

VisTex: An Investigation on the Role of Visual and Textual Programming Languages When Learning to Program



MARK NOONE

A thesis submitted for the degree of
Master of Computer Science
Department of Computer Science
Maynooth University

October 2023

Supervisor: Dr. Aidan Mooney
Head of Department: Prof. Ronan Farrell

CONTENTS

I	INTRODUCTION & BACKGROUND	1
1	INTRODUCTION & BACKGROUND	2
1.1	Introduction	2
1.2	Motivation	4
1.3	Goals and Research Questions	5
1.4	Contributions	6
1.5	List of Publications	7
1.6	Ethical Approval	8
1.7	Thesis Overview	9
2	RELATED WORK	10
2.1	Overview of Systematic Literature Review	10
2.1.1	Introduction and Motivation	11
2.1.2	Research Questions	12
2.1.3	Background	13
2.2	Method	14
2.2.1	Introduction	14
2.2.2	Resources Searched	14
2.2.3	Search Terms	15
2.2.4	Document Selection	15
2.2.5	Quality Assessment	17
2.2.6	Data Extraction and Synthesis	19
2.3	Dataset	20
2.3.1	Types of Studies	20
2.3.2	Timeline of chosen publications	20
2.3.3	Data Sources	21

2.3.4	Dataset Discussion	21
2.4	Results	22
2.4.1	Research Question 1: Are there any benefits of learning a visual programming language over a traditional text-based language?	23
2.4.2	Research Question 2: Does the choice of First Programming Language make a difference? What languages are the best ones to teach?	34
2.4.3	Discussion	43
2.5	Summary	45
II	COURSE DEVELOPMENT & TESTING	46
3	INITIAL COURSE DEVELOPMENT	47
3.1	Overview	47
3.2	Initial Test Sessions	47
3.2.1	Language Choice	48
3.2.2	Sessions Overview	50
3.2.3	Initial Pilot Test	51
3.2.4	Data Collection	53
3.2.5	Main Session Delivery	53
3.2.6	Summer Camp Outcomes	54
3.3	Full Courses	58
3.3.1	Courses Overview	59
3.3.2	Schools Testing	60
3.4	Need for a Hybrid Model	71
3.5	Summary	72
III	HYBRID PROGRAMMING	73
4	HYBRID JAVA	74
4.1	The Need for Hybrid Java and Introduction	74
4.2	Additional Hybrid Programming Background	76

4.3	Hybrid Java Development	78
4.3.1	Block Sections	81
4.4	Hybrid Java Testing	83
4.4.1	Test 1: Undergraduate Survey	83
4.4.2	Test 2: Computer Science Summer Camp	86
4.4.3	Test 3: UKICER Workshop	89
4.4.4	Iterative Development	90
4.5	Building the Hybrid Java Curriculum	91
4.6	Other Uses for Hybrid Java	92
4.7	The Hybrid Java Package	94
4.8	Summary	94
IV	CONCLUSIONS & FUTURE WORK	96
5	CONCLUSIONS AND FUTURE WORK	97
5.1	Conclusions	97
5.1.1	Research Questions	98
5.1.2	Place for Hybrid Java	99
5.2	Challenges	100
5.3	Future Work	101
5.3.1	Full Hybrid Java Delivery	101
5.3.2	Further Testing Phases	102
5.3.3	Object Oriented Programming	103
5.3.4	Hybrid Java to Java translation	103
5.3.5	Visual Feedback	104
5.3.6	Other Language Options	105
5.3.7	HJP Sharing	105
	Bibliography	106
	Appendix	118
A	APPENDIX 1 - SUMMER CAMP SESSIONS	118
A.1	Initial Session Materials	118
A.1.1	Java Session Material	118

A.1.2	Snap! Session Material	127
A.1.3	Survey Questions	134
A.1.4	Data Sheet	136
B	APPENDIX 2 - FULL COURSES	137
B.1	Full Course Materials	137
C	APPENDIX 3 - HYBRID JAVA BLOCKS	138
C.1	Full List of all Hybrid Java Blocks	138

LIST OF FIGURES

Figure 2.1	List of accepted papers, their year and their QA Scores (Y = 1, P = 0.5, N = 0, score of 3 out of 4 required)	18
Figure 2.2	Timeline of Accepted Papers	21
Figure 2.3	Distribution of Accepted Papers by Source	22
Figure 3.1	Advanced Code Solutions	52
Figure 3.2	Language Difficulty Ratings	55
Figure 3.3	Language Mean Difficulty Ratings by Age	56
Figure 3.4	Responses to Favourite and Least Favourite Course Elements	57
Figure 3.5	Example homework question (Java and Snap!) . . .	60
Figure 3.6	Java Examination Question	62
Figure 3.7	Snap! Examination Question	63
Figure 3.8	Computer Science at University Opinions - Java . .	66
Figure 3.9	Computer Science at University Opinions - Snap! .	66
Figure 3.10	Hardest Session in Java	68
Figure 3.11	Easiest Session in Java	69
Figure 3.12	Hardest Session in Snap!	70
Figure 3.13	Easiest Session in Snap!	71
Figure 4.1	For Loop Backend in Snap!	79
Figure 4.2	For Loop Frontend in Snap!	80
Figure 4.3	Sample Hybrid Java program with output	82
Figure 4.4	Undergraduate System Test Question	84
Figure 4.5	Graph of responses to Q3, Q4 and Q5 respectively .	85
Figure 4.6	Simple Calculator program in both Java and Hybrid Java	88

Figure C.1	"Control" Blocks	139
Figure C.2	"Sensing" Blocks	139
Figure C.3	"Operators" Blocks	140
Figure C.4	"Variables" Blocks – Variables, Strings, Scanner . . .	141
Figure C.5	"Variables" Blocks – Random, Parse, Arrays	142
Figure C.6	"Variables" Blocks – 2D Arrays	143

LIST OF TABLES

Table 3.1	Session Topics	50
Table 3.2	Marking Schemes for Java and Snap!	64

ABSTRACT

Visual programming languages are seen by many as the best way to teach programming at primary and secondary level, while at third level textual languages tend to be taught due in part to the dependence of these languages to industry. In general, there is a trade-off in programming language design between ease of use and expressiveness. Writing a simple "Hello, world" program in a visual language, such as Snap, Scratch or App Inventor, is generally more straightforward than writing it in a general-purpose programming language such as Java or Python. The underlying syntax of visual languages tends to be hidden from students thus eliminating the possibility of syntax errors by making it impossible to create syntactically invalid programs.

In our current society vast opportunities exist for graduates in Information Technology and Software Development but dropout rates from year one to year two in Computer Science courses at higher education are worryingly high; as indeed is the low ratio of females studying these courses. The timeliness of this project is in line with considerable interest and growth in teaching programming at secondary schools (e.g. Computer Science at Leaving Certificate) but, we must endeavour to ensure that we are not turning students off pursuing programming in higher education, if the first language they encounter at secondary school is not optimal.

Numerous studies and much debate, particularly in the ACM CS Education community forum discuss the optimal first language to teach computer programming to students. These studies are often small in scale, have specific settings and are not repeated, and thus must be interpreted with caution. Despite all of this the answer remains elusive. A solid systematic project composed of multiple repeated studies that would provide

scientific evidence on the efficacy of the examined languages and thus direct future pedagogic approaches is needed and is the first component of this research.

The overall aims of the project include investigating the role that different types of programming languages (textual, visual and “hybrid”) play when a learner is first introduced to programming. A systematic analysis of the gathered data will be carried out to determine if educational, age, gender and other differences lead to a preference in language choice and difference in performance in those languages. When the initial phases of curriculum testing are complete, a hybrid version of Java (with a blocks-based approach) will be developed and delivered as a curriculum. Finally, a conclusion will be made as to whether visual programming languages result in more effective and higher performance outputs than textual programming languages, and if the hybrid approach provides a “middling” difficulty level.

ACKNOWLEDGMENTS

This work was undertaken with support from the Maynooth University John and Pat Hume Scholarship 2016-2019, and 2023.

I wish to thank my wife, Jessica Noone, who has always been there for me and supported me through everything over the years.

I would also like to thank my supervisor, Dr Aidan Mooney. He was always around when needed, and gave the perfect amount of support, never pushing too hard but never under supporting either. He was a pleasure to work with.

I am grateful to Emily O' Regan, a past project student who aided in the development of the Hybrid Java tool. I would also like to thank Emlyn Hegarty-Kelly, Keith Nolan, Amy Thompson, Sam O' Neill and many other Postgraduate students who I have worked with over the last few years.

I would like to thank a number of co-authors that I met during this project including Frank Glavin, Monica Ward, Emer Thornbury and others.

Finally, I would like to acknowledge all of the students in schools where my curricula were tested, students who attended the Computer Science Summer Camp during numerous test sessions, many demonstrators of the CS Summer Camp who aided in teaching and attending focus groups. They were all instrumental in the completion of this research.

ACRONYMS

The following is a list of frequently used acronyms that will appear in this thesis. Please refer back here if there are any acronyms you do not recognise when reading. They are listed in alphabetical order.

- BYOB - "Bring Your Own Blocks" - The original name for the Snap! programming environment. The name was eventually changed for obvious reasons.
- CS - "Computer Science".
- CSo - "Computer Science o" - The name given to a pre-University level programming course, often using a visual programming language.
- CS₁ - "Computer Science 1" - An introductory programming course covering all of the basic threshold concepts.
- CS₁₆₁ - The name for the CS₁ course in the Department of Computer Science at Maynooth University.
- CS₁₆₂ - The name for the CS₂ course in the Department of Computer Science at Maynooth University.
- CS₂ - "Computer Science 2" - The followup course to a CS₁. This usually covers more advanced topics, for example object orientation.
- CSC - "Computer Science Centre" - A support centre ran by the Department of Computer Science at Maynooth University which aids struggling students with their programming skills.
- FPL - "First Programming Language".

- HJP - "Hybrid Java Package" which will be detailed in Chapter 4 and Chapter 5.
- IDE - "Integrated Development Environment" - A tool used to write and run programs.
- JC - "Junior Certificate" - The first set of state examinations in Ireland, taken by 14-15 year olds.
- LC - "Leaving Certificate" - The final set of state examinations in Ireland, taken by 17-18 year olds. These examinations determine your University prospects.
- MU - "Maynooth University".
- RQ - "Research Question".
- SLR - "Systematic Literature Review".
- TPL - "Textual Programming Language".
- TY - "Transition Year" - An intermediate year between the JC and LC in Ireland which is optional for students. Traditionally, students will do more experimental classes during this year.
- VisTex - "Visual and Textual Programming Languages" - The name of this study.
- VPL - "Visual Programming Language".

DECLARATION

I confirm that this is my own work and the use of all material from other sources has been properly cited and fully acknowledged.

Mark Noone

October 2023

Part I

INTRODUCTION & BACKGROUND

INTRODUCTION & BACKGROUND

Visual and Textual Programming Languages (VisTex) are the two primary language types used in programming education today. A Visual Programming language (VPL) is one which uses more than just text in their workflow (images, blocks, animation etc.). Some examples of VPL include Scratch, Alice and flowchart based systems. A Textual Programming Language (TPL) on the other hand is what most consider a "traditional" programming language, one that might be used in third level education or industry. Examples of TPL include Java, Python and C++.

In this thesis, a comparative study will be done examining the usage of visual and textual programming language in a second-level educational setting. In particular, Snap! will be used as the VPL and Java as a TPL. This will be discussed in more detail later. This study will involve the undertaking of a Systematic Literature Review, development and testing of curricula, examining the need for another "hybrid" option, the development of this and then data analysis and conclusions.

In this introductory chapter, some of the initial ideas and motivations will be described, research questions will be set and an overview of the remaining chapters in the thesis will be presented.

1.1 INTRODUCTION

It has been shown that the frontal lobes in the brain, home to key components of the neural circuitry underlying "executive functions" such as planning, working memory and abstract thinking, are among the last ar-

eas of the brain to mature; they may not be fully developed until halfway through the third decade of life [37, 88].

Computer programming is a skill that many people struggle with. The Computer Science Education Research team (CSEd) at Maynooth University (MU), with over 15 years of experience predicting programming performance, have pioneered PreSS#, which is an early warning system that predicts student performance based on three main factors; self-efficacy, mathematics performance and hours spent playing computer games. Despite the strong links between Computer Science and Mathematics and the proven importance of mathematics in the PreSS# system, most programming languages tend to be taught using a theoretical approach like Second Language Acquisition rather than the theoretical approach used in mathematics. This focus on abstract thinking may not be relevant given the brain development of the learner.

Programming ability rests on the foundation of knowledge about a language through comprehension. Visual programming languages are widely used to introduce young people to programming. However, the majority of higher education programming courses are delivered through text-based languages, due in part to the reliance in industry on these language types.

It could be argued that learning a programming language is similar to learning a second natural language. Krashen [44], a very influential researcher in the area of Second Language Acquisition, suggests in his Input Hypothesis that language acquisition is driven solely by comprehensible input at a comprehensible level. According to Krashen there are two independent systems of second language performance: 'the acquired system' and 'the learned system'. The 'acquired system' requires meaningful interaction in the target language. The 'learned system' results in conscious knowledge about the language, e.g. knowledge of grammar rules.

1.2 MOTIVATION

Computer Science as a third level subject has a history of volatility. While it has generally provided a very high graduate to employment ratio, it has often struggled with retaining students in the early years, particularly from first to second year [77]. According to a study undertaken by the Irish Times newspaper in 2016, "about one-third of (Irish) Computer Science students across all institutes of technology are dropping out after first year in college" [11]. This is something that researchers and educators continually try to mitigate.

In modern day society, the ability to code is a highly desirable skill. So much so that the current supply from third level institutes across the world does not meet the high demands of industry. One of the major issues is the low progression rates from first to second year in third level Computer Science courses with introductory programming courses proving to be a high contributing factor. This is something that needs to be addressed. One such way to address the issue is to get children involved and engaged with computing at young ages.

The motivation for this study relates closely to the concept that tackling the task of introducing students to Computer Science, and more specifically programming, should be done at an early age in order to best pique and maintain their interest. There are multitude of methodologies, language choices and educational styles when it comes to teaching a First Programming language (FPL). This thesis will attempt to suggest a "best approach" to FPL teaching, and to continually supporting our students while they are learning.

1.3 GOALS AND RESEARCH QUESTIONS

This project will attempt to determine if there is a particular programming language type that is an appropriate starting language for learners to enhance their likelihood of success. The premise is that different learners will perform better in one language over another with a number of key factors influencing this. In a study from the Anita Borg Institute [22], it was found that 18% of undergraduate degrees awarded in computer science were to women, a drop of 19% since 1985, due in part to misconceptions amongst younger females. Therefore, getting the language choice right, especially for females who are programming for the first time, is pertinent.

The following are the major goals of the project:

- Investigate the role that different types of programming languages (textual and visual) play when a learner is first introduced to programming. The goal is to determine if visual and textual languages result in different levels of comprehension and different levels of academic performance.
- Carry out a systematic analysis on the gathered data to determine if educational level (primary, secondary, tertiary) differences lead to a preference in language choice.
- Carry out a systematic analysis on whether gender differences result in a preference in language choice.

Three research questions will be addressed, namely:

1. Does the choice of language type affect the performance of learners as they learn programming for the first time? (RQ1)
2. Do visual programming languages, given their close interconnection with mathematics in terms of delivery, result in more effective and higher performance outputs than textual languages? (RQ2)

3. Is there an alternative approach to the traditional programming language types, and if so, how effective would this approach be? (RQ₃)

Using statistical analysis techniques learner data will be analysed. We will attempt to determine whether educational, gender and efficacy differences exist across the language types.

Additionally, does the absence of the iconic, as in textual languages make a difference? During the reporting phase, the results will be disseminated to appropriate parties, namely the Eye Movements and CS Education communities.

Upon completion of this work, verifiable curricula will have been created which can be reused by other teachers with ease. With the introduction of Computer Science to the Leaving Certificate, this work is very timely. We will be able to see if these curricula have any effect on learning versus the existing trial curriculum in place from 2018 – 2020 [83].

1.4 CONTRIBUTIONS

The main contributions of this thesis are:

- To examine the background research in this area in the form of a large-scale Systematic Literature Review.
- To determine the effectiveness of current teaching methodologies and tools within Computer Science Education at Secondary Level.
- To demonstrate the creation of a Hybrid Programming tool and its use cases. It will be shown that this tool fits the need for a "middling" level of difficulty in youth education.
- To create a "package" of material that includes the Hybrid Java Tool, a short course with all materials (class plans, slides, assessments) which second-level educators can take and utilise, and to create a bevy of examples to help with student support at third-level.

