## Contents

1 Preface 3

2 Introduction 5  
2.1 What is programming ................................................................. 7  
2.2 Goal of this book ........................................................................ 8

3 Problem Solving 9  
3.1 Steps in Problem Solving ............................................................. 11  
3.2 Example Problems ..................................................................... 13  
3.3 Top Down Design ....................................................................... 15  
3.4 Flow-Charts ................................................................................. 16  
3.5 Pseudocode ................................................................................ 17  
3.6 Computer Problem Solving .......................................................... 18

4 Structured Problem Solving 19  
4.1 Top Down Design in Programming ............................................ 21  
4.2 Variables .................................................................................... 22  
4.3 Constants ................................................................................... 24  
4.4 Assignment Statement ............................................................... 25  
4.5 Scope of Variables ...................................................................... 26  
4.6 Sequence .................................................................................... 28  
4.7 Selection ..................................................................................... 33  
4.8 Repetition ................................................................................... 52

5 Procedures and Functions 53  
5.1 What is programming ................................................................. 53  
5.2 How to contribute to Toga ............................................................ 54

6 Holding Data 61  
6.1 What is programming ................................................................. 62  
6.2 How to contribute to Toga ............................................................ 62

7 Representing and Sorting Data 71  
7.1 What is programming ................................................................. 71  
7.2 How to contribute to Toga ............................................................ 72

8 Using OOP 81
The goal of this book is to take students from the point of never having done any formal programming and lead them first through a structured method of problem solving (Input-Process-Output and Top-Down design), then into basic Structured Programming and then into the early basics of Object Oriented Programming (or OOP). If this book is used to teach a high school course in computer programming, there are likely many other learning outcomes that students are required to do that are not presented in this book. The focus of this book is strictly on solving problems with computer programming.

A PDF version of this textbook can be downloaded here, for offline reading.
CHAPTER 1

Preface

I'M READY WITH THE IMPLEMENTATION!

GREAT! NOW YOU CAN START WITH THE DOCUMENTATION

I GUESS I'VE FORGOTTEN TO IMPLEMENT SOME COOL FEATURES

GEEKS LOVE TO SAVE THE BEST FOR THE VERY END
It should be remembered that the focus of this book is to teach students how to program, not to just teach them a programming language. To do this the focus is on “Problem Solving”, using a computer program as a problem solving aid. Programming languages change over time and come and go but a good foundation of programming concepts and how to solve a problem will allow anyone to get over the syntax of a new programming language.

This book does not include any instructions on how to load, use, create GUIs or any other housekeeping of any particular language. There are many other resources that can aid both students and teachers alike for this.

Within the text you will see words or groups of words that are hyperlinked to Wikipedia. The point of linking to Wikipedia is to give additional information about a topic if the reader is unsure about the concept. Please note that I do not have control over what is placed on Wikipedia and although it seemed useful and correct when I looked at the link, these pages are changing all the time. Despite this, the information is usually correct and can be very helpful.
CHAPTER 2

Introduction
Problems have been around for as long as people have been around. The process of solving a problem is not something new. Using a computer to aid in solving a problem is new. Modern electronic computers have only been around since the Second World War (1939-1945), which might seem like a long time ago to you but in the history of the human race it is a very short time. The purpose of this book is to help you learn to structure your problem solving method, so that you can consistently develop a verifiable solution that will solve a problem and in the process, use the computer to help you more easily and quickly solve that problem.

2.1 What is programming

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart computers are. Actually, computers are not very smart at all! A computer, broken down into its most basic form is nothing more than a bunch of tiny electronic switches, called transistors, that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you both usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quite difficult, easy to make mistakes in and tedious. People quickly realized they needed a better way. Assemble languages were created that used very simple instructions (like DCR C ; which decreases C counter by one). This assemble language is then run through a program that converts it to machine code. Eventually people wanted an even easier to understand language. To help people talk to a computer a high-level programming language is normally used, that is then translated into machine language so that the computer can understand what to do. This high-level language comes in many different variations and is normally just called a programming language and you have probably already heard of some of them (Java, C++, Python, …). Just like there are many different languages that people speak around the world (English, French, Spanish, …), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs these actions.
2.2 Goal of this book

The goal of this book is to make you a “good” programmer. Despite the fact that a normal high school semester courses run for about 90 days, you will not become an expert programmer in just one semester. It has been said that it takes around 10,000 hours to really become proficient at anything and programming is no different. By the end of this book you will be very much on your way and have a good foundation in the skills you will need. The important thing to remember is that the point is not to teach you a specific programming language, since programming languages come and go and change over time. This is just like real languages.

Although it is the official language of the Catholic Church, not too many people go around on the streets and speak Latin to their friends. Many years ago, it might have been common but not today. The tools you will learn from this book are good programming techniques. These tools will be useful no matter what programming language you are using. Just like in the real world, you cannot be called a “linguist” if you only know one language. The same thing is true for a programmer; knowing more than one language is essential. The fortunate thing is that if you know how to program and know one programming language, picking up a second one is much easier. The cornerstone of being a good programmer is to be able to solve problems in a logical and systematic way and hopefully have fun in the process.
CHAPTER 3

Problem Solving

I DON'T GET YOUR CODE. WHAT ARE THESE LINES FOR?

I HAVE NO IDEA. BUT IT DOES NOT WORK WITHOUT THEM

THE ART OF PROGRAMMING - PART 2: KISS
As previously mentioned problems have been around forever. The use of a computer to help in solving problems is new but computers do not solve problems, people still solve problems. Computers can be used to aid in solving problems but they are just a tool. People have been creating tools to help them solve problems for 1,000’s of years. The key to remember is that a computer is just a tool, just like a hammer is a tool to help people put nails in a board.

My father is a joiner by trade but worked in construction, building houses when he first immigrated to Canada. He has often told me stories of the “good old days” when they built houses completely with hand tools (using no electricity). They use to have competitions to see who could be the most accurate on estimating the length of a board, by cutting it first and then measuring it. They were usually within less than 1⁄2 inch (half the width of your thumb on a board 8 or 10 feet long!). It use to take dozens of men months to build a house this way. Then power tools were developed (electric drills, power saws, nailing guns, . . . ). Now a house can be built by a fraction of the men it use to take, in a fraction of the time. Power tools have revolutionized the housing industry.

Computers have also revolutionized many of the ways people solve problems, as compared to the past. The first modern electronic computer the ENIAC was built to calculate tables for firing artillery shells. (Some, and maybe correctly, argue that the first electronic computer was actually the Colossus). The ENIAC was developed because it took too long and there were too many mistakes when people were doing the mathematics to calculate the firing tables by hand. The ENIAC could do the calculations in 30 seconds that it would have taken one person 20 hours to do! Today the same book of firing tables could be produced in a modern computer in a few seconds!

### 3.1 Steps in Problem Solving

There are many ways to solve a problem but having a process to follow can help make problem solving easier. If you do not think through a problem logically, then you end up just going around in circles and never solving it. The following is just one of many Six Step Problem Solving Systems, which can be used to solve any type of problem, not just ones that will be solved on a computer. The good thing is that the system translates nicely to computer problems, which is very useful, since the focus of the book is to solve problems on a computer.

The six steps in this system are:

1. What is the problem
2. Make a model
3. Analyze the model
4. Find the solution
5. Check the solution
6. Document the solution

#### 3.1.1 What is the Problem

Before you can solve a problem correctly, you have to ensure that you understand the problem thoroughly. Many times you will have to go back to the source of the problem and confirm information or ask additional questions. You might have to have them restate the problem so that it is very clear what they are asking. Here are things to remember:

- What am I trying to find?
- What do I know / don’t know?
- State the problem in your own words.
- Get them to restate the problem.
3.1.2 Make a Model

Making a model of a problem is a great way to see what is really going on and to lead you to a solution. It can show you patterns or you might recognize the problem from before. The model might be a drawing, picture, chart, a physical 3D scale model, or something else. Most “good” problems are too complex to be solved simply; they need to be broken down into smaller pieces, solve each of the smaller pieces and then bring all the small solutions back together to solve the original problem. Here are things to remember:

- Draw or create a model.
- Break the problem down into pieces.
- Is there a pattern?
- Have you seen something similar?

3.1.3 Analyze the Model

Once you have broken the problem down into more manageable pieces and made a model of the problem or the pieces of the problem, the next step is to understand what is really going on. If you do not become an expert at the problem, you might miss an important aspect. It is always a good idea to go back, not to the person that asked the question but the person that will be using the solution, to get information from them. Each piece might have a pattern that can be followed. You might have seen a solution for one of the pieces before. Here are things to remember:

- Ensure the model does what you think it does.
- Look for patterns you have seen before.
- Go back to the user to get more information.

3.1.4 Find the Solution

The hard part is now to find a solution. Hopefully you are well on your way by doing the above three steps. Can you find a pattern? Do you know the solution to one of the pieces? Can you find the solution somewhere (internet!)? In the world of programming there are books called, Patterns and Practices, that are full of common problems and their solutions. Once you have all your solutions, the next step is to bring them all together to a final overall solution to the original problem. Here are things to remember:

- Find a solution to each piece of the problem.
- Find other people’s solutions to similar problems.
- Make sure all the pieces fit back together.

3.1.5 Check the Solution

You now (hopefully) have a solution to the original problem that you are pretty sure works. The next step is to ensure it actually solves exactly what the problem was. You might need to go through, step by step, to ensure it works. You might need to work through the solution and confirm the answer you get is correct. You might have to work through the solution several times and ensure you always get the same (or similar) answer. Here are things to remember:

- Is the answer reasonable?
- Work through the solution and check for errors.
- Go through the solution several times and compare results.
3.1.6 Document the Solution

So you have come up with a brilliant solution to a problem. If you do not share the solution with anyone, what was the point? You have to ensure that you answer is verifiable and reproducible, so anyone can use it. Here are things to remember:

- Document what the problem and solution is.
- Ensure anyone can follow your steps.

3.2 Example Problems

Here are some examples of problems and a possible solution using the six step method. Note that these problems are not necessarily problems that you would use a computer to help you solve. This is an important thing to remember, not all problems should be solved with a computer. Although this book is about solving problems using a computer, there are many problems that are better solved not using a computer and that is perfectly ok. Sometimes your job as a programmer might be to identify that you should not be solving the problem on a computer. Latter on these six steps will be translated into six step used to solve problems that a computer program will be used to help solve.

3.2.1 Folding Paper

How small can a piece of paper be made?

1. What is the problem?

The wording in this problem is a little vague and could be confusing. What is meant by “made”? Is it cutting, burning or shredding? You should go back to the source of the question to find out. You might find out that the real question is, “How many times can a piece of paper be folded?”

2. Make a model.

For this problem our model will just be our actual physical piece of paper. It is not always possible to make a model using that “real” item. What would happen if the question was to fold paper made of gold, can you afford gold leaf paper? You might have to use a suitable substitute.

3. Analyze the model.

You should check to make sure that your model will be correct. What is your size of paper? Would that not make a difference in how many times you can fold it? Maybe not? Once again you might have to go back to the source of the question and get more information. Maybe the real question is, “How many times can a piece of 8½”x 11” paper be folded?”

4. Find the solution.

In this case to find the solution we will take our piece of 8½”x 11” piece of paper and keep folding it until we cannot longer do it. Should we try more than once?
5. Check the solution.

It is always important to check your solution to see if it can be reproduced with accuracy. Maybe you could get someone else to fold a piece of paper and see how many times they can do it. Is it different from your answer? Maybe they have some special technique?


Now that you have proven that a piece of 8½” x 11” can only be folded X number of times (where X is your answer), the next step is to document the solution so that other people can benefit from your analysis and can reproduce your experiment.

So what did you get as your answer? When you got someone else to do the experiment, did they get a higher number than you? Here are a few web links to also look at:

- Folding paper to the moon
- Can you really only fold a piece of paper 7 times

3.2.2 The Salmon Swimming

A salmon swims 3 km upstream and the current brings her back 2 km each day. How long does it take her to swim 100 km?

