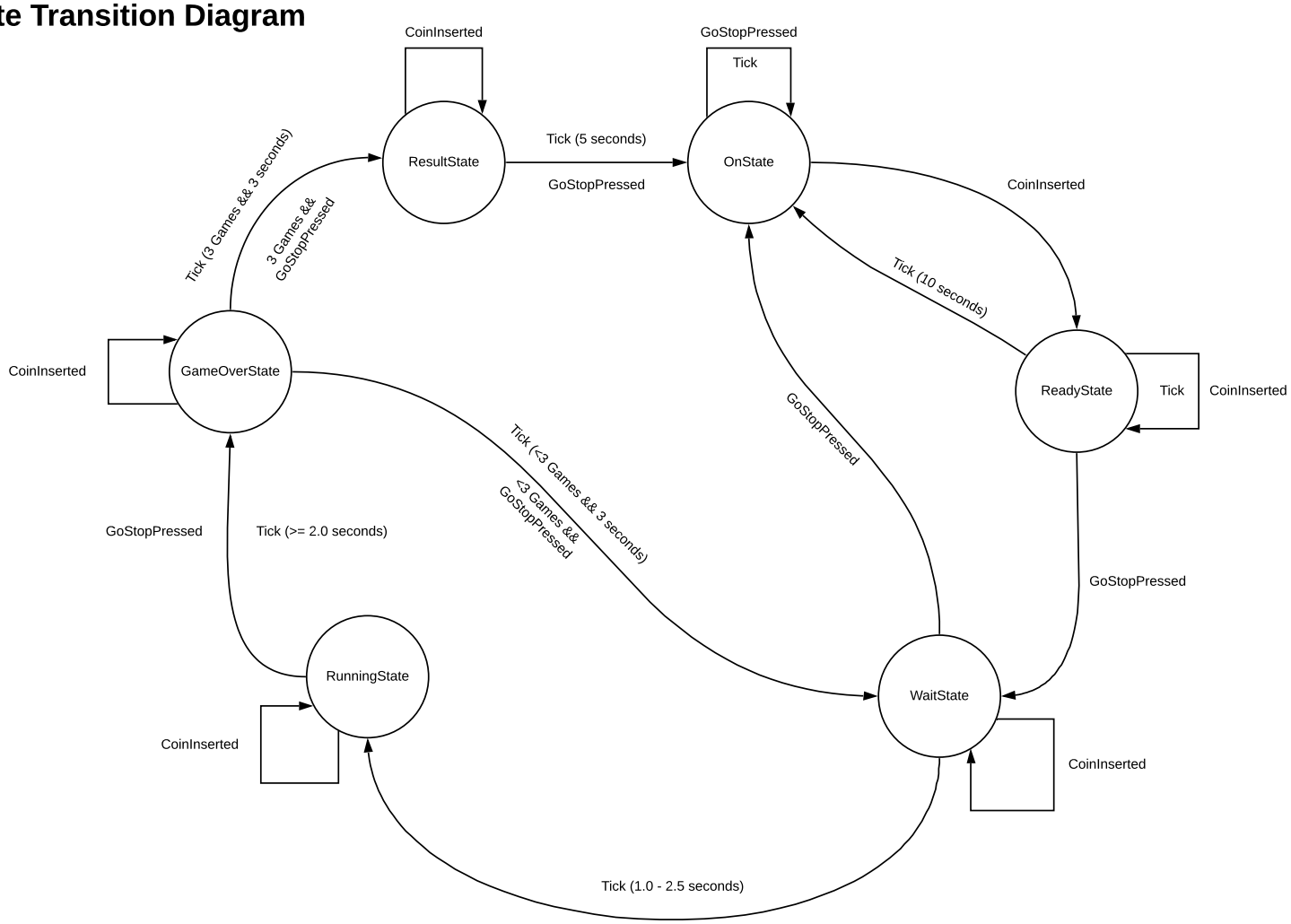
```
372
373             for (int t = 0; t < RandomNumber + 160; t++) controller.Tick();
374             controller.GoStopPressed();
375             Assert.AreEqual("1.60", displayText);
376             controller.GoStopPressed();
377             Assert.AreEqual("Wait...", displayText);
378
379             for (int t = 0; t < RandomNumber + 165; t++) controller.Tick();
380             controller.GoStopPressed();
381             Assert.AreEqual("1.65", displayText);
382             controller.GoStopPressed();
383             Assert.AreEqual("Average: 1.60", displayText);
384         }
385
386         [TestMethod]
387         public void Test_ResultState_CoinInserted()
388         {
389             controller = new EnhancedReactionController();
390             gui = new DummyGui();
391             rng = new RndGenerator();
392             InitialiseToRunningState(controller, gui, rng);
393
394             // Play 3 games
395             for (int t = 0; t < 10; t++) controller.Tick();
396             controller.GoStopPressed();
397             controller.GoStopPressed();
398
399             for (int t = 0; t < RandomNumber + 15; t++) controller.Tick();
400             controller.GoStopPressed();
401             controller.GoStopPressed();
402
403             for (int t = 0; t < RandomNumber + 20; t++) controller.Tick();
404             controller.GoStopPressed();
405             controller.GoStopPressed();
406
407             // Inserting a coin in the ResultState has no effect
408             Assert.AreEqual("Average: 0.15", displayText);
409             controller.CoinInserted();
410             Assert.AreEqual("Average: 0.15", displayText);
411         }
412
413         [TestMethod]
414         public void Test_ResultState_Ticks()
415         {
416             controller = new EnhancedReactionController();
417             gui = new DummyGui();
418             rng = new RndGenerator();
419             InitialiseToRunningState(controller, gui, rng);
420
421             // Play 3 games
422             for (int t = 0; t < 10; t++) controller.Tick();
423             controller.GoStopPressed();
424             controller.GoStopPressed();
```

```
425
426             for (int t = 0; t < RandomNumber + 15; t++) controller.Tick();
427             controller.GoStopPressed();
428             controller.GoStopPressed();
429
430             for (int t = 0; t < RandomNumber + 20; t++) controller.Tick();
431             controller.GoStopPressed();
432             controller.GoStopPressed();
433
434             // Ticks displays the average reaction time for 5 seconds
435             // and then the controller is set to OnState
436             // Display should then be "Insert coin"
437             Assert.AreEqual("Average: 0.15", displayText);
438             for (int i = 0; i < 499; i++) controller.Tick();
439             Assert.AreEqual("Average: 0.15", displayText);
440             controller.Tick();
441             Assert.AreEqual("Insert coin", displayText);
442         }
443
444         [TestMethod]
445         public void Test_ResultState_GoStopPressed()
446         {
447             controller = new EnhancedReactionController();
448             gui = new DummyGui();
449             rng = new RndGenerator();
450             InitialiseToRunningState(controller, gui, rng);
451
452             // Play 3 games
453             for (int t = 0; t < 10; t++) controller.Tick();
454             controller.GoStopPressed();
455             controller.GoStopPressed();
456
457             for (int t = 0; t < RandomNumber + 15; t++) controller.Tick();
458             controller.GoStopPressed();
459             controller.GoStopPressed();
460
461             for (int t = 0; t < RandomNumber + 20; t++) controller.Tick();
462             controller.GoStopPressed();
463             controller.GoStopPressed();
464
465             // GoStopPressed displays the average reaction time for 5 seconds
466             // and then the controller is set to OnState
467             // Display should then be "Insert coin"
468             Assert.AreEqual("Average: 0.15", displayText);
469             controller.GoStopPressed();
470             Assert.AreEqual("Insert coin", displayText);
471         }
472
473         // sets the controller to the OnState
474         private void InitialiseToOnState(IController controller, IGui gui, IRandom
            ↪ rng)
475         {
476             gui.Connect(controller);
```

```
477             controller.Connect(gui, rng);
478             gui.Init();
479             controller.Init();
480         }
481 
482         // sets the controller to the ReadyState
483         private void InitialiseToReadyState(IController controller, IGui gui,
            ↪ IRandom rng)
484         {
485             InitialiseToOnState(controller, gui, rng);
486             controller.CoinInserted();
487         }
488 
489         // sets the controller to the WaitState
490         private void InitialiseToWaitState(IController controller, IGui gui,
            ↪ IRandom rng)
491         {
492             InitialiseToReadyState(controller, gui, rng);
493             controller.GoStopPressed();
494         }
495 
496         // sets the controller to the RunningState
497         private void InitialiseToRunningState(IController controller, IGui gui,
            ↪ IRandom rng)
498         {
499             InitialiseToWaitState(controller, gui, rng);
500             for (int t = 0; t < RandomNumber; t++)
501                 controller.Tick();
502         }
503 
504         // Mock Gui, as implementation of the IGui, to allow the controller
505         // to Connect and SetDisplay
506         private class DummyGui : IGui
507         {
508             private IController _controller;
509             public void Connect(IController controller) => _controller = controller;
510             public void Init() => displayText = "?reset?";
511 
512             public void SetDisplay(string msg)
513             {
514                 displayText = msg;
515             }
516         }
517 
518         // IRandom implementation that also sets the RandomNumber property
519         // to allow the test to access it, as well as the values being
520         // passed to the controller
521         private class RndGenerator : IRandom
522         {
523             Random rnd = new Random(42);
524 
525             public int GetRandom(int from, int to)
526             {
```

```
527                     RandomNumber = rnd.Next(from) + to;
528                     return RandomNumber;
529                 }
530             }
531         }
532 }
```

SIT232 - Task 5.4HD
State Transition Diagram



21 Abstract Transactions

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

Outcome	**Weight**
Principles	♦♦♦◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

Outcome	**Weight**
Build Programs	♦♦◊◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

Outcome	**Weight**
Design	♦♦◊◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

Outcome	Weight
Justify	♦♦♦◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

Date	Author	Comment
2020/04/30 08:01	Peter Stacey	Ready to Mark
2020/05/01 13:47	Peter Stacey	Ready to Mark
2020/05/01 13:47	Peter Stacey	Fixed an error in rolling back transfers found in additional testing.
2020/05/03 17:05	Dipto Pratyaksa	great early submission, looking forward for your video
2020/05/03 17:05	Dipto Pratyaksa	Discuss
2020/05/04 14:24	Peter Stacey	Added additional option to include the account details in the transaction history.
2020/05/04 14:24	Peter Stacey	Video Likn: https://youtu.be/uzQnwz-DNLA
2020/05/04 22:14	Dipto Pratyaksa	very well written code, always enjoyable video. I like how you clearly document your codes in comments. Overall outstanding submission. You are the king of the wolfgang pack of this semester!
2020/05/04 22:15	Dipto Pratyaksa	:+1::smile:
2020/05/04 22:15	Dipto Pratyaksa	:wolf: Go Peter Go!
2020/05/04 22:15	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Abstract Transactions

Submitted By:
Peter Stacey
pstacey
2020/05/04 14:24

Tutor:
Dipto Pratyaksa

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦◊◊◊
Design	♦♦◊◊◊
Justify	♦♦♦◊◊

This task involves extending the bank account by adding a base class for all transactions. That aspect requires evaluating the existing code in order to change it appropriately to integrate the new parent class into the design and derive the more specific transaction classes from the new base class. It also involves integrating the new class into the bank class and the addition of methods into the bank system. With the addition of a base class for all transactions, we have added further core concepts of OOP into the program, which aligns with the learning outcomes of the subject, and in doing so, removed some duplication across classes, by abstracting aspects the transactions into the base class. My code is evidence of meeting the required outcomes and will be supported by additional diagrams in my video.