1.5 LIST OF PUBLICATIONS

The following is a list of all publications (and other relevant academic works, e.g. workshops) created by the author throughout the duration of the project. All of these publications can be found at the following google scholar link: https://scholar.google.com/citations?view_op=list_works&hl=en&authuser=1

- **Visual and Textual Programming Languages: a Systematic Review of the Literature** *published in* Journal of Computers in Education, 2018 [68]. This paper will form the backbone of Chapter 2. It had 106 citations at time of thesis submission.
- **First programming language: Visual or textual?** *published in* International Conference on Engaging Pedagogy, 2017 [67]. This paper will be discussed in Chapter 3. It had 12 citations at time of thesis submission.
- **First Programming Language - Java or Snap? A Short Course Perspective** *published in* Computer Science Education: Innovation and Technology, 2019 [69]. This paper will be presented in Chapter 3. It had 1 citation at time of thesis submission.
- **Hybrid Java: The Creation of a Hybrid Programming Environment** *published in* the Irish Journal of Technology Enhanced Learning [70]. This paper will be examined in Chapter 4. It had 5 citations at time of thesis submission.
- **Hybrid Java Programming: A Visual-Textual Programming Language Workshop** *held at* the UK and Ireland Computing Education Research Conference, 2019 [34]. This workshop will be explained in Chapter 4.

- **Creation of a Hybrid Programming Language** *published in* the Ed Tech Book of Abstracts, 2019 [61]. This abstract covers the initial idea of the Hybrid Java tool, and will be mentioned in Chapter 4.
- **An Overview of the Redevelopment of a Computer Science Support Centre and the Associated Pedagogy Impacts** *published in* All Ireland Journal of Higher Education, 2021 [72]. This paper looks at the redevelopment of the Maynooth University Computer Science Centre, and will be touched on in Chapter 5. It had 1 citation at time of thesis submission.
- **A Review of the Supports Available to Third-Level Programming Students in Ireland** *published in* All Ireland Journal of Higher Education, 2022 [71]. This paper studies the existing state of Computing Support in Ireland currently, and will be used as a metric for the need for Hybrid Programming as a support tool in Chapter 5. It had 2 citations at time of thesis submission.

1.6 ETHICAL APPROVAL

Ethical approval was applied for and approved by Maynooth University in the early stages of this project. The application number for this approval was SRESC-2017-037. This was a Tier 3 application, given that the project would involve working with children under the age of 18. This application had strict ethical protocols to ensure the correct management of data and to ensure the safety of all involved with the project. This ethical approval allowed us to engage with students in a wide range of settings including school visits, students under 18 attending Maynooth University for events (such as Summer Camps) and other scenarios with University students over 18 also.

1.7 THESIS OVERVIEW

This section will provide a brief overview of what will be discussed in each of the remaining thesis chapters.

Chapter 2 provides an overview of other work in the relevant areas. In particular, a Systematic Literature Review published in the Journal of Computers in Education will be discussed, along with some other relevant literature from more recent years. All of this literature looks at the existing state of visual and textual programming usage in education.

Chapter 3 describes the creation of the initial curricula in both Java and Snap!, and their deliveries to schools, summer camps, etc. Introductory sessions were first developed and tested for teaching at a summer camp, and to inform future course development. Afterwards, full 8 week short courses were developed and then tested. Finally, this chapter will highlight the need for a "middling" difficulty level of language, namely Hybrid Programming Languages.

Chapter 4 discusses the creation of a Hybrid Programming tool, "Hybrid Java". After development, an overview of the creation of a short course utilising the tool to mirror the Java and Snap! variants is presented. There was then a short testing phase for this tool including a summer camp session, a conference workshop and focus groups.

Chapter 5 will look at the conclusions of this project including some additional analysis elements (most analysis is done throughout the thesis). Hybrid Java, as the primary output of this body of work can be delivered as a complete package for educators, with accompanying curricula, class plans and more. It will be shown that Hybrid Programming fills in a gap in the pedagogy where a certain age of student could effectively be taught programming, and the efficacy of Hybrid Java as a support tool will be discussed. Future work that could be done in this area will also be examined.

RELATED WORK

As outlined in Chapter 1, this project is focused on the relationship between VPL and TPL in terms of learning a First Programming Language (FPL). As such, an in depth research process was deemed necessary to examine the existing state of research in this area. A Systematic Literature Review (SLR) was determined to be the best approach.

This chapter will begin by examining an SLR entitled “Visual and Textual Programming Languages: A Systematic Review of the Literature”, work on which was undertaken in the first year of this project. This paper was published in 2018 in the *Journal of Computers in Education* and has been quite successful, with over 100 citations to this date. What will follow after this SLR is a discussion of some additional publications which were published post 2018 and are relevant to the project, particularly around the concepts of hybrid programming and student support.

2.1 OVERVIEW OF SYSTEMATIC LITERATURE REVIEW

It is well documented and has been the topic of much research as well that Computer Science courses tend to have higher than average drop-out rates at third level, particularly so, for students advancing from first year to second year. This is a problem that needs to be addressed not only with urgency but also with caution. The required number of Computer Science graduates is growing every year, but the number of graduates is not meeting this demand, and one way that this problem can be alleviated is to encourage students, at an early age, towards studying Computer Sci-

ence courses. This paper presents an SLR that examines the role of visual and textual programming languages when learning to program, particularly as an FPL. The approach is systematic in that a structured search of electronic resources has been conducted, and the results are presented and quantitatively analysed. This study will provide insight into whether or not the current approaches to teaching young learners programming are viable, and examines what we can do to increase the interest and retention of these students as they progress through their education.

2.1.1 INTRODUCTION AND MOTIVATION

The usage of Computer Science is becoming much more prevalent in society today. In Ireland, a high number of technology companies choose to set up due to the quality of our third-level Computer Science graduates. However, the demand for a highly educated workforce is so great that the required numbers of graduates are not coming through the system to meet the demand.

According to a study undertaken by the Irish Times newspaper in 2016, "about one-third of Computer Science students across all institutes of technology are dropping out after first year in college" [11]. Similarly, the report discusses high drop-out rates among students progressing from first to second year in universities. It is well accepted that a high contributor to this lower progression rate is that incoming students to CS struggle to master fundamental concepts in their FPL modules [77].

What can we do to help solve these problems? There are two things that we believe must be considered. Firstly, we must educate students in the subject area of Computer Science at an earlier age so that they have an inherent interest when it comes to choosing a college/university course. In Ireland, steps have been taken at second level to address this. From the beginning of the 2017–2018 school year, Irish secondary schools will begin

teaching a short course in coding and other aspects of Computer Science to "Junior Cycle" students (approximately 12–15 years of age). In 2018, a full Computer Science option to "Senior Cycle" students (approximately 16-18 years of age) will be offered [20]. Teaching programming at an earlier age is becoming prevalent in many other countries too as the importance of Computer Science becomes more evident. The second thing we need to ensure is that we are teaching students correctly. This means, using the correct methodologies, using the right programming language and starting with the correct basis. All of these are challenges we aim to discuss in this paper, with a particular focus on language choice.

This paper contains the findings of a systematic literature review that was performed between October 2016 and March 2017. In it, two research questions were asked relating to Computer Science retention and what languages/tools we should be using to get the best performance/interest from students of various ages. These questions will help to inform us as to whether visual or textual languages or a hybrid of both is the best choice as a teaching language. It will also determine whether this choice has any bearing on future decisions about (and ability with) programming.

2.1.2 RESEARCH QUESTIONS

This study is focused on the relationship between language choice and learning to program. In particular, we want to discover what effects visual programming languages have on the learning process as well as how they compared with the performance of students using traditional text-based languages. To that end, the following research questions were defined:

1. Are there any benefits of learning a visual programming language over a traditional text-based language?
2. Does the choice of First Programming Language make a difference?
What languages are the best ones to teach?

2.1.3 BACKGROUND

Systematic literature reviews provide an unbiased and comprehensive approach to answering broad research questions. They offer a strict set of guidelines for how to extract information from relevant databases and process it in a detailed manner. This allows for an exhaustive analysis of available papers balanced with time required to process them. For the questions we will raise in this paper, we anticipated that a very large amount of material could be found. As such, a systematic review was the best approach for us to take.

With Computer Science making its way onto second level school curricula in Ireland, this research is very timely. The topics of language choice and the "best" First Programming Language and teaching approach is often asked, for example by Davies et al. [19], Eid and Millham [23], Ivanović et al. [35], Mannila and de Raadt [55], Quille et al. [77], but rarely answered. With this review, we aim to address the topics in a detailed manner and compile the opinions and results of many researchers.

Some other literature reviews that have informed this study include the work of Nolan and Bergin [64] on anxiety in programming. This well structured review along with Kitchenham's guidelines [38] provided a number of key methodologies for undertaking the review. Some other reviews were also read, but were a lot more specialised. Major et al. [52] looked at teaching introductory programming using robots, for example. Our review covers the full spectrum of introductory programming language opinions and visual/textual language comparisons. Its broad nature will be of great use to researchers and educators alike trying to decide what approaches to use in their classes.

2.2 METHOD

2.2.1 INTRODUCTION

The methodology used to perform this literature review is based on Barbara Kitchenham's approach, as modified by Keele [38]. This procedure was chosen due to its high focus on removing human bias from the search process. This ensures that, to the highest possible level of certainty, no false positive answers to the research questions will be found.

The method involves performing the following thorough steps, which are followed throughout this paper:

1. Identify the need for a review (see Sections 2.1.1 and 2.1.3).
2. Specify the Research Questions (see Section 2.1.2).
3. Develop a review protocol (see Sections 2.2.1-2.2.3).
4. Identification of research (see Section 2.2.4).
5. Study quality assessment (see Section 2.2.5).
6. Data extraction and synthesis (see Sections 2.2.6-2.3.3).
7. Report on found results (see Section 2.4).

2.2.2 RESOURCES SEARCHED

Between October 2016 and November 2016, searches were performed on numerous publication databases, namely, the ACM Digital Library, IEEE Xplore, the Education Resources Information Centre (ERIC) and Google Scholar. These particular databases were chosen due to the high level of regard achieved in their respective industries. ACM and IEEE both contain a very wide range of Computer Science papers. ERIC is primarily

an educational database, which is also important for this study. Google Scholar was used as a backup database to ensure that all important papers were found.

2.2.3 SEARCH TERMS

The methodology used to perform these searches involved taking each primary term and searching for it in each database. If the primary search term alone yielded less than 400 results, all of those papers were extracted for later filtering. If the search was too broad, it was combined with each respective secondary search term and those results were then chosen for filtering.

Due to the broad nature of this study, an extensive list of search terms was used. This list included 10 primary search terms and 16 secondary search terms. These terms were chosen in order to cover a broad spectrum of age groups and language types. The goal was to be as fully comprehensive as possible. The terms used were:

Primary Terms: *"Visual Programming", "Iconic Programming", "Visual Versus Textual", "Visual vs. Textual", "Graphical Programming", "Textual Programming", "First Programming Language", "Introductory Programming", "Novice Programmers" and "Programming Education"*

Secondary Terms: *"Scratch", "Alice", "Primary Education", "National School", "Elementary School", "First Level", "Secondary Education", "High School", "Second Level", "Third Level", "College", "University", "CS1", "Kids", "Children", "Education" and "Teaching"*

2.2.4 DOCUMENT SELECTION

The initial searches on each database produced a very large number of results. In total (combining the amount of responses for each pair of search

terms), ACM returned 2252 papers, IEEE returned 1713 papers, ERIC returned 486 papers and Google Scholar returned 655 papers. This is a total of 5106 potential papers (though there may be overlap between the sources). The first step to minimise these numbers was to perform a "Title Filtering" on the related papers. This removed any titles where it was immediately obvious they would have nothing to do with the research questions posed. This process cut the number of possible papers down to 661 (all sources merged).

After obtaining full copies of the filtered papers, the next stage involved an "Abstract Filtering". This was performed in much the same way as "Title Filtering", but the full abstract of each paper was read. If the content of a paper's abstract did not relate to either of the research questions, it was excluded. This process was undertaken in December 2016. After its completion, 124 possible papers remained.

At this stage, each paper needed to be read in full. Inclusion and Exclusion Criteria were defined as well as a quality assessment (see Section 2.2.5) undertaken at the same time during this phase. The requirements for a paper to be included were that the paper:

- Focused on the topic of at least one research question.
- Focused on specific programming languages, either visual, textual, or a combination of both. Specifically, a study/verification needed to be undertaken with students.
- Detailed the learning of a First Programming Language.
- Was NOT grey literature / blog / a PhD thesis.
- Did NOT examine students under the age of 10.

Each paper had to meet all of the applicable above requirements. For papers that were borderline, author discretion was used based on their content. For example, some papers were kept, which described the development of certain VPL tools despite not detailing any studies.

2.2.5 QUALITY ASSESSMENT

After reading each paper in full, a final decision was made as to whether it would be included in the results section of this study. For each paper, a rigorous quality assessment protocol was applied. This process was undertaken manually during the reading phase.

Kitchenham [38] lists 18 possible quality assessment questions in her guidelines. For this study, a small subset of four questions were chosen. These were:

- How credible are the findings?
- If credible, are they important?
- Is the scope of the study sufficiently wide? (modified)
- How well can the route to any conclusions be seen?

For each of these questions and each paper, a score was applied. A score of 1 was given if the paper completely satisfied the question (Y). A score of 0.5 was given if the paper partially satisfied the question (P). A score of 0 was given if the paper failed to satisfy the question (N). Upon first full read through of a paper, these questions were answered. This involved a certain amount of objectivity. For Q₁ and Q₂, the credibility of the papers had to closely relate to the research questions. For Q₃, small studies that contained very little content or had very small experimental groups were excluded. For Q₄, it was important that the paper had a logical route to its conclusions and did not make any assumptions. This information was all double checked and adjusted where necessary, before making the final decision on the included papers.

For a paper to be included, it must achieve a score of at least three (out of four). This ensures that a paper is of sufficient quality without rashly excluding one that misses a single element. Between this Quality

Assessment (QA) and the inclusion criteria, a final list of 53 papers were selected for inclusion. The full list of accepted papers as well as details of their QA scores are presented in Figure 2.1.

TITLES OF THE ACCEPTED PAPERS	YEAR	Q1	Q2	Q3	Q4	Score
A Minimal, Extensible, Drag-and-drop Implementation of the C Programming Language	2011	P	Y	Y	Y	3.5
A Snapshot of Current Practices in Teaching the Introductory Programming Sequence	2011	Y	Y	P	Y	3.5
A Study of the Difficulties of Novice Programmers	2005	Y	Y	P	Y	3.5
An Objective Comparison of Languages for Teaching Introductory Programming	2006	Y	Y	Y	Y	4
B#: The Development and Assessment of an Iconic Programming Tool	2006	P	Y	Y	P	3
Block-C: A Block-Based Visual Environment for Supporting the Teaching of C	2013	Y	Y	Y	Y	4
Blocks, text, and the space between: The role of representations in novice programming	2015	P	Y	Y	Y	3.5
C++ or Python? Which One to Begin with: A Learner's Perspective	2014	Y	Y	Y	Y	4
Catroid: A Mobile Visual Programming System for Children	2012	P	Y	Y	Y	3.5
Code Club: Bringing Programming to UK Primary Schools Through Scratch	2014	Y	Y	Y	Y	4
Comparing Textual and Block Interfaces in a Novice Programming Environment	2015	Y	Y	P	Y	3.5
Computer Gaming at Every Age: A Comparative Evaluation of Alice	2008	Y	P	Y	P	3
Define and Visualize Your First Programming Language	2008	P	P	Y	Y	3
Determining the Effectiveness of the 3D Alice Programming Environment	2007	P	Y	Y	Y	3.5
Does the Choice of the First Programming Language Influence Students' Grades?	2015	Y	P	P	Y	3
Dual instructional support materials for introductory object-oriented programming	2010	Y	Y	P	Y	3.5
Empirical Comparison of Visual to Hybrid Formula Manipulation	2014	P	Y	P	Y	3
Evaluating Visual Programming Environments to Teach Computing	2013	P	Y	Y	Y	3.5
Filling the Gap in Programming Instruction	2009	P	Y	P	Y	3
From Scratch to "Real" Programming	2015	Y	Y	Y	Y	4
From Scratch to Patch: Easing the Blocks-Text Transition	2016	N/A	N/A	N/A	N/A	N/S
Habits of Programming in Scratch	2011	P	P	Y	Y	3
How Kids Code and How We Know: An Exploratory Study on the Scratch Repository	2016	Y	N	Y	Y	3
How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch	2010	Y	Y	Y	Y	4
Introductory Programming: Let Us Cut Through the Clutter!	2016	Y	P	P	Y	3
Language Migration in non-CS Introductory Programming	2015	Y	Y	Y	Y	4
Learning to Program is Easy	2016	Y	Y	Y	Y	4
Minimizing to Maximize: An Initial Attempt at Teaching Introductory Programming Using AI	2011	Y	Y	P	Y	3.5
Pedagogy and Processes for a Computer Programming Outreach Workshop	2010	Y	Y	Y	Y	4
PILeT: An Interactive Learning Tool To Teach Python	2015	P	P	Y	Y	3
Programming web-course analysis: How to introduce computer programming?	2014	P	Y	P	Y	3
Python and Roles of Variables in Introductory Programming	2007	Y	Y	Y	Y	4
Python in CS1 - Not	2015	P	Y	P	Y	3
Python Prevails	2009	Y	Y	P	Y	3.5
RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving	2005	Y	Y	Y	Y	4
Scratch: A Way to Logo and Python	2015	Y	Y	Y	Y	4
Some Deficiencies of C++ in Teaching CS1 and CS2	2003	Y	Y	Y	N	3
Student opinions of Alice in CS1	2008	P	P	Y	Y	3
STUDY OF ALICE: A VISUAL ENVIRONMENT FOR TEACHING OBJECT-ORIENTED PROGRAMMING	2012	P	P	Y	Y	3
Teaching Experiences with Alice for High School Students	2011	Y	Y	N	Y	3
The Design of Alice	2010	Y	Y	P	Y	3.5
The Effect of Integrating an Iconic Programming Notation into CS1	2005	Y	Y	P	Y	3.5
The Scratch Programming Language and Environment	2010	Y	Y	Y	Y	4
To Block or Not to Block, That is the Question	2015	Y	Y	Y	Y	4
Use of cutting edge educational tools for an initial programming course	2014	Y	Y	Y	Y	4
Using Alice 2.0 As a First Language	2009	Y	Y	Y	Y	4
Using Alice in CS1: A Quantitative Experiment	2010	Y	Y	Y	Y	4
Using Alice in Overview Courses to Improve Success Rates in Programming I	2008	Y	P	P	Y	4
Visual Learning Environments for Computer Programming	2011	N/A	N/A	N/A	N/A	N/S
What about a Simple Language? Analyzing the Difficulties in Learning to Program	2006	Y	P	Y	Y	3.5
What is a Good First Programming Language?	2004	Y	Y	Y	Y	4
Which Introductory Programming Approach Is Most Suitable for Students	2012	Y	Y	P	Y	3.5
Why Complicate Things?: Introducing Programming in High School Using Python	2006	Y	Y	Y	Y	4

Figure 2.1: List of accepted papers, their year and their QA Scores (Y = 1, P = 0.5, N = 0, score of 3 out of 4 required)

2.2.6 DATA EXTRACTION AND SYNTHESIS

Throughout the process, all important information was extracted and stored in a number of Microsoft Excel documents. For each search in the initial stages of the study (before coming to the final 53 titles), each individual database search was stored in its own Excel sheet. After title filtering of each document occurred, a master list of titles that passed was created. This document was the primary one used from then on. During the abstract processing stage, papers were highlighted in green if they were to be read in full, highlighted in yellow if they needed further examination and highlighted in red if they were deemed unrelated to the study.