1) What is the problem?

The first thing that should be asked is, does the fish swim up stream 3 km during the day and fall back 2 km at night or does she swim continuously all day and would go 3 km if it was not for the 2 km current so she only ever gets 1 km? Once again you will have to go back to the source of the problem to find out. We will say she swims 3 km during the day and then drifts back 2 km at night.

2) Make a model.

For this problem our model will be a picture of a piece of what is happening.
3) Analyze the model.

You should check to make sure that your model will be correct. We will follow what is going on in a table:

<table>
<thead>
<tr>
<th>Day Number</th>
<th>Distance at end of Swim</th>
<th>Distance after being moved back</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>#2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>#3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>#4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>#5</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>#96</td>
<td>98</td>
<td>96</td>
</tr>
<tr>
<td>#97</td>
<td>99</td>
<td>97</td>
</tr>
<tr>
<td>#98</td>
<td>100</td>
<td>98</td>
</tr>
<tr>
<td>#99</td>
<td>101</td>
<td>99</td>
</tr>
<tr>
<td>#100</td>
<td>102</td>
<td>100</td>
</tr>
</tbody>
</table>

4) Find the solution

In this case to find the solution we need to know how many days it took to get to 100 km. Your first reaction might be 100 days BUT if you look at the table on day 98 after the fish swam the 3 km, it is actually at 100 km mark, so that is the answer, 98 days not 100 days.

5) Check the solution

It is always important to check your solution. In this case since our solution came from the table, check to make sure there is no error in the table. It might be a good idea to let it sit for a few days and then come back to look at it or get somebody else to look at your solution and see if it is correct.

6) Document the solution

Now that you have proven that the answer is 98 days, make sure you document it, so that someone else does not have to figure it out but can just refer to your answer and check your solution.

Remember not to always go with your gut instinct and thing because it is following a patter you know the answer instantly without following through with the steps. Do all six steps and always check your answer.

3.3 Top Down Design

Top-down design is a method of breaking a problem down into smaller, less complex pieces from the initial overall problem. Most “good” problems are too complex to solve in just one step, so we divide the problem up into smaller manageable pieces, solve each one of them and then bring everything back together again. The process of making the steps more and more specific in top-down design is called stepwise refinement.

As mentioned before my father use to work in construction building houses. If someone gave you a piece of land and told you to, “Build me a house” you would not immediately go over and start nailing 2 x 4’s together. Building a house is a very complex adventure, not to mention there are many rules, codes and laws that must be followed. To build a house you could break the project up into smaller jobs, plan and do each one of these jobs (in the correct order) and in the end you would have a house. You will notice in the following top-down design diagram that some jobs get broken down several times, until they are a manageable size.
We will be using top-down design (and top-down design diagrams, like the one above) to help us understand a problem and all its components.

### 3.4 Flow-Charts

Some people think that there is no need to do flow-charts before writing a program; that you can just go to the computer and start writing code. Any “interesting” computer problem is so complex though, that without planning you would just end up spinning your wheels and have to throw out most of your code. In our six step problem solving model, the second step was to create a model and flow-charts are an excellent tool to make a model of what happens in most computer problems. Remember that a computer program is just a set of steps that the computer follows to solve a problem. A flow-chart is just a pictorial representation of a sequence of steps.

A flow-chart is a set of different shapes that each represent a certain type of action. These shapes are connected together with arrows so that you can see the flow of logic. The shapes are:

- **Start / Stop**: Every flow-chart must start and end with one of these.
- **Operation**: Doing something, also known as a process.
- **Decision**: Answering a yes or no question. There are usually 2 branches and you take 1 of them.
- **Information**: Getting data from the user or presenting data back to a user.
- **Connectors**: Used if the flow-chart is longer than one page.
- **Is Assigned**: Used to represent a value (or variable) being placed into another variable.

Here is an example of a flowchart for a none-computer based problem:
3.5 Pseudocode

Pseudocode is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

We will be translating our flow-chart that we create from the previous step into pseudocode to aid us in writing our computer program. In general the vocabulary used in pseudocode should be the vocabulary of the problem, not written in “computer speak”. A non-computer scientist should be able to read and understand what is going on. The pseudocode is a narrative for someone who knows the problem and is trying to learn how the solution is organized. Several keywords are often used to indicate common input, output, and processing operations.

Input: READ, OBTAIN, GET
Output: PRINT, DISPLAY, SHOW
Compute: COMPUTE, CALCULATE, DETERMINE
Initialize: SET, INIT
Add one: INCREMENT, BUMP

Here is the lamp example from the flow-chart section as pseudocode:

GET lamp does not work
IF (lamp plugged in == yes) THEN
    plug in lamp
ELSE
    IF (bulb burnt out == yes) THEN
        replace bulb
    ELSE
        repair lamp
    ENDIF
ENDIF

3.6 Computer Problem Solving

The initial goal of learning problem solving was to help us to solve problems that a computer program could be used for to help solve. The initial six step problem solving model that was presented can be used to help solve any type of problem. If we know that we are going to use a computer program to help solve the problem, the six steps can be translated into six steps that are more tailored for computer programming problems. They are the same basic six steps; they are just more focused on computer programming problems.

The table below shows the initial six steps that we have been using for generic problems. They are then translated into the six steps we will be following for computer programming problems.

<table>
<thead>
<tr>
<th></th>
<th>Generic Step</th>
<th>Computer Programming Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>What is the problem</td>
<td>Top-Down Chart</td>
</tr>
<tr>
<td>#2</td>
<td>Make a model</td>
<td>Flow Chart &amp; StoryBoard</td>
</tr>
<tr>
<td>#3</td>
<td>Analyze the model</td>
<td>Pseudocode</td>
</tr>
<tr>
<td>#4</td>
<td>Find the solution</td>
<td>Actually type in the code</td>
</tr>
<tr>
<td>#5</td>
<td>Check the solution</td>
<td>Style check &amp; test for errors</td>
</tr>
<tr>
<td>#6</td>
<td>Document the solution</td>
<td>Document the code – comments &amp; GitHub</td>
</tr>
</tbody>
</table>

These six steps are here to help you. Most people have the urge when they are given a programming assignment to just go to the computers and start coding. This is NOT a good idea. If you have not thought through the problem first and worked through these steps, you will make too many mistakes, get lost and waste too much time.
CHAPTER 4

Structured Problem Solving

Just because you named a class
SorryJimButUntypedLanguagesAreJustWrongPeriod
does not completely convince me!

ONLY THE CODE TELLS THE TRUTH
Teachers often hear students complain that they “...don’t know where to begin.” when they are expected to solve what seem to be straightforward problems. Obviously they are not straightforward to the students for reasons that we are now beginning to understand, knowing where to begin is usually the hardest part. Structured problem solving is a set of tools to help you guide yourself though the process of solving computer related problems that seem to be impossible to solve.

### 4.1 Top Down Design in Programming

As we have seen above, top-down design is a process of breaking a complicated problem down into simpler steps that actually can be solved. In programming one of the easiest models to follow for beginner programmers is the Input-Process-Output Model (IPO Model). You are most likely already familiar with this model, even if you do not realize you are using it. In life there are many examples of getting information, doing something with it and then returning the result. The most common example is probable the problems you do in math class every day. You get information, do calculations and then return the answer. We are going to use this model to help us solve our computer problems. The first step in our top-down design will always be to break the problem up into input-process-output.

![Diagram of IPO Model]

#### 4.1.1 Sometimes I need to Store Something

As mention above, “Input-Process-Output” is a very useful model to help solve problems. It is kind of misleading though because one very important point is missing, storage. What actually really happens is that we get information and then place it somewhere (write it down in a math problem or use a variable like x to hold some important number), we then process it (do some calculations), we get the answer and then give it back. We need storage as a temporary location to keep information. In really long problems many pieces of information might be kept, we might even have information that we just need to keep for some intermediate step. The real model looks more like:
In computer science programmers call these temporary storage locations variables.

### 4.2 Variables

A **variable** is a name that we use to represent a value that we want the computer to store in its memory for us. When solving problems, we often need to hold some valuable pieces of information we will use in our solution. From the “Input-Process-Output” model above, an example of variables we would be placing in storage is the input from the user. The pieces of information could be people’s names, important numbers or a total in a purchase. We use a name that means something to us (and hopefully the people that come after us and read our source code, commonly referred to as just code) so that we know right away when we read our code what it is holding. In math class you might be familiar with equations that involve variables like “x or y”. We would not name a variable x, if it is holding the number of students in class, we might call it numberOfStudents or number_of_students (depending on the style guide for a particular language). This has much more meaning to us and other people that also look at our code. Some people are still tempted to use a variable name like “x”, because they say it will save space. But once our code is converted to machine language, it does not matter what you called your variable it will be converted to something that takes the same space. So be a “nice” programmer and always use meaningful variable names.

Depending on the type of programming language you are using, you might need to declare your variable (warn the computer we will be using a variable before we use it) before you use it in a program. Some programming languages do not enforce this rule, other do. Since you are new to programming, it is really good programming style to always declare a variable before using it, if that is possible. The process of declaring a variable is called a declaration statement.

In most programming languages you will have an **identifier**, which is the name you are giving your variable (ex. numberOfStudents or number_of_students) and the **data type**. The identifier will be the way that you refer to this piece of information latter in your program. The data type determines what kind of data can be stored in the variable, like a name, a number or a date. In the computer world you will come across data types like **integer**, **character**, **string** & **boolean**. It is always important that you ensure you select the right kind of data type for the particular data that it is going to hold. You would not use an integer to hold your name and vice versa, you would not use a string to hold your age. The following is a table of some common built in types from several different languages that you might use:
<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Type Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>True or False (1 or 0)</td>
</tr>
<tr>
<td>Unsigned Byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>Signed Byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>Character</td>
<td>A single character (like A or % or @)</td>
</tr>
<tr>
<td>String</td>
<td>Variable length number of characters</td>
</tr>
</tbody>
</table>

Variable declaration usually should be grouped at the beginning of a section of code (sub, procedure, function, method...), after the initial comments. A blank line follows the declaration and separates the declaration from the rest of your code. This makes it easy to see where the declaration starts and ends. Ensuring that your code is easy to read and understand is as important in computer science as it is in English. It is important to remember that your code has two audiences, the computer that needs to compile or interpret it so that the computer can run your program and even more important, you and everyone else that looks at your source code that are trying to figure out how your program works. Here are some examples of declaring a variable:

C++

```cpp
// variable definition
bool isCurrent = true;
int age = 32;
float area;
string someWords = "Hello, World!";
```

Go

```go
// variable definition
var isCurrent bool = true
var age int = 32
var float32 area
var someWords string = "Hello, World!"
```

Java

```java
// variable definition
boolean isCurrent = true;
int age = 32;
float area;
String someWords = "Hello, World!"
```

JavaScript

```javascript
// variable definition
let isCurrent = true;
let age = 32;
float area;
let someWords = "Hello, World!"
```

Python3

```python
// variable definition
is_current = True
age = 32
area = 32.45
some_words = "Hello, World!"
```

Ruby

```ruby
```

4.2. Variables
# variable definition
is_current = true
age = 32
area = 32.45
some_words = "Hello, World!"

Swift

```swift
// variable definition
var isCurrent: Bool = true
var age: Int = 32
var area: Float = 32.45
var someWords: String = "Hello, World!"
```

## 4.3 Constants

There are times in a computer program where you have a value that you need the computer to save that does not change or changes very rarely. Some examples are the value of \( \pi \) (3.14...) and the HST (the HST in Ontario is currently 13%, but it has changed once). When you have values like these, you do not want them to be saved as a variable (since they do not vary or change) but you place them in a constant.