May 4, 2020

```
using System;

namespace Task_7_1P
{
    /// <summary>
    /// Prototype for a deposit transaction
    /// </summary>
    class DepositTransaction : Transaction
    {
        // Instance variables
        private Account _account;

        public Account Account { get => _account; }

        /// <summary>
        /// Constructs a deposit transaction object
        /// </summary>
        /// <param name="account">Account to deposit into</param>
        /// <param name="amount">Amount to deposit</param>
        public DepositTransaction(Account account, decimal amount) : base(amount)
        {
            _account = account;
        }

        /// <summary>
        /// Returns the name of the account receiving the deposit
        /// </summary>
        /// <returns>
        /// String of the account name
        /// </returns>
        public override string GetAccountName()
        {
            return _account.Name;
        }

        /// <summary>
        /// Prints the details and status of a deposit
        /// </summary>
        public override void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "DEPOSIT AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
             ↪  _amount.ToString("C"));
            if (!Executed)
            {
                Console.Write("{0, 20}|", "Pending");
            }
            else if (Reversed)
            {
                Console.Write("{0, 20}|", "Deposit reversed");
```

```
53              }
54              else if (Success)
55              {
56                  Console.Write("{0, 20}|", "Deposit complete");
57              }
58              else if (!Success)
59              {
60                  Console.Write("{0, 20}|", "Invalid deposit");
61              }
62              Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
63              Console.WriteLine(new String('-', 85));
64          }
65
66          /// <summary>
67          /// Executes a deposit transaction
68          /// </summary>
69          public override void Execute()
70          {
71              base.Execute();
72
73              _success = _account.Deposit(_amount);
74              Console.WriteLine(Success);
75              if (!_success)
76              {
77                  throw new InvalidOperationException("Deposit amount invalid");
78              }
79          }
80
81          /// <summary>
82          /// Reverses a deposit if previously executed successfully
83          /// </summary>
84          public override void Rollback()
85          {
86              base.Rollback();
87              bool complete = _account.Withdraw(_amount); // Withdraw returns boolean
88              if (!complete) // Withdraw didn't occur
89              {
90                  throw new InvalidOperationException("Insufficient funds to
                    ↪  rollback");
91              }
92              base.Reversed = true;
93          }
94      }
95  }
```

```
using System;

namespace Task_7_1P
{
    /// <summary>
    /// Prototype for a Withdraw transaction
    /// </summary>
    class WithdrawTransaction : Transaction
    {
        // Instance variables
        private Account _account;

        /// <summary>
        /// Constructs a WithdrawTransaction
        /// </summary>
        /// <param name="account">Account to withdraw from</param>
        /// <param name="amount">Amount to withdraw</param>
        public WithdrawTransaction(Account account, decimal amount) : base(amount)
        {
            _account = account;
        }

        /// <summary>
        /// Returns the name of the account being debited
        /// </summary>
        /// <returns>
        /// String of the account name
        /// </returns>
        public override string GetAccountName()
        {
            return _account.Name;
        }

        /// <summary>
        /// Prints the details and status of the withdrawal
        /// </summary>
        public override void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, 20}|{2, 20}|{3, 20}|",
                "ACCOUNT", "WITHDRAW AMOUNT", "STATUS", "CURRENT BALANCE");
            Console.WriteLine(new String('-', 85));
            Console.Write("|{0, -20}|{1, 20}|", _account.Name,
            ↪  _amount.ToString("C"));
            if (!Executed)
            {
                Console.Write("{0, 20}|", "Pending");
            }
            else if (Reversed)
            {
                Console.Write("{0, 20}|", "Withdraw reversed");
            }
            else if (Success)
```

```
53              {
54                  Console.Write("{0, 20}|", "Withdraw complete");
55              }
56              else if (!Success)
57              {
58                  Console.Write("{0, 20}|", "Insufficient funds");
59              }
60              Console.WriteLine("{0, 20}|", _account.Balance.ToString("C"));
61              Console.WriteLine(new String('-', 85));
62          }
63
64          /// <summary>
65          /// Executes the withdrawal
66          /// </summary>
67          /// <exception cref="System.InvalidOperationException">Thrown
68          /// when the withdraw is already complete or insufficient funds</exception>
69          public override void Execute()
70          {
71              base.Execute();
72
73              _success = _account.Withdraw(_amount);
74              if (!_success)
75              {
76                  throw new InvalidOperationException("Insufficient funds");
77              }
78          }
79
80          /// <summary>
81          /// Reverses the withdraw if previously executed successfully
82          /// </summary>
83          /// <exception cref="System.InvalidOperationException">Thrown
84          /// if already rolled back or if there are insufficient
85          /// funds to complete the rollback</exception>
86          public override void Rollback()
87          {
88              base.Rollback();
89              bool complete = _account.Deposit(_amount); // Deposit returns boolean
90              if (!complete) // Deposit didn't occur
91              {
92                  throw new InvalidOperationException("Invalid amount");
93              }
94              base.Reversed = true;
95          }
96      }
97  }
```

```cs
using System;

namespace Task_7_1P
{
    /// <summary>
    /// Prototype for a transfer transaction
    /// </summary>
    class TransferTransaction : Transaction
    {
        // Instance variables
        private Account _fromAccount;
        private Account _toAccount;
        private DepositTransaction _deposit;
        private WithdrawTransaction _withdraw;

        // Properties
        public new bool Success { get => (_deposit.Success && _withdraw.Success); }

        /// <summary>
        /// Constructor for a transfer transaction
        /// </summary>
        /// <param name="fromAccount">The account to transfer from</param>
        /// <param name="toAccount">The account to transfer to</param>
        /// <param name="amount">The amount to transfer</param>
        /// <exception cref="System.ArgumentOutOfRangeException">Thrown
        /// when the amount is negative</exception>
        public TransferTransaction(Account fromAccount, Account toAccount, decimal
        ↪ amount) : base(amount)
        {
            _fromAccount = fromAccount;
            _toAccount = toAccount;
            _withdraw = new WithdrawTransaction(_fromAccount, _amount);
            _deposit = new DepositTransaction(_toAccount, _amount);
        }

        /// <summary>
        /// Returns the name of the account(s) in the transaction
        /// </summary>
        /// <returns>
        /// String of the account names
        /// </returns>
        public override string GetAccountName()
        {
            return "From: " + _fromAccount.Name + ", To: " + _toAccount.Name;
        }

        /// <summary>
        /// Prints the details of the transfer
        /// </summary>
        public override void Print()
        {
            Console.WriteLine(new String('-', 85));
            Console.WriteLine("|{0, -20}|{1, -20}|{2, 20}|{3, 20}|",
```

```
53                  "FROM ACCOUNT", "To ACCOUNT", "TRANSFER AMOUNT", "STATUS");
54              Console.WriteLine(new String('-', 85));
55              Console.Write("|{0, -20}|{1, -20}|{2, 20}|", _fromAccount.Name,
                ↪  _toAccount.Name, _amount.ToString("C"));
56              if (!Executed)
57              {
58                  Console.WriteLine("{0, 20}|", "Pending");
59              }
60              else if (Reversed)
61              {
62                  Console.WriteLine("{0, 20}|", "Transfer reversed");
63              }
64              else if (Success)
65              {
66                  Console.WriteLine("{0, 20}|", "Transfer complete");
67              }
68              else if (!Success)
69              {
70                  Console.WriteLine("{0, 20}|", "Transfer failed");
71              }
72              Console.WriteLine(new String('-', 85));
73          }
74
75          /// <summary>
76          /// Executes the transfer
77          /// </summary>
78          /// <exception cref="System.InvalidOperationException">Thrown
79          /// when previously executed or deposit or withdraw fail</exception>
80          public override void Execute()
81          {
82              base.Execute();
83
84              try
85              {
86                  _withdraw.Execute();
87              }
88              catch (InvalidOperationException exception)
89              {
90                  Console.WriteLine("Transfer failed with reason: " +
                    ↪  exception.Message);
91                  _withdraw.Print();
92              }
93
94              if (_withdraw.Success)
95              {
96                  try
97                  {
98                      _deposit.Execute();
99                  }
100                 catch (InvalidOperationException exception)
101                 {
102                     Console.WriteLine("Transfer failed with reason: " +
                        ↪  exception.Message);
```

```
103                     _deposit.Print();
104                     try
105                     {
106                         _withdraw.Rollback();
107                     }
108                     catch (InvalidOperationException e)
109                     {
110                         Console.WriteLine("Withdraw could not be reversed with
                            ↪  reason: " + e.Message);
111                         _withdraw.Print();
112                         return;
113                     }
114                 }
115             }
116             _success = true;
117         }
118 
119         /// <summary>
120         /// Rolls the transfer back
121         /// </summary>
122         /// <exception cref="System.InvalidOperationException">Thrown
123         /// when the rollback has already been executed or it fails</exception>
124         public override void Rollback()
125         {
126             base.Rollback();
127 
128             if (this.Success)
129             {
130                 try
131                 {
132                     _deposit.Rollback();
133                 }
134                 catch (InvalidOperationException exception)
135                 {
136                     Console.WriteLine("Failed to rollback deposit: "
137                         + exception.Message);
138                     return;
139                 }
140 
141                 try
142                 {
143                     _withdraw.Rollback();
144                 }
145                 catch (InvalidOperationException exception)
146                 {
147                     Console.WriteLine("Failed to rollback withdraw: "
148                         + exception.Message);
149                     return;
150                 }
151             }
152             base.Reversed = true;
153         }
154     }
```

```
155 }
```

```cs
using System;

namespace Task_7_1P
{
    /// <summary>
    /// Baseclass for transaction classes
    /// </summary>
    abstract class Transaction
    {
        // Instance variables
        protected decimal _amount;
        protected Boolean _success;
        private Boolean _executed;
        private Boolean _reversed;
        private DateTime _dateStamp;

        // public properties
        public Boolean Success { get => _success; }
        public Boolean Executed { get => _executed; }
        public Boolean Reversed { get => _reversed; set => _reversed = value; }
        public DateTime DateStamp { get => _dateStamp; }
        public decimal Amount { get => _amount; } // Added for the transaction
        ↪ history print

        /// <summary>
        /// Creates a new Transaction object
        /// </summary>
        /// <param name="amount">The amount of the transaction</param>
        public Transaction(decimal amount)
        {
            if (amount > 0)
            {
                _amount = amount;
            }
            else
            {
                amount = 0;
                throw new ArgumentOutOfRangeException("Amount must be > $0.00");
            }
            // _executed, _success, _reversed false by default
        }

        /// <summary>
        /// Provides a virtual method to return the name of the account(s)
        /// involved in the transaction
        /// </summary>
        /// <returns>
        /// String of the account name
        /// </returns>
        public virtual string GetAccountName()
        {
            return "Unknown name";
        }
```

```
53
54          /// <summary>
55          /// Writes the amount and status to the Console
56          /// </summary>
57          public virtual void Print()
58          {
59              Console.WriteLine(
60                  "Transaction amount: {0}, Executed: {1}, Success: {2}, Reversed:
                    ↪ {3}",
61                  _amount.ToString("C"), _executed, _success, _reversed);
62          }
63
64          /// <summary>
65          /// Records execution of the transaction if not previously executed
                ↪ successfully
66          /// </summary>
67          public virtual void Execute()
68          {
69              if (_executed && _success)
70              {
71                  throw new InvalidOperationException("Transaction previously
                        ↪ executed");
72              }
73              _dateStamp = DateTime.Now;
74              _executed = true;
75          }
76
77          /// <summary>
78          /// Records rolling back of the transaction if not previously rolled back
79          /// </summary>
80          public virtual void Rollback()
81          {
82              if (_reversed)
83              {
84                  throw new InvalidOperationException("Transaction already reversed");
85              }
86              else if (!_success)
87              {
88                  throw new InvalidOperationException(
89                      "Transaction not successfully executed. Nothing to rollback.");
90              }
91              _dateStamp = DateTime.Now;
92          }
93      }
94  }
```

```
using System;

namespace Task_7_1P
{
    enum MenuOption
    {
        CreateAccount,
        Withdraw,
        Deposit,
        Transfer,
        Rollback,
        Print,
        Quit
    }

    /// <summary>
    /// BankSystem implements a banking system to operate on accounts
    /// </summary>
    class BankSystem
    {
        // Reads string input in the console
        /// <summary>
        /// Reads string input in the console
        /// </summary>
        /// <returns>
        /// The string input of the user
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static String ReadString(String prompt)
        {
            Console.Write(prompt + ": ");
            return Console.ReadLine();
        }

        // Reads integer input in the console
        /// <summary>
        /// Reads integerinput in the console
        /// </summary>
        /// <returns>
        /// The input of the user as an integer
        /// </returns>
        /// <param name="prompt">The string prompt for the user</param>
        public static int ReadInteger(String prompt)
        {
            int number = 0;
            string numberInput = ReadString(prompt);
            while (!(int.TryParse(numberInput, out number)))
            {
                Console.WriteLine("Please enter a whole number");
                numberInput = ReadString(prompt);
            }
            return Convert.ToInt32(numberInput);
        }
```

```
54
55          // Reads integer input in the console between two numbers
56          /// <summary>
57          /// Reads integer input in the console between two numbers
58          /// </summary>
59          /// <returns>
60          /// The input of the user as an integer
61          /// </returns>
62          /// <param name="prompt">The string prompt for the user</param>
63          /// <param name="minimum">The minimum number allowed</param>
64          /// <param name="maximum">The maximum number allowed</param>
65          public static int ReadInteger(String prompt, int minimum, int maximum)
66          {
67              int number = ReadInteger(prompt);
68              while (number < minimum || number > maximum)
69              {
70                  Console.WriteLine("Please enter a whole number from " +
71                                    minimum + " to " + maximum);
72                  number = ReadInteger(prompt);
73              }
74              return number;
75          }
76
77          // Reads decimal input in the console
78          /// <summary>
79          /// Reads decimal input in the console
80          /// </summary>
81          /// <returns>
82          /// The input of the user as a decimal
83          /// </returns>
84          /// <param name="prompt">The string prompt for the user</param>
85          public static decimal ReadDecimal(String prompt)
86          {
87              decimal number = 0;
88              string numberInput = ReadString(prompt);
89              while (!(decimal.TryParse(numberInput, out number)) || number < 0)
90              {
91                  Console.WriteLine("Please enter a decimal number, $0.00 or
                    ↪  greater");
92                  numberInput = ReadString(prompt);
93              }
94              return Convert.ToDecimal(numberInput);
95          }
96
97          /// <summary>
98          /// Displays a menu of possible actions for the user to choose
99          /// </summary>
100         private static void DisplayMenu()
101         {
102             Console.WriteLine("\n**********************");
103             Console.WriteLine("*       Menu         *");
104             Console.WriteLine("**********************");
105             Console.WriteLine("*  1. New Account    *");
```

```
106             Console.WriteLine("*  2. Withdraw     *");
107             Console.WriteLine("*  3. Deposit      *");
108             Console.WriteLine("*  4. Transfer     *");
109             Console.WriteLine("*  5. Rollback     *");
110             Console.WriteLine("*  6. Print        *");
111             Console.WriteLine("*  7. Quit         *");
112             Console.WriteLine("********************");
113         }
114
115         /// <summary>
116         /// Returns a menu option chosen by the user
117         /// </summary>
118         /// <returns>
119         /// MenuOption chosen by the user
120         /// </returns>
121         private static MenuOption ReadUserOption()
122         {
123             DisplayMenu();
124             int option = ReadInteger("Choose an option", 1,
125                 Enum.GetNames(typeof(MenuOption)).Length);
126             return (MenuOption)(option - 1);
127         }
128
129         /// <summary>
130         /// Attempts to deposit funds into an account at a bank
131         /// </summary>
132         /// <param name="bank">The bank holding the account to deposit into</param>
133         static void DoDeposit(Bank bank)
134         {
135             Account account = FindAccount(bank);
136             if (account != null)
137             {
138                 decimal amount = ReadDecimal("Enter the amount");
139                 DepositTransaction deposit = new DepositTransaction(account,
                    ↪ amount);
140                 try
141                 {
142                     bank.Execute(deposit);
143                 }
144                 catch (InvalidOperationException)
145                 {
146                     deposit.Print();
147                     return;
148                 }
149                 deposit.Print();
150             }
151         }
152
153         /// <summary>
154         /// Attempts to withdraw funds from an account at a bank
155         /// </summary>
156         /// <param name="bank">The bank holding account to withdraw from</param>
157         static void DoWithdraw(Bank bank)
```

```
158         {
159             Account account = FindAccount(bank);
160             if (account != null)
161             {
162                 decimal amount = ReadDecimal("Enter the amount");
163                 WithdrawTransaction withdraw = new WithdrawTransaction(account,
                    ↪ amount);
164                 try
165                 {
166                     bank.Execute(withdraw);
167                 }
168                 catch (InvalidOperationException)
169                 {
170                     withdraw.Print();
171                     return;
172                 }
173                 withdraw.Print();
174             }
175         }
176 
177         /// <summary>
178         /// Attempts to transfer funds between accounts
179         /// </summary>
180         /// <param name="bank">The bank holding the accounts
181         /// to transfer between</param>
182         static void DoTransfer(Bank bank)
183         {
184             Console.WriteLine("Transfer from:");
185             Account from = FindAccount(bank);
186             Console.WriteLine("Transfer to:");
187             Account to = FindAccount(bank);
188             if (from != null && to != null)
189             {
190                 decimal amount = ReadDecimal("Enter the amount");
191                 try
192                 {
193                     TransferTransaction transfer = new TransferTransaction(from,
                        ↪ to, amount);
194                     bank.Execute(transfer);
195                     transfer.Print();
196                 }
197                 catch (Exception)
198                 {
199                     // Currently this is handled in the TransferTransaction. This
                        ↪ will be changed
200                 }
201             }
202         }
203 
204         /// <summary>
205         /// Outputs the account name and balance
206         /// </summary>
207         /// <param name="account">The account to print</param>
```

```
208         static void DoPrint(Bank bank)
209         {
210             Account account = FindAccount(bank);
211             if (account != null)
212             {
213                 account.Print();
214             }
215         }
216 
217         /// <summary>
218         /// Prints a list of transactions and allows them to be rolled back
219         /// if necessary
220         /// </summary>
221         /// <param name="bank">The bank to rollback transactions for</param>
222         static void DoRollback(Bank bank)
223         {
224             bank.PrintTransactionHistory();
225             int result = ReadInteger(
226                 "Enter transaction # to rollback (0 for no rollback)",
227                 0, bank.Transactions.Count);
228 
229             if (result == 0)
230                 return;
231 
232             bank.Rollback(bank.Transactions[result - 1]);
233         }
234 
235         /// <summary>
236         /// Creates a new account and adds it to the Bank
237         /// </summary>
238         /// <param name="bank">The bank to create the account in</param>
239         static void CreateAccount(Bank bank)
240         {
241             string name = ReadString("Enter account name");
242             decimal balance = ReadDecimal("Enter the opening balance");
243             bank.AddAccount(new Account(name, balance));
244         }
245 
246         private static Account FindAccount(Bank bank)
247         {
248             Account account = null;
249             string name = ReadString("Enter the account name");
250             account = bank.GetAccount(name);
251             if (account == null)
252             {
253                 Console.WriteLine("That account name does not exist at this bank");
254             }
255             return account;
256         }
257 
258         static void Main(string[] args)
259         {
260             Bank bank = new Bank();
```

```
261
262             do
263             {
264                 MenuOption chosen = ReadUserOption();
265                 switch (chosen)
266                 {
267                     case MenuOption.CreateAccount:
268                         CreateAccount(bank); break;
269
270                     case MenuOption.Withdraw:
271                         DoWithdraw(bank); break;
272
273                     case MenuOption.Deposit:
274                         DoDeposit(bank); break;
275
276                     case MenuOption.Transfer:
277                         DoTransfer(bank); break;
278
279                     case MenuOption.Rollback:
280                         DoRollback(bank); break;
281
282                     case MenuOption.Print:
283                         DoPrint(bank); break;
284
285                     case MenuOption.Quit:
286                     default:
287                         Console.WriteLine("Goodbye");
288                         System.Environment.Exit(0); // terminates the program
289                         break; // unreachable
290                 }
291             } while (true);
292         }
293     }
294 }
```

```cs
using System;
using System.Collections.Generic;

namespace Task_7_1P
{
    /// <summary>
    /// Prototype for a bank to hold accounts
    /// </summary>
    class Bank
    {
        // Instance variables
        private List<Account> _accounts;
        private List<Transaction> _transactions;

        public List<Transaction> Transactions { get => _transactions; }

        /// <summary>
        /// Creates an empty bank object with a list for accounts
        /// </summary>
        public Bank()
        {
            _accounts = new List<Account>();
            _transactions = new List<Transaction>();
        }

        /// <summary>
        /// Adds an account to the Bank accounts register
        /// </summary>
        /// <param name="account"></param>
        public void AddAccount(Account account)
        {
            _accounts.Add(account);
        }

        /// <summary>
        /// Returns the first Account corresponding to the name, or
        /// null if there is no account matching the criteria
        /// </summary>
        /// <param name="name"></param>
        /// <returns>
        /// Account matching the provided name, or null
        /// </returns>
        public Account GetAccount(string name)
        {
            foreach (Account account in _accounts)
            {
                if (account.Name == name)
                {
                    return account;
                }
            }
            return null;
        }
```

```
54
55          /// <summary>
56          /// Executes a transaction
57          /// </summary>
58          /// <param name="transaction">Transaction to execute</param>
59          public void Execute(Transaction transaction)
60          {
61              _transactions.Add(transaction);
62              try
63              {
64                  transaction.Execute();
65              }
66              catch (InvalidOperationException exception)
67              {
68                  Console.WriteLine("An error occurred in executing the transaction");
69                  Console.WriteLine("The error was: " + exception.Message);
70              }
71          }
72
73          /// <summary>
74          /// Rolls a transaction back
75          /// </summary>
76          /// <param name="transaction">Transaction to execute</param>
77          public void Rollback(Transaction transaction)
78          {
79              try
80              {
81                  transaction.Rollback();
82              }
83              catch (InvalidOperationException exception)
84              {
85                  Console.WriteLine("An error occurred in rolling the transaction
                    ↪ back");
86                  Console.WriteLine("The error was: " + exception.Message);
87              }
88          }
89
90          /// <summary>
91          /// Helper function for PrintTransactionHistory that converts the
92          /// type of the transaction to a string
93          /// </summary>
94          /// <param name="transaction">The transaction to return the
95          /// type of</param>
96          /// <returns>
97          /// The type as a string representation
98          /// </returns>
99          public string TransactionType(Transaction transaction)
100         {
101             switch (transaction.GetType().ToString())
102             {
103                 case "Task_7_1P.DepositTransaction":
104                     return "Deposit";
105                 case "Task_7_1P.WithdrawTransaction":
```

```
106                     return "Withdraw";
107                 case "Task_7_1P.TransferTransaction":
108                     return "Transfer";
109             }
110             return "Unknown";
111         }
112 
113         /// <summary>
114         /// Helper function for PrintTransactionHistory that converts the
115         /// current status to a string representation
116         /// </summary>
117         /// <param name="transaction">The transaction to return the
118         /// type of</param>
119         /// <returns>
120         /// The status as a string representation
121         /// </returns>
122         public string TransactionStatus(Transaction transaction)
123         {
124             if (!transaction.Executed)
125             {
126                 return "Pending";
127             }
128             else if (transaction.Reversed)
129             {
130                 return "Reversed";
131             }
132             else if (!transaction.Success)
133             {
134                 return "Incomplete";
135             }
136             else
137             {
138                 return "Complete";
139             }
140         }
141 
142         /// <summary>
143         /// Writes the list of transactions to the Console in a table format
144         /// </summary>
145         public void PrintTransactionHistory()
146         {
147             string transactionType = "";
148             string transactionStatus = "";
149             Console.WriteLine(new String('-', 127));
150             Console.WriteLine(
151                 "| {0,2} |{1,-25} | {2,-15}| {3,-40}|{4,15} | {5,15} |", "#",
152                 "DateTime", "Type", "Account Details", "Amount", "Status");
153             Console.WriteLine(new String('=', 127));
154             for (int i = 0; i < _transactions.Count; i++)
155             {
156                 transactionType = TransactionType(_transactions[i]);
157                 transactionStatus = TransactionStatus(_transactions[i]);
158                 Console.WriteLine(
```

```
159                     "| {0,2} |{1,-25} | {2,-15}| {3,-40}|{4,15} | {5,15} |",
160                     i + 1, _transactions[i].DateStamp, transactionType,
161                     _transactions[i].GetAccountName(),
162                     _transactions[i].Amount.ToString("C"), transactionStatus);
163             }
164             Console.WriteLine(new String('=', 127));
165         }
166     }
167 }
```

```cs
using System;

namespace Task_7_1P
{
    /// <summary>
    /// A bank account class to hold the account name and balance details
    /// </summary>
    class Account
    {
        // Instance variables
        private String _name;
        private decimal _balance;

        // Read-only properties
        public String Name { get => _name; }
        public decimal Balance { get => _balance; }


        /// <summary>
        /// Class constructor
        /// </summary>
        /// <param name="name">The name string for the account</param>
        /// <param name="balance">The decimal balance of the account</param>
        public Account(String name, decimal balance = 0)
        {
            _name = name;
            if (balance < 0)
                return;
            _balance = balance;
        }

        /// <summary>
        /// Deposits money into the account
        /// </summary>
        /// <returns>
        /// Boolean whether the deposit was successful (true) or not (false)
        /// </returns>
        /// <param name="amount">The decimal amount to add to the balance</param>
        public Boolean Deposit(decimal amount)
        {
            if ((amount < 0) || (amount == decimal.MaxValue))
                return false;

            _balance += amount;
            return true;
        }

        /// <summary>
        /// Withdraws money from the account (with no overdraw protection currently)
        /// </summary>
        /// <returns>
        /// Boolean whether the withdrawal was successful (true) or not (false)
        /// </returns>
```

```
54          /// <param name="amount">The amount to subtract from the balance</param>
55          public Boolean Withdraw(decimal amount)
56          {
57              if ((amount < 0) || (amount > _balance))
58                  return false;
59
60              _balance -= amount;
61              return true;
62          }
63
64          /// <summary>
65          /// Outputs the account name and current balance as a string
66          /// </summary>
67          public void Print()
68          {
69              Console.WriteLine("Account Name: {0}, Balance: {1}",
70                  _name, _balance.ToString("C"));
71          }
72      }
73  }
```