At this point, the 124 abstract-filtered papers were split into a new tab on the excel document. Here, the full title, source, publication location, which research question the paper covered and any additional notes were stored. A similar highlighting system was used on this tab as well, when papers were deemed to have failed the quality assessment checks or when they did not cover any research questions.

As well as the use of Microsoft Excel, Mendeley reference manager [58] was used to store every full paper and summaries from the reading of that paper. This tool was chosen as it allowed one to keep track of which papers had been read, and to "favourite" those ones that passed the quality assessment. A folder structure was used to separate each set of papers into their initial sources (ACM, IEEE, ERIC, Google Scholar). Mendeley also makes it very easy to see where and in what year a paper was published. At the writing stage, Mendeley allowed for easy generation of the list of references for BiBTeX.

2.3 DATASET

2.3.1 TYPES OF STUDIES

Many of the included studies involve quantitative experiments detailing the results following the teaching of some form of curriculum using a given language. Some authors also used a mixed model approach for data collection (feedback surveys / questionnaires as well as tangible results). Some of the accepted papers were borderline in their Quality Assessment scoring but were still accepted due to the fact that the original developers wrote it, despite not containing any study.

2.3.2 TIMELINE OF CHOSEN PUBLICATIONS

Programming, and in particular programming languages, are a very volatile thing. What may be relevant today might not have been even ten years ago. As such, it was decided to set a hard timeline for acceptable papers. Any paper that passed all other checks and was written any time after 2002 was kept for analysis. This gave a 15-year range for acceptance. This timeline provides a high chance for papers to still be relevant without too many irrelevant studies being kept. Although a lot can change in a 15-year range in terms of Computer Science we felt that in the domain of Computer Science Education there would not be as dramatic a change, as techniques used 15 years ago may still be used today.

The profile of when the accepted papers were published is shown in Figure 2.2. A large number of these are from between 2008 and 2018. Additionally, 33% were published after 2014. This tends to suggest that our 15-year range is a valid range based on our research questions.

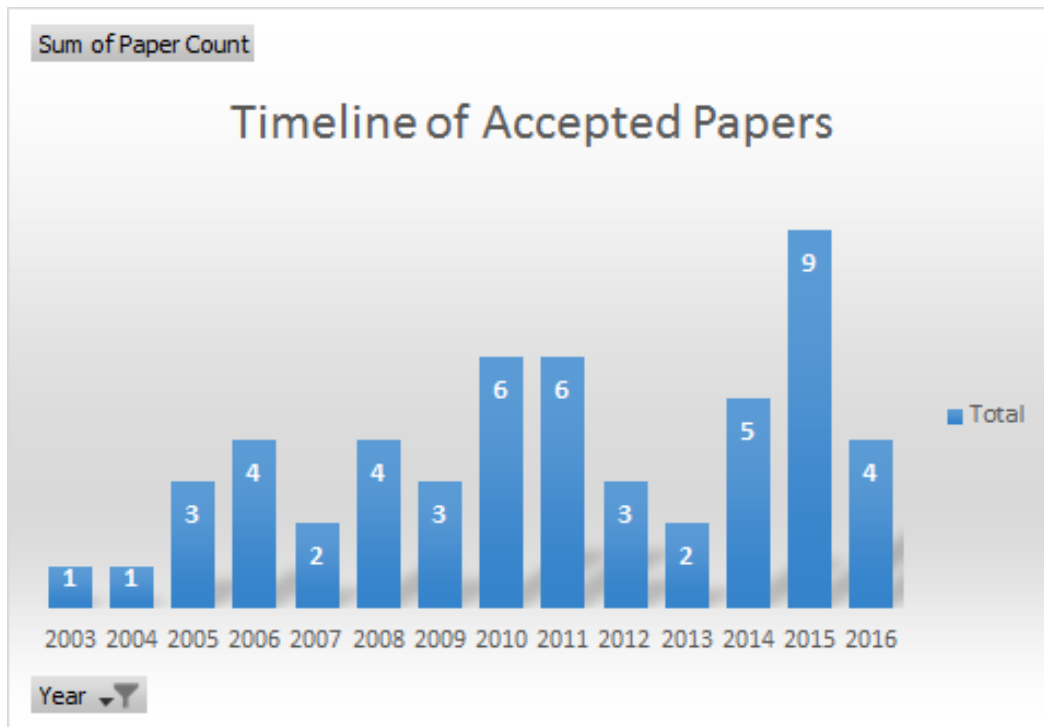


Figure 2.2: Timeline of Accepted Papers

2.3.3 DATA SOURCES

The largest quantity of accepted papers came from the ACM database. Figure 2.3 shows the breakdown of which database the 53 accepted papers were found in. For comparison, during the reading stage (before the final filtering), 73 papers were retrieved from ACM, 36 from IEEE, seven from ERIC and eight from Google Scholar. All accepted papers were disseminated via a conference or a journal.

2.3.4 DATASET DISCUSSION

This study was performed systematically in order to ensure the answers to the research questions were comprehensive, unbiased and valid. As discussed in Section 2.3.2, a 15-year range of acceptable papers was set. This allows for examination of the evolution of teaching methodologies

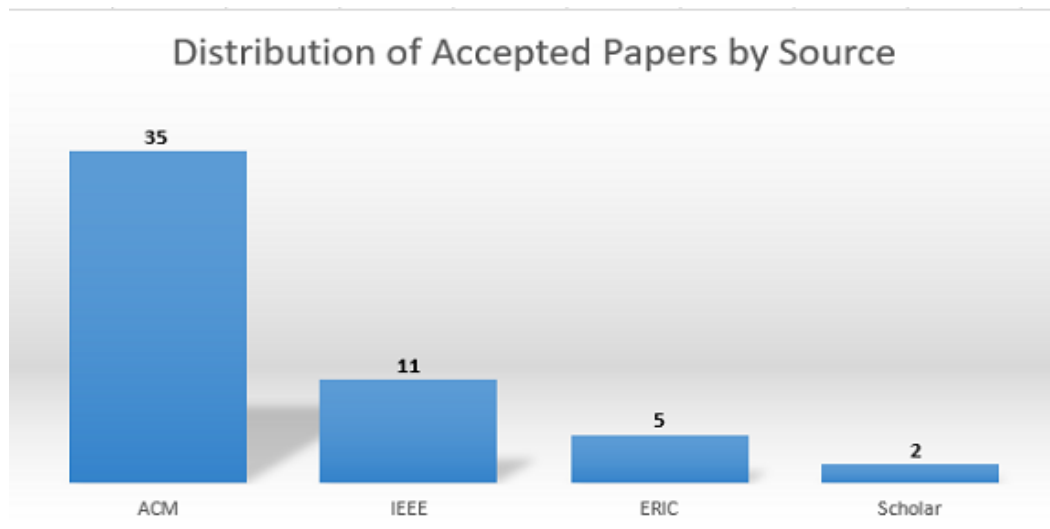


Figure 2.3: Distribution of Accepted Papers by Source

and languages within recent history. With this timeline, we can see what has changed and perhaps more importantly, what has stayed the same. Similarly, with the inclusion of secondary search terms, we are able to look at different levels of education (primary, secondary, tertiary). As will be seen throughout the results, different levels of education tend to converge towards certain language types or teaching styles. This trend appears to be universal. Even though the final papers are from different countries, their results have key similarities. Through this systematic process and with this information in mind, we believe the results returned were of high quality.

2.4 RESULTS

In this section, analysis of the 53 approved papers will be performed. This analysis will involve a second full read-through of each paper (having first read them in the QA stage along with the papers rejected at this stage). While reading the papers, key points will be extracted and noted down in the Mendeley Reference Manager notes section. Twenty-nine papers inform the first research question, with 24 informing the second. It is important to note that the level of contribution that some papers will have

will be greater than that of others. Some papers that were included may have only raised one strong point, but if it was a point worth making, it was included.

2.4.1 RESEARCH QUESTION 1: ARE THERE ANY BENEFITS OF LEARNING A VISUAL PROGRAMMING LANGUAGE OVER A TRADITIONAL TEXT-BASED LANGUAGE?

As a reminder, a VPL is any programming language where users are able to manipulate the underlying code in some graphical fashion rather than the traditional text-based approach. Some examples of widely used VPLs today include Scratch [53] and Alice [16]. Before discussing these, let us look at a more traditional approach.

Flowchart approach

A more traditional approach to visual languages came in the form of using flowcharts. Most "modern" languages don't use this methodology, but for completeness, results from the search that covered this style of design are included here. Greyling et al. [28] discuss the concept of the B# language that they developed. B# uses an iconic flowchart approach to give students two options in developing their code, via drag-and-drop of code pieces, or by the traditional textual approach. As a user is building a flowchart, code is generated in parallel in C++, Pascal or Java. Flow chart icons are connected by lines, making the ordering and structure of the program obvious. While this methodology worked well in the early stages of a CS1 course, the authors note that "unfortunately initial evaluation sessions showed that many students did not succeed in developing adequate coding skills while working with B#". Another example of a flowchart-based VPL is RAPTOR [12]. RAPTOR's goal is to improve problem solving skills while reducing the emphasis on syntax. It uses a similar approach to B#, except without

a textual counterpart. Different elements are built up via drag-and-drop, ensuring that program structure is correct. From a study of 959 test subjects, the authors found that students prefer to express their algorithms visually, with 95% choosing to use a flowchart on the final exam over a textual language. This lends credence to the concept of a VPL, allowing more advanced tools to be built.

Scratch

Scratch was developed by the Lifelong Kindergarten Group at the MIT Lab. Scratch's primary goal is to give young people an accessible way to introduce themselves to programming. It uses a "drag-and-drop" approach, where users drag "blocks" from a predefined list of commands into a script area. These blocks essentially fit together and make syntax errors impossible. This reduces the mental load of the student and allows them to focus on concepts rather than becoming bogged down with the technicalities of the language.

There have been many studies performed to verify the efficacy of Scratch as a teaching tool for young audiences. Tangney et al. [90] used Scratch in a project-based after-school workshop for 15–16-year-old students. Their goal was to see if they could engage students at an early stage and put them on a path to CS courses. They had 39 students with high maths performance attend the workshops, and the results were favourable. The majority of participants enjoyed the content of the workshop, and the authors noted that "participants left with a favourable and more realistic impression of both CS courses and the CS profession".

Lewis [49] performed a brief comparison study of Scratch and Logo. This comparison focused on programming concepts. Lewis noted that "the Scratch environment provided a relative improvement in learning outcomes for students learning the construct of conditionals". Meerbaum-Salant et al. [57] point out, however that bad habits can still happen even

in a VPL. They discuss how, if these are not caught, they could actually affect performance in later textual language courses. We as teachers still need to portray good methodologies for students to demonstrate success.

These papers cover just a small sample of the work that has been done verifying the usefulness of Scratch, and it is well established in its field. Perhaps the clearest indicator of Scratch's success is the sheer number of projects that are connected on their website with 136,157,507 total projects shared when checked in August 2023 (up from 20,695,116 in March 2017 at time of original publication) at the MIT Media Lab [51].

Alice

Alice, while working in a very similar manner to Scratch has somewhat different targets. The developers Cooper [16] aim was to provide students with a "serious pre-CS1 programming experience". Alice allows users to build up a functional animated world using drag-and-drop code blocks. It contains a "Virtual World Editor", which allows users to lay out a set of objects in 3D space. All the underlying code is still dealt with in a drag-and-drop manner after this initial visual setup. Cooper has noted that, "opposed to algorithm animation, program visualisation systems allow the student to create their own animations". One of Alice's biggest differences from Scratch is that it supports the Object Oriented approach to programming, although in a limited fashion.

Among the studies that investigate Alice as a VPL is the study performed by Parker [73]. This study involved a week long workshop with 15 high school students, with a goal of encouraging them towards a degree in Computer Science. The participants did connect with the course, with many stating they enjoyed the video game development aspect and their enthusiasm was encouraging. Larger studies include that of Sykes [89] which focused on the Objects First approach that Alice allows. A CS1 course based on Alice was directly compared to two iterations of a CS1

course based on C. The author accepts that it is harder to perform computations with Alice, and that the lack of visible syntax could be an issue in later courses, but at the same time, it is noted that Alice makes it very easy to understand the fundamentals, which is exactly what one would want from a CS1 language. The Alice students outperformed the control groups significantly in the exams.

Johnsgard and McDonald [36] present another success story using Alice. They were experiencing low grades in their C++-based CS1 course. They implemented a CSo (pre-CS1) using Alice. The average grade in the following year of C++ rose to 70.3% from 46.4% in the previous year, a statistically significant increase. Students also expressed their enjoyment of the Alice course. Anniroot and de Villiers [4] found that Alice helped their students better their problem solving abilities and gave them a stronger understanding of programming concepts. They found that through "quantitative analysis of the closed-ended questions, 81% of experimental learners were found to agree that the visual effects in Alice provide meaningful contexts for understanding classes, objects, methods, and events".

Of course, not everyone can have a positive experience. Garlick and Cankaya [25] felt that while Alice was a nice tool, students didn't necessarily focus on programming concepts enough and were more just enjoying building a world. The alternative they offered was a "pseudo code" CSo — in other words, a course that focused purely on algorithmic design with no programming language used at all. The participants in the course had similar results in Alice and pseudo code assignments, with a worse result on the Alice exam, and the Alice group declared less confidence as well via collected survey responses. Given the large number of positive feelings towards Alice in multiple studies (a small sample of which are discussed here), Alice's failings in this course could possibly be attributed to the teaching techniques employed or the fact that pseudo code might not be comparable to a full language (i.e. programming is more complex).

After School Clubs

After school clubs, such as CoderDojo, often teach Scratch or a similar VPL as one of their modules. The authors have had the pleasure to watch some students progress from a local CoderDojo to the CS1 course on offer at their university. These students often have a strong advantage and perform very highly at college level.

In 2014, Smith et al. [86] analysed the effect of 1000 Code Club locations in UK schools. They chose to primarily teach Scratch due to its "known ease of use by primary school children". Step by step instructions were given at first, but as the children progressed, they were expected to create their own scripts and make their own choices. Surveys were collected at the end of the year, with some positive results. The children had demonstrated a good knowledge of some programming concepts. Smith analysed 22 final projects at random and discovered that "(some) children coped remarkably easily with difficult programming concepts". To push this idea further, Seals et al. [82] had 8–9 year olds working on assignments in Alice that were of an equivalent difficulty level to that which 18–19-year-old college students would undertake. This is quite remarkable. Something must be helping these young people understand things so clearly.

Blocks

The most prevalent thing noticeable in the analysis of both Alice and Scratch is that the block-based approach to teaching seems to resonate strongly with younger cohorts. This may not be a surprise due to the fact that a large number of people are believed to be visual learners, and young students generally have more creative minds. What else is it that makes this blocks approach so strong?