Constants just like variables hold a particular value in memory for a programmer that will be used in the program. The difference is that the computer will not let the programmer change the value of a constant during the running of the program. This prevents errors from happening if the programmer accidently tries to change the value of a constant. It should always be declared, just as a variable is declared to warn the computer and people reading your code that it exists. Constants should be declared right before variables, so that they are prominent and easy to notice. Here are some examples of declaring constants:

**C++**

```cpp
// constant definition
const int NUMBER_OF_LIVES = 3;
const double HST = 0.13;
```

**Go**

```go
// constant definition
const int NUMBER_OF_LIVES = 3
const float32 HST = 0.13
```

**Java**

```java
// constant definition
private static int NUMBER_OF_LIVES = 3;
p~ivate static double HST = 0.13;
```

**JavaScript**

```javascript
// constant definition
const NUMBER_OF_LIVES = 3;
const HST = 0.13;
```

**Python3**

```python
// constant definition
const NUMBER_OF_LIVES = 3;
const HST = 0.13;
```
# constant definition
# Note: in python usually constants are declared in a separate module
# In this example the module is called "constants"
NUMBER_OF_LIVES = 3
HST = 0.13

# then in main program
constants.NUMBER_OF_LIVES
constants.HST

Ruby

// constant definition
NUMBER_OF_LIVES = 3
HST = 0.13

Swift

// constant definition
let numberOfLives = 3
let HST = 0.13

4.4 Assignment Statement

Programs can have many variables. Usually information is gathered from the user, stored in variable, processed with other variables, saved back to one/some variable(s) and then returned to the user. Variables are changed or initially assigned a value by the use of an assignment statement. Assignment statement are usually written in reverse from what we are use to in math class. A variable on the left side of the assignment statement will receive the value that is on the right hand side of the assignment statement. Note that different programming languages use different symbols to represent the assignment statement (for example in Alpha it is"←", in Pascal it is":="). No matter what the symbol is, you always read it as, “is assigned”. This is particularly important in languages like Visual Basic, where the assignment symbol is an equal sign ( = ) and people are use to reading this as “is equal to”. C# also uses and equals sign ( = ) as its assignment operator. To make it even more confusing, the same symbol ( = ) is used in another context in Visual Basic and in that one it actually is an equal sign and is read as such. In C# the equals symbol is actually two equals signs next to each other (==). Here are a few examples of assignment statements:

C++

// assignment statement
numberOfStudents = numberOfStudents + 5;
width = 32;
length = 10;
areaOfRectangle = length * width;
someWords = "Hello, World!";

Go

// assignment statement
numberOfStudents = numberOfStudents + 5
width = 32
length = 10
areaOfRectangle = length * width
someWords = "Hello, World!"

Java

4.4. Assignment Statement
4.5 Scope of Variables

Where a variable is declared is important because it determines its scope. The scope refers to where it is visible or can be used within a program. Usually you would declare a variable at the beginning of a function (for example a click event on a button or menu or the “main” function). Since it is declared at the beginning of a function, it can only be used within that function. Once the flow of your program exits this function, the variable is removed from memory (actually it is just de-allocate most likely) and can no longer be used. This type of variable is referred to as a local variable. Any other function in your program can not use or refer to this variable.

What if for some reason you needed a variable to be accessible to several different functions within a single program. In this case declaring it within a single function is no good. Another option is to declare the variable at the top of the form class or module, just before any function. If this is done then any function within that program can see and use
this variable. This type of variable is called a **global variable**. Global variables should only be used when absolutely necessary; if only one function needs a variable, it should be declared within the function. This is good programming style and also saves computer memory. The following is an example where you can see variables with the same name, being used as global and local variables. Type it in and follow the variables by stepping through the program.

C++

```cpp
// Copyright (c) 2019 St. Mother Teresa HS All rights reserved.
//
// Created by: Mr. Coxall
// Created on: Sep 2019
// This program shows how local and global variables work

#include <iostream>

// global variable
int variableX = 25;

void localVariable() {
    // this shows what happens with local variables

    int variableX = 10;
    int variableY = 30;
    int variableZ = variableX + variableY;
    std::cout << "Local variableX, variableY, variableZ: " << variableX
               << " + " << variableY << " = " << variableZ << std::endl;
}

void globalVariable() {
    // this shows what happens with global variables

    variableX = variableX + 1;
    int variableY = 30;
    int variableZ = variableX + variableY;
    std::cout << "Local variableX, variableY, variableZ: " << variableX
               << " + " << variableY << " = " << variableZ << std::endl;
}

int main() {
    // this function calls local and global

    localVariable();
    globalVariable();
}
```

Go

```go
// This program shows how local and global variables work
```

Java

```java
// This program shows how local and global variables work
```

JavaScript

```javascript
// This program shows how local and global variables work
```

Python3

```python
// This program shows how local and global variables work
```
#!/usr/bin/env python3

# Created by: Mr. Coxall
# Created on: Sept 2019
# This program shows how local and global variables work

# global variable
variable_X = 25

def local_variable():
    # this shows what happens with local variables
    variable_X = 10
    variable_Y = 30
    variable_Z = variable_X + variable_Y
    print("Local variable_X, variable_Y, variable_Z: {0} + {1} = {2}".format(variable_X, variable_Y, variable_Z))

def global_variable():
    # this shows what happens with global variables
    global variable_X
    variable_X = variable_X + 1
    variable_Y = 30
    variable_Z = variable_X + variable_Y
    print("Global variable_X, variable_Y, variable_Z: {0} + {1} = {2}".format(variable_X, variable_Y, variable_Z))

def main():
    # this function shows how local and global variables work
    local_variable()
    global_variable()

if __name__ == "__main__":
    main()

Ruby

# This program shows how local and global variables work

Swift

// This program shows how local and global variables work

4.6 Sequence

Teachers often hear students complain that they “...don’t know where to begin.” when they are expected to solve what seem to be straightforward problems. Obviously they are not straightforward to the students for reasons that we are now beginning to understand, knowing where to begin is usually the hardest part. Structured problem solving is a set of tools to help you guide yourself through the process of solving computer-related problems that seem to be impossible
4.6.1 Example Sequence Problem

The following is an example problem that has been solved using the six step computer based problem solving method mentioned above. The goal is to show you how a sequential program works and also to show how the six steps are used to develop a program from a given problem.

Sequence Programming Example

Here is the problem: Write a program that will allow the user to enter a subtotal and the program will calculate the final total including tax.

1. Top-Down Chart

The first thing we need to find out is how to calculate tax in your particular province. If the program is to be used anywhere in Canada, it could get really confusing since each province has a different tax rate and some even calculate taxes differently than other provinces. To help simplify the problem, we are just going to do it for Ontario. The Harmonized Sale Tax (HST) in Ontario is currently 13%. The following is a top-down design breaking the problem up into smaller manageable pieces:

2. Flow Chart

The next step is to convert the top-down design into a flowchart. To help create the flow chart, use the bottom boxes of each of the arms, in order from left to right from the top-down design. This would mean Get subtotal, Calculate HST, Add 2 values together and Put final total. Remember that every flowchart has the “Start” oval at the top and the “Stop” oval at the bottom. This is so people know where to begin and end. The arrows are used so people can follow the flow. The words in the flow charts are not full sentences but simplified pieces of code. Ensure that you include any formulas and use the variable names that you will plan on using in your code.
3. **Pseudo-code**

Pseudo-code converts your flowchart into something that more resembles the final code you will write. Once again though it is not code (hence the name pseudo-code), so it is generically written so that it can be translated into any language. It should be understood by anyone that can write a computer program, not just people that use the same programming language that you do. The first word on each line should be a verb (an action word), since you want the computer to do something for you. By convention the first verb is also in all caps (capital letters). Here is the pseudo-code for the problem:

GET subtotal from user
CALCULATE tax ← subTotal * HST
CALCULATE total ← subTotal + tax
SHOW taxes back to user
SHOW total back to user

4. **Code**

Once you have the pseudo-code done, the hardest part of solving the problem should be finished. Now you just convert your pseudo-code into the specific programming language you have chosen:

**C++**

```cpp
#include <iostream>
#include <iomanip>

int main() {
    // this function calculates total from subtotal and tax
    const float HST = 0.13;
    float tax;
    float subTotal;
    float total;

    // input
    std::cout << "Enter the subtotal: ";
    std::cin >> subTotal;

    // process
    tax = + subTotal * HST;
    total = subTotal + tax;

    // output
    std::cout << "The HST is: ", std::fixed << std::setprecision(2) << std::setfill('0') << tax << ", and the total cost is: ", std::fixed << std::setprecision(2) << std::setfill('0') << total << "." << std::endl;
}
```

**Go**

4.6. **Sequence**
This program calculates total from subtotal and tax

Java

// This program calculates total from subtotal and tax

JavaScript

// This program shows how local and global variables work

Python3

#!/usr/bin/env python3
# Created by: Mr. Coxall
# Created on: Sep 2019
# This program calculates total from subtotal and tax

import constants

def main():
    # this function calculates total from subtotal and tax

    # input
    sub_total = float(input("Enter the subtotal: ")

    # process
    tax = sub_total * constants.HST
    total = sub_total + tax

    # output
    print("")
    print("The HST is ${0:,.2f}, and the total cost is: ${1:,.2f}"
        .format(tax, total))

if __name__ == "__main__":
    main()

Ruby

# This program calculates total from subtotal and tax

Swift

This program calculates total from subtotal and tax

5. Debug

It is hard to show the debugging step, since I ensured that the program above worked correctly before I pasted it into the page. When programmers write code it is extremely unlikely that it will work right away the first time. This is why the development environment has tools to help the programmer fix simple mistakes. The two main kinds of mistakes are syntax errors and logical errors.
In modern languages high level languages and IDEs, syntax errors are usually easy to see and fix. A syntax error is a piece of code that the compiler or interpreter does not understand. It would be like speaking to you and one of the sentences did not make any sense to you. A modern IDE will nicely place a squiggly line under the code (or some other way of showing you) it does not understand, so that you can fix the problem. A logical error is a lot harder to find. This is a problem with the way you solved the problem. The code will still compile or be interpreted and run but the program will give you the wrong answer (or maybe just the wrong answer some times!). There is not easy way to solve these problems than to step through your code one line at a time.

6. Document the code

This is hopefully not done just at the end of your programming but as you write your code. All the same it is good practice to go over you code at the end to ensure that someone else looking at it will understand what is going on. In the above example you can see that there is a comment at the start of the program and in the function as well. Also I have used a naming convention that is hopefully easy to understand what the variables are holding. In addition, the value of the HST is places in a constants, since they only change very infrequently.

The above six steps are an example of how you should go about solving a compute based problem. Ensure when you are given a problem, you do not make the mistake that most people do and go directly to the computer and start coding. If you have not first been able to break the problem down into smaller pieces and solve the problem on paper, going to the computer and starting to code will not help you. You will just end up going in circles, wasting time, creating bad code and getting nowhere. Programming is just problem solving on a computer but you have to have solved the problem before you actually get to the computer to help you get the answer.

4.7 Selection

If computer programs could only do just a linear sequence of steps and nothing else, they would not be very useful. One additional thing that computers are very good at doing is conditional control or making a decision, as long as you provide all the parts it needs to figure out the decision. This gives a computer program the ability to make a decision, based on a boolean expression.