22 Documenting the Banking System

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Outcome	Weight
Evaluate Code	♦♦◇◇◇

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

Outcome	Weight
Principles	♦♦♦◇◇

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

Outcome	Weight
Build Programs	♦◇◇◇◇

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

Outcome	Weight
Design	♦♦♦♦♦

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

Outcome	Weight
Justify	♦◇◇◇◇

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

Date	Author	Comment
2020/05/04 19:39	Peter Stacey	Ready to Mark
2020/05/04 19:48	Peter Stacey	Ready to Mark
2020/05/04 19:48	Peter Stacey	Small mistake compared to my code (I initially had FindAccount as a public static void method, while it is actually a private static void method).
2020/05/04 22:10	Dipto Pratyaksa	Very nice work Peter! I always enjoy your submission which I think lead the whole wolf-gang pack this trimester!
2020/05/04 22:12	Dipto Pratyaksa	In your video, please answer the following questions as if you were getting real interview:How do you represent classes,fields,metods,and relationships in terms of the Universal Modelling Language?Describe the relationship between the diagram and you code.How could you use this to think through a solution before you write the code?What would be the advantage to doing this?
2020/05/04 22:13	Dipto Pratyaksa	Discuss
2020/05/08 13:18	Peter Stacey	Hope everything is covered ok.Video Link: https://youtu.be/emw2P6u3jUI
2020/05/08 15:35	Dipto Pratyaksa	Boom! You can make a big hit with videos like that.
2020/05/08 15:35	Dipto Pratyaksa	Complete