Sandoval-Reyes et al. [81] asked themselves this same question. They performed an analysis of three major block programming environments:

Scratch, Alice and App Inventor, while also looking at Greenfoot. They put forward the idea that this kind of environment provides such strong pedagogy due to "connecting users with their interests", direct mapping of ideas to instructions on screen and the hiding of unnecessary complexities from the novice user. The blocks approach can work in all kinds of environments, as demonstrated by Catroid [85]. Catroid allows users to develop programs directly on their phone, they can even develop controllers for other devices such as the Lego Mindstorms NXT robot. For many young people, this is a really exciting prospect. Price and Barnes [76] undertook an interesting study that "seeks to isolate the effect of a block interface on the experience of novices". Half of a group of middle school students were assigned to a "block" group, while the other half were assigned to a "text" group. During a half-day session, the students were given programming exercises to do in their respective environments. At the end of the session, data via surveys and logged interactions with the tools were analysed. The block group performed better than the text group, and they also had a slightly higher self-efficacy at the end of the session as well. This provides strong data pointing to blocks having some positive effect on the performance of students.

Transition from Visual to Textual

A number of researchers tend to agree that, while Visual Programming is a very strong concept for introductory courses, it has a tendency to fall short when the time comes to deal with complex topics. Some researchers agree that VPL are more of a "gateway" to learning textual languages.

Dorling and White [21], for example, examined a scenario in which graphical languages were taught "in conjunction with, not in place of, text-based programming languages". This study involved beginning a ten week curriculum using Scratch and algorithmic concepts, and working towards introducing Python. By showing students Python code side-by-side with

Scratch code, their understanding of the textual language was made much stronger. It was noted that "this transition process has been a factor in an increased uptake of Computer Science".

Giordano and Maiorana [26] looked at a similar approach that involved using multiple languages in the same course. This study was done with a group of 28 10th grade students in Italy between the ages of 14 and 16. The course was taught over 28 weeks. The early stages used Scratch and similar tools. According to the authors, "This is done in order to relieve students from the burden of learning all the syntax-related details and instead to let them focus on the concepts and problem solving skills". In later weeks, the C language was introduced to give students a proper experience with a textual language. The results were positive across the course. In particular, when the C language section began, students made less errors than would normally be expected upon first exposure. This shows that using the VPL first has allowed students to familiarise themselves with the concept of programming.

Weintrop and Wilensky [93] asked "To Block or not to Block". They wanted to determine if high school students found the blocks approach easier than the textual approach and why. To examine this, they taught a course using five weeks of Snap! and five weeks of Java. Fifty-eight percent of students found Snap! easier to use. Some participants reported (via a survey) that the blocks approach was easier to read and the shapes were also determined to be helpful. There are also some drawbacks, for example, blocks languages are less powerful. At some point, you will hit a barrier you cannot pass with the tools you have. The author suggests an interesting point: "Why not add a similar browsability to introductory text-based environments"? Multiple authors have examined this concept in detail; we will call this type of language a "Hybrid Language".

Hybrid Languages

These languages involve either an interface that shows both visual and textual elements at the same time, or the merging of a textual language into a blocks style interface. Weintrop [92] describes this idea in detail. A pilot study involving the use of block-based, text-based and hybrid programming environments was performed in order to compare the effects of all the three. The analysis (while ongoing) showed promise for the concept of hybrid languages. There are certainly benefits to both visual and textual approaches, hence why combining them might have the best possible effect on young learners.

The earliest found study on the concept of hybrid languages was undertaken by Cilliers et al. [14]. The authors wanted to examine what effect the integration of an iconic notation into a textual development environment would have. They recognised that "visual programming notations offer benefits over textual programming notations" while also recognising that VPL were not a standalone solution. In order to verify their thoughts, they implemented a course that compared a control group using exclusively PASCAL as their language of choice to a study group using B# as their language of choice (first mentioned in Section 2.4.1.1). B# was designed with the intention of only being valid for the initial stages of CS₁, after which the students would progress to a purely textual approach (once they became familiar with the concepts). Participants using B# performed statistically significantly better upon final assessment, particularly amongst students deemed to be high risk. In other words, it helped those who would have struggled quite a lot with the traditional approach, without having a negative effect on those who didn't necessarily need the extra help.

Koitz and Slany [41] also asked a similar question when comparing Scratch and "Pocket Code". Pocket Code is a mobile development environment that uses a mix of textual programming and Scratch elements.

After performing four tasks using both languages (17 participants), the results showed that Pocket Code's hybrid approach was more beneficial than the purely visual approach of Scratch.

In recent years, there has been a large amount of research into hybrid languages that use existing textual languages merged with a block style model. In 2011, Federici [24] combined the C programming language into a Scratch-like blocks system. This was made possible by a Scratch mod known as BYOB. This tool allows for the creation of custom blocks and functions within the Scratch environment. The author implemented blocks such as `printf`, `scanf`, integers, etc. The goal of this research was to "lower the student effort required in advancing from introductory tools, such as Scratch, to regular programming languages, such as C". The tool they designed was named blockC. Kyfonidis et al. [45] developed a very similar implementation called Block-C. The authors wanted a visual methodology without skewing away from teaching a "general purpose programming language". This was tested with a two hour tutorial and 32 first year university students. The Block-C group performed much stronger than a textual C group.

This concept expands past just the C language of course, with tools existing to allow blocks to be modified for any language. Matsuzawa et al. [56] provide an example of a Java-based blocks language. With their study, the tool allowed for direct translation between blocks and text-based Java. The author posited that students should begin with this blocks approach, and gradually move towards a fully textual environment. Those who continued to use Blocks for the entirety of the study turned out to be the weaker students, showing that the visual approach might have a threshold. Finally, Robinson [79] looked at a tool called "Patch" which combined Scratch with elements of Python. Again, the goal was to minimise the gap between visual and textual languages in young learners. No verification

was done on this particular tool, but it follows much the same patterns as the other tools discussed.

Academic Benefit

Based on all of the discussed approaches to visual programming, can we see any academic benefit to teaching such a language? It is well established that younger students can take a tool like Scratch and really thrive using it, but what about second level students? Cheung et al. [13] believes that there are key age groups for each type of language to be most successful. High school students respond better to textual programming, students younger than 14 find VPL is the most beneficial, and those from around 15–17 would most benefit from a hybrid environment. They ran summer workshops that back up this fact on hybrid languages. Andujar et al. [3] also wanted to see if there was any benefit of teaching high school students visual programming. They came to a similar conclusion as Cheung et al. [13], in that teaching Alice to them did not provide any significant benefit over other languages. However, Alice did increase the retention rate of students. This could come down to the enjoyment of using a VPL, and this effect could hold true for all courses using VPL.

Conclusion

From the literature, it is clear that Visual Programming Languages present many benefits over traditional text-based programming languages. As presented in Section 2.4.1.8, all types of language have their benefits.

There are many factors that lead to VPL being beneficial. For one, they are highly accessible. They are available both online and as downloadable tools for free. This makes it easy for a curious individual to find and give it a go. If we search "Programming for Kids" on Google, Scratch among other educational programming websites are indexed on the first page of results. For those who have a more general interest in computers, they

might attend an instructor-led after-school club (see Section 2.4.1.4). This gives them an introduction to beginning programming. Clubs such as CoderDojo encourage young kids to get involved with programming at an early age. The authors have been involved with such a club for a number of years. From our own perspective, we have seen multiple students progress to our CS1 course and perform at the top end of the class having learned a VPL first.

The familiarity of VPL is another key element. Many young learners will watch animated shows or play games, and, from our experience, they love the feeling of seeing their own ideas and animations come to fruition. Similarly, the presentation style often resonates with young students. The WYSIWYG / drag-and-drop approach to learning fosters creativity in a way that might not be possible with text-based languages. You can experiment more and easier when you have a sprite visible showing the outcomes of what you have created. The level of knowledge overhead with this approach is much lower.

As presented in Section 2.4.1.5, middle school students both performed better and had higher self-efficacy when using the blocks-based approach to programming [76]. In Section 2.4.1.6, multiple examples are provided showcasing the effects learning a VPL can have on learners as they progress towards a TPL. Having this knowledge and skill is a key factor.

For the purpose of this research question, we can conclude from the above evidence that teaching a Visual Programming Language or hybrid programming language to the right age group can have a very positive effect on their interest and retention in Computer Science. This might also have a positive effect on the retention rates in college level courses.

2.4.2 RESEARCH QUESTION 2: DOES THE CHOICE OF FIRST PROGRAMMING LANGUAGE MAKE A DIFFERENCE? WHAT LANGUAGES ARE THE BEST ONES TO TEACH?

In this section, we will examine whether or not the choice of First Programming Language (FPL) has a significant effect on outcome in an introductory programming course. Specific languages will be examined, and conclusions will be drawn.

A "Good" First Programming Language

The first question that must be asked is, what constitutes a good FPL? Gupta [29] examined this question in detail and believes that the choice of FPL is a big decision, one that will have a "profound impact" on future learning. He concluded that the "ideal" language will depend on the age of the target audience among other things. He posits that it is important to "focus on problem based learning, allowing students to focus on techniques rather than on the language syntax itself". Some of the important elements of a FPL that he discusses are:

- The language should have a clear and intuitive syntax.
- The language should cover all common syntactic and semantic constructs.
- The language should be consistent in its handling of things like errors and provide meaningful error messages.
- The language should not have excess brevity (functional languages) or excess verbosity
- The language should be customisable and allow for changing needs over time.

Ateeq et al. [6] examined this research question specifically in the context of C++ or Python. They agree with many of Gupta's definitions of important features in a FPL, particularly regarding notation overhead, verbosity, target audience and use of simple syntax. They found that Python met many of these requirements (which will be discussed in more detail in Section 2.4.2.4). To test this, a study was run with CS1 students comparing their thoughts of both languages (via surveys). In most criteria, Python was held in a higher regard which further proves that the checklist above is an important factor. Mannila and de Raadt [55] also examined objectively what languages might be the best to use as a FPL. A list of 17 criteria was developed by educational language writers. Eleven well-known languages were examined against these criteria. Those that came out on top were Python (meeting 15 of 17 criteria), Eiffel (15/17) and Java (14/17).

Ranade [78] furthers these ideas by talking about his college's use of a C++ language that has been graphically augmented with a logo turtle style view. He believes that the focus of a FPL course should be taken away from syntax / semantics and directed towards the more fun aspects of computing as well as algorithmic thinking. One key example from their work was the use of this tool to demonstrate how recursion works using a visual tree that keeps splitting its branches in two as the structure grows deeper. This tree was drawn in real time. The author believes the visual nature of this allows for easier comprehension of complex concepts.

Difficulties with CS1

The attrition rates in CS1 courses are often quite high; there must be some attributing factors to this. Lahtinen et al. [47] used a survey to help discover what some of the key difficulties students experienced were. This survey was distributed to 559 students and 34 teachers from a group of multinational universities and colleges. Most were students of C++, but some had used others as well. Most of the key results agree with the ideas

of a good FPL in Section 2.4.2.1. Recursion, pointers, abstract types and error handling were determined to be the hardest concepts. Getting familiar with structures, syntax, algorithm design and how to divide into functions / classes are the elements that need to be done to be successful. In general, however, "the teaching language did not seem to affect the learning situations". Mannila et al. [54] also analysed Java and Python programs with the intent of determining the difficulties the writers experienced. Sixty programs written by 16–19-year-old novices were used. Common errors that were found involved poor error checking, bad use of variable types in Java and mismatching brackets in Java. The authors also agreed that Python had the potential to be a strong CS1 as it had less errors than the verbose Java code.

Luxton-Reilly [50] on the other hand posits that learning to program is actually an easy endeavour, and that we, as educators, expect too much from students in a CS1 module. Could we in fact be scaring people away from CS by overestimating how much can be learned in a short period? He raises the point that "There is nothing intrinsic to the subject that makes it difficult to learn, but rather our subjective assessment of how much a student "should" be able to achieve by the end of the course that determines the difficulty". This is something to keep in mind as we discuss FPL next.

The Commonly Chosen Languages

Davies et al. [19] conducted a survey of 371 institutions in the US in 2011. This will give a reasonable snapshot of FPL choice in general in this region. They broke the survey down into CS0, CS1 and CS2. The most commonly used CS0 language was Alice, followed by Python and Java. For CS1, the primary focus for this paper, Java was the most used with 48.2% of institutions adopting it, 28.8% offered C++ and 12.9% offering Python. Alice only maintained 4.3% usage as a CS1 language. For CS2, the usage of Java strengthened further to 55.8%, with C++ also increasing to 36.1%. All

other languages of note fell to usage rates of below 4%. This tells us that advanced topics are much better suited to object oriented environments. In the following sections, the efficacy of a subset of these languages will be discussed in detail.

Textual FPL

By far, the most commonly found language in the literature was Python. This may come down to the fact that Python is reasonably new when compared to C++ and Java. These languages have already cemented themselves in the pedagogy of CS1. Python still needs to convince educators of its efficacy, however it could be a strong choice for a FPL.

Grandell et al. [27] discuss their attempts at a Python-based FPL course with high school students. They recognised that Python met a lot of the requirements that make it "easy" to learn. They implemented a curriculum and tested it on 42 boys. Eighty-five percent of students passed the course, with an average grade of 77.1%. This was compared to a similar High School course in Java they previously taught, with the Python average being much higher. Survey results showed that students strongly agreed that Python was easy to learn. Nikula et al. [63] also considered Python to be an easy language to learn. They tested this at three institutions that previously used different languages (C, Java, Delphi). In all cases, Python was found to be a better choice. This was determined by a higher average grade on a course of comparable difficulty. Leping et al. [48] used Python as their FPL with a subset of their class in 2008. The rest of the class was still taught using Java. They felt that Python was "elegant, simple and practical" with clean and easy to read syntax. The results showed similar outcomes for both Java and Python students. One interesting outcome however was that a lower percentage of people outright failed the course in Python, but more students were perhaps scraping by.

Hunt [33] disagrees with Grandell et al. [27], Nikula et al. [63] and Leping et al. [48]. In 2014, his department switched from teaching Java to Python. In 2015, they decided to switch back due to problems they experienced that hadn't been noted in literature studied. In particular, the lack of arrays, the difficulty of transitioning to Java in CS2, and the inability to focus on an "objects first" approach were cited as the reasons for this.

According to the TIOBE index Software [87], Java is the most commonly used programming language in the world (at least it was in 2017, currently in July 2023 it is 4th with Python now sitting atop the rankings), so it makes sense that it is a frequently used FPL also. Ivanović et al. [35] took up teaching of Java after a number of years of using Modula-2. They decided to do a comparison study. While they liked Java as a language, there were no statistically significant differences in grades. The author posits "this result suggests that the choice of the introductory programming language does not matter if we use students' performance as the criterion of suitability". Again, it is worth noting that many other papers in the literature mentioned Java as their FPL without discussing why. This could be complacency due to it being such a widely used language that not many researchers are discussing its efficacy.

Not much information was found relating to C++ as a FPL. This is likely due to the rise in Java in the last two decades. One paper that discusses C++ was Bergin et al. [8] which covers some of the issues related to C++ in CS1. It is noted that C++ contains many verbose and over-complicated elements such as include statements, unnecessary typecasting and string comparison. A lot of these issues are also present in other common FPL. The author is not trying to discourage the use of C++, but merely pointing out some likely pitfalls that could be experienced.

Visual FPL

Scratch is commonly used to teach programming to young students, but is it effective? Aivaloglou and Hermans [1] performed an analysis on a database of 250,166 scraped Scratch projects to see how children make use of the tool. While most projects were small, conditionals and variables were frequently applied. There were also some example of large projects using multiple sprites and many blocks of code. This shows that Scratch has the potential to be used as a FPL, or in general as a first-exposure programming environment. They also found a high count of clones within the data, suggesting that it already is being highly used for teaching purposes. Armoni et al. [5] noted that learning Scratch at an early age did affect retention. These students chose to continue on to a Java / C# course later into their school lives. They also appeared to pick up information faster and grasp the tougher concepts before their peers.

As presented in Section 2.4.2.3, Alice is the most frequently used language in college level CSo courses. Mullins et al. [62] discussed one such course. The authors see the importance of students being able to see and manipulate objects directly in the editor, a benefit they would not experience in a textual language. Upon examining collected data from the course, it was noted that results varied, but for the most part using Alice increased pass rates, sometimes with a lower average grade however. In general, Alice proved most helpful for those students who would traditionally struggle with the material, without having a negative effect on other students who don't need the extra help. Retention and interest also increased with those who undertook this course.

Comparison of Textual and Visual FPL

A multitude of studies have been performed that compare the usage of VPL and textual languages as FPL. da Silva Ribeiro et al. [84] wanted to determine if, and how, VPL can help learners understand and transition

to a textual language. This was tested with the help of a pair of Moodle-based web courses. The visual course was taught using Visual iVProg and the textual course used C. The Moodle "Virtual Programming Lab" environment allowed for automatic evaluation of student submissions. These courses were voluntary (public), and only lasted four weeks. There was a total of 144 participants, split between both teaching styles. The content in both courses was essentially the same. From an analysis of workload, the authors conclude that "visual programming seems to be a nice option to introduce programming concepts".