4.7.1 Boolean Expressions

A boolean expression is an expression (or equation) that has two possible values or outcomes, either a “True” or a “False” (or you can look at it as a 1 or a 0). An example of a boolean expression is “A Volkswagen beetle is a car.” Clearly this statement is true, so that is how it is evaluated to “True”. You could also have “a frog is a mammal”. This statement is not correct, so it evaluates to “False”. You could also have mathematical expression, “$3+2 = 6$”. This equation (and yes it is an equations so you read the “=” as “is equal”) is not correct, so it is also evaluated to “False”. There are many other types of expressions that can be checked, besides equality. You could have “$3+2 \leq 6$”. This time the inequality is evaluated as “True”, since 5 is $\leq$ to 6. Some of the most common operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>== or eq</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>
4.7.2 If...Then

The If...Then structure is a conditional statement, or sometimes referred to as a decision structure. It is used to perform a section of code if and only if the condition is true. The condition is checked by using a Boolean statement. If the condition is not true (meaning false), then the section of code is not performed it is just passed over. The form of an If...Then statement is:

\[
\text{IF (boolean expression) THEN} \\
\quad \text{Statements to be performed} \\
\text{ENDIF}
\]

The indentation (usually 4 spaces, NOT A TAB) used in the If...Then statement is a coding convention used in almost every language. It is there to make the statement easier to read. It has no effect on how the code works and could be ignored; however, it is REALLLY BAD programming style not to have it. You will also notice that some programming languages like to place the Boolean expression in brackets, while others do not. It is just style, but you should follow the language’s style.

Here is a problem that can be solved using an If...Then statement. I have a class that can only hold 30 students because that is how many chairs I have. As the user to enter a number of students and tell me if I have too many students for chairs.

You top-down design will have a decision logic in it. You do not use a diamond in a top-down design, you still only use rectangles. Here is what a top-down design might look like for this problem:

![Diagram](image)

Remember from the section on flowcharts, the diamond shape represented decisions. The If...Then statement is the translation of a decision in a flowchart to code. The above examples would look like the following in a flowchart:
You will also be using If...Then statements in pseudo-code. The above problem looks like this in pseudo-code. Note that you do indent when you are inside an If...Then statement in pseudo-code. Also note that “IF”, “THEN” and “ENDIF” are all bold and caps:

GET number_of_students
IF (number_of_students > 30) THEN
    SHOW “Too many students!”
ENDIF

In the code examples below, if the variable numberOfStudents (or number_of_students) happens to be a number that is greater than 30 (say 32), the next line of code is performed (print(“Too many Students!”)). If the variable is not greater than 30 (say it is exactly 30), then the next line of code is skipped over and NOT performed.

C++

// Copyright (c) 2019 St. Mother Teresa HS All rights reserved.
//
// Created by: Mr. Coxall
// Created on: Sep 2019
// This program checks if there is over 30 students

(continues on next page)
```c++
#include <iostream>

int main() {
    // this function checks if there is over 30 students
    const int MAX_STUDENT_NUMBER = 30;
    int numberOfStudents;

    // input
    std::cout << "Enter the number of students: ";
    std::cin >> numberOfStudents;
    std::cout << "\n" << std::endl;

    // process
    if (numberOfStudents > MAX_STUDENT_NUMBER) {
        // output
        std::cout << "Too many students!\n";
    }
}
```

Go

```go
// if ... then example
if numberOfStudents > 30 {
    fmt.Println("Too many students!")
}
```

Java

```java
// if ... then example
if (numberOfStudents > 30) {
    print("Too many students!");
}
```

JavaScript

```javascript
// if ... then example
if (numberOfStudents > 30) {
    print("Too many students!");
}
```

Python3

```python
#!/usr/bin/env python3

# Created by: Mr. Coxall
# Created on: Sep 2019
# This program checks if there is over 30 students

import constants

def main():
    # this function checks if there is over 30 students
    # input
```
number_of_students = int(input("Enter the number of students: "))

# process & output
if number_of_students > 30:
    print("Too many students!")

if __name__ == "__main__":
    main()

Ruby

// if ... then example
if numberOfStudents > 30
    puts "Too many students!"

Swift

// if ... then example
if numberOfStudents > 30 {
    print("Too many students!")
}

4.7.3 If...Then...Else

In the previous section we looked at the If...Then statement that is used for making a decision. When used a section of code is either performed or not performed, depending if the boolean statement is true or not. In some situations, if the statement is false and the section of code is not performed you would like an alternative piece of code to be performed instead. In this case an optional Else statement can be used. The If...Then...Else statement (in most computer programming languages) takes the generic form of:

IF (boolean expression) THEN
    Statements to be performed
ELSE
    Alternate statements to be performed
ENDIF

An example of what this would look like in a specific programming language is:

C++

// if ... then ... else example
if (numberOfStudents == 30) {
    std::cout << "Exactly 30 students!"
} else {
    std::cout << "Not 30 students."
}

Go
In the above examples, if the variable `numberOfStudents` happens to be exactly equal to 30, the next line of code is performed (print(“Exactly 30 students!”)). If the variable is not equal to 30 (say it is 32 or 17), then the next line of code is skipped over and NOT performed but the following line of code will be performed (print(“Exactly 30 students!”)). Once again the diamond shape represented decision, even if it has a statement if it is true and a different one if it is false. The above examples would look like the following in a flow-chart:
4.7.4 If...Then... ElseIf...Else

In some problems there are not just two different outcomes but more than two. If this is the case, then a simple If...Then...Else structure will not work. In these situations an If...Then... ElseIf...Else might be used. In this type of structure there can be more than just one Boolean condition and each is checked in sequence. Once a Boolean expression is met (the value is true), then the specified section of code for that Boolean expression is executed. Once executed, all other conditions are skipped over and the flow of logic goes right down to the bottom of the structure. It is important to remember that one and only one section of code can be executed. Even if several of the Boolean conditions happen to be met, the first and only the first one will be evaluated and run.

The structure can contain many ElseIfs, as many as the programmer needs. Another optional piece of the structure is the “Else”. If none of the above boolean conditions are met, the programmer might want a section of code to be executed. If this is the case, then the code is placed in the else section. Any code in the else section is run if and only if none of the Boolean expressions were met. It should also be noted that there is no Boolean condition associated with the else. That is because it is run only if all the above boolean conditions are not met. The If...Then... ElseIf... Else statement (in most computer programming languages) takes the generic form of:

IF (boolean expression #1) THEN
    First potential statement to be performed
ELSEIF (boolean expression #2) THEN
    Second potential statement to be performed

4.7. Selection
ELSEIF (boolean expression #3) THEN
    Third potential statement to be performed
...
ELSEIF (boolean expression #n) THEN
    Nth potential statement to be performed
ELSE
    Alternate statements to be performed
ENDIF

An example of what this would look like in a specific programming language is:

C++

```
// if ... then ... elseif ... else example
if (colourOfLight == "red") {
    std::cout << "Stop!";
} else if (colourOfLight == "yellow") {
    std::cout << "Slow Down.";
} else if (colourOfLight == "green") {
    std::cout << "Go, if all clear.";
} else {
    std::cout << "No idea!";
}
```

Go

```
// if ... then ... elseif ... else example
```

Java

```
// if ... then ... elseif ... else example
```

JavaScript

```
// if ... then ... elseif ... else example
```

Python3

```
# if ... then ... elseif ... else example
if colour_of_light == "red":
    print("Stop!")
elif colour_of_light == "yellow":
    print("Slow Down.")
elif colour_of_light == "green":
    print("Go, if all clear.")
else:
    print("No idea!")
```

Ruby

```
// if ... then ... elseif ... else example
```

Swift

```
// if ... then ... elseif ... else example
```
In the above examples, if the variable `colourOfLight` is red, yellow or green than the appropriate section of code is executed. If the variable does not equal any of these, then the last statement is executed, “No idea!” The above examples would look like the following in a flow-chart:

![Flow-chart](image)

### 4.7.5 Select Case

As you have seen from the If...Elseif...Elseif...Else statement, when there are many choices, the structure can be hard to follow. Some programming languages have an alternative structure when this happens. The Select Case or Switch Case statement is also a decision structure that is sometimes preferred because code might be easier to read and understand, by people.

The Select Case structure takes a variable and then compares it to a list of expressions. The first expressions that is evaluated as “True” is executed and the remaining of the select case structure is skipped over, just like an If...ElseIf...Else statement. There are several different ways to create your expression. You can just use a value (a single digit for example), several digits, a range or having a regular expression (Is < 10). Just like the If structure, there is an optional “Else” that can be placed at the end as a catch all. If none of the expressions is evaluated to “True”, then the flow will go to the else. The general form of a Select...Case statement (in most computer programming languages) takes the generic form of:

```
SELECT (variable)
    CASE valueOne
        //statements
    CASE valueTwo
        //statements
    CASE valueThree
        //statements
    ...
    ELSE //optional
        //statements
```
An example of what this would look like in a specific programming language is:

C++

```cpp
#include <iostream>

int main() {
  // this function checks your grade
  char gradeLevel;  // a single character

  // input
  std::cout << "Enter grade mark as a single character (ex: A, B, ...): ";
  std::cin >> gradeLevel;

  // switch in C++ can not support strings, only numbers and char
  // also note you need the break in C++ or it will move to next
  // line in switch statement and might be true again
  switch (gradeLevel) {
    case 'A':
      std::cout << "Excellent!" << std::endl;
      break;
    case 'B':
      std::cout << "Really good!" << std::endl;
      break;
    case 'C':
      std::cout << "Well done" << std::endl;
      break;
    case 'D':
      std::cout << "You passed" << std::endl;
      break;
    case 'F':
      std::cout << "Better try again" << std::endl;
      break;
    default:
      std::cout << "Invalid grade" << std::endl;
  }
}
```

Go

```go
// select ... case example
```

Java

```java
// select ... case example
```

JavaScript

```javascript
// select ... case example
```

Python3

```python
# Chapter 4. Structured Problem Solving
```
# select ... case example
# python does not actually have a select ... case structure

Ruby

```
// select ... case example
```

Swift

```
// select ... case example
```

In the above examples, if the variable `colourOfLight` is red, yellow or green then the appropriate section of code is executed. If the variable does not equal any of these, then the last statement is executed, “No idea!” The above examples would look like the following in a flow-chart:
4.7.6 Try Catch

When a runtime error or exception occurs, the program will terminate and generate an error message. This is not exactly ideal for the user. What would be better is if we could catch these errors and then do what is necessary to get fix the problem. A common example is when we ask the user to enter a number, but for some reason they entered text. Ideally we would not want the program just to crash, we would want to explain to the user they entered something incorrectly.

To catch these runtime errors, a portion of code is enclosed in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored. The try statement (in python) takes the generic form of:

```
TRY
 some statement(s) to be performed
EXCEPT (type of error)
 some statement(s) to be performed
ELSE
 some statement(s) to be performed
FINALLY
 some statement(s) to be performed
END
```

You can define as many exception blocks as you want (e.g. if you want to execute a special block of code for a special kind of error). You can use the else keyword to define a block of code to be executed if no errors were raised. The finally block, if specified, will be executed regardless if the try block raises an error or not.

An example of what this would look like in a specific programming language is:

C++

```cpp
// Copyright (c) 2019 St. Mother Teresa HS All rights reserved.
//
// Created by: Mr. Coxall
// Created on: Sep 2019
// This program uses a try statement

#include <iostream>
#include <string>

int main() {
    // this function uses a try statement
    std::string integer_as_string;
    int integer_as_number;

    // input
    std::cout << "Enter a number: ";
    std::cin >> integer_as_string;

    // process & output
    try {
        integer_as_number = std::stoi(integer_as_string);
        std::cout << "That number as an integer is " << integer_as_number;
    } catch (std::invalid_argument) {
        std::cout << "That was not a valid integer";
    }
}
```

(continues on next page)
In the above examples, if you do enter in an integer, it will let you know. If you enter in a string for example, the program will not crash, but give you a warning. The above examples would look like the following in a flow-chart:
4.7.7 Compound Boolean Expressions

Just before we looked at the If . . . Then statement we looked at Boolean expressions. Boolean expressions have two and only two potential answers, either they are true or false. So far we have looked at just simple boolean expression, with just one comparison. A boolean expression can actually have more than one comparison and be quiet complex. A compound boolean expression is generated by combining more than one simple boolean expression together with a logical operator And or Or. And is used to form an expression that evaluates to True only when both operands are true. Or is used to form an expression that evaluates to true when either operand is true. Here is a truth table for each operator:

AND Truth Table

4.7. Selection
### Computer Based Problem Solving

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**OR Truth Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

In some programming languages the operators are simply the words “AND and OR”. In others they are “&&” for AND and “||” for OR. The following are some examples of compound boolean expressions:

**C++**

```cpp
// compound boolean expressions
```

**Go**

```go
// compound boolean expressions
```

**Java**

```java
// compound boolean expressions
```

**JavaScript**

```javascript
// compound boolean expressions
```

**Python3**

```python3
#!/usr/bin/env python3

# Created by: Mr. Coxall
# Created on: Sep 2019
# This program uses a compound boolean statement

def main():
    # this function uses a compound boolean statement

    # input
    term_mark = int(input("Enter term mark: "))
    project_mark = int(input("Enter project mark: "))
    print("")

    # process & output
    if term_mark > 50 and project_mark > 50:
        print("You passed the course")
    else:
        print("You did not pass the course")
```

(continues on next page)
if \texttt{\_\_name\_\_} == "\texttt{\_\_main\_\_}":
    \texttt{\_main\_()}

Ruby

// compound boolean expressions

Swift

// compound boolean expressions

Besides these two logical operators, there is one more, the NOT. NOT is used most often at the beginning of a Boolean expression to invert its evaluation. It does not compare 2 values but just inverts a single one.