Deakin University

Object Oriented Development

OnTrack Submission

Documenting the Banking System

Submitted By:
Peter Stacey
pstacey
2020/05/04 19:48

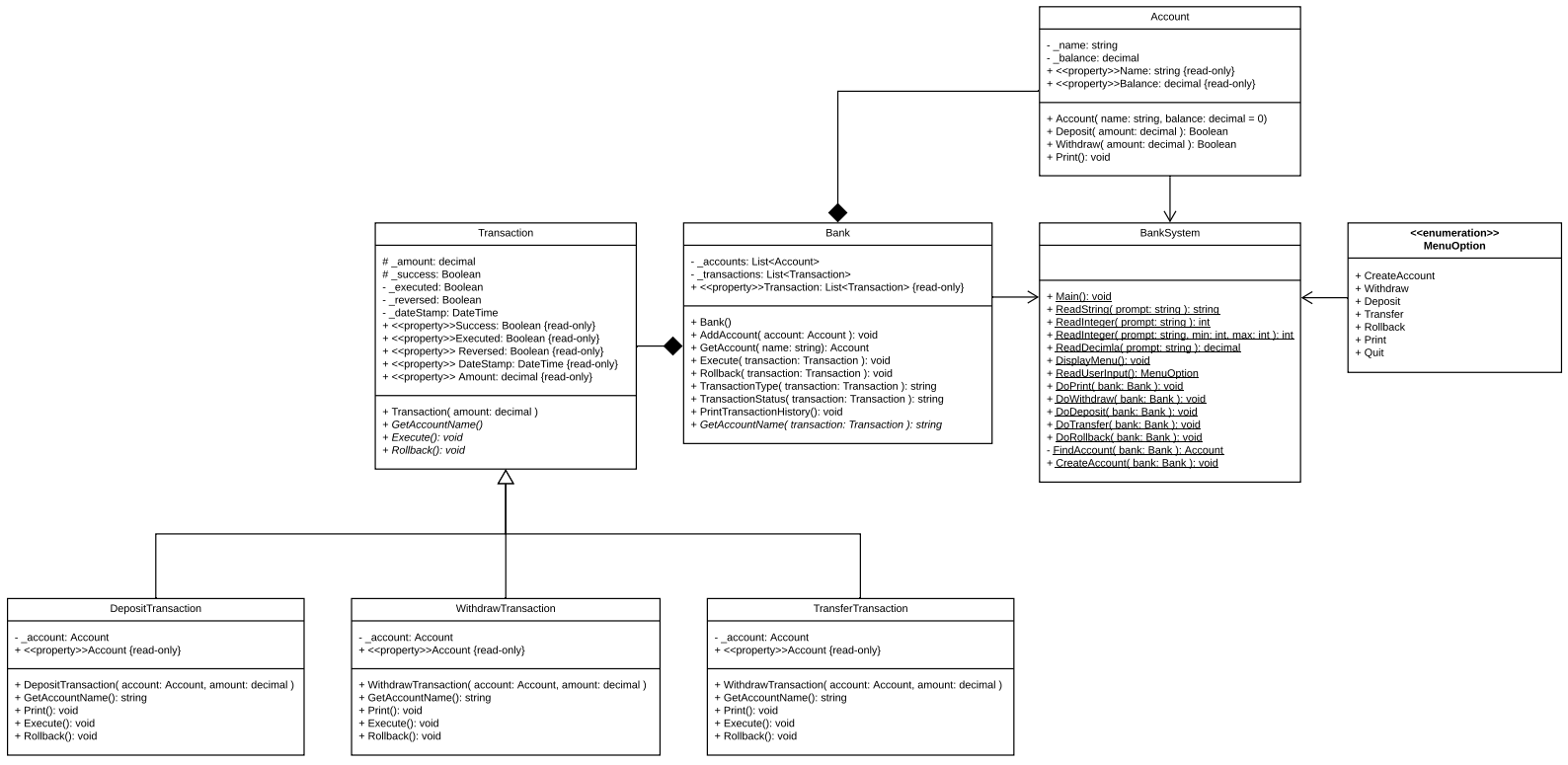
Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦◊◊
Build Programs	♦◊◊◊◊
Design	♦♦♦♦♦
Justify	♦◊◊◊◊

As a task that involves turning the Banking Application code into a design that conforms with the UML Standard, this task directly involves producing a diagram as evidence of understanding, which perfectly aligns with the Design outcome. Additionally, as the ability to produce an accurate UML requires understanding how to interpret code and represent virtual methods, composition, inheritance, association, dependencies and public, private and protected variables and properties, this also aligns with the Evaluate outcome, by requiring us to evaluate our own code to represent it as a UML.

May 4, 2020

SIT232 - Banking Application Design



23 Helping Your Peers

Note that we will not accept your report after the submission deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit.

Outcome	Weight
Evaluate Code	♦♦◊◊◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

Outcome	Weight
Principles	♦♦♦♦◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

Outcome	Weight
Build Programs	♦♦◊◊◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

Outcome	Weight
Design	♦♦◊◊◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

Outcome	Weight
Justify	♦♦◊◊◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

Date	Author	Comment
2020/05/17 14:19	Peter Stacey	Ready to Mark
2020/05/17 14:28	Peter Stacey	Ready to Mark
2020/05/17 14:28	Peter Stacey	Fixed 2 grammatical errors
2020/05/17 14:37	Peter Stacey	Ready to Mark
2020/05/17 14:37	Peter Stacey	Sorry. Added some additional evidence from Teams

Deakin University

Object Oriented Development

OnTrack Submission

Helping Your Peers

Submitted By:
Peter Stacey
pstacey
2020/05/17 14:37

Tutor:
Dipto Pratyaksa

Outcome	Weight
Evaluate Code	♦♦◊◊◊
Principles	♦♦♦♦◊
Build Programs	♦♦◊◊◊
Design	♦♦◊◊◊
Justify	♦♦◊◊◊

In being able to assist others, an abilities to both understand the principles covered in the subject, and to explain them simply to others is needed. This directly aligns with the Principles outcome. Additionally, in doing this, assisting others often involves reading small snippets of code, interpreting errors, explaining designs and using graphics to help outline concepts. These aspects all relate to the other four outcomes.

May 17, 2020

SIT232 – Object Oriented Development Task 1.3D - Helping Others

Student Name: Peter Stacey

Student ID: 219011171

In helping others, my contributions have been both on the SIT232 forums of Cloud Deakin and on the SIT232 Teams site.

Since the start of Trimester I have participated in assisting on the Deakin Forums for almost every task. For example, the count of posts by forum area:

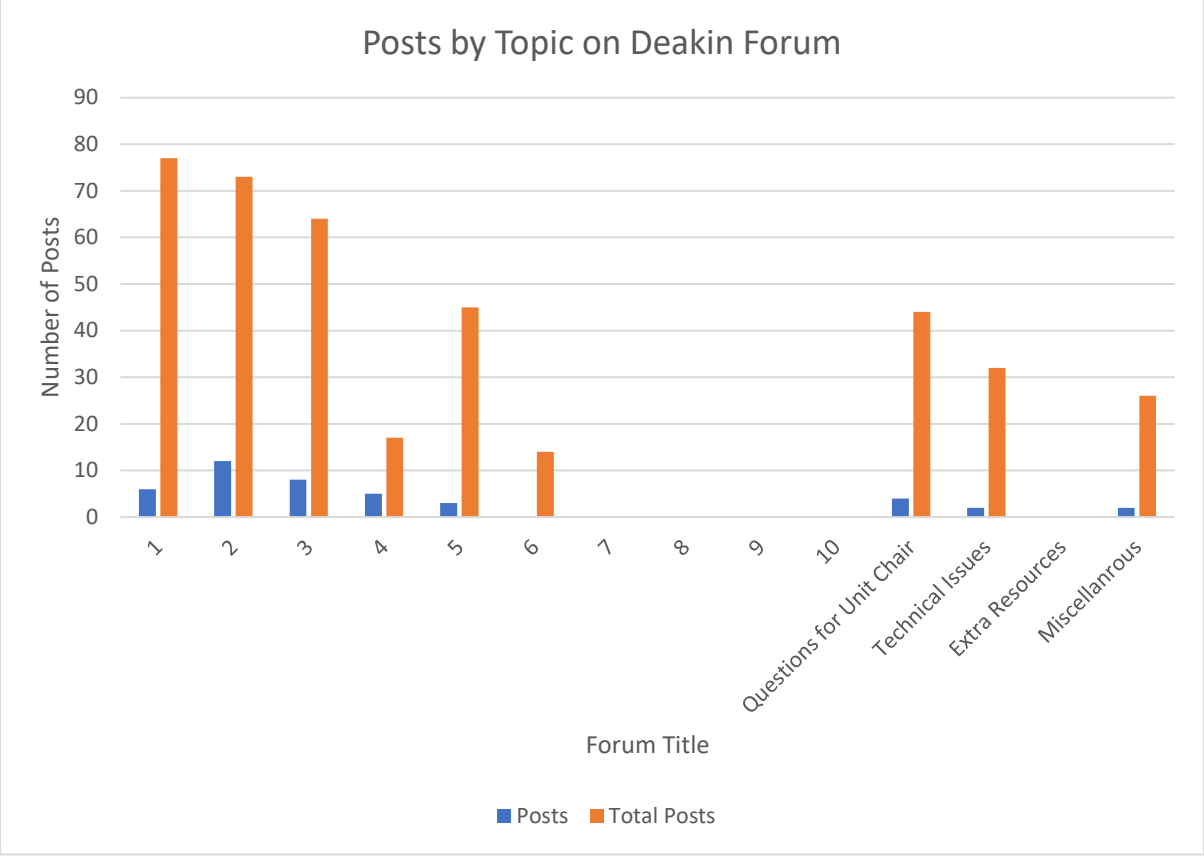
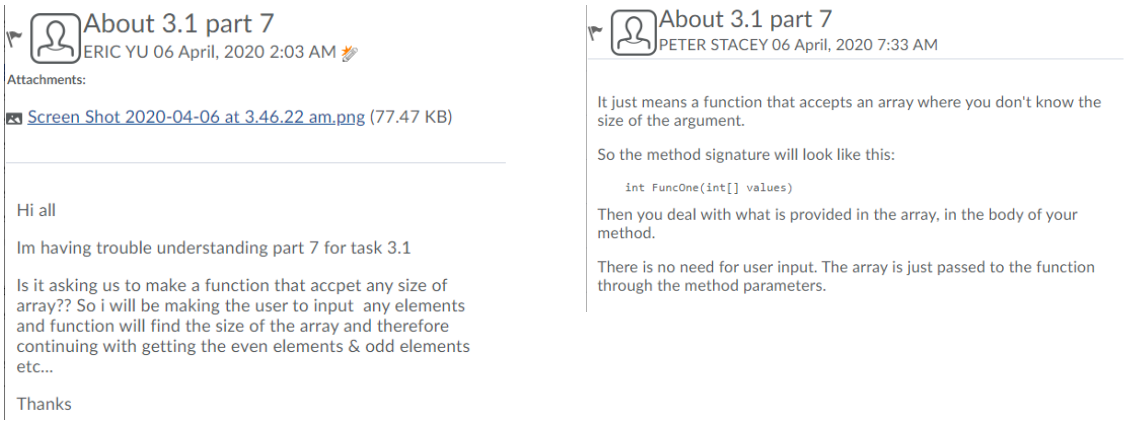


Figure 1: Involvement in discussion on the Deakin Forum

In some weeks, my contribution in terms of numbers of posts approached 30% of posts in a specific topic (ie. For the Week 4 discussion forum, my contribution is 30% of posts and for weeks 2 and 3, it's around the 15% of all posts contributed).

Volume of posts alone isn't necessarily a great indicator of usefulness, only of participation, however some example posts demonstrate the usefulness of the assistance provided:



About 3.1 part 7
ERIC YU 06 April, 2020 2:03 AM
Attachments:
Screen Shot 2020-04-06 at 3.46.22 am.png (77.47 KB)

Hi all

Im having trouble understanding part 7 for task 3.1

Is it asking us to make a function that accpet any size of array?? So i will be making the user to input any elements and function will find the size of the array and therefore continuing with getting the even elements & odd elements etc...

Thanks

About 3.1 part 7
PETER STACEY 06 April, 2020 7:33 AM

It just means a function that accepts an array where you don't know the size of the argument.