Cliburn [15] discusses a CS1 course that taught Alice and Java together in the same term. A total of 84 participants took this course, of which 59.5% found Alice helped them in their understanding of Java. Despite this, the author argues that this outcome was not good enough. If Alice truly made a lasting difference, apart from just the effect of knowing elements of a programming language before beginning Java, then the figure should be much higher. One interesting response from a provided survey was "the programming concepts it (Alice) taught were mostly so simplistic that it really would have been better to spend only a little time on them and the more complex concepts did not make sense until I learned them in Java". It was due to responses like this that the author decided to revert to a full Java course. He still believes that Alice can be useful, but perhaps not in a two language, one semester style course.

Daly [18] also compared the effects of teaching Alice side-by-side with Java. The focus was on confidence levels and if they have an effect. There were a total of 29 participants who took part in this online study. Eighteen took the pure Java course, with 11 taking a course that entailed six weeks of Alice followed by six weeks of Java. The author found that "the students in the Alice / Java course had a higher level of confidence overall when compared to the pure Java course". More importantly, confidence

did seem to imply success in the course and also led to higher retention and enjoyment.

Eid and Millham [23] wanted to investigate if learning introductory concepts in a textual language was better than doing so in a VPL. Two groups of students were examined. One group started with a textual language and proceeded to a high level visual programming course. The other group started with a low level visual programming course and proceeded to the high level one. This allows for the focus on concepts first and lets students understand what is happening at a basic level. The authors found that there was statistically significantly better test results for those whose FPL was text-based.

Textual Augmentation

A number of authors also looked at the concept of textual augmentation, which is akin to the hybrid languages discussed earlier. Laakso et al. [46] wanted to look at the concept of an executable pseudo language. The authors believe that this allows you to take the focus away from verbose syntax while still allowing the run time nature of programming to shine through. Their solution involved a tool called ViLLE, which runs on a subset of Python, and allows for visualisation. The authors tested this tool on a class of 72, with 32 students using ViLLE. The results showed enhanced learning in those who used ViLLE.

Montero et al. [59] looked at Greenfoot which allows for visualisation of object oriented Java concepts using animation. They chose Greenfoot as it allowed for both visual and textual editing of the program. In their study, 15 students used the Greenfoot environment while 18 used only textual materials. There was a statistically significant difference at the end of the course in the knowledge of Greenfoot students versus the knowledge of the control group about Object Oriented principles. Greenfoot was also liked by the students, which is always a positive thing.

Alshaigy et al. [2] developed a tool called PILET, which is an interactive learning tool for Python. The goal of this tool was to be adaptable to the learning style of the student. If they were a visual learner, they could use a visual tool, similarly a textual model and a puzzle-based model were included. As you use the tool, it builds up a knowledge database about how you learn in order to present the user with the best material first time as they progress. The authors goal is to avoid a single pedagogical learning style, which would not necessarily meet everyone's needs. Based on the literature in this paper, and the amount of different approaches that different institutions take, this might be a very strong concept. The analysis of this tool is still ongoing, but early results are promising.

Conclusion

Many researchers believe that using correct teaching methodologies, independent of what particular tool you are using is more important than the actual choice of First Programming Language. Educators have had success with a broad range of different FPL, and equally others have had failings with many languages. As discussed in Section 2.4.2.1, there are certain criteria that a "Good" FPL might have. If these guidelines are followed (by not picking an overly "difficult" language), along with if the teacher is familiar with a given language, this might lead to the best quality of course. For us, if one particular language of each type had to be chosen, Python, however, seems to be the most highly supported textual language from the literature, possibly due to its relative newness as a programming language. Java would also be a strong choice as it is currently (one of) the most used language(s) in the world [87] and has proven itself to be a strong FPL [55]. Scratch is also held in high regard as a VPL. Based on the knowledge that hybrid languages provide the best of both worlds, the "ideal" language choice might be a combination of Python (or Java) and Scratch. If a course

can be made stimulating and interesting for the students, then the choice of programming language is not as important as many people think.

2.4.3 DISCUSSION

Throughout this review, we have discussed the benefits of learning a Visual Programming Language and whether or not the First Programming Language choice has a profound effect on student performance and interest. It is clear that the most important thing educators can do is make their course interesting, and ensure it covers all the important elements needed to truly "know" programming.

It has been demonstrated through the answers to the research questions that the actual choice of what tools to use does not matter, within reason. The use of a Visual Programming Language will in most cases, be very helpful to a student. It may not be something to pursue for a longitudinal time frame, but as an introduction to CS, it is clearly beneficial and will generally lead to higher retention of knowledge and interest. Some other general recommendations in terms of First Programming Language choice are detailed next.

In Sections 2.4.2.1 and 2.4.2.3, we discussed both the elements that lead to a "good" FPL choice and the frequency of common language uses. These factors should be your main consideration when choosing a language. You want a language that allows you to teach all threshold concepts in an easy to understand manner. At the same time, you do not want a student's primary language to be something that only 1% of the workforce use. These reasons are why we will primarily focus on Java / Python / Scratch for the rest of the thesis as they have a good balance of all the requirements and usage rates.

It is important to find your programming comfort zone. When you are comfortable with your material, it will come across in your teaching

and will give your students more confidence. If you have been using and teaching Java for years, you are likely best to stick with it. There is no need to reinvent the wheel since the concepts are the most important thing and the syntax can be relearned by the student in the future. This is better rather than their having a negative first experience that turns them away from Computer Science forever.

In general, it is always good to follow local conventions in order to best prepare young students for what they will be undertaking in the future. In Ireland, Computer Science is currently in the process of being introduced as an examinable, optional, Leaving Certificate subject. An initial phase with a small number of schools will commence in September 2018, with full roll-out to all schools commencing in 2020. For this course, students will be taught using both Python and Javascript [17]. As such, these might become more highly considered languages of choice for educators within Ireland.

If a student of any age enjoys what they are doing, there is a better chance that they are going to understand it and continue studying it. An interactive and fun environment fosters the best learning for young students as it allows them to feel they are involved in the process. With particular reference to VPL (although it holds true for TPL as well), Armoni et al. [5] noted that after learning and experimenting with Scratch at an early age, students were more likely to continue with programming. Scratch mostly involves game making and animation and is generally considered fun.

Most of all, we would suggest that you ensure you enjoy teaching your material and engage with it and your students, because if you do not, it is unlikely that they are enjoying learning it [7]. This initial analysis aims to guide educators at all levels and in all institution types to examine the options available to them when they are teaching programming. This review may go some way to informing their decisions about what the First

Programming Language to use and the benefits of both text-based and visual-based programming languages.

2.5 SUMMARY

This chapter began to answer two of the overall research questions. For RQ₁, the literature mostly shows that language choice is not the single most important element (but is still something to be considered). For RQ₂, the literature suggests that there is an age where VPL usage wanes. In Chapter 4 we will look at a "hybrid" programming approach that fills in the gap years between when VPL appeal to students and when TPL become more approachable.

The chapter overall demonstrated the need for a consensus approach in FPL teaching. There are many different approaches with differing outcomes, but the most agreed upon elements are making programming education interesting, some level of VPL intervention and following good methodologies and structure. Throughout the rest of this thesis, we will begin to examine the author's approach to this problem.

Part II

COURSE DEVELOPMENT & TESTING

INITIAL COURSE DEVELOPMENT

3.1 OVERVIEW

This chapter will look at the development of two initial pilot sessions using Java and Snap! as the programming languages of choice. Testing of these sessions was undertaken at a Summer Camp ran by Maynooth University. With the knowledge gained from these sessions, further development will then be described on full eight-week courses in both languages. Finally, we will examine the testing phase of these curricula in both a public and private secondary school in Ireland, with multiple class groups.

3.2 INITIAL TEST SESSIONS

With the knowledge gained from the Systematic Literature Review findings, development of pilot sessions began in April 2017. Java and Snap! were chosen as the languages of choice for these sessions due to their familiarity and frequency in the literature. They were designed to be instructor led and cover a high number of topics in a short time frame (90 minutes).

Both sessions were designed to be as identical as possible, in terms of content. They both contained key concepts in programming such as variables, selection statements, loops and a key task (drawing shapes in Snap! and creating a basic calculator in Java). This would ensure that the primary difference in opinions of the session material rested on the language choice itself.

Presentations were designed to accompany these sessions. A survey was also created using Google Forms for attendees of the summer camp (and any future tests) to fill out. All slides from the summer camp as well as the survey and related documents can be seen in Appendix A. These materials were all used at the Computer Science Summer Camp 2017, Maynooth University to teach the developed sessions. In Section 3.2, a paper entitled "First Programming Language: Visual or Textual" [67] will be referenced. This was published in the proceedings of the International Conference on Engaging Pedagogy (ICEP) in December 2017. It details the creation of the initial test sessions and their first delivery.

3.2.1 LANGUAGE CHOICE

The languages that were decided upon to be used in this study were Snap! and Java. Snap! was chosen as an alternative to the popular VPL Scratch as Snap! is a clone of Scratch that offers some more advanced options which will be useful for future work in the development of a curriculum. Java was chosen because, according to the TIOBE index [87], Java is (one of) the most commonly used programming language in the world. This includes an aggregate of both educational and industry based popularity.

A major finding of the literature review was that teaching methodologies are often more important than the actual languages of choice. Furthermore, familiarity is a large factor in teaching ability. Both of these languages are very familiar to the author. While there is a multitude of literature that pointed to these languages being the "best choice", there were some key points that added further weight to using them as the languages of choice in this study. Given that language choice isn't the most important aspect of teaching, picking a language that is widespread has clear advantages for both teacher and learner. There is also more established material

available for teaching Java and Snap!. Snap! provides all the functionality of Scratch along with the ability to add additional features.

To further this point, a survey conducted in the USA in 2011 [19] investigated 371 institutions of education who were asked what language they used for their beginner programming courses (CS1 or CS0). The results were that 48.2% of institutions had adopted Java. While this doesn't inform us of the current rates today, nor does it inform us of data outside of the USA, it still gives a strong idea of where the numbers have been in recent years.

In many countries, CoderDojo and other similar after school club organisations often use Scratch or another visual language as the backbone of their teaching. It is well established that VPL are considered more fun for young learners to experiment with when compared to the more complex TPL. The author has had experience with running such a CoderDojo and has seen first-hand the effect learning Scratch, and other blocks-based languages, has had on some students. Other researchers have examined this further by looking at retention of students [5]. They found that after learning Scratch at an early age, the students that chose to progress to a Java / C# course in later years appeared to pick up information faster and grasp some of the tougher concepts earlier than their peers.

However, which of these two approaches makes the learning process easier for the student? As mentioned earlier, language choice and approach is not as important as methodology; however, some elements of each language type might be easier for different age groups to learn. Due to this, a combination of both languages might be a good, if not the best, approach to take. A study undertaken in 2015 which involved teaching five weeks of Snap and five weeks of Java found that 58% of students thought Snap! was easier to use [93]. Some students reported that the blocks approach was easier to read. Some drawbacks were also noted; for example, blocks were identified as being less powerful giving less implicit customisation

(but this isn't a huge issue for CS₁, given the implied simplicity of first principles programming).

These are but a few examples of why these languages were chosen. In general, no matter what language or tool is chosen it will have both positives and negatives. For this study, the two languages were chosen due to pedagogical evidence of their success.

3.2.2 SESSIONS OVERVIEW

Once the two languages had been decided on, work could commence on developing initial sessions for both. The goal was to create two sessions that were close to identical in terms of content and level of difficulty, while still managing to showcase the important elements of each respective language. The sessions would be designed to be delivered to students aged 10-18 in 90-minute sessions. As well as the author, multiple demonstrators would be employed at the summer camp to provide help to the students whenever they struggled. This would allow for even detailed topics to be covered in a very short time frame. In terms of content, both sessions would cover the topics presented in Table 3.1. These sessions were developed as pilots to test the methodology with an aim to develop full curricula in the next stage (which will be discussed in Section 3.3).

Table 3.1: Session Topics

<i>Language Tools (BlueJ / Snap!)</i>	<i>Java boilerplate / Snap! run blocks</i>	<i>"Hello World" for the language</i>	<i>Selection statements (if / else)</i>
<i>Basic Math</i>	<i>Variables</i>	<i>Loops</i>	<i>An advanced topic</i>

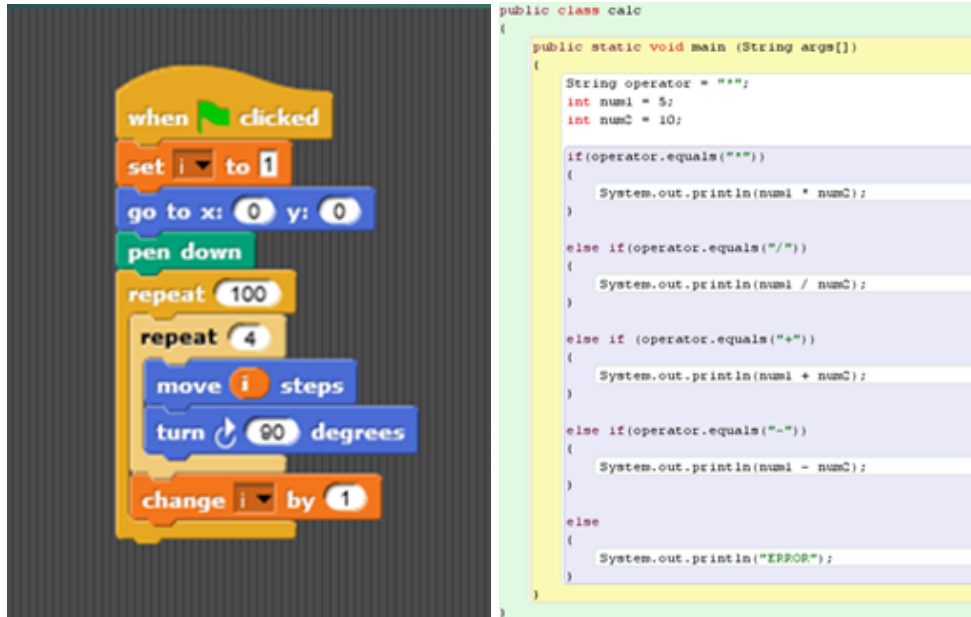
For Java, the session closely followed a shortened, but expedited version of the CS₁ course delivered in the Computer Science department at Maynooth University. For Snap!, elements were taken from the “Beauty and Joy of Computing course” [10] as well as from personal experiences with Scratch. The key element that allows these sessions to be delivered in such a short time frame (compared to usually spending weeks learning these topics) lies in the expectations. The students are not expected to become experts in the material. We simply aim to give them an overview of what programming entails and introduce some threshold concepts.

At the end of each topic in a session, an exercise would be displayed on the slides giving the students a chance to trial what they have learned. The students’ copy of the material would not contain the answers to these exercises. Once the students had sufficient time to work on the exercise, the answer would be given on the lecturer’s copy of the slides on screen so they could see the optimal solution.

For the final section of each session, the advanced topic was covered. This is the only element of the sessions that would be significantly different. For Snap!, the concept of drawing shapes was chosen since it would utilise a part of everything they had learned so far and also demonstrate some nice animation features of the software (See Figure 3.1a). For Java, the concept of creating a very basic calculator was chosen. Again, this would combine everything they had learned together, while showing something functional that they would understand (See Figure 3.1b).

3.2.3 INITIAL PILOT TEST

Before the commencement of the main testing phase at the Computer Science Summer Camp, a pilot test for each session was conducted. The Snap! session was tested on a cohort of approximately 20 Junior Cycle students. These students were attending Maynooth University to experience taster



(a) Snap! Advanced Code

(b) Java Advanced Code

Figure 3.1: Advanced Code Solutions

courses in multiple disciplines. The material was well received based on anecdotal feedback. Many students followed along with the session material, while some others lost focus. This was not a major concern as it would be expected with a group who had not decided themselves to attend the sessions.

The Java session was also tested on a small group of Senior Cycle students by a member of our research group. The anecdotal feedback from them was generally positive. Some students struggled with the loops concept (a threshold concept within programming). This was rectified in the final session by moving it to later in the teaching process for those who wanted a challenge after completing the main phase.

The feedback received from both of these pilot tests was vital as it helped to verify that the timing of the sessions was correct as well as revealing some enhancements that were needed in the material.

3.2.4 DATA COLLECTION

Since there would be no official testing of the students, feedback relating to the success or failure of the sessions would come from the results of a survey. This survey was compiled, using Google Forms, after the initial pilot tests were completed. The goal of the survey was to learn about how the students felt about the sessions; how the sessions compared was a key element of this. The questions that were decided upon for the survey can be seen in Appendix A.

These questions would help to gather feedback from the students with as little bias as possible. It would allow them to express their preferences within each individual session as well as make a fair comparison of the two sessions. The ability to filter the results by age and gender is also key to determining if there are any preference patterns. In line with ethical requirements, consent forms were given to parents before the commencement of the camp to ensure data collection and possibly publication was allowed by their parents. On the day, consent from the students was also collected.