NOT Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>NOT(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

For example:

C++

// NOT boolean expressions

Go

// NOT boolean expressions

Java

// NOT boolean expressions

JavaScript

// NOT boolean expressions

Python3

# NOT boolean expressions

Ruby

// NOT boolean expressions

Swift

// NOT boolean expressions
4.7.8 Nested If Statements

Sometimes a single if statement, even a long If...Then...ElseIf...ElseIf...Else is not a suitable structure to model your problem. Sometimes after one decision is made, there is another second decision that must follow. In these cases, if statements can be nested within if statements (or other structures as we will see later). Here is a problem:

A school is going to sell chocolate bars to raise money. If a student sells over 20 boxes, they get a prize. If they sell 20 to 10, they get a “small” prize. If they sell less than 10, they get honorable mention. Create a program that will let the user input the number of boxes sold and then state what the reward would be and use nested if statements. The flowchart for this type of problem will look something like this:

The Select...Case statement (in most computer programming languages) takes the generic form of:

\[
\text{IF (boolean expression \#1) THEN} \\
\text{First potential statement to be performed} \\
\text{ELSEIF (boolean expression \#2) THEN} \\
\text{Second potential statement to be performed} \\
\text{ELSEIF (boolean expression \#3) THEN} \\
\text{Third potential statement to be performed} \\
\ldots \\
\text{ELSEIF (boolean expression \#n) THEN} \\
\text{Nth potential statement to be performed} \\
\text{ELSE} \\
\text{Alternate statements to be performed} \\
\text{ENDIF}
\]

An example of what this would look like in a specific programming language is:

C++

```cpp
// if ... then ... elseif ... else example
if (colourOfLight == "red") {
    std::cout << "Stop!";
} else if (colourOfLight == "yellow") {
    std::cout << "Slow Down."
} else if (colourOfLight == "green") {
    std::cout << "Go, if all clear.";
} else {
    std::cout << "No idea!";
}
```

Go

```go
// if ... then ... elseif ... else example
```

Java

```java
// if ... then ... elseif ... else example
```

JavaScript

```javascript
// if ... then ... elseif ... else example
```

Python

```python
// if ... then ... elseif ... else example
```

Chapter 4. Structured Problem Solving
# if ... then ... elseif ... else example
if colour_of_light == "red"
    print("Stop!")
elif (colour_of_light == "yellow")
    print("Slow Down.")
elif (colour_of_light == green)
    print("Go, if all clear.")
else
    print("No idea!")
}

// if ... then ... elseif ... else example

Swift

// if ... then ... elseif ... else example

Note: notice how the second if statement is indented within the first one. This is because each if statement is a "structure" and gets indented, within the structure it is already in.
4.8 Repetition
5.1 What is programming

5.1.1 Quickstart

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart they are. Actually, computers are not very smart at all! A computer, broken down into its most basic form, is nothing more than a bunch of tiny electronic switches that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though. A person has to instruct it to do this.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quiet difficult, easy to make mistakes in and tedious. To help people talk to a computer an intermediate language is used, that is translated into machine language so that the computer can understand what to do. This intermediate language is called a programming language and you have probably already heard of some of them (Java, C++, Visual Basic...). Just like there are many different languages that people use around the world (English, French, Spanish...), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs the action.
5.2 How to contribute to Toga

If you experience problems with Toga, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

5.2.1 Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

macOS

```
$ python3 --version
$ pip3 --version
```

Linux

```
$ python3 --version
$ pip3 --version
```

Windows

```
C:\...>python3 --version
C:\...>pip3 --version
```

Next install any additional dependencies for your operating system:

macOS

No additional dependencies

Linux

```
$ sudo apt-get update
$ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev
```

Windows

No additional dependencies

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

macOS

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Linux

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Windows

```
C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate
```
Your prompt should now have a (venv) prefix in front of it.

Next, go to the Toga page on Github, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on “Clone or Download”. If you have the Github desktop application installed on your computer, you can select “Open in Desktop”; otherwise, copy the URL provided, and use it to clone using the command line:

macOS
Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

Linux
Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

Windows
Fork the Toga repository, and then:

```
(venv) C:\...>git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

Now that you have the source code, you can install Toga into your development environment. The Toga source repository contains multiple packages. Since we’re installing from source, we can’t rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

macOS
```
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

Linux
```
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

Windows
```
(venv) C:\...>cd toga
(venv) C:\...>pip install -e src/core
(venv) C:\...>pip install -e src/dummy
```

Then, we can install the code for the specific platform we want to use:

macOS
```
(venv) $ pip install -e src/cocoa
```

Linux

5.2. How to contribute to Toga
Computer Based Problem Solving

(venv) $ pip install -e src/gtk

Windows

(venv) C:\...>pip install -e src/winforms

You can then run the core test suite:

macOS

(venv) $ cd src/core
(venv) $ python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)

Linux

(venv) $ cd src/core
(venv) $ python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)

Windows

(venv) C:\...>cd src/core
(venv) C:\...>python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)

You should get some output indicating that tests have been run. You shouldn’t ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don’t merge the patch. If you do find a test error or failure, either there’s something odd in your test environment, or you’ve found an edge case that we haven’t seen before - either way, let us know!

Now you are ready to start hacking on Toga!

5.2.2 What should I do?

The src/core package of toga has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren’t currently tested (or aren’t tested thoroughly). To work out what isn’t tested, we’re going to use a tool called coverage. Coverage allows you to check which lines of code have (and haven’t) been executed - which then gives you an idea of what code has (and hasn’t) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

macOS
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/"

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Linux

(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/"

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Windows

(venv) C:\...>pip install coverage
(venv) C:\...>coverage run setup.py test
(venv) C:\...>coverage report -m --include=toga/

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

What does this all mean? Well, the “Cover” column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of toga/app.py was executed; but only 77% of lines in toga/window.py were executed. Which lines were missed? They’re listed in the next column: lines 58, 75, 87, and so on weren’t executed.

That’s what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in src/core/tests, you should find a test file that matches the name of the file that has insufficient coverage. If you don’t, it’s possible the entire test file is missing - so you’ll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you’ve written a test, re-run the test suite to generate fresh coverage data. Let’s say we added a test for line 58 of toga/window.py - we’d expect to see something like:

macOS
(venv) $ coverage run setup.py test  
running test  
...  
Ran 101 tests in 0.343s  
 OK (skipped=1)  
(venv) $ coverage report -m --include="toga/*"  

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| toga/window.py            | 79    | 17   | 78%   | 75, 87, 92, 104, 141, 155, ...
|                           |       |      |       | 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257 |
|                           |       |      |       |               |
| TOTAL                     | 1034  | 257  | 75%   |               |

Linux

(venv) $ coverage run setup.py test  
running test  
...  
Ran 101 tests in 0.343s  
 OK (skipped=1)  
(venv) $ coverage report -m --include="toga/*"  

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| toga/window.py            | 79    | 17   | 78%   | 75, 87, 92, 104, 141, 155, ...
|                           |       |      |       | 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257 |
|                           |       |      |       |               |
| TOTAL                     | 1034  | 257  | 75%   |               |

Windows

(venv) C:\...>coverage run setup.py test  
running test  
...  
Ran 101 tests in 0.343s  
 OK (skipped=1)  
(venv) $ coverage report -m --include=toga/*  

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| toga/window.py            | 79    | 17   | 78%   | 75, 87, 92, 104, 141, 155, ...
|                           |       |      |       | 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257 |
|                           |       |      |       |               |
| TOTAL                     | 1034  | 257  | 75%   |               |
That is, one more test has been executed, resulting in one less missing line in the coverage results.
Submit a pull request for your work, and you’re done! Congratulations, you’re a contributor to Toga!

5.2.3 How does this all work?

Since you’re writing tests for a GUI toolkit, you might be wondering why you haven’t seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a “dummy” backend - a platform backend that implements the full API, but doesn’t actually do anything (i.e., when you say display a button, it creates an object, but doesn’t actually display a button).

In this way, it’s possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the dummy backend is invoked correctly, then any other backend will be handled correctly, too.

One error you might see…

When you’re running these tests - especially when you submit your PR, and the tests run on our continuous integration (CI) server - it’s possible you might get and error that reads:

```python
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you’ve found an bug in the way the widget you’re testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a “factory” to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you’re on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the “dummy” factory. The Dummy factory isn’t the “native” platform anywhere - it’s just a placeholder. As a result, the dummy factory won’t be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a factory argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, some widgets - like Icon - are “late loaded” - the implementation isn’t created until the widget is actually used. Late loaded widgets don’t accept a factory when they’re created - but they do have an _impl() method that accepts a factory.

If these factory arguments aren’t being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform- default factory, rather than the “dummy” factory. If you’ve installed the appropriate platform default backend, you won’t (necessarily) get an error, but your tests won’t use the dummy backend. On our CI server, we deliberately don’t install a platform backend so we can find these errors.

If you get the ModuleNotFoundError, you need to audit the code to find out where a widget is being created without a factory being specified.

5.2.4 It’s not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn’t just about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket…
but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally consistent as well. If you notice any method names that aren’t internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn’t being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you’re confident that you know what needs to be done, create a pull request that fixes the problem you’ve found.

One example of the type of consistency we’re looking for is described in this ticket.

### 5.2.5 What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for another coverage line!

If you’re feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you’re on macOS), and look for a widget that isn’t implemented (a missing file in the `widgets` directory for that platform, or an API `on` a widget that isn’t implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

### 5.2.6 Advanced Mode

If you’ve got expertise in a particular platform (for example, if you’ve got experience writing iOS apps), or you’d like to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.

- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.

- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don’t validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we’d have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you’ve got experience with automated GUI testing, we’d love to hear your suggestions.

- **Improve the testing API for application writers** The dummy backend exists to validate that Toga’s internal API works as expected. However, we would like it to be a useful resource for application authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.
CHAPTER 6

Holding Data

THE ART OF BUGFIXING

I'VE FIXED THE BUG

GREAT! WHAT HAVE YOU DONE?

I HAVE NO IDEA

CHAPTER 1: SOMETIMES IT'S BETTER TO NOT EVEN TRY TO UNDERSTAND
Problems have been around for

6.1 What is programming

6.1.1 Quickstart

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart they are. Actually, computers are not very smart at all! A computer, broken down into its most basic form, is nothing more than a bunch of tiny electronic switches that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though. A person has to instruct it to do this.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quiet difficult, easy to make mistakes in and tedious. To help people talk to a computer an intermediate language is used, that is translated into machine language so that the computer can understand what to do. This intermediate language is called a programming language and you have probably already heard of some of them (Java, C++, Visual Basic...). Just like there are many different languages that people use around the world (English, French, Spanish...), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs the action.