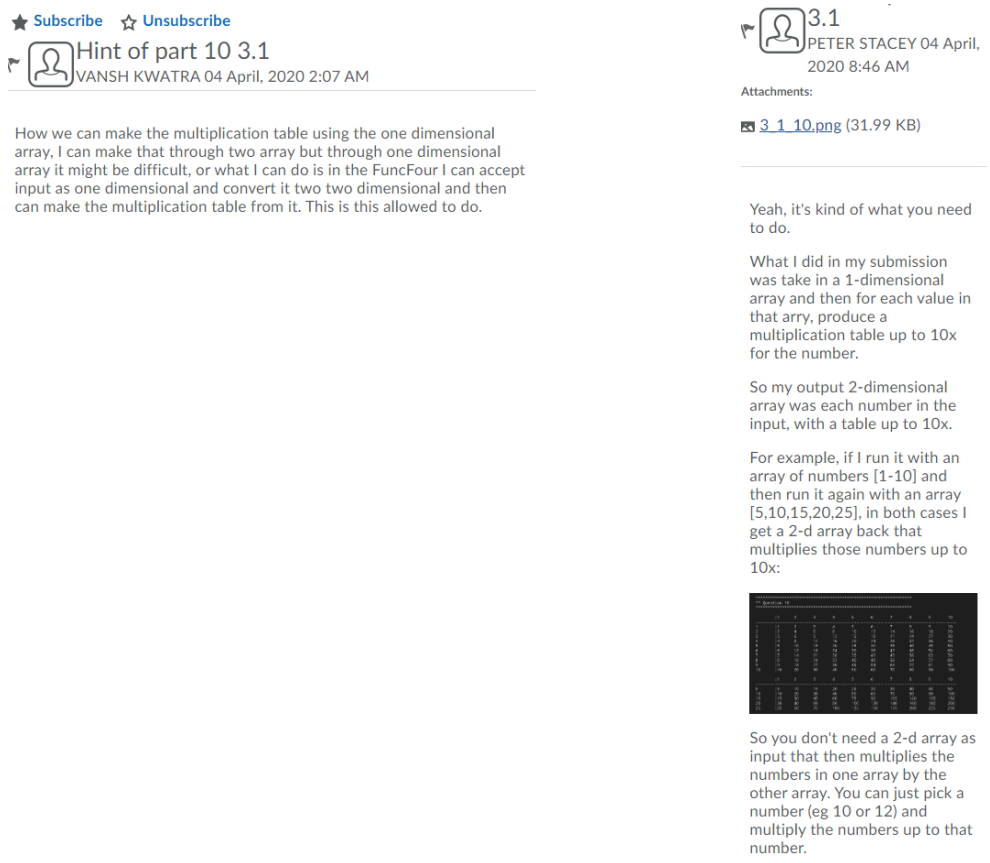
So the method signature will look like this:

```
int FuncOne(int[] values)
```

Then you deal with what is provided in the array, in the body of your method.

There is no need for user input. The array is just passed to the function through the method parameters.

In this example, I provide an example method signature to demonstrate the type of input required for the function, and directly answer the question. Additionally, as Eric indicated he was having difficulty understanding the requirement, I framed my reply by trying to explain the requirement in a new way, so that he had a second frame of reference to interpret the question from.



Subscribe Unsubscribe
Hint of part 10 3.1
VANSH KWATRA 04 April, 2020 2:07 AM

How we can make the multiplication table using the one dimensional array, I can make that through two array but through one dimensional array it might be difficult, or what I can do is in the FuncFour I can accept input as one dimensional and convert it two two dimensional and then can make the multiplication table from it. This is this allowed to do.

3.1
PETER STACEY 04 April, 2020 8:46 AM
Attachments:
3_1_10.png (31.99 KB)

Yeah, it's kind of what you need to do.

What I did in my submission was take in a 1-dimensional array and then for each value in that arry, produce a multiplication table up to 10x for the number.

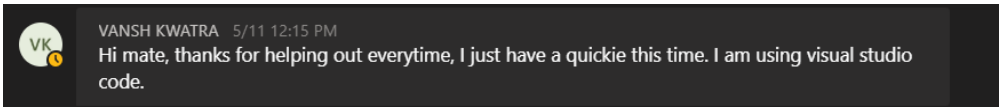
So my output 2-dimensional array was each number in the input, with a table up to 10x.

For example, if I run it with an array of numbers [1-10] and then run it again with an array [5,10,15,20,25], in both cases I get a 2-d array back that multiplies those numbers up to 10x:

So you don't need a 2-d array as input that then multiplies the numbers in one array by the other array. You can just pick a number (eg 10 or 12) and multiply the numbers up to that number.

In this example, I both provided an example of the output and my approach to solving the problem, as well as directly answered the query that Vansh had.

Of note, since we moved to Teams as a result of the move to online study for everyone, Vansh has continued to engage me privately in direct messages, to provide assistance as he found my assistance useful:



VANSH KWATRA 5/11 12:15 PM
Hi mate, thanks for helping out everytime, I just have a quickie this time. I am using visual studio code.

5.3 Tester Program
JAMES MARTIN 11 May, 2020 9:45 PM

I've found that the tester program seems to act differently from the supplied VS Studio App in 5.3.

More specifically the tester program frequently references the randomNumber variable. This is a reference to the random wait time after the timer starts. However I store my random number in my SimpleReactionController class. Changing the randomNumber in the tester class does not change it in the SimpleReactionController class.

Then when the tester class uses this new value to predict outcomes, my program acts differently than how it expects, because they are using two separate randomNumbers.

Additionally, I've noticed that the random number generation in the supplied VS App, and the Tester program is different. One merely returns randomNumber, the other returns a random number based on supplied parameters.

Would anyone be able to give me some direction as to how to understand this?

5.3 Tester Program
PETER STACEY 12 May, 2020 12:31 AM

None of the random numbers in the tester are random. They are all fixed, in the randomNumber variable.

Then, the RndGenerator.GetRandom method in lines 210-215 doesn't actually pick a random number. It just returns the number that has already been set, which guarantees that in your controller, the number you receive as the "random" number, is exactly the same as what the fixed value is.

The method has the same signature as the random generator expects, otherwise it wouldn't be an implementation of the interface and your controller wouldn't be able to call it. However internally it isn't using a random number at all.

So, your controller should behave exactly as expected, because it receives the pre-determined number, not a random one.

Hope that makes sense.

In this example, I evaluate the code we are provided in Task 5.3D, to explain how the tester is able to pass a non-random number to the controller, through the RngGenerator contained in the SimpleReactionMachine.cs class.

In all cases on the forum before we primarily moved to Teams, my posts attempted to assist, after first trying top understand the problem the person asking the question was having.

In addition, on the Discord site run by students, I've also contributed positively to answering questions.

For example:

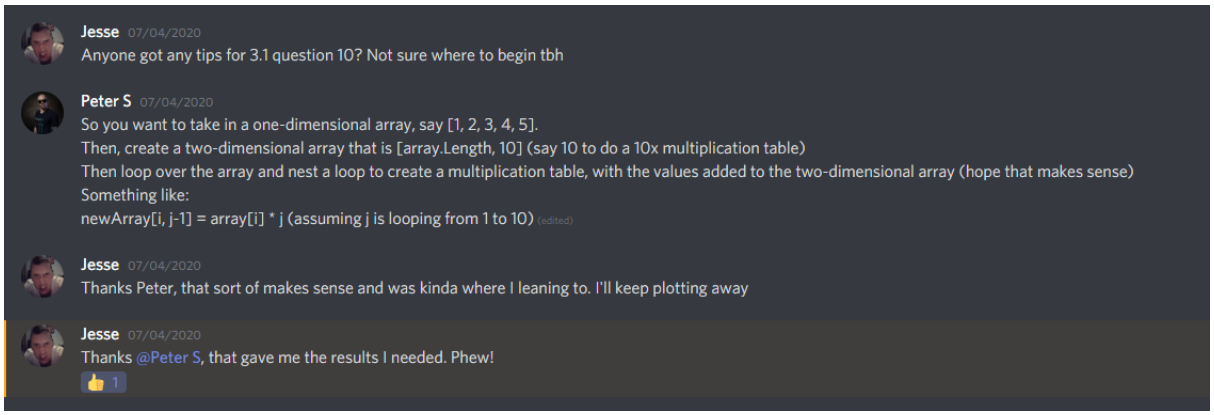


Figure 2: Example assistance provided on the student run Discord

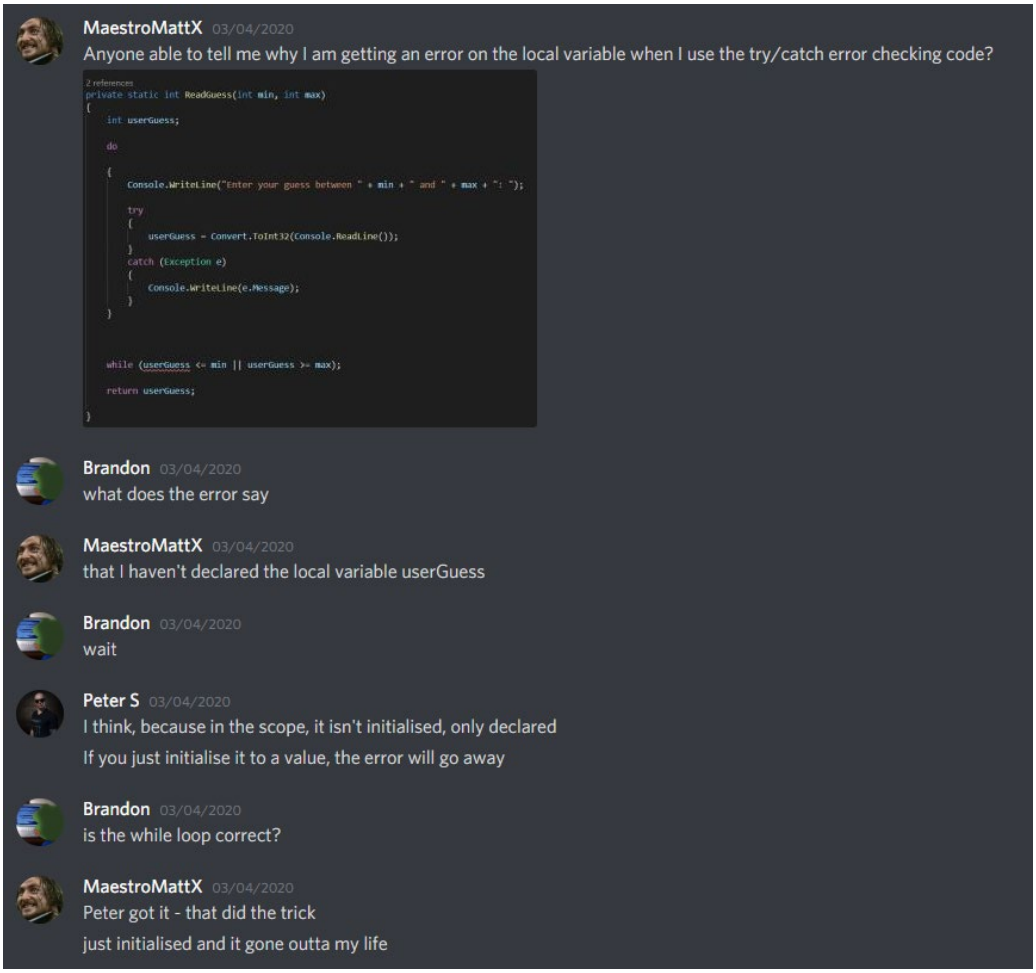
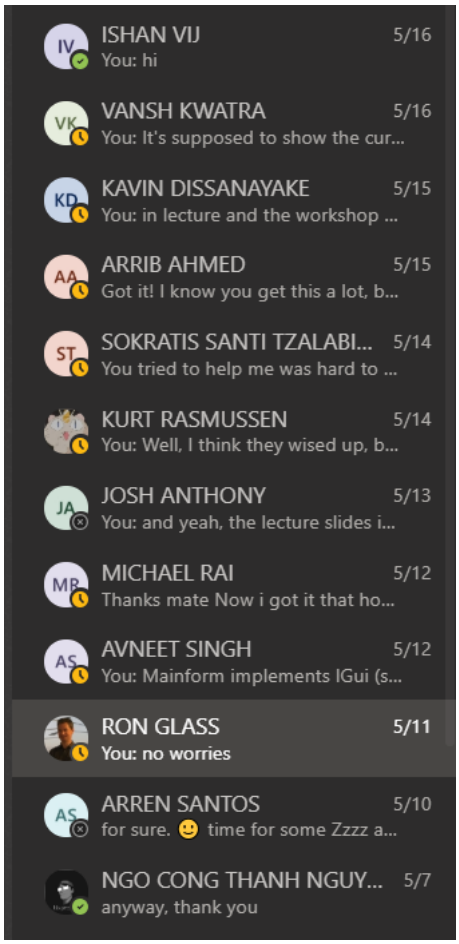


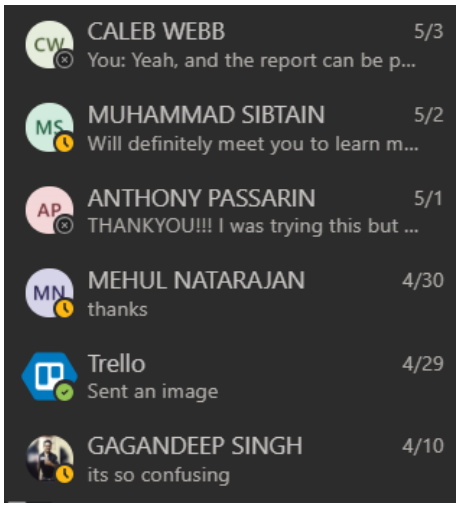
Figure 3: Example assistance provided on the student run Discord

Figure 4: Example of assistance provided on the student run Discord

Since we moved to Teams, I have continued to engage in discussions and provide assistance where I can.