3.2.5 MAIN SESSION DELIVERY

In June 2017, the departmental summer camp began. The camp is broken down into three separate weeks where students can choose to do any of the weeks individually or all three weeks. All the content was unique in each week. The Java and Snap! sessions were both scheduled to run on the same day in week two. In total, there were 35 students sitting the sessions on the day with ages varying from 10 to 18 years of age.

The Snap! session was the first session delivered. The students all chose a PC which had the Snap! website preloaded as well as a copy of the material opened. After a short introduction, the slides were presented and

delivered at a slow pace. The majority of students followed along with no issues, with minimal assistance from the demonstrators. Some students were on the wrong track with some of the exercises, but understood the answer once it was shown on the screen. This session ran for 90 minutes.

After a short 30-minute lunch break, the students returned and immediately started working with Java. Java was chosen to go second due to the perceived extra difficulty it would present. Since the material of the two sessions mostly matched, they would only be learning the syntax of Java in the first part of the Java session rather than both the concepts and the syntax. To further assist with the learning of Java, some parts of the exercises were live coded after the students made their attempt rather than being static on the screen. After the completion of the Java course, the survey was immediately administered while all the material was still fresh in their minds. If there was more time available, another study would have been ran using Java first as well.

3.2.6 SUMMER CAMP OUTCOMES

All 35 students who were present on the day of the study provided a response to the survey. The demographic of the participants varied slightly with 88.6% of the participants being male and 11.4% being female. In terms of the age groups, 8.6% were between 10 and 12, 68.6% were between 13 and 15, and 22.9% were 16 or older. To examine if these demographics show patterns in their perceptions of the different languages and styles, most feedback will be broken down in relation to age groups.

The most encouraging outcome from the survey was that 88.6% of respondents said they wanted to learn more programming in the future. The other 11.4% said that they “maybe” want to learn more in the future. This shows that the perception of younger learners towards the topic is very positive.

The main question that arose from this study is whether one language was perceived to be harder than the other or not. The results of this question can be seen in Figure 3.2a and Figure 3.2b. The mean difficulty rating for Snap! was 3.57/10 while Java had a mean difficulty rating of 6.94/10. Based on a one-tail paired two sample t-test, this represents a clear statistical inference that Java was harder to learn than Snap! ($p = 6.8E-10$). We wanted to check how this breaks down across the age groups. Is there a clear upward trend of the languages getting easier as you get older? From the results of this study, there seems not to be a clear trend, as can be seen in Figures 3.3a and 3.3b. Unsurprisingly, for all age groups, Java remained more difficult than Snap! (10-12 year olds: $p=0.0471$, 13-15 year olds: $p=1.03E-06$ and 16+ year olds: $p=0.0013$).

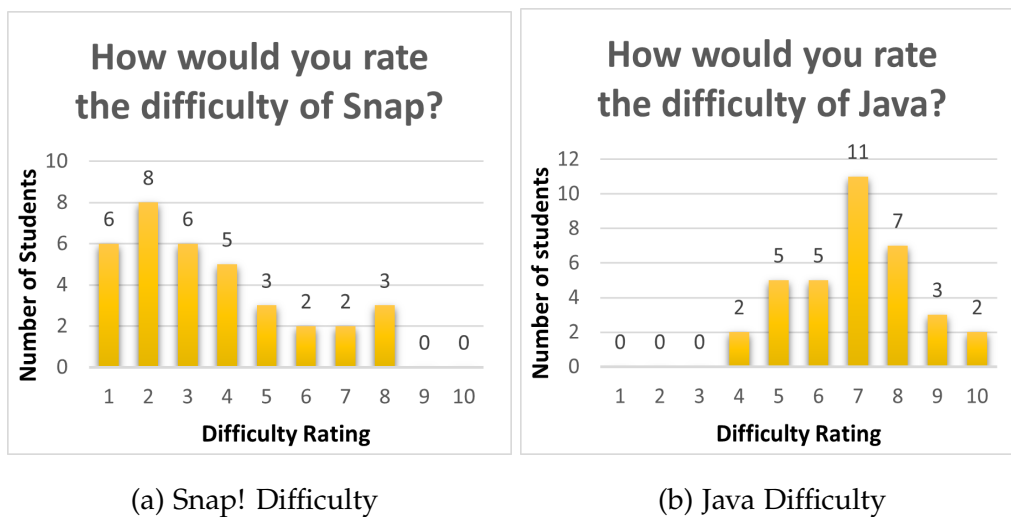


Figure 3.2: Language Difficulty Ratings

This is not conclusive given that the number of students who fell outside the 13-15-year-old range was very low. Given that the courses were created with the same core content (with the exception of the one advanced topic in each), no external bias is being added to the difficulty ratings. This is purely based on the syntax and content of the languages themselves and how they compare to each other.

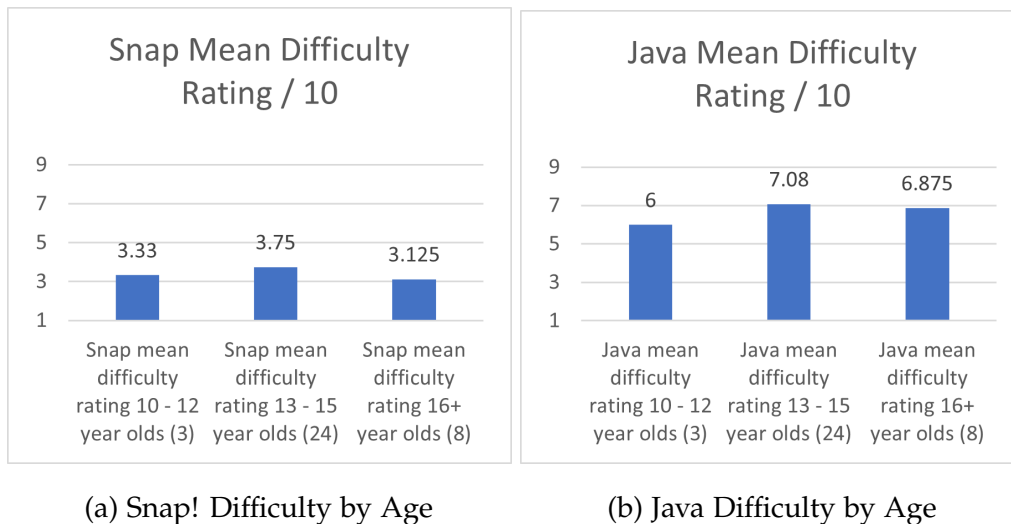


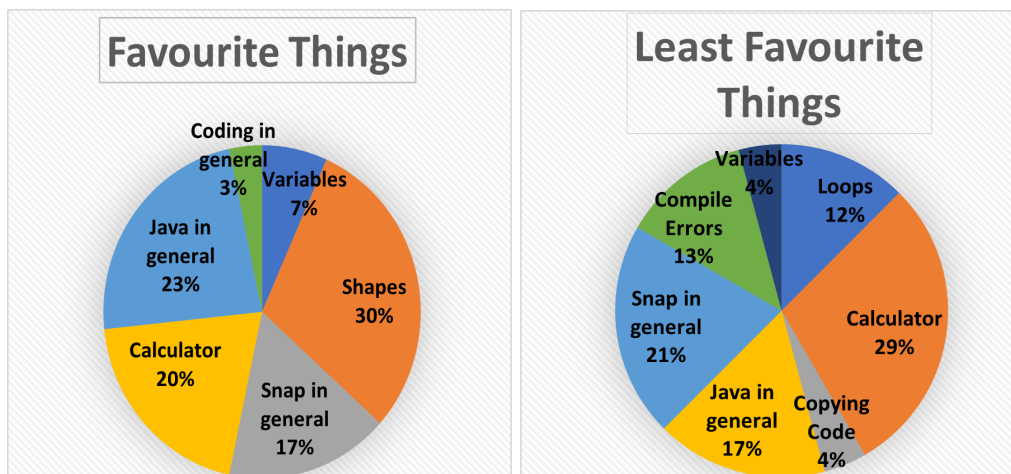
Figure 3.3: Language Mean Difficulty Ratings by Age

To further analyse, the students were asked what they considered to be the hardest aspect of each course. For Snap!, 34.3% said variables were, 28.6% said drawing shapes and 22.9% said loops. For Java, 68.6% said making the calculator, 17.1% said loops and 11.4% said selection statements. Putting the shapes and the calculator on this section of the survey was an oversight as they involved using all the core elements of the course to make. However, this still gives us a good idea of what students were finding difficulty with and were mostly in the expected places.

Finally, when asked whether they preferred one of the styles of programming (text based or visual based), 37.1% of students said they preferred text based and 31.4% said visual based and 31.4% said that they had no preference. However, analysing the age brackets for this question it was observed that 13-15 year olds had a larger preference for the visual based language (41.6%) over the text based language (29.2%) with 19.2% having no preference. Conversely, 16+ year olds overwhelmingly preferred text based programming (62.5%) with 0% choosing the visual based option and 37.5% having no preference. This gives some credence to the theory that younger students enjoy visual programming more and older students' prefer textual programming.

As well as this numerical data, the students could make some optional comments on the course related to their favourite and least favourite elements as well as anything else they wished to add. When asked if they enjoyed each session, 60% of students responded that they did (for both sessions). Similarly, 34.3% (Snap!) and 25.7% (Java) said that they thought the sessions were “OK”. The remaining 5.7% (Snap!) and 14.3% (Java) did not enjoy the courses. In terms of a preferred course, 51.4% of the students preferred the Snap! session with 48.6% preferring Java.

In terms of students’ favourite and least favourite things in each session, some students chose not to respond, and others gave spoiled answers. For favourite things, there were 30 valid answers, while for least favourite things there were 24 valid answers. These were then summarised into categories and the results can be seen in Figure 3.4a and Figure 3.4b.



(a) Students’ Favourite Things

(b) Students’ Least Favourite Things

Figure 3.4: Responses to Favourite and Least Favourite Course Elements

A few other key anecdotal comments from the survey included:

- “*Make the programming bit more interesting*” – This was a fair comment. It is difficult to find the balance between learning and fun in an introductory course.

- *"Spend more time on Java"* – Given the higher difficulty of Java, it is completely fair that students would need more time to learn the Java material.
- *"Overall it was really interesting and enjoyable!"*

The results of this study, even though the number in attendance was reasonably small and a lot of the feedback was anecdotal, are still important. When and why one might use a VPL as a teaching tool are questions that are often asked and rarely answered. This study has shown that a language like Snap! has as much potential for learning as Java, provided the target audience is correct.

Some elements of the study could have been done differently, and will be rectified in the next phase. The time available for delivery of the course was not ideal with some students feeling like they needed longer for certain concepts. There were some minor mistakes in the survey such as asking the students what the most difficult element was and including the element that encompassed everything.

3.3 FULL COURSES

Following on from the successful delivery and testing of the short sessions, plans were laid out for the development of two curricula that expanded on them. The target audience for these full courses would be Transition Year students (approximately 15 years old students who are doing an intermediate year in school between Junior Cycle and Senior Cycle).

A full suite of material was included in the development (slides, class plans, assessments, etc). Just like the short courses, these would be as identical as possible. These courses would be built to last eight weeks and cover a number of key threshold concepts in Computer Programming, including:

- Language Introduction (Hello World)
- Variables and Operators
- Selection
- Loops
- Strings and User Input
- Arrays (omitted in the course delivery)

This section will discuss this process in detail, as well as outline the testing phase for these courses. The full course material can be seen as part of Appendix B. Throughout Section 3.3, a paper entitled "First Programming Language: - Java or Snap!? A Short Course Perspective" [69] will be referenced. This was published in the proceedings of the 10th Annual International Conference on Computer Science Education: Innovation and Technology in 2019.

3.3.1 COURSES OVERVIEW

In May 2018, work began on developing the eight-week courses in both Snap! and Java based on all findings from the summer camp short courses. For each session mentioned in Section 3.3, the material was created in parallel for both languages to ensure that they were equivalent. The material created for each week included a class plan, teaching slides, in class practice questions, a homework sheet and a homework answer sheet. Two surveys were also drafted; one as a pre-course survey which collected basic data and opinions on Computer Science and one as a post-course survey which would collect opinions on the lessons, the courses and Computer Science in general again.

The questions asked both in class and as part of the homework sheets were partially inspired by the existing CS1 course at Maynooth University

(as well as ideas from the Systematic Literature Review) but simplified for the target audience. A sample of a question asked in both courses during the “Selection” session is shown in Figure 3.5.

Create a program that checks if a person is able to vote. You will need to create an int that holds a value to represent a person’s age. Print a statement that says “They can vote” if they’re of age and a statement that says “They cannot vote” if they are too young.

Figure 3.5: Example homework question (Java and Snap!)

Once the course development was completed in June 2018, the material was given to some colleagues for equivalence testing. This involved reading the course materials in parallel and deciding if they were equivalent, as we needed others to verify our thoughts and decisions. Results of this testing were favourable, and some feedback was also obtained which led to further iterative development of the curriculum materials.

As well as the six major topics, the week 7 and week 8 material also had to be developed. Week 7 was a revision week which allowed the students to reflect on all of the concepts they had touched on over the previous weeks. This material contained worksheets with multiple short questions (one per topic) and two longer questions (using elements from all weeks) to pick from. Week 8 was an examination so a question and marking scheme needed to be written up for this session. The material for both courses can be seen on Padlet (see Appendix B).

3.3.2 SCHOOLS TESTING

During July and August of 2018, contact was made with a number of schools under the PACT initiative [60] to offer them the option of host-

ing one of the courses in the first term of the 2018 – 2019 academic year. The supervisor of this project is a founding member of the PACT initiative at Maynooth University and hence this is why this methodology was explored. These courses would be delivered in Transition Year. An agreement was made with two chosen schools, with the following make up:

- School #1 – Mixed gender public school, four Transition Year classes, 15-16-year-old students, two classes would undertake the Java course and two classes would undertake the Snap! course, approximately 40 students for each course.
- School #2 - All girls private school, one Transition Year class, 15-16-year-old students, approximately 20 students total, would undertake the Java course.

All five classes began in mid-October 2018 with the intention to run until the final week before Christmas for a total of eight weeks. There were some challenges with this which ended up pushing the last two sessions to late January 2019 after a month-long break from material, but all material was still delivered to all students.

One exception to this is that the “Arrays” sessions were not delivered to any class. This was due to the fact that Strings took longer for the students to comprehend and practice than originally expected. As such, Strings and User Input (originally a single week session) was expanded out to two weeks and arrays had to be dropped from the course. The revision and final exam were adjusted to account for this.

The first session began with an introduction to the programming environment of choice (JCreator for Java, Snap! UI for Snap!). The students were also given an overview of what the rest of the course would entail. Finally, they were asked to complete a short survey on their opinions of CS and what they expected from the course. Data was collected anonymously with each student being given an ID number to link their data with the survey in the final week.

All intermediate sessions began with a recap of what was covered in the previous week and a run through of the homework questions that were assigned. The new material was then delivered. Week 7 gave the students control to look over any material they previously struggled with. They were able to ask questions or simply work away on questions. The final session began with the post course survey and the course completed with a final examination.

Efficacy Outcomes

The final examination was delivered in week 8 of the course delivery. Students were given 45 minutes to answer a short question which used an element of each week's material. The questions posed in Java can be seen in Figure 3.6 and the question posed in Snap! can be seen in Figure 3.7.

Write a Java program which:

1. Creates a String
2. Gets the value for this String using user input (use any sentence when testing)
3. Using a loop and selection, go through this String and only print every even positioned character.

For example:

Hello World -> HloWrd (0, 2, 4, 6, 8, 10)

Figure 3.6: Java Examination Question

Once the examination was completed, a photograph was taken of each student attempt, saved as their ID number. After the course was completed, these attempts were then graded based on a marking scheme. The marking schemes gave either 0 (no attempt made), 5 (decent attempt made at using the element) or 10 (good or perfect use of the element) for each of

Write a Snap program which:

1. Creates a String variable
2. Gets the value for this String using user input (use any sentence when testing)
3. Using a loop and selection, go through this String and only print every even positioned character.

For example:

Hello World -> el ol (2, 4, 6, 8, 10)

Figure 3.7: Snap! Examination Question

ten different requirements to give a total of 100 marks. The ten marking elements for each language's examination are given in Table 3.2.

The average results from these tests were as follows:

- Snap! - 34.4 / 100
- Java - 22.8 / 100
- Java in School #1 - 22.8 / 100
- Java in School #2 - 42.5 / 100

From these results, we were able to discern some important information. First of all, the overall grades were quite low. There are many possible factors for this including the limited time frame for practice, the general difficulty of learning programming [47], the fact that the last two sessions (revision and examination) needed to be delivered after the winter holidays, and in school #1's case the lack of a permanent teacher present in the room. Additionally, it is worth noting that no marks would be awarded to students towards their end of year marks for Transition Year so a lack of motivation may be present.

Table 3.2: Marking Schemes for Java and Snap!