6.2 How to contribute to Toga

If you experience problems with Toga, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

6.2.1 Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

macOS

```
$ python3 --version
$ pip3 --version
```

Linux

```
$ python3 --version
$ pip3 --version
```

Windows
Next install any additional dependencies for your operating system:

macOS
No additional dependencies

Linux

$ sudo apt-get update
$ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev

Windows
No additional dependencies

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

macOS

$ python3 -m venv venv
$ source venv/bin/activate

Linux

$ python3 -m venv venv
$ source venv/bin/activate

Windows

C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate

Your prompt should now have a (venv) prefix in front of it.

Next, go to the Toga page on Github, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on “Clone or Download”. If you have the Github desktop application installed on your computer, you can select “Open in Desktop”; otherwise, copy the URL provided, and use it to clone using the command line:

macOS
Fork the Toga repository, and then:

(venv) $ git clone https://github.com/<your username>/toga.git

(substituting your Github username)

Linux
Fork the Toga repository, and then:

(venv) $ git clone https://github.com/<your username>/toga.git

(substituting your Github username)

Windows
Fork the Toga repository, and then:
Now that you have the source code, you can install Toga into your development environment. The Toga source repository contains multiple packages. Since we’re installing from source, we can’t rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

**macOS**

```bash
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

**Linux**

```bash
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

**Windows**

```bash
(venv) C:\...>cd toga
(venv) C:\...>pip install -e src/core
(venv) C:\...>pip install -e src/dummy
```

Then, we can install the code for the specific platform we want to use:

**macOS**

```bash
(venv) $ pip install -e src/cocoa
```

**Linux**

```bash
(venv) $ pip install -e src/gtk
```

**Windows**

```bash
(venv) C:\...>pip install -e src/winforms
```

You can then run the core test suite:

**macOS**

```bash
(venv) $ cd src/core
(venv) $ python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)
```

**Linux**

```bash
(venv) $ cd src/core
(venv) $ python setup.py test
...
```

(continues on next page)
Ran 181 tests in 0.343s
OK (skipped=1)

### 6.2.2 What should I do?

The `src/core` package of `toga` has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren’t currently tested (or aren’t tested thoroughly). To work out what isn’t tested, we’re going to use a tool called **coverage**. Coverage allows you to check which lines of code have (and haven’t) been executed - which then gives you an idea of what code has (and hasn’t) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

**macOS**

```bash
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

**Linux**

```bash
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

You should get some output indicating that tests have been run. You shouldn’t ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don’t merge the patch. If you do find a test error or failure, either there’s something odd in your test environment, or you’ve found an edge case that we haven’t seen before - either way, let us know!

Now you are ready to start hacking on Toga!
What does this all mean? Well, the “Cover” column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of toga/app.py was executed; but only 77% of lines in toga/window.py were executed. Which lines were missed? They’re listed in the next column: lines 58, 75, 87, and so on weren’t executed.

That’s what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in src/core/tests, you should find a test file that matches the name of the file that has insufficient coverage. If you don’t, it’s possible the entire test file is missing - so you’ll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you’ve written a test, re-run the test suite to generate fresh coverage data. Let’s say we added a test for line 58 of toga/window.py - we’d expect to see something like:

macOS

(venv) $ coverage run setup.py test
running test

Ran 101 tests in 0.343s

OK (skipped=1)

(venv) $ coverage report -m --include="toga/"

Name Stmts Miss Cover Missing
-----------------------------------------------
toga/__init__.py 29 0 100%
toga/app.py 50 0 100%
...
toga/window.py 79 17 78% 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
-----------------------------------------------
TOTAL 1034 257 75%

Linux

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>17</td>
<td>78%</td>
<td>75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>257</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>
6.2.3 How does this all work?

Since you’re writing tests for a GUI toolkit, you might be wondering why you haven’t seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a “dummy” backend - a platform backend that implements the full API, but doesn’t actually do anything (i.e., when you say display a button, it creates an object, but doesn’t actually display a button).

In this way, it’s possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the dummy backend is invoked correctly, then any other backend will be handled correctly, too.

That is, one more test has been executed, resulting in one less missing line in the coverage results.

Submit a pull request for your work, and you’re done! Congratulations, you’re a contributor to Toga!
One error you might see...

When you’re running these tests - especially when you submit your PR, and the tests run on our continuous integration (CI) server - it’s possible you might get and error that reads:

```python
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you’ve found an bug in the way the widget you’re testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a “factory” to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you’re on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the “dummy” factory. The Dummy factory isn’t the “native” platform anywhere - it’s just a placeholder. As a result, the dummy factory won’t be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a `factory` argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, some widgets - like Icon - are “late loaded” - the implementation isn’t created until the widget is actually used. Late loaded widgets don’t accept a `factory` when they’re created - but they do have an `_impl()` method that accepts a factory.

If these factory arguments aren’t being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform-default factory, rather than the “dummy” factory. If you’ve installed the appropriate platform default backend, you won’t (necessarily) get an error, but your tests won’t use the dummy backend. On our CI server, we deliberately don’t install a platform backend so we can find these errors.

If you get the `ModuleNotFoundError`, you need to audit the code to find out where a widget is being created without a factory being specified.

6.2.4 It’s not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn’t just about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket... but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally consistent as well. If you notice any method names that aren’t internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn’t being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you’re confident that you know what needs to be done, create a pull request that fixes the problem you’ve found.

One example of the type of consistency we’re looking for is described in this ticket.

6.2.5 What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for another coverage line!

If you’re feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you’re on macOS), and look for a widget that isn’t implemented (a missing file in the `widgets` directory for that platform, or an API on a widget that isn’t implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out
how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

### 6.2.6 Advanced Mode

If you’ve got expertise in a particular platform (for example, if you’ve got experience writing iOS apps), or you’d like to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.

- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.

- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don’t validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we’d have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you’ve got experience with automated GUI testing, we’d love to hear your suggestions.

- **Improve the testing API for application writers** The dummy backend exists to validate that Toga’s internal API works as expected. However, we would like it to be a useful resource for application authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.
CHAPTER 7

Representing and Sorting Data

Problems have been around for as long as people have been around. The process of solving a problem is not something new. Using a computer to aid in solving a problem is new. Modern electronic computers have only been around since the Second World War (1939-1945), which might seem like a long time ago to you but in the history of the human race it is a very short time. The purpose of this book is to help you learn to structure your problem solving method, so that you can consistently develop a verifiable solution that will solve a problem and in the process, use the computer to help you more easily and quickly solve that problem.

7.1 What is programming

7.1.1 Quickstart

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart they are. Actually, computers are not very smart at all! A computer, broken down into its most basic form, is nothing more than a bunch of tiny electronic switches that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though. A person has to instruct it to do this.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quiet difficult, easy to make mistakes in and tedious. To help people talk to a computer an intermediate language is used,
that is translated into machine language so that the computer can understand what to do. This intermediate language is called a programming language and you have probably already heard of some of them (Java, C++, Visual Basic...). Just like there are many different languages that people use around the world (English, French, Spanish...), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs the action.

### 7.2 How to contribute to Toga

If you experience problems with Toga, [log them on GitHub](https://github.com). If you want to contribute code, please [fork the code](https://github.com) and submit a pull request.

#### 7.2.1 Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

**macOS**

```bash
$ python3 --version
$ pip3 --version
```

**Linux**

```bash
$ python3 --version
$ pip3 --version
```

**Windows**

```bash
C:\...>python3 --version
C:\...>pip3 --version
```

Next install any additional dependencies for your operating system:

**macOS**

No additional dependencies

**Linux**

```bash
$ sudo apt-get update
$ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev
```

**Windows**

No additional dependencies

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

**macOS**
```bash
$ python3 -m venv venv
$ source venv/bin/activate
```

**Linux**

```bash
$ python3 -m venv venv
$ source venv/bin/activate
```

**Windows**

```bash
C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate
```

Your prompt should now have a (venv) prefix in front of it.

Next, go to the Toga page on Github, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on “Clone or Download”. If you have the Github desktop application installed on your computer, you can select “Open in Desktop”; otherwise, copy the URL provided, and use it to clone using the command line:

**macOS**

Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

**Linux**

Fork the Toga repository, and then:

```
(venv) $ git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

**Windows**

Fork the Toga repository, and then:

```
(venv) C:\...>git clone https://github.com/<your username>/toga.git
```

(substituting your Github username)

Now that you have the source code, you can install Toga into your development environment. The Toga source repository contains multiple packages. Since we’re installing from source, we can’t rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

**macOS**

```
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

**Linux**

```
(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy
```

7.2. How to contribute to Toga
Then, we can install the code for the specific platform we want to use:

macOS

```
(venv) $ pip install -e src/cocoa
```

Linux

```
(venv) $ pip install -e src/gtk
```

You can then run the core test suite:

macOS

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)
```

Linux

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)
```

Windows

```
(venv) C:\...>cd src/core
(venv) C:\...>python setup.py test
...
Ran 181 tests in 0.343s
OK (skipped=1)
```

You should get some output indicating that tests have been run. You shouldn’t ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don’t merge the patch. If you do find a test error or failure, either there’s something odd in your test environment, or you’ve found an edge case that we haven’t seen before - either way, let us know!

Now you are ready to start hacking on Toga!
7.2.2 What should I do?

The src/core package of toga has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren’t currently tested (or aren’t tested thoroughly). To work out what isn’t tested, we’re going to use a tool called coverage. Coverage allows you to check which lines of code have (and haven’t) been executed - which then gives you an idea of what code has (and hasn’t) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

### macOS

```
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

### Linux

```
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

### Windows

```
(venv) C:\...\>pip install coverage
(venv) C:\...\>coverage run setup.py test
(venv) C:\...\>coverage report -m --include=toga/*
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

What does this all mean? Well, the “Cover” column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of toga/app.py was executed; but only 77% of lines in toga/window.py were executed. Which lines were missed? They’re listed in the next column: lines 58, 75, 87, and so on weren’t executed.

7.2. How to contribute to Toga
That’s what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in src/core/tests, you should find a test file that matches the name of the file that has insufficient coverage. If you don’t, it’s possible the entire test file is missing - so you’ll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you’ve written a test, re-run the test suite to generate fresh coverage data. Let’s say we added a test for line 58 of toga/window.py - we’d expect to see something like:

```bash
(venv) $ coverage run setup.py test
running test
...
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include="toga/*"
Name Stmts Miss Cover Missing
------------------------------------------
toga/__init__.py  29  0  100%
toga/app.py  50  0  100%
...  
toga/window.py  79  17  78%  75, 87, 92, 104, 141, 155, ...
-164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
------------------------------------------
TOTAL  1034  257  75%
```

Linux

```bash
(venv) $ coverage run setup.py test
running test
...
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include="toga/*"
Name Stmts Miss Cover Missing
------------------------------------------
toga/__init__.py  29  0  100%
toga/app.py  50  0  100%
...  
toga/window.py  79  17  78%  75, 87, 92, 104, 141, 155, ...
-164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
------------------------------------------
TOTAL  1034  257  75%
```

Windows

```bash
(venv) C:\...> coverage run setup.py test
running test
...
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include=toga/*
```

(continues on next page)
That is, one more test has been executed, resulting in one less missing line in the coverage results.

Submit a pull request for your work, and you’re done! Congratulations, you’re a contributor to Toga!

### 7.2.3 How does this all work?

Since you’re writing tests for a GUI toolkit, you might be wondering why you haven’t seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a “dummy” backend - a platform backend that implements the full API, but doesn’t actually do anything (i.e., when you say display a button, it creates an object, but doesn’t actually display a button).

In this way, it’s possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the dummy backend is invoked correctly, then any other backend will be handled correctly, too.