This has resulted in many students starting private discussions to assist them with tasks. For example, the list of people I am involved in private messages with is indicative of the number of people I have been directly assisting where I can:





These discussions have been filled with comments acknowledging the helpfulness of the assistance provided. For example:

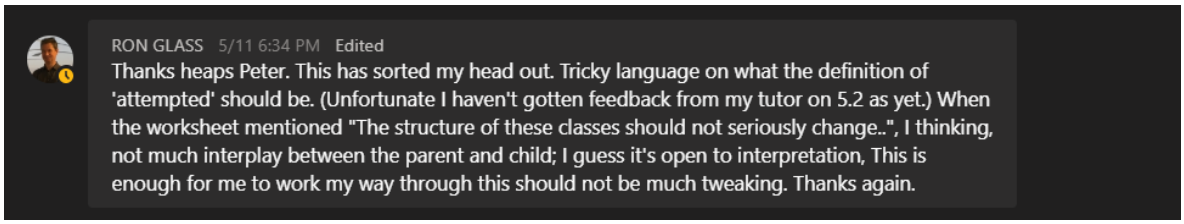


Figure 5: Example 1 of appreciation for assistance given

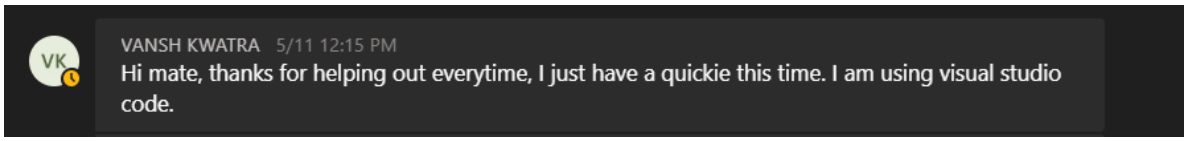


Figure 6: Example 2 of appreciation for assistance given

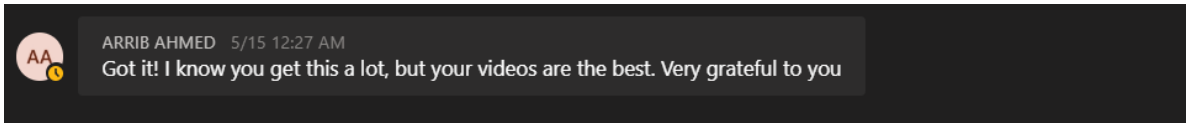


Figure 7: Example 3 of appreciation for assistance given

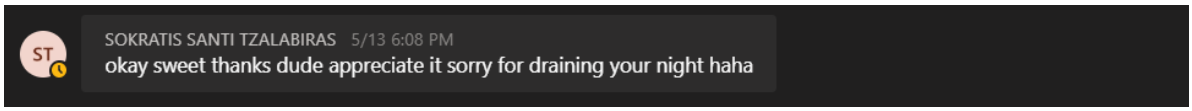


Figure 8: Example 4 of appreciation for assistance given

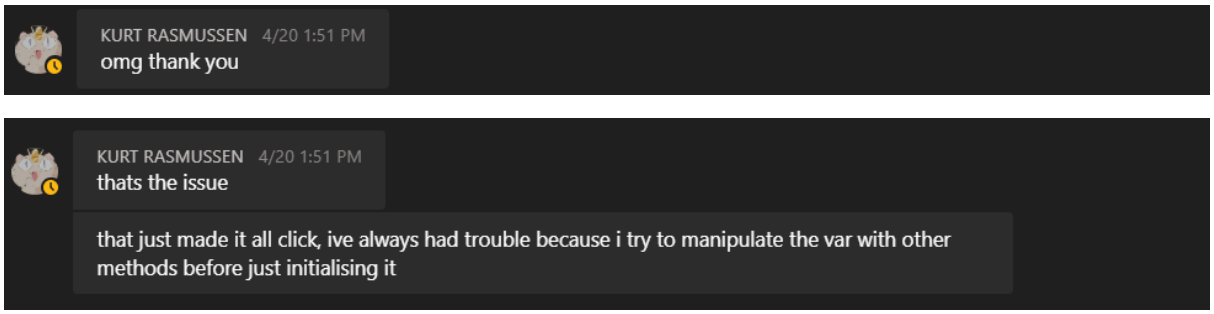


Figure 9: Example 5 of appreciation for assistance given

Figure 10: Example 6 of appreciation for assistance given

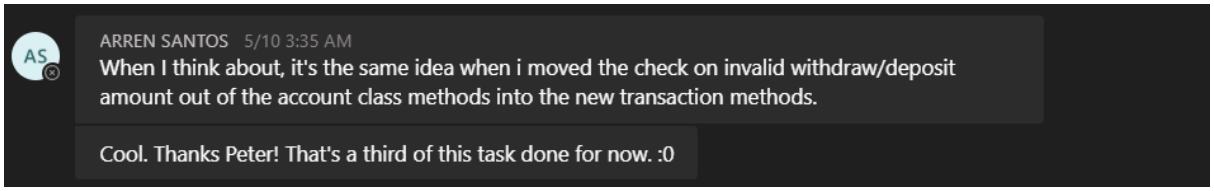


Figure 11: Example 7 of appreciation for assistance given

There are many more similar expressions of gratitude. Importantly however, in private messaging I never just supply an answer and always try to help other person understand the principles or concepts involved so they can solve it themselves.

For example:

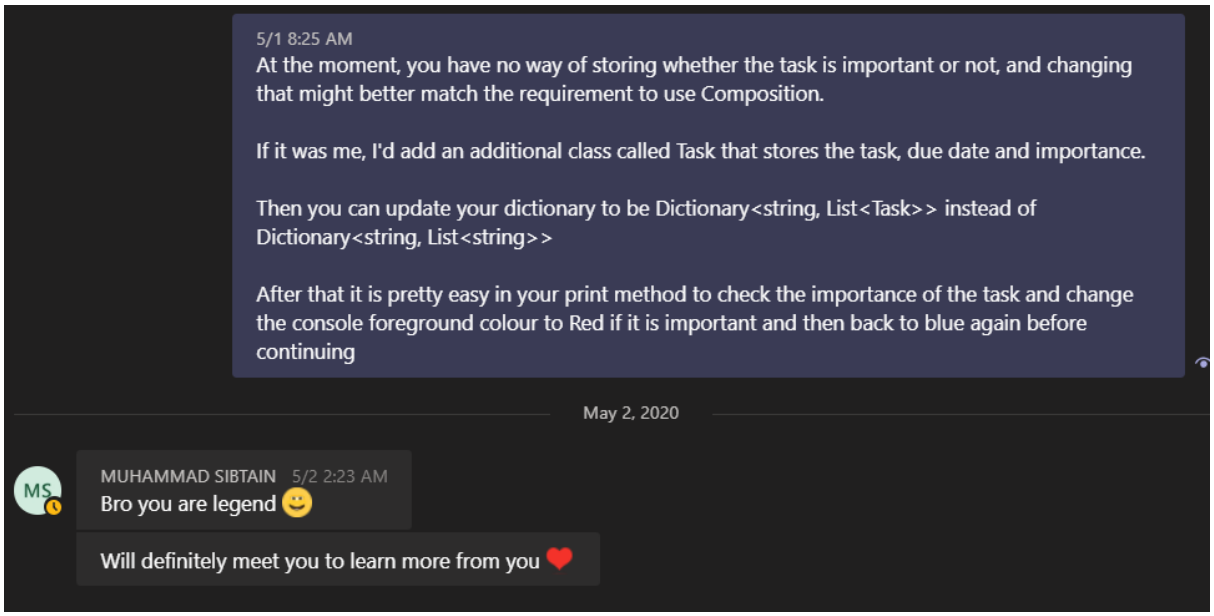


Figure 12: Example of explaining an error in a private chat and how someone might go about fixing their problem

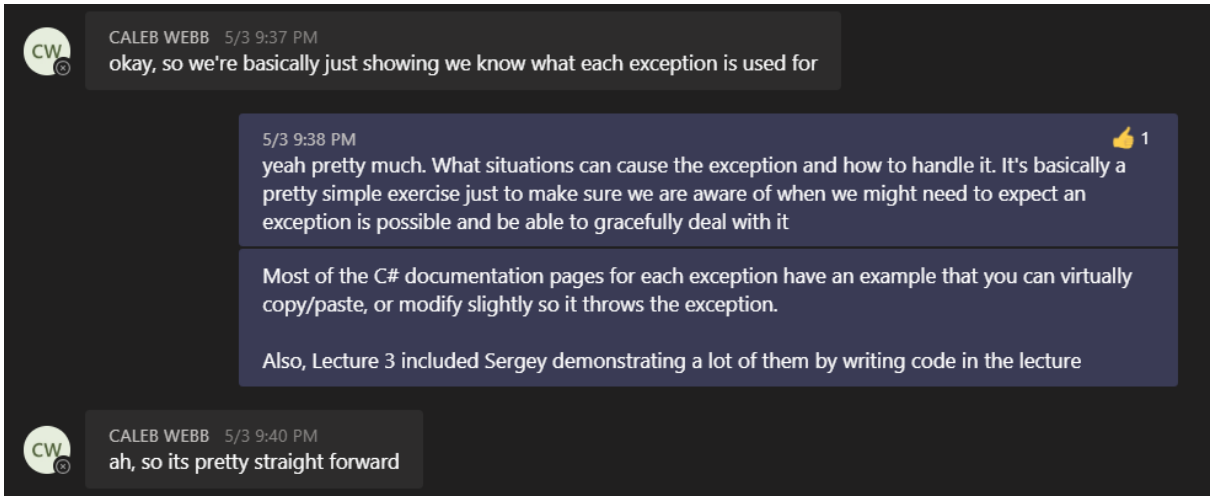


Figure 13: Example of responding in a private chat with general details and not just the answer

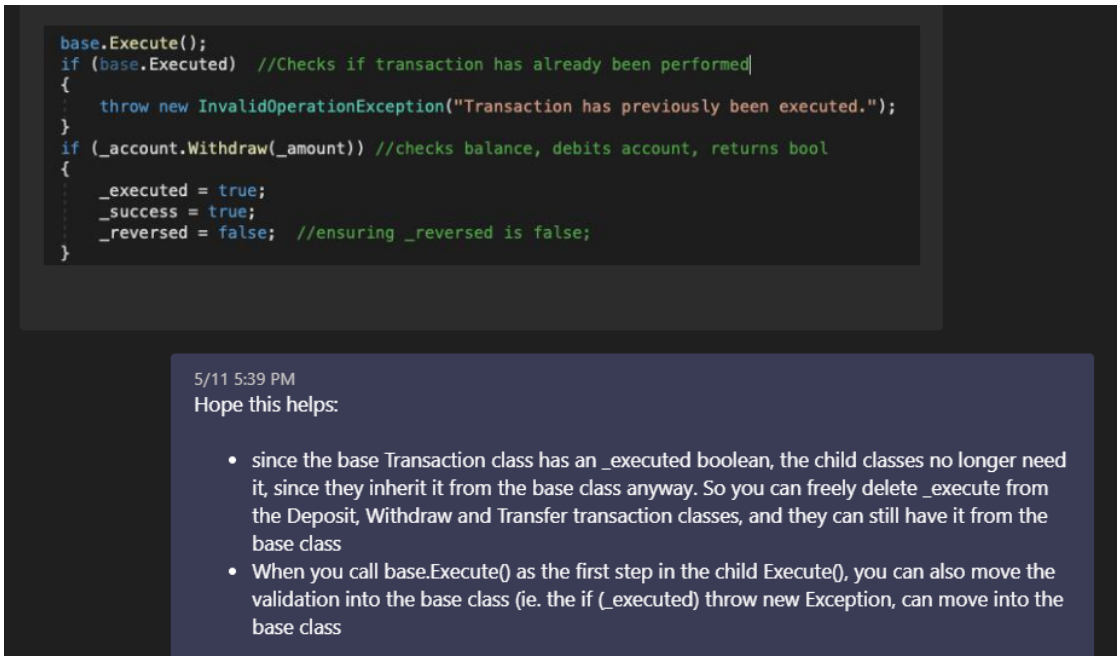


Figure 14: Example of explaining an approach in a private chat and not just providing the answer

Similarly, in the peer-to-peer and general channels on Teams, I've also tried to always be helpful, explaining concepts where I can.