Marks (/100)	Java	Snap!
10	Use of Import	Use of a "When Click" Block
10	Use of Class	Use of the "Answer" Element
10	Use of a Main Method	Use of an Update Statement
10	Use of a String Variable	Use of a Variable With Text
10	Use of the Scanner	Use of "Ask and Wait"
10	Use of a Loop	Use of a Loop
10	Use of an If Statement	Use of an If Statement
10	Use of Modulus	Use of Modulus
10	Use of a Print Statement	Use of "Say" Block to Print
10	Use of charAt	Use of "letter X of" Block

For the null hypothesis that the Java examination would not be more difficult than the Snap! examination; the results show that this is possibly the case. This was shown through the usage of a two tailed t-test which was the metric used for all further data analysis in this paper. A minimum p-value of 0.05 will be required to reject any null hypotheses. In this instance, we had $p = 0.3420$ ($p > 0.05$). This is not a statistically significant enough result to say if one exam was more challenging than the other. If this were true, it would imply that Java is not actually implicitly harder to learn than Snap! is.

Interestingly though, when we break this down further and assume that the two schools were both significantly different environments, the results look a little different. We can affirm this by comparing the Java results of school #1 with school #2. When we do, we see a statistically significant difference in outcomes with $p = 0.0009$ ($p < 0.01$). This means that there's

a greater than 99% chance that the differing environments between the schools had a significant effect on the study.

With this in mind, we can now compare the outcomes of the Java examination and the Snap! examination in school #1 (which was also the school with the larger dataset with two classes each). We found that the Snap! grades were significantly higher with $p = 0.01239$ ($p < 0.05$). This tells us that there's a very high probability that the Java examination was in fact more difficult to perform well in, and we can reject the null hypothesis for school #1.

This would align with the previous findings [67] that Java was a significantly harder language to learn. This is the key point of the study.

Survey Outcomes

The examination results were only one metric analysed as part of this process. As previously mentioned, two surveys were also administered; the first one prior to the first session of the course which collected opinions on what they might expect from the course, if they might consider studying Computer Science at University and other collected survey data.

The second survey was administered right before the final examination to determine the easiest and hardest sessions from the student's vantage, their overall enjoyment of and difficulties with the course and once again whether they would consider studying Computer Science at University.

One important point to note as we delve into some of the results from these surveys is that the cohort who were there for the pre-survey did not exactly match the cohort who undertook the post-survey. There is much overlap but due to absences on both days, there are some who only sat one of the two surveys.

In Figure 3.8, the opinions of the Java students on studying Computer Science at University can be seen and in Figure 3.9, the opinions of the Snap! students on studying Computer Science at University can be seen.

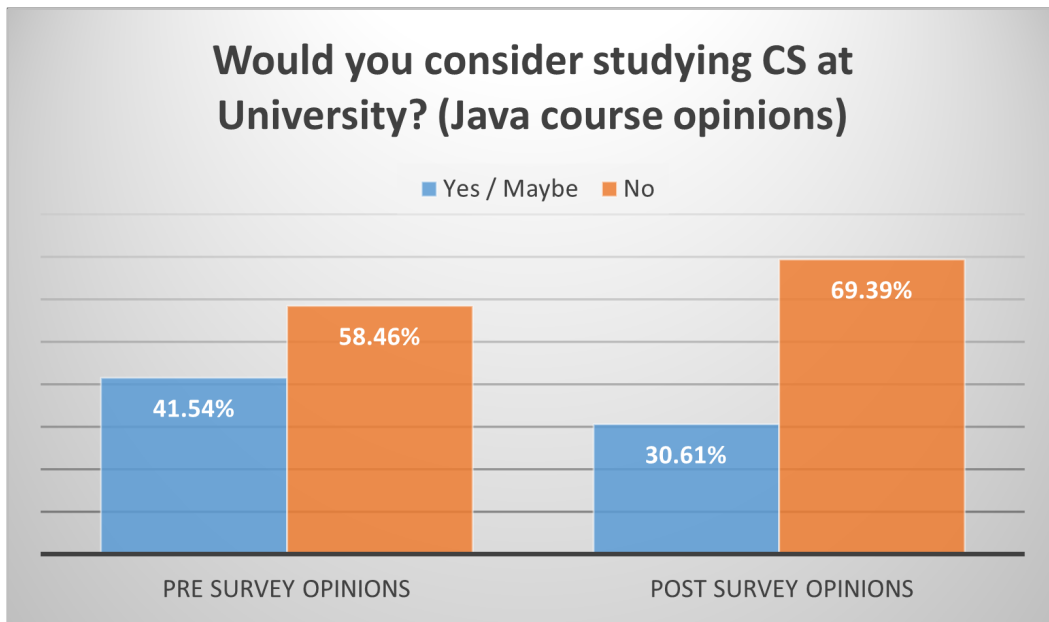


Figure 3.8: Computer Science at University Opinions - Java

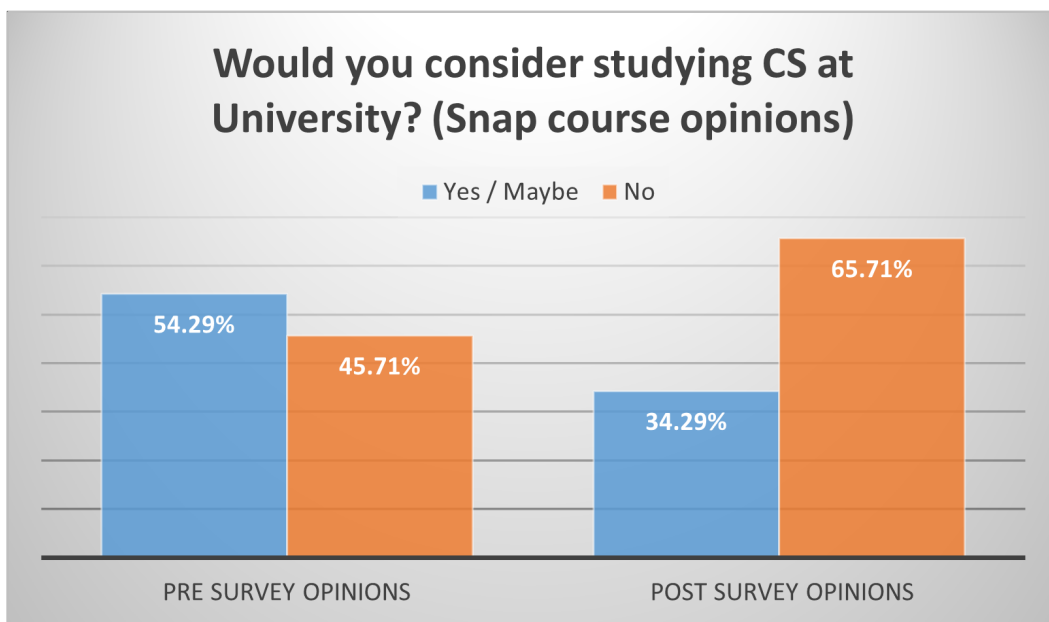


Figure 3.9: Computer Science at University Opinions - Snap!

In terms of the Java students, we found that with $p = 0.2349$ for the change in opinions between the pre-course survey and the post-course survey that there was no significant decrease. This implies that the course itself did not negatively affect their views of programming and Computer Science. In terms of the Snap! students, with $p = 0.1795$ we can make the same conclusion.

What these results did show however is that some of the students were able to make their mind up over the duration of the course. This is an important thing given the current dropout rates in Computer Science course at University [11]. It is vital that students are aware of what Computer Science is before committing to attending third level and having exposure to courses like these ones will ensure this.

In Figure 3.10, Figure 3.11, Figure 3.12 and Figure 3.13, the outcomes are shown for which course sessions the students found the easiest and most difficult. In Figure 3.10 and Figure 3.12 it is interesting to note that loops were considered the hardest to learn in Java, but not as many deemed it the hardest in Snap!. Instead, Strings and User Input seemed to be the most difficult Snap! session (which was still a highly chosen session in Java). In Figure 3.11 and Figure 3.13 it was clear that the introductory sessions eased students into each course and were overwhelmingly considered the easiest. This is much in line with what we would have expected.

The two other key questions in the post-course survey were “How much did you enjoy the course? (1= Not at all, 5 = It was OK, 10 = I loved it)” and “How difficult did you find the course? (1 star = easy, 5 stars = difficult)”.

In terms of enjoyment, the average rating for Snap! was 5.6 and the average rating for Java was 5.4694. The difference between the results for both courses was not significant with $p = 0.8027$. In other words, there was no difference at all in enjoyment levels of the courses. This is logical since the courses had identical content so enjoyment levels would be expected to be similar.

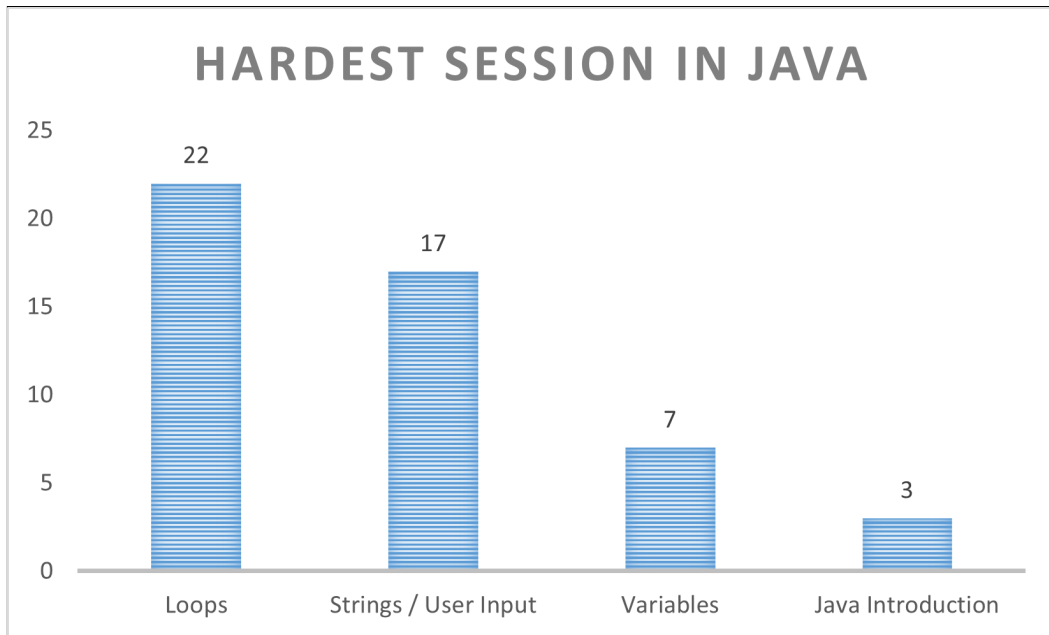


Figure 3.10: Hardest Session in Java

When it came to difficulty though, the results were more interesting. Java was rated 3.51 / 5 on average for difficulty. Snap however was only rated a 2.63 / 5. The courses were as identical in content as possible. Comparing the rated difficulties, we have $p = 0.0002$ which is significant at $p < 0.01$. This means that students found Snap! to be much easier than Java, which again aligns with previous findings.

If we look even deeper at school #1 only (as they would have a similar cohort of students, they rated Java 3.69 / 5 in terms of difficulty) the result only gets more significant with $p = 0.0001$. This leaves very little room for the results to be chance meaning that Java is clearly more difficult than Snap!. The only way we could be more certain is if the exact same students rated each language.

This is once again key. It further exposes the fact that Java is more challenging to learn. This can only mean that some element of Java (verbosity, overhead or something else entirely) caused it to be more difficult to learn given the nature of the study with identical courses. If this is the case, why do we not teach CS1 in a VPL all of the time? Of course, this study

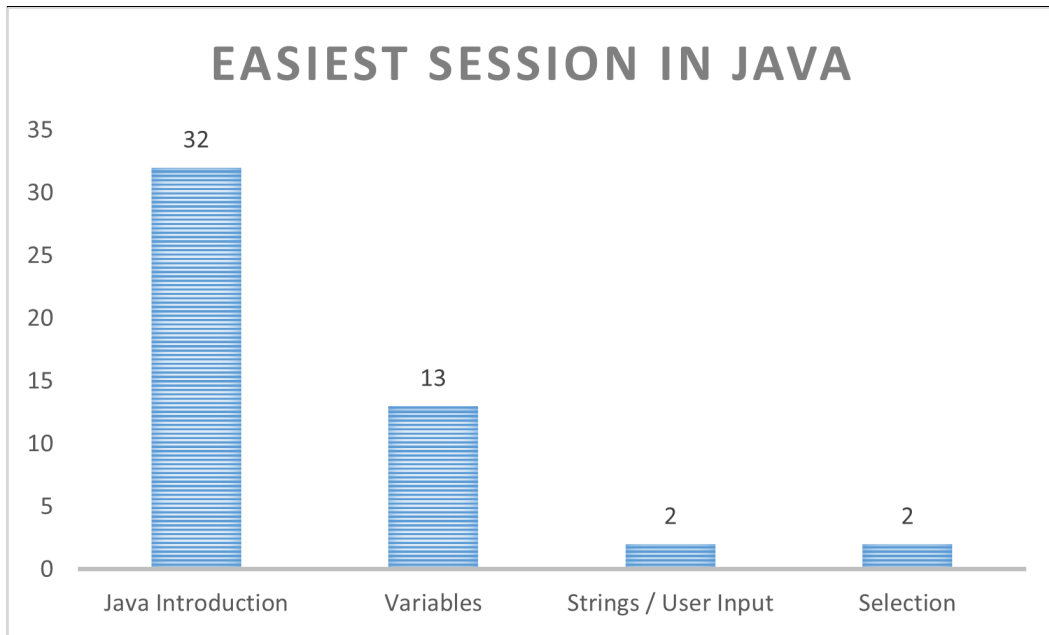


Figure 3.11: Easiest Session in Java

can only make this assertion for students aged 15 – 16 years old. This is a younger cohort than we would see at University level.

We also examined whether gender made a difference to the difficulty the students perceived with either course. We found that there was no measurable difference in the difficulty rating for girls against boys in the Java course ($p = 0.5514$) or the Snap! course ($p = 0.5033$). These results mean that within the same language groupings, gender did not have an effect on how students perceived the course.

Due to the low number of participants who took both surveys in certain cases (Males taking Java course $n = 8$, females taking Snap! course $n = 9$) the results were inconclusive as to whether Java was more difficult than Snap! amongst only males and only females respectively. The data did seem to be trending towards the overall conclusion that Java was more challenging, however. Females rated Java 3.36 / 5 on average for difficulty compared to their average Snap! difficulty rating of 2.89 / 5. Likewise, males rated Java 3.63 / 5 on average compared to Snap! 2.54 / 5.

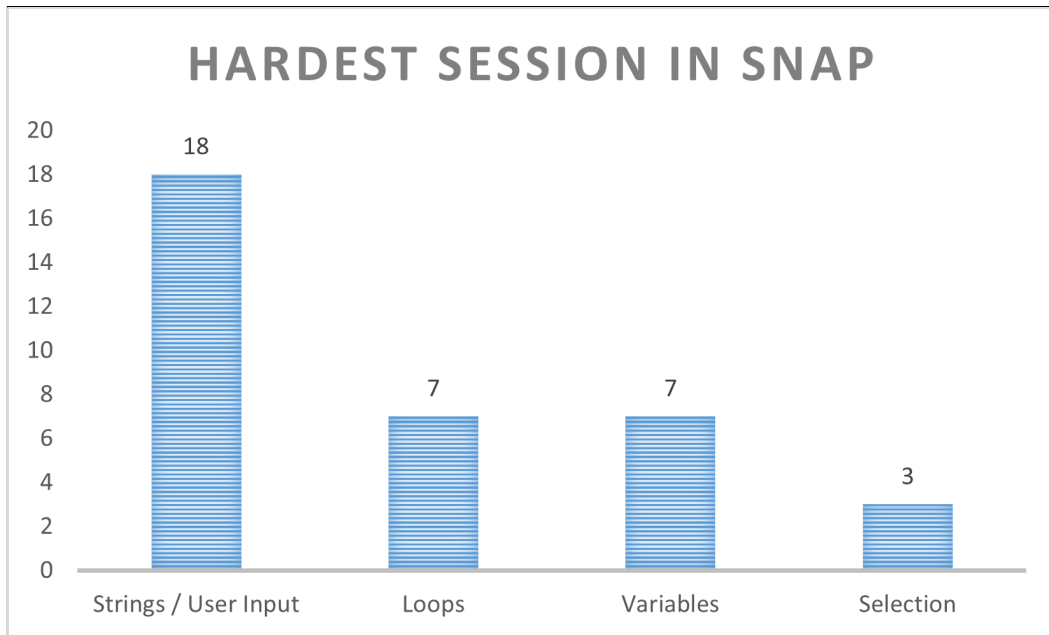


Figure 3.12: Hardest Session in Snap!

Finally, we note some interesting comments that the students of the course made. These comments were made under the post-course survey question “Do you have any other comments or suggestions?”.