#### One error you might see...

When you’re running these tests - especially when you submit your PR, and the tests run on our continous integration (CI) server - it’s possible you might get and error that reads:

```
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you’ve found an bug in the way the widget you’re testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a “factory” to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you’re on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the “dummy” factory. The Dummy factory isn’t the “native” platform anywhere - it’s just a placeholder. As a result, the dummy factory won’t be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a factory argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, some widgets - like Icon - are “late loaded” - the implementation isn’t created until the widget is actually used. Late loaded widgets don’t accept a factory when they’re created - but they do have an _impl() method that accepts a factory.

If these factory arguments aren’t being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform- default factory, rather than the “dummy” factory. If you’ve installed the appropriate
platform default backend, you won’t (necessarily) get an error, but your tests won’t use the dummy backend. On our CI server, we deliberately don’t install a platform backend so we can find these errors.

If you get the `ModuleNotFoundError`, you need to audit the code to find out where a widget is being created without a factory being specified.

### 7.2.4 It’s not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn’t just about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket... but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally consistent as well. If you notice any method names that aren’t internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn’t being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you’re confident that you know what needs to be done, create a pull request that fixes the problem you’ve found.

One example of the type of consistency we’re looking for is described in this ticket.

### 7.2.5 What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for another coverage line!

If you’re feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you’re on macOS), and look for a widget that isn’t implemented (a missing file in the `widgets` directory for that platform, or an API on a widget that isn’t implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

### 7.2.6 Advanced Mode

If you’ve got expertise in a particular platform (for example, if you’ve got experience writing iOS apps), or you’d like to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.

- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.

- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don’t validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we’d have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you’ve got experience with automated GUI testing, we’d love to hear your suggestions.

- **Improve the testing API for application writers** The dummy backend exists to validate that Toga’s internal API works as expected. However, we would like it to be a useful resource for application authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one
that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.
Problems have been around for as long as people have been around. The process of solving a problem is not something new. Using a computer to aid in solving a problem is new. Modern electronic computers have only been around since the Second World War (1939-1945), which might seem like a long time ago to you but in the history of the human race it is a very short time. The purpose of this book is to help you learn to structure your problem solving method, so that you can consistently develop a verifiable solution that will solve a problem and in the process, use the computer to help you more easily and quickly solve that problem.

8.1 What is programming

8.1.1 Quickstart

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart they are. Actually, computers are not very smart at all! A computer, broken down into its most basic form, is nothing more than a bunch of tiny electronic switches that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though. A person has to instruct it to do this.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quiet difficult, easy to make mistakes in and tedious. To help people talk to a computer an intermediate language is used,
that is translated into machine language so that the computer can understand what to do. This intermediate language is called a programming language and you have probably already heard of some of them (Java, C++, Visual Basic...). Just like there are many different languages that people use around the world (English, French, Spanish...), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs the action.

8.2 How to contribute to Toga

If you experience problems with Toga, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

8.2.1 Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

macOS

```bash
$ python3 --version
$ pip3 --version
```

Linux

```bash
$ python3 --version
$ pip3 --version
```

Windows

```bash
C:\...>python3 --version
C:\...>pip3 --version
```

Next install any additional dependencies for your operating system:

macOS

No additional dependencies

Linux

```bash
$ sudo apt-get update
$ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev
```

Windows

No additional dependencies

The recommended way of setting up your development environment for Toga is to install a virtual environment, install the required dependencies and start coding. To set up a virtual environment, run:

macOS
```bash
$ python3 -m venv venv
$ source venv/bin/activate
```

**Linux**

```bash
$ python3 -m venv venv
$ source venv/bin/activate
```

**Windows**

```bash
C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate
```

Your prompt should now have a (venv) prefix in front of it.

Next, go to the Toga page on Github, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on “Clone or Download”. If you have the Github desktop application installed on your computer, you can select “Open in Desktop”; otherwise, copy the URL provided, and use it to clone using the command line:

- **macOS**
  
  Fork the Toga repository, and then:

  ```bash
  (venv) $ git clone https://github.com/<your username>/toga.git
  ```

  (substituting your Github username)

- **Linux**
  
  Fork the Toga repository, and then:

  ```bash
  (venv) $ git clone https://github.com/<your username>/toga.git
  ```

  (substituting your Github username)

- **Windows**
  
  Fork the Toga repository, and then:

  ```bash
  (venv) C:\...>git clone https://github.com/<your username>/toga.git
  ```

  (substituting your Github username)

Now that you have the source code, you can install Toga into your development environment. The Toga source repository contains multiple packages. Since we’re installing from source, we can’t rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

- **macOS**
  
  ```bash
  (venv) $ cd toga
  (venv) $ pip install -e src/core
  (venv) $ pip install -e src/dummy
  ```

- **Linux**
  
  ```bash
  (venv) $ cd toga
  (venv) $ pip install -e src/core
  (venv) $ pip install -e src/dummy
  ```

### 8.2. How to contribute to Toga
Then, we can install the code for the specific platform we want to use:

**macOS**

```
(venv) $ pip install -e src/cocoa
```

**Linux**

```
(venv) $ pip install -e src/gtk
```

**Windows**

```
(venv) C:...>cd src/winforms
```

You can then run the core test suite:

**macOS**

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
---------------------------------------------------------------------------
Ran 181 tests in 0.343s
OK (skipped=1)
```

**Linux**

```
(venv) $ cd src/core
(venv) $ python setup.py test
...
---------------------------------------------------------------------------
Ran 181 tests in 0.343s
OK (skipped=1)
```

**Windows**

```
(venv) C:...>cd src/core
(venv) C:...>python setup.py test
...
---------------------------------------------------------------------------
Ran 181 tests in 0.343s
OK (skipped=1)
```

You should get some output indicating that tests have been run. You shouldn’t ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don’t merge the patch. If you do find a test error or failure, either there’s something odd in your test environment, or you’ve found an edge case that we haven’t seen before - either way, let us know!

Now you are ready to start hacking on Toga!
8.2.2 What should I do?

The src/core package of toga has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren’t currently tested (or aren’t tested thoroughly). To work out what isn’t tested, we’re going to use a tool called coverage. Coverage allows you to check which lines of code have (and haven’t) been executed - which then gives you an idea of what code has (and hasn’t) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

macOS

```
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Linux

```
(venv) $ pip install coverage
(venv) $ coverage run setup.py test
(venv) $ coverage report -m --include="toga/*"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Windows

```
(venv) C:\...>pip install coverage
(venv) C:\...>coverage run setup.py test
(venv) C:\...>coverage report -m --include=toga/"
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td></td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

What does this all mean? Well, the “Cover” column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of toga/app.py was executed; but only 77% of lines in toga/window.py were executed. Which lines were missed? They’re listed in the next column: lines 58, 75, 87, and so on weren’t executed.
That’s what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in \textit{src/core/tests}, you should find a test file that matches the name of the file that has insufficient coverage. If you don’t, it’s possible the entire test file is missing - so you’ll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you’ve written a test, re-run the test suite to generate fresh coverage data. Let’s say we added a test for line 58 of \texttt{toga/window.py} - we’d expect to see something like:

\begin{verbatim}
macOS
(venv) $ coverage run setup.py test
running test
... 
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include="toga/*"
Name Stmts Miss Cover Missing
------------------------------------------
toga/__init__.py 29 0 100%                    
toga/app.py 50 0 100%                    
...   toga/window.py 79 17 78% 75, 87, 92, 104, 141, 155, ...
       164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
------------------------------------------
TOTAL 1034 257 75%
\end{verbatim}

\begin{verbatim}
Linux
(venv) $ coverage run setup.py test
running test
... 
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include="toga/*"
Name Stmts Miss Cover Missing
------------------------------------------
toga/__init__.py 29 0 100%                    
toga/app.py 50 0 100%                    
...   toga/window.py 79 17 78% 75, 87, 92, 104, 141, 155, ...
       164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
------------------------------------------
TOTAL 1034 257 75%
\end{verbatim}

\begin{verbatim}
Windows
(venv) C:...>coverage run setup.py test
running test
... 
Ran 101 tests in 0.343s
OK (skipped=1)
(venv) $ coverage report -m --include=to
\end{verbatim}
That is, one more test has been executed, resulting in one less missing line in the coverage results.

Submit a pull request for your work, and you’re done! Congratulations, you’re a contributor to Toga!

### 8.2.3 How does this all work?

Since you’re writing tests for a GUI toolkit, you might be wondering why you haven’t seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a “dummy” backend - a platform backend that implements the full API, but doesn’t actually do anything (i.e., when you say display a button, it creates an object, but doesn’t actually display a button).

In this way, it’s possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the dummy backend is invoked correctly, then any other backend will be handled correctly, too.

**One error you might see…**

When you’re running these tests - especially when you submit your PR, and the tests run on our continuous integration (CI) server - it’s possible you might get an error that reads:

```
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you’ve found an bug in the way the widget you’re testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a “factory” to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you’re on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the “dummy” factory. The Dummy factory isn’t the “native” platform anywhere - it’s just a placeholder. As a result, the dummy factory won’t be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a `factory` argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, some widgets - like Icon - are “late loaded” - the implementation isn’t created until the widget is actually used. Late loaded widgets don’t accept a `factory` when they’re created - but they do have an `_impl()` method that accepts a factory.

If these factory arguments aren’t being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform- default factory, rather than the “dummy” factory. If you’ve installed the appropriate
platform default backend, you won’t (necessarily) get an error, but your tests won’t use the dummy backend. On our CI server, we deliberately don’t install a platform backend so we can find these errors.

If you get the `ModuleNotFoundError`, you need to audit the code to find out where a widget is being created without a factory being specified.

### 8.2.4 It’s not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn’t *just* about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket... but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally consistent as well. If you notice any method names that aren’t internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn’t being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you’re confident that you know what needs to be done, create a pull request that fixes the problem you’ve found.

One example of the type of consistency we’re looking for is described in this ticket.

### 8.2.5 What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for another coverage line!

If you’re feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you’re on macOS), and look for a widget that isn’t implemented (a missing file in the `widgets` directory for that platform, or an API *on* a widget that isn’t implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

### 8.2.6 Advanced Mode

If you’ve got expertise in a particular platform (for example, if you’ve got experience writing iOS apps), or you’d *like* to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.

- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.

- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don’t validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we’d have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you’ve got experience with automated GUI testing, we’d love to hear your suggestions.

- **Improve the testing API for application writers** The dummy backend exists to validate that Toga’s internal API works as expected. However, we would like it to be a useful resource for *application* authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one
that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.
Creating Objects

Problems have been around for as long as people have been around. The process of solving a problem is not something new. Using a computer to aid in solving a problem is new. Modern electronic computers have only been around since the Second World War (1939-1945), which might seem like a long time ago to you but in the history of the human race it is a very short time. The purpose of this book is to help you learn to structure your problem solving method, so that you can consistently develop a verifiable solution that will solve a problem and in the process, use the computer to help you more easily and quickly solve that problem.

9.1 What is programming

9.1.1 Quickstart

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart they are. Actually, computers are not very smart at all! A computer, broken down into its most basic form, is nothing more than a bunch of tiny electronic switches that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though. A person has to instruct it to do this.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them. To keep things simple, you usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1’s and 0’s mentioned above. Trying to talk in machine language is quiet difficult, easy to make mistakes in and tedious. To help people talk to a computer an intermediate language is used,
that is translated into machine language so that the computer can understand what to do. This intermediate language
is called a programming language and you have probably already heard of some of them (Java, C++, Visual Basic...).
Just like there are many different languages that people use around the world (English, French, Spanish...), there
have been many different programming languages developed to help people instruct computers in what to do. The
purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to
talk machine language.

A person that uses a programming language to instruct a computer what to do is called a programmer. The program-
mer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the
programming language that they have chosen. Then the computer translates the instructions into machine language
(the language that the computer actually understands) and the computer performs the action.

9.2 How to contribute to Toga

If you experience problems with Toga, log them on GitHub. If you want to contribute code, please fork the code and
submit a pull request.