For example:

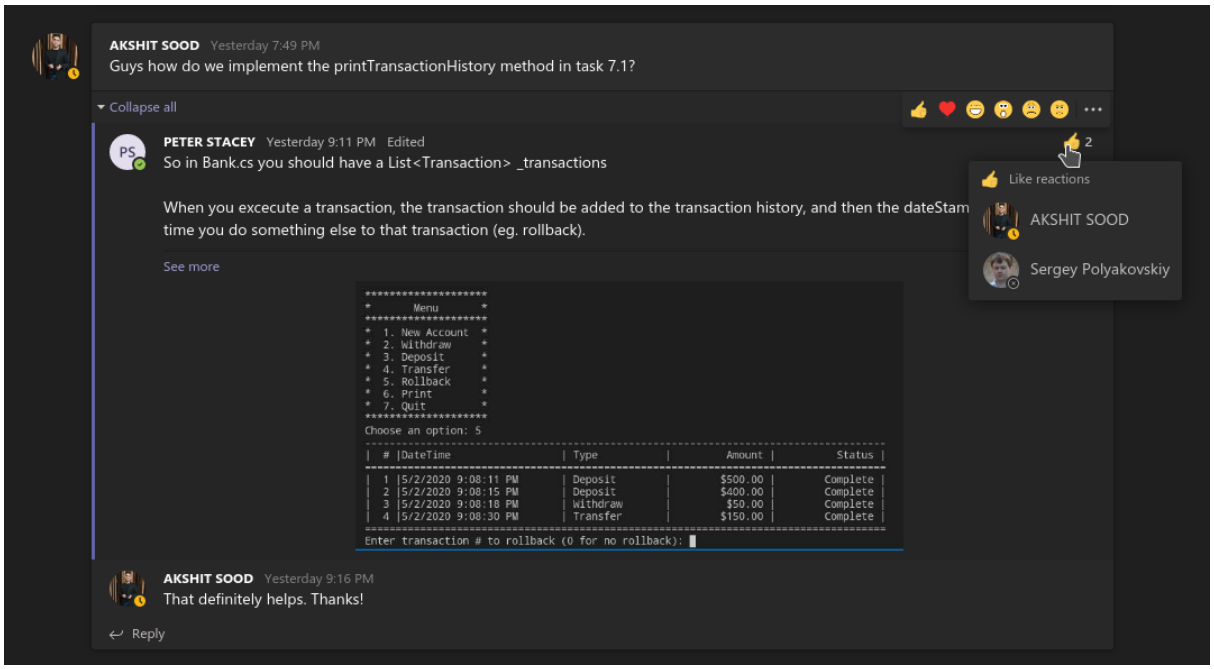


Figure 15: Example of assistance provided in the Peer-to-Peer channel

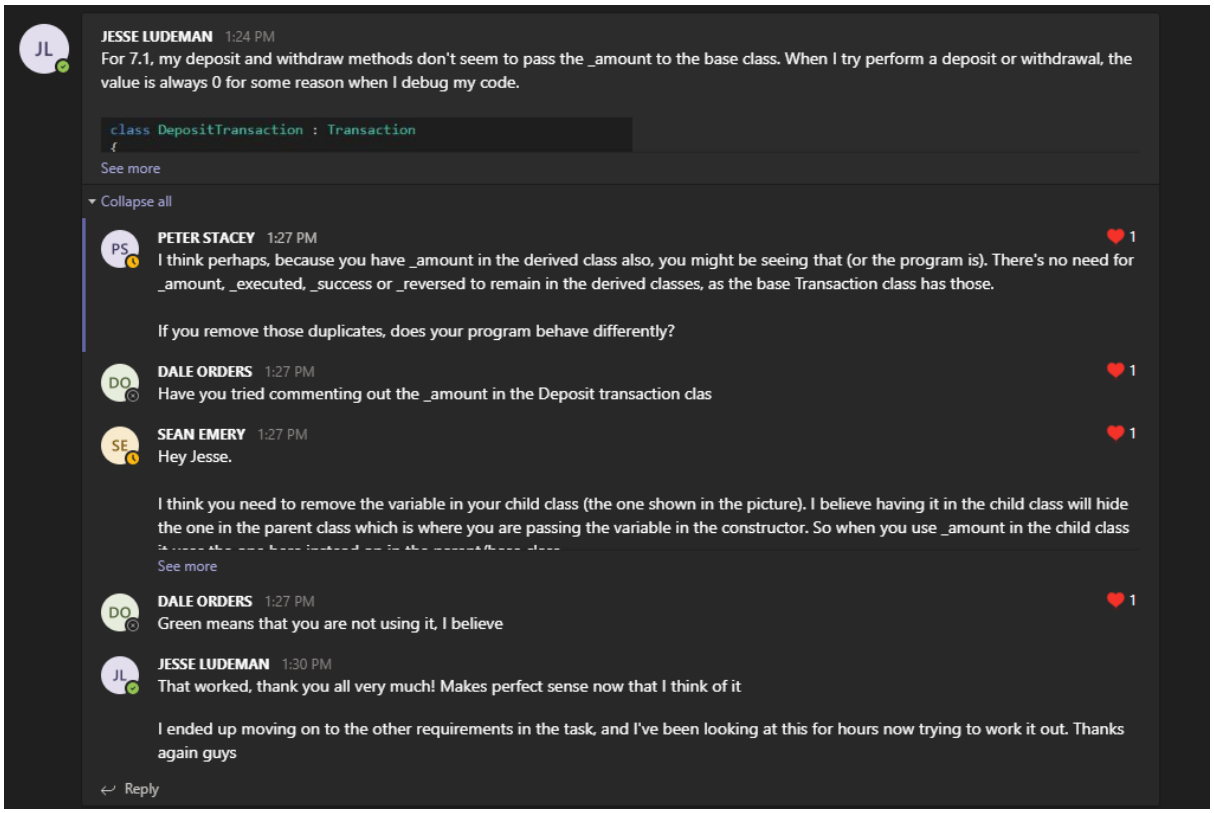


Figure 16: Example of assistance provided on Teams Peer-to-Peer channel

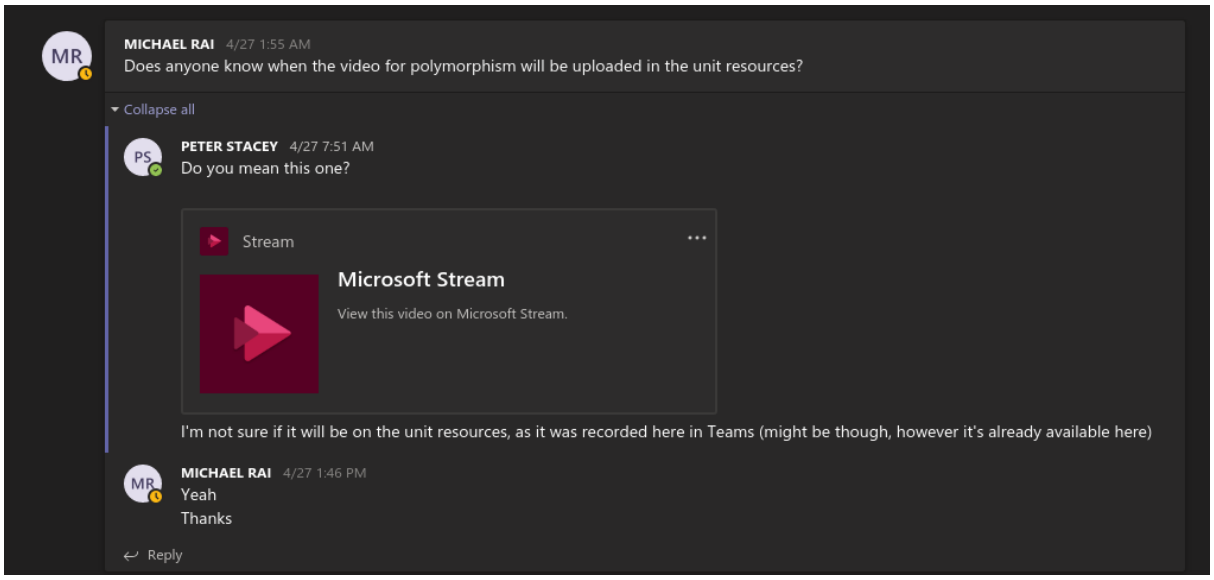


Figure 17: Example of assistance with links to material people are looking for

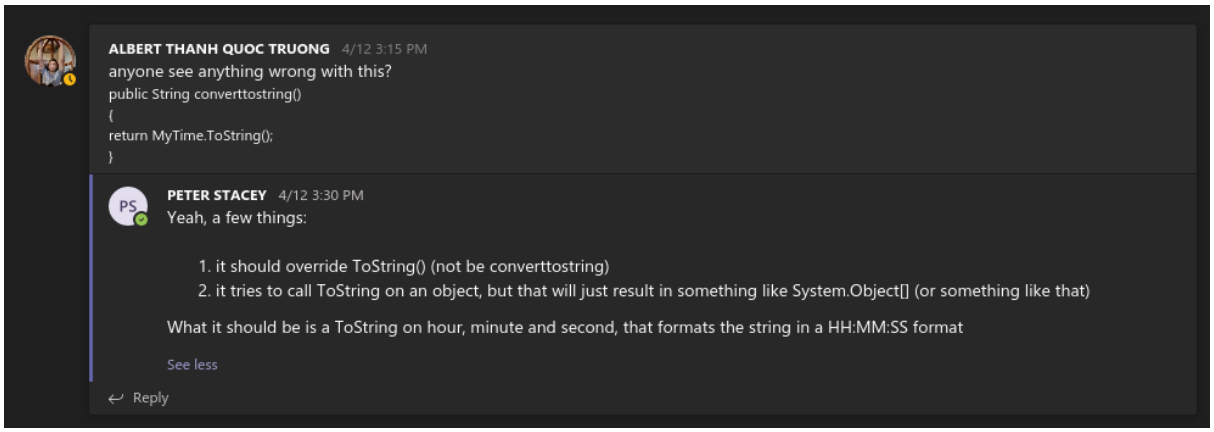


Figure 18: Example of assistance provided in the Peer-to-Peer channel

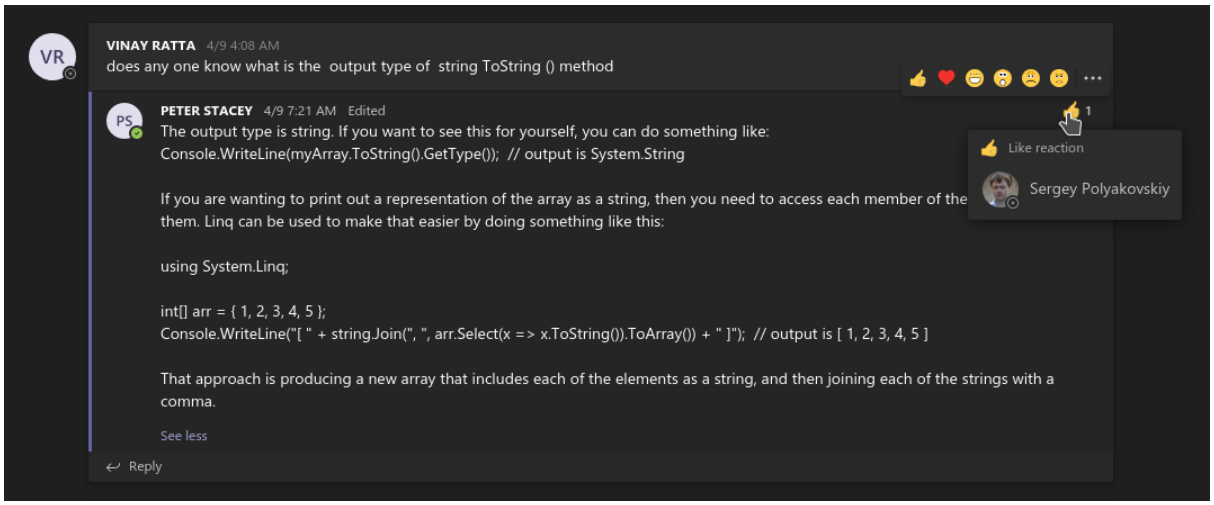
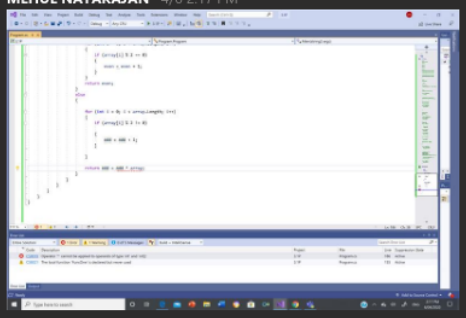


Figure 19: Example of explaining concepts involved in solving a problem, in the Peer-to-Peer channel

MEHUL NATARAJAN 4/6 2:17 PM



n

Collapse all

MEHUL NATARAJAN 4/6 2:18 PM
I'm getting an error for some reason

PETER STACEY 4/6 2:50 PM
Yeah because array is an array (ie. a collection of numbers) and you are trying to multiple it by an int.
Returning the odd product should return the product of all odd numbers multiplied together.
So above the for loop, just initialise an integer to 1 and then whenver you find an odd number, multiple that variable by the odd number and just return it at the end.

MEHUL NATARAJAN 4/6 3:02 PM
I made a variable called int number and set it to 1

but now when I run my program I get no output

PETER STACEY 4/6 3:03 PM
Are you returning the number from the function? Are you also multiplying it by any odd number in your for loop?

PETER STACEY 4/6 3:03 PM
Are you returning the number from the function? Are you also multiplying it by any odd number in your for loop?

MEHUL NATARAJAN 4/6 3:04 PM
This is what I wrote

```
return odd = odd * number;
```

PETER STACEY 4/6 3:07 PM Edited
Which isn't going to give you the right answer. In the for loop you already check if a number is odd, but then you are increasing the count of odd by 1. You should be multiplying "number" by the value from the array that is odd

For example:

- if you have an array [1, 2, 3, 4, 5]
  and you set int number = 1
  then in your for loop you identify each odd number, then:
  each time:
  number * array[i]
- In the end, that array above would end up returning 15 as the odd product
- then just return number at the end

  Really the best thing to do is work it through on paper so you understand the logic and what you want to achieve. Once you understand that fully, writing that into code is much easier

See less

MEHUL NATARAJAN 4/6 3:08 PM
Wait which part of the code are you referring to?

Here is the code below

```
int[] array = { 1, 2, 3, 4, 5, 6 };
        static int FuncOne(int[] array)
        {
          {
              int number = 1;
              int even = 1;
```

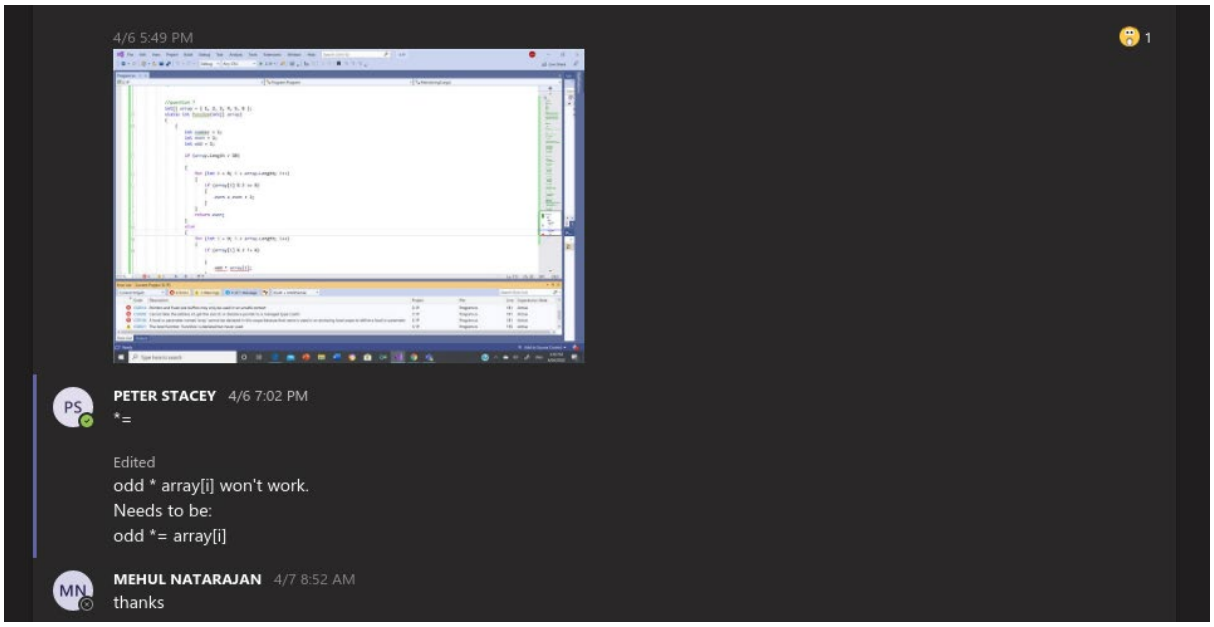


Figure 20: Example of staying with an issue to assist until the concept is understood

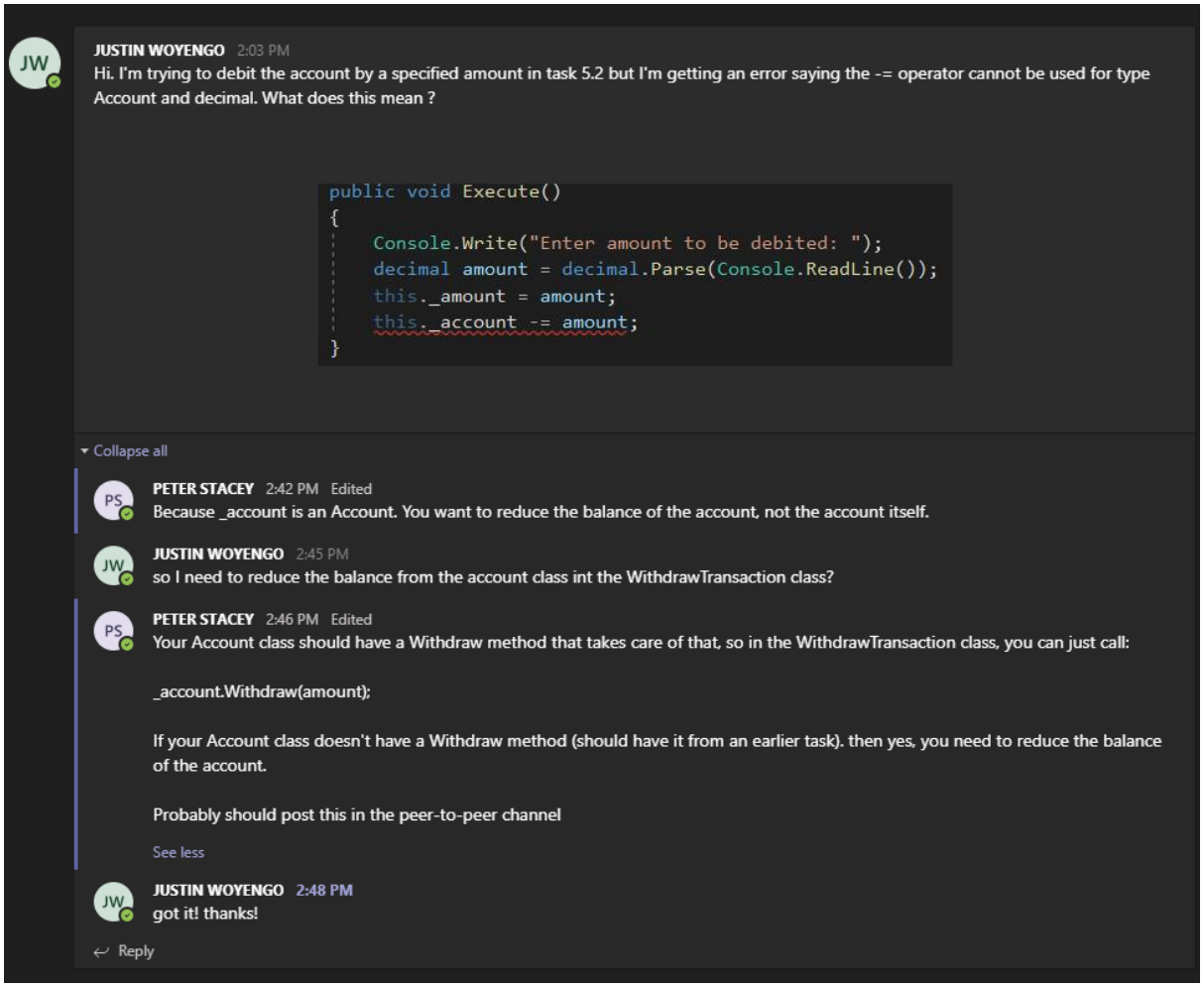


Figure 21: Example of assistance provided in the Teams Peer-to-Peer channel

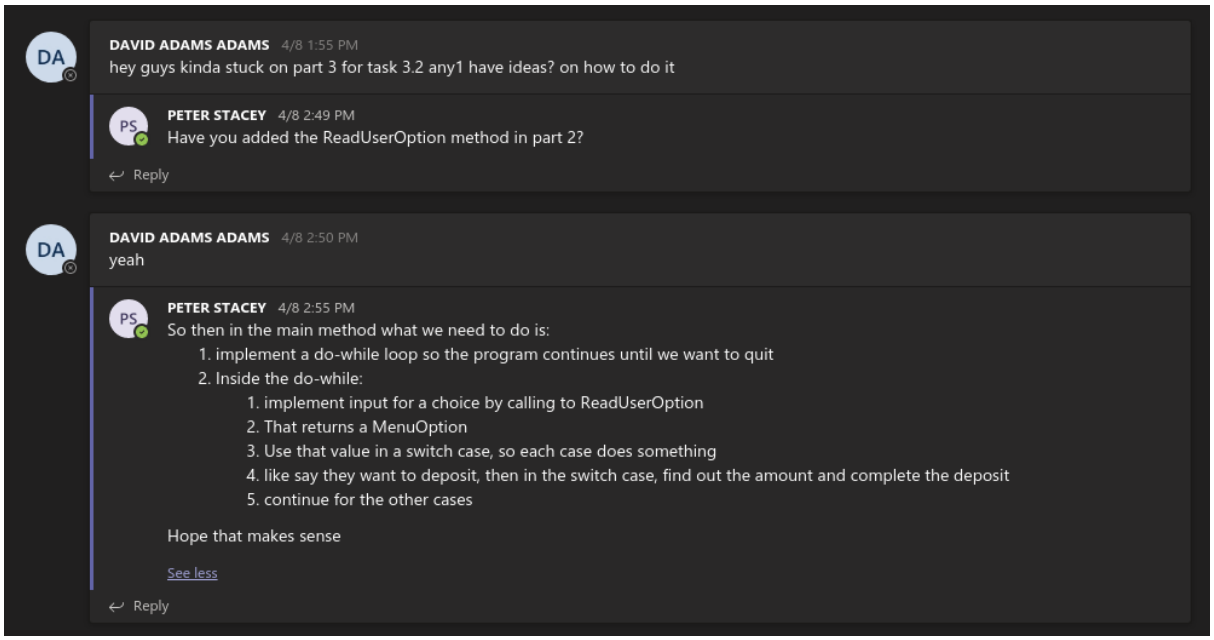


Figure 22: Example of assistance provided in the Teams Peer-to-Peer channel

For the whole Trimester, I have constantly and consistently engaged in assisting other students on the Forums, Discord and in Teams; and in engaging is discussion where my own view is changed by the valuable input of others.

In addition, throughout the semester, as I've created many graphs and images to assist me in explaining my solution in my videos, I've also shared those. Some examples include:

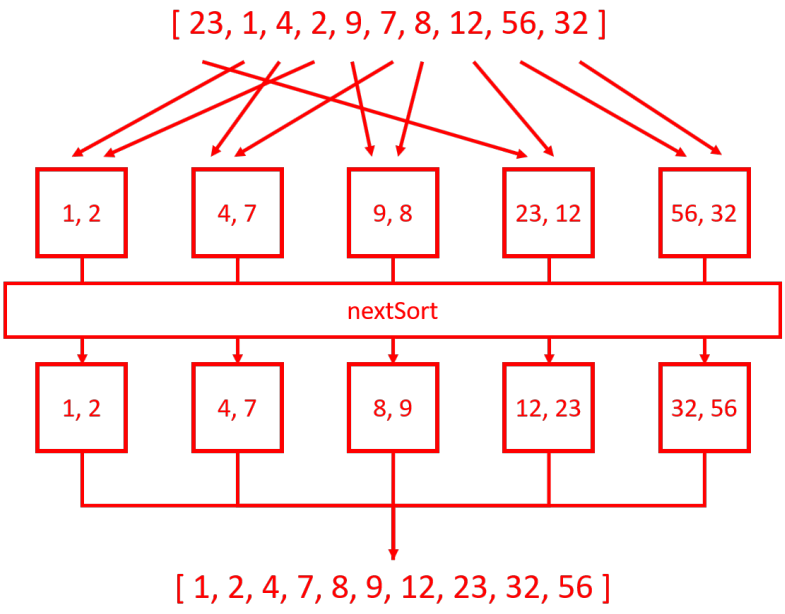


Figure 23: Example graphic that has been used to help explain the initial sorting process for the bucket sort task

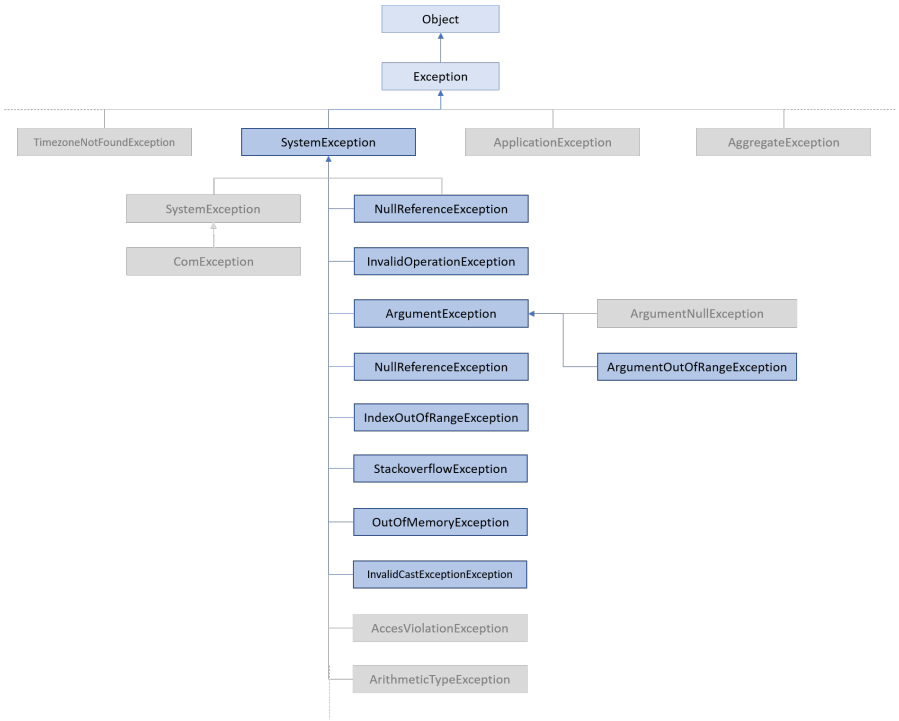


Figure 24: Example graphic that has been used to explain how the exceptions we covered in Task 4.1, relate to each other and to the broader exceptions provided by C#