9.2.1 Set up your development environment

First thing is to ensure that you have Python 3 and pip installed. To do this run the following commands:

macOS

$ python3 --version
$ pip3 --version

Linux

$ python3 --version
$ pip3 --version

Windows

$ python3 --version
$ pip3 --version

Next install any additional dependencies for your operating system:

macOS
No additional dependencies

Linux

$ sudo apt-get update
$ sudo apt-get install python3-dev libgirepository1.0-dev libcairo2-dev

Windows
No additional dependencies

The recommended way of setting up your development environment for Toga is to install a virtual environment, install
the required dependencies and start coding. To set up a virtual environment, run:

macOS
$ python3 -m venv venv
$ source venv/bin/activate

Linux

$ python3 -m venv venv
$ source venv/bin/activate

Windows

C:\...>python3 -m venv venv
C:\...>venv/Scripts/activate

Your prompt should now have a (venv) prefix in front of it.

Next, go to the Toga page on Github, and fork the repository into your own account, and then clone a copy of that repository onto your computer by clicking on “Clone or Download”. If you have the Github desktop application installed on your computer, you can select “Open in Desktop”; otherwise, copy the URL provided, and use it to clone using the command line:

macOS

Fork the Toga repository, and then:

(venv) $ git clone https://github.com/<your username>/toga.git

(substituting your Github username)

Linux

Fork the Toga repository, and then:

(venv) $ git clone https://github.com/<your username>/toga.git

(substituting your Github username)

Windows

Fork the Toga repository, and then:

(venv) C:\...>git clone https://github.com/<your username>/toga.git

(substituting your Github username)

Now that you have the source code, you can install Toga into your development environment. The Toga source repository contains multiple packages. Since we’re installing from source, we can’t rely on pip to install the packages in dependency order. Therefore, we have to manually install each package in a specific order. We start with the core packages:

macOS

(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy

Linux

(venv) $ cd toga
(venv) $ pip install -e src/core
(venv) $ pip install -e src/dummy

9.2. How to contribute to Toga
Windows

```bash
(venv) C:\...>cd toga
(venv) C:\...>pip install -e src/core
(venv) C:\...>pip install -e src/dummy
```

Then, we can install the code for the specific platform we want to use:

macOS

```bash
(venv) $ pip install -e src/cocoa
```

Linux

```bash
(venv) $ pip install -e src/gtk
```

Windows

```bash
(venv) C:\...>pip install -e src/winforms
```

You can then run the core test suite:

macOS

```bash
(venv) $ cd src/core
(venv) $ python setup.py test
... 
Ran 181 tests in 0.343s
OK (skipped=1)
```

Linux

```bash
(venv) $ cd src/core
(venv) $ python setup.py test
... 
Ran 181 tests in 0.343s
OK (skipped=1)
```

Windows

```bash
(venv) C:\...>cd src/core
(venv) C:\...>python setup.py test
... 
Ran 181 tests in 0.343s
OK (skipped=1)
```

You should get some output indicating that tests have been run. You shouldn’t ever get any FAIL or ERROR test results. We run our full test suite before merging every patch. If that process discovers any problems, we don’t merge the patch. If you do find a test error or failure, either there’s something odd in your test environment, or you’ve found an edge case that we haven’t seen before - either way, let us know!

Now you are ready to start hacking on Toga!
9.2.2 What should I do?

The src/core package of toga has a test suite, but that test suite is incomplete. There are many aspects of the Toga Core API that aren’t currently tested (or aren’t tested thoroughly). To work out what isn’t tested, we’re going to use a tool called coverage. Coverage allows you to check which lines of code have (and haven’t) been executed - which then gives you an idea of what code has (and hasn’t) been tested.

Install coverage, and then re-run the test suite – this time, in a slightly different way so that we can gather some data about the test run. Then we can ask coverage to generate a report of the data that was gathered:

macOS

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Linux

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

Windows

<table>
<thead>
<tr>
<th>Name</th>
<th>Stmts</th>
<th>Miss</th>
<th>Cover</th>
<th>Missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>toga/<strong>init</strong>.py</td>
<td>29</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>toga/app.py</td>
<td>50</td>
<td>0</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toga/window.py</td>
<td>79</td>
<td>18</td>
<td>77%</td>
<td>58, 75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1034</td>
<td>258</td>
<td>75%</td>
<td></td>
</tr>
</tbody>
</table>

What does this all mean? Well, the “Cover” column tells you what proportion of lines in a given file were executed during the test run. In this run, every line of toga/app.py was executed; but only 77% of lines in toga/window.py were executed. Which lines were missed? They’re listed in the next column: lines 58, 75, 87, and so on weren’t executed.

9.2. How to contribute to Toga
That’s what you have to fix - ideally, every single line in every single file will have 100% coverage. If you look in `src/core/tests`, you should find a test file that matches the name of the file that has insufficient coverage. If you don’t, it’s possible the entire test file is missing - so you’ll have to create it!

Your task: create a test that improves coverage - even by one more line.

Once you’ve written a test, re-run the test suite to generate fresh coverage data. Let’s say we added a test for line 58 of `toga/window.py` - we’d expect to see something like:

```plaintext
   macos
   (venv) $ coverage run setup.py test
      running test
      ...
      ____________________________________________________
      Ran 101 tests in 0.343s
      OK (skipped=1)
   (venv) $ coverage report -m --include="toga/*"
   Name          Stmts  Miss  Cover  Missing
   --------------  ------  ----  ------  ----------
      toga/__init__.py  29     0   100%     
      toga/app.py       50     0   100%     
      ...              
      toga/window.py   79    17    78%    75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
      ...              
   TOTAL           1034  257    75%     

   linux
   (venv) $ coverage run setup.py test
      running test
      ...
      ____________________________________________________
      Ran 101 tests in 0.343s
      OK (skipped=1)
   (venv) $ coverage report -m --include="toga/*"
   Name          Stmts  Miss  Cover  Missing
   --------------  ------  ----  ------  ----------
      toga/__init__.py  29     0   100%     
      toga/app.py       50     0   100%     
      ...              
      toga/window.py   79    17    78%    75, 87, 92, 104, 141, 155, 164, 168, 172-173, 176, 192, 204, 216, 228, 243, 257
      ...              
   TOTAL           1034  257    75%     

   windows
   (venv) C:\...>coverage run setup.py test
      running test
      ...
      ____________________________________________________
      Ran 101 tests in 0.343s
      OK (skipped=1)
   (venv) $ coverage report -m --include=toga/*
```
That is, one more test has been executed, resulting in one less missing line in the coverage results.

Submit a pull request for your work, and you’re done! Congratulations, you’re a contributor to Toga!

### 9.2.3 How does this all work?

Since you’re writing tests for a GUI toolkit, you might be wondering why you haven’t seen a GUI yet. The Toga Core package contains the API definitions for the Toga widget kit. This is completely platform agnostic - it just provides an interface, and defers actually drawing anything on the screen to the platform backends.

When you run the test suite, the test runner uses a “dummy” backend - a platform backend that implements the full API, but doesn’t actually do anything (i.e., when you say display a button, it creates an object, but doesn’t actually display a button).

In this way, it’s possible to for the Toga Core tests to exercise every API entry point in the Toga Core package, verify that data is stored correctly on the interface layer, and sent through to the right endpoints in the Dummy backend. If the dummy backend is invoked correctly, then any other backend will be handled correctly, too.

**One error you might see…**

When you’re running these tests - especially when you submit your PR, and the tests run on our continous integration (CI) server - it’s possible you might get an error that reads:

```
ModuleNotFoundError: No module named 'toga_gtk'.
```

If this happens, you’ve found an bug in the way the widget you’re testing has been constructed.

The Core API is designed to be platform independent. When a widget is created, it calls upon a “factory” to instantiate the underlying platform-dependent implementation. When a Toga application starts running, it will try to guess the right factory to use based on the environment where the code is running. So, if you run your code on a Mac, it will use the Cocoa factory; if you’re on a Linux box, it will use the GTK factory.

However, when writing tests, we want to use the “dummy” factory. The Dummy factory isn’t the “native” platform anywhere - it’s just a placeholder. As a result, the dummy factory won’t be used unless you specifically request it - which means every widget has to honor that request.

Most Toga widgets create their platform-specific implementation when they are created. As a result, most Toga widgets should accept a factory argument - and that factory should be used to instantiate any widget implementations or sub-widgets.

However, some widgets - like Icon - are “late loaded” - the implementation isn’t created until the widget is actually used. Late loaded widgets don’t accept a factory when they’re created - but they do have an _impl() method that accepts a factory.

If these factory arguments aren’t being passed around correctly, then a test suite will attempt to create a widget, but will fall back to the platform- default factory, rather than the “dummy” factory. If you’ve installed the appropriate...
platform default backend, you won’t (necessarily) get an error, but your tests won’t use the dummy backend. On our CI server, we deliberately don’t install a platform backend so we can find these errors.

If you get the `ModuleNotFoundError`, you need to audit the code to find out where a widget is being created without a factory being specified.

### 9.2.4 It’s not just about coverage!

Although improving test coverage is the goal, the task ahead of you isn’t just about increasing numerical coverage. Part of the task is to audit the code as you go. You could write a comprehensive set of tests for a concrete life jacket... but a concrete life jacket would still be useless for the purpose it was intended!

As you develop tests and improve coverage, you should be checking that the core module is internally consistent as well. If you notice any method names that aren’t internally consistent (e.g., something called `on_select` in one module, but called `on_selected` in another), or where the data isn’t being handled consistently (one widget updates then refreshes, but another widget refreshes then updates), flag it and bring it to our attention by raising a ticket. Or, if you’re confident that you know what needs to be done, create a pull request that fixes the problem you’ve found.

One example of the type of consistency we’re looking for is described in this ticket.

### 9.2.5 What next?

Rinse and repeat! Having improved coverage by one line, go back and do it again for another coverage line!

If you’re feeling particularly adventurous, you could start looking at a specific platform backend. The Toga Dummy API defines the API that a backend needs to implement; so find a platform backend of interest to you (e.g., cocoa if you’re on macOS), and look for a widget that isn’t implemented (a missing file in the `widgets` directory for that platform, or an API on a widget that isn’t implemented (these will be flagged by raising `NotImplementedError()`). Dig into the documentation for native widgets for that platform (e.g., the Apple Cocoa documentation), and work out how to map native widget capabilities to the Toga API. You may find it helpful to look at existing widgets to work out what is needed.

Most importantly - have fun!

### 9.2.6 Advanced Mode

If you’ve got expertise in a particular platform (for example, if you’ve got experience writing iOS apps), or you’d like to have that experience, you might want to look into a more advanced problem. Here are some suggestions:

- **Implement a platform native widget** If the core library already specifies an interface, implement that interface; if no interface exists, propose an interface design, and implement it for at least one platform.

- **Add a new feature to an existing widget API** Can you think of a feature than an existing widget should have? Propose a new API for that widget, and provide a sample implementation.

- **Improve platform specific testing** The tests that have been described in this document are all platform independent. They use the dummy backend to validate that data is being passed around correctly, but they don’t validate that on a given platform, widgets behave they way they should. If I put a button on a Toga app, is that button displayed? Is it in the right place? Does it respond to mouse clicks? Ideally, we’d have automated tests to validate these properties. However, automated tests of GUI operations can be difficult to set up. If you’ve got experience with automated GUI testing, we’d love to hear your suggestions.

- **Improve the testing API for application writers** The dummy backend exists to validate that Toga’s internal API works as expected. However, we would like it to be a useful resource for application authors as well. Testing GUI applications is a difficult task; a Dummy backend would potentially allow an end user to write an application, and validate behavior by testing the properties of the Dummy. Think of it as a GUI mock - but one
that is baked into Toga as a framework. See if you can write a GUI app of your own, and write a test suite that uses the Dummy backend to validate the behavior of that app.