- "Nope but maybe have a school teacher in the classroom" – From a student in the Snap! course. This is very important given the difference between school #1 (no teacher present) and school #2.
- "I found that when I was doing the tasks with instructions it was easy, but when I had to do them on my own, I found it to be much more difficult." - From a student in the Java course. This is a common issue that faces many students in CS1 / CS2 courses.
- "It was a useful experience to get under my belt! I'm not sure if programming is what I want for my future, but it was good to get to know the basics and try it out." – From a student in the Java course. Programming courses in schools are important to help students decide early on if it would interest them as a degree choice.

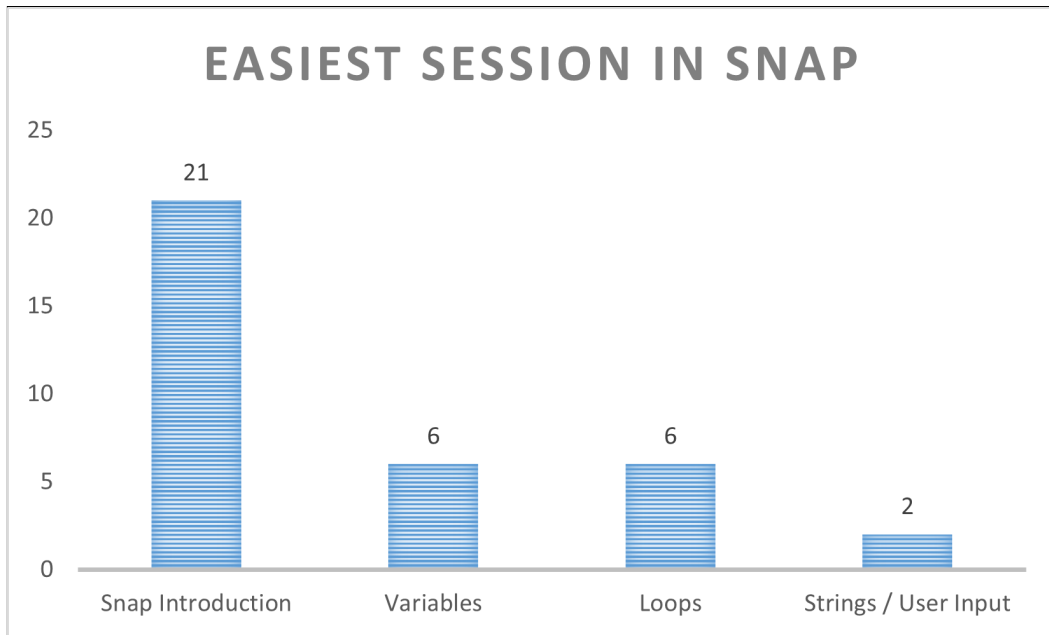


Figure 3.13: Easiest Session in Snap!

3.4 NEED FOR A HYBRID MODEL

The results from this study show promise. At least with the 15 – 16 age group, it does seem that Snap! was easier to learn than Java was. This outcome informed the next phases of the research. The courses presented will be slightly adjusted. More importantly though, given the confirmation that Java is statistically significantly more difficult than Snap!, it was decided that work would now begin on a hybrid programming model.

The concept of a hybrid programming language has been examined by multiple educators in recent years with promising results. In particular, Weintrop [93], Poole [74] and Harken [30] have looked at the concept in different ways. Weintrop [93] has created one of these environments with some promising early results. Harken [30] has commented that “browsability” is a key feature of visual blocks-based environments that reduced the complexity of learning them.

Our hypothesis is that a hybrid, drag and drop implementation of the Java programming language will reduce the difficulty of learning Java. We

have already proven in this study that Snap! was certainly easier, for 15 – 16 year old students, than Java. Using Snap!'s "build your own blocks" feature, we will be able to create custom blocks to match the syntax and semantics of Java keywords and code sections. With this created, we can create a third version of the course using the hybrid environment.

We hope to then see that Java is harder than Java Hybrid, and Java Hybrid is harder than Snap! in terms of learning difficulty. We infer that this could be the case due to the browsability and blocks-based environments having enough of an effect to ease the complexity of Java's verbosity without trivialising the learning of the language. It will also reduce the cognitive load of the student when assessing a problem. We call this the "linear line hypothesis". However, from looking at results of similar studies, we do not believe that it would make learning Java easy enough to put it on level ground with Snap! and other purely visual blocks-based languages.

3.5 SUMMARY

Overall, the work presented in this chapter has shown the promise for considering other forms of FPL rather than just a text-based approach. All approaches have their merits but with increasing drop-out rates and student difficulty with traditional CS1 courses as shown by Watson & Li [91], considering other approaches might just be the key to improving retention. These initial courses are one of the major deliverables of this project, but in the next chapter we will begin discussing the development and testing of "Hybrid Java", a tool and curriculum combined that fills in a gap in the pedagogy.

Part III

HYBRID PROGRAMMING

HYBRID JAVA

This chapter will discuss the development, verification and testing of "Hybrid Java". Hybrid Java is a Hybrid Programming Language (or tool) built on top of the Snap! programming language. It aims to combine the power of a traditional textual programming language with the visual features of the Snap! language. It was developed after the previous phases of the study showed a need for, and gap in the market of a Hybrid Programming tool based on Java. The initial concept and the subsequent development will be discussed, along with a workshop where feedback was gathered along with an initial test at the Maynooth University Computer Science Summer Camp.

This chapter will refer to a number of academic papers. In particular, "Hybrid Java: The Creation of a Hybrid Programming Environment" [70], "Hybrid Java Programming: A Visual-Textual Programming Language Workshop" [34] and "Creation of a Hybrid Programming Language" [61]. The final Hybrid Java tool can also be found online at [66].

4.1 THE NEED FOR HYBRID JAVA AND INTRODUCTION

Some visual languages (such as ScratchJr and Snap!) have a reach to children as young as five. It has been well documented that there exists a gap in the education of students in their mid- to late-teenage years where perhaps visual languages are no longer suitable and textual languages may involve too steep of a learning curve.

There is an increasing need for languages that combine the powerfulness of a text-based language with the simplistic design of a visual language. These so-called hybrid programming languages would allow for the introduction of more complex programming concepts to students by having a more welcoming and more suitable interface. A need for a hybrid language is growing alongside the increasing interest among young people in the field of Computer Programming. This follows from the research of Cheung et al. which shows that there is a certain age group (13-16 years old) where many students begin to consider visual programming languages too limited; but they are still at a point where they consider text-based languages too verbose and difficult to learn [13].

This tool attempts to address this need. For the purpose of this project, the platform Snap! is utilised to create a hybrid language. Snap! is a visual programming language which employs "blocks" to allow users to build programs. Snap! is also considered a platform and runs in the user's browser and presents an interface on which the user can program. Snap! was originally known as BYOB (Build Your Own Blocks) and was heavily influenced by the blocks-based visual language Scratch. Both Scratch and Snap! give the user access to libraries of pre-existing blocks with pre-set functionalities and allow the user to build programs using these blocks. The main additional feature that Snap! offers is the ability to create one's own blocks and extend the functionality of those blocks to create more complex and powerful programs. This is something that will prove very useful when it comes to development of the tool.

The newly created hybrid language presented combines the textual programming language of Java with the visual "drag-and-drop" programming language of Snap!. Snap! allows the user build their own blocks, which supports the integration of Java into the platform. A "drag-and-drop" interface is presented to the user, with each "block" representing a corresponding concept in Java. Samples of concepts developed with this new lan-

guage will be presented along with some of the main considerations and constraints involved. User experience and feedback was gathered from a subject pool of 174 first year Computer Science students in Maynooth University where these participants were given instructions to work with the hybrid programming language and provide a feedback to evaluate their experience using the language. This informed additional development phases. Further testing was then performed via the Computer Science Summer Camp at Maynooth University, the workshop at UKICER and through additional feedback channels, such as focus groups.

4.2 ADDITIONAL HYBRID PROGRAMMING BACKGROUND

There have been numerous studies undertaken to evaluate the benefits and disadvantages related to teaching programming using visual programming languages [9, 39, 80, 93, 94]. One such study [93] was devised to determine if a visual blocks-based programming language would be fitting as a first programming language for students to learn. The study was focused around high school students and answered three main questions:

- *Is a block-based language considered easy? If so, why?*
- *What are the difference between blocks-based and text-based languages?*
- *What could be considered as weaknesses within blocks-based languages?*

The results found that over half of the student participants surveyed found the block-based language Snap! easier to use than the text-based language Java. The reasons given for the increased ease-of-use included:

- "The lack of obscure punctuation".
- The provision of "graphical cues" given by the shape of the blocks, assisting the user in determining how to use them.

- The "act of dragging-and-dropping" the blocks resulting in less errors than the traditional typing of commands.

Perhaps the strongest advantage of a visual programming language to emerge from this study is the "browsability" of its commands as reported by Weintrop & Wilensky [93]. By having an easily accessible list of commands that are available to the user, the complexity of a language is reduced. This is something that text-based programming languages tend not to have, apart from libraries for more complicated elements.

Although the block-based language was seen as easier to use than the traditional text-based language, weaknesses were also identified, including:

- The limitation attached to programs that can be created, which prohibits the creation of more complex programs, a fact also discussed by Preidel et. al [75].
- Some students found that more time was required to complete a program in a blocks-based environment as opposed to a text-based environment.
- The students recounted how text-based languages often require less lines of code to be written in comparison to the number of blocks needed in a blocks-based language.
- The lack of authenticity held by blocks-based programming languages in the sense that the blocks-based language was not similar enough to traditional text-based languages to effectively educate others in the ways of computer programming.

Some efforts have been undertaken by researchers to bridge this gap in the past. One large area of focus has been the creation of text-based programming environments with some visual cues or elements. For example,

Kölling, Quig, Patterson, & Rosenberg have developed the BlueJ programming system [43] and Kölling has developed the Greenfoot programming environment [42]. Both environments seek to solve a particular problem. BlueJ is presented as a learning tool to aid with the difficulty of teaching object-oriented programming to novice programmers. Greenfoot has similar goals but targets itself at younger students and uses topics such as game development to help teach the concepts. Both tools aim to fill this educational gap, but neither are marketed as complete solutions, with the assumption that students will still migrate to a text-based language afterwards.

A pertinent question that could be asked here is "Is there a tool with a more longitudinal focus, or one which can be used interchangeably with a text-based language?". From the results of the literature review (see Chapter 2), it was found that visual programming languages are extremely beneficial when taught to the "right age group". The study also further pointed to the existence of this "educational gap" around the ages of 14-17 where neither language type is ideal. This was again further backed up by the results of the initial course delivery (see Chapter 3). Therefore, we have discerned that text-based languages have their weaknesses, but visual block-based languages also have their issues. This led to the logical conclusion of taking the best parts of both language types and merging them together into a so-called hybrid programming language. In particular, a hybrid blocks-text environment.

4.3 HYBRID JAVA DEVELOPMENT

Hybrid Java is built using the Snap! Platform [31]. Snap! was developed as an extensible reimplementaion of Scratch [53], with the added benefit of being able to create one's own custom visual blocks. This allows one to create some backend code using Snap!'s blocks, and then customise the

display (frontend) for the block. Using this toolset, the goal was to create blocks that mirror the syntax of the Java programming language.

Figure 4.1 presents an example of the "build your own block" feature of Snap!, where the functionality of a Java "for loop" is defined using Snap! blocks. The first line creates the visual display for the block, in this case the structure "for (int", followed by a variable "i", then an initial value, a condition and finally some update predicate. The next line puts the "init" number into the "i" variable. Finally, for the remaining lines, a Snap! "while loop" is used to mirror the functionality of the Java "for loop" by repeatedly iterating the statements and updating the variables until the loop condition is no longer met. Figure 4.2 presents the user facing display of the code block from Figure 4.1 (with some additional elements included). In terms of functionality, it is equivalent to a Java "for loop" which runs a piece of code several times.

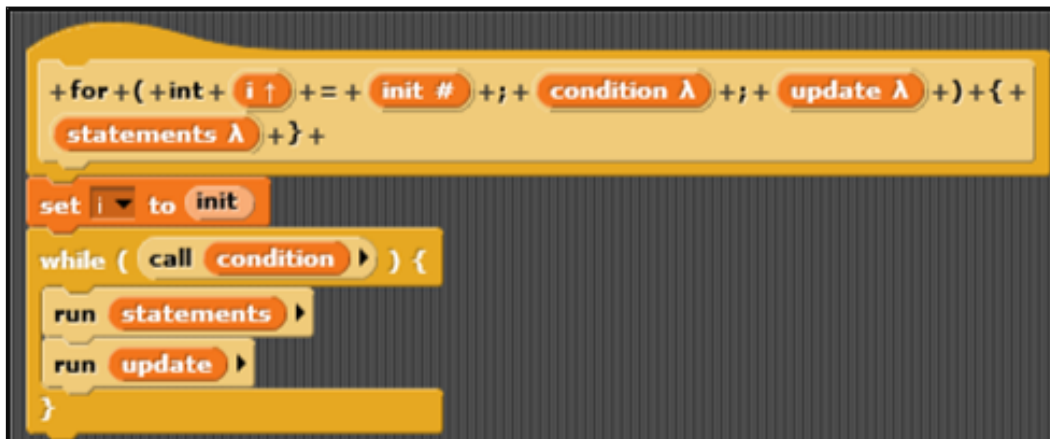


Figure 4.1: For Loop Backend in Snap!

With this process of creating blocks in Snap! the overall goal for the environment was to allow a user to create functional programs that mirror their Java counterparts. For the first major version of the environment, it should have enough blocks available to teach a CS1 module.

This led to the identification of the below list of concepts, which would need to be replicated for Hybrid Java.



Figure 4.2: For Loop Frontend in Snap!

1. A Java like structure (classes, main method)
2. Variable initialisation and modification
3. User Output (printing)
4. Operators
5. Comments
6. Conditional Statements
7. Iteration
8. Strings and String Methods
9. User Input - Scanners and Scanner methods
10. Arrays
11. Random Numbers

All of these concepts were included in the first iteration of the environment. A full set of all the finished code blocks are provided in Appendix C. For each code block, the backend code that would make the block work needed to be conceptualised, which proved more challenging in some instances than in others. For example, the Hybrid Java "if" block was easy to replicate as Snap! has its own "if" block concept. In this instance, the Java like frontend can simply call the Snap! version of if in the backend. Other concepts like "2D arrays" did not have a direct match in Snap! leading to a more detailed development, in this case, forcing Snap! to create a list of lists.

4.3.1 BLOCK SECTIONS

Once all of the blocks were created, a major aspect relating to the user experience with the environment related to the presentation of the blocks. By default, the Snap! user interface provides the following categories under which blocks can be placed: "Motion", "Looks", "Sound", "Pen", "Control", "Sensing", "Operators" and "Variables". These categories are usually filled with the existing Snap! programming blocks. However, for this programming environment, all of the original blocks could be hidden (to avoid confusion between Snap! blocks and Hybrid Java blocks). Hybrid Java blocks were then placed into the following categories, linking back in with the previously discussed list of concepts:

- *Control* - All of the Java structure (1), imports, printing (3), selection (6) and looping (7) blocks,
- *Sensing* - Blocks for commenting code (5),
- *Operators* - All operator (4) blocks (simple operations, incrementing, logical operators),
- *Variables* - All remaining variable blocks (basic types (2), Strings (8), User Input (9), Arrays (10), 2D Arrays (10), Random numbers (11)).

The Motion, Looks, Sound and Pen categories were left empty. These sections mostly refer to the movement of the Sprite in Snap! and as such, it was decided that they should be left empty. There were no obvious mappings between the Java language snippets and any of these sections.

It is important to note that by simply deleting blocks from the Hybrid Java command list, one can create a version of Hybrid Java with as many or as few blocks as desired by the teacher. A copy of this version can then be saved for use in a specific setting. Using this approach, one can ask students to create a program and give them a version of the Hybrid Java interface with the exact blocks they will need to use to create the required

program, potentially reducing the complexity of solving the problem. For example, we could ask the student to "write a program that prints all of the even numbers between 1 and 10". For a struggling novice, this might not be enough information to succeed in writing this program. However, if the student was given the Hybrid Java environment with only a for loop, if statement and some operator / variable blocks they might be able to piece it together in an easier fashion having now realised what blocks the required code will comprise of.

To create a program in the Hybrid Java environment, users simply drag and drop blocks to construct their code. Once a user has a completed piece of code, they simply need to click on the top block in the chain to run the program. For those familiar with blocks-based languages this will be very familiar. An example of a simple completed program and its corresponding output is shown in Figure 4.3.



Figure 4.3: Sample Hybrid Java program with output

It is also important to note that the "class" and "main" blocks do not have the same functionality in Hybrid Java as they do in Java. This is because the concept of a "class" in Snap! would simply be the entire programming area, and the concept of a method in Snap! would be simply having multiple separated lists of programming commands. These blocks are still retained however to aid the transition from Hybrid Java to Java and vice versa. They also delete old temporary variables like the one seen in 4.3.

4.4 HYBRID JAVA TESTING

Multiple phases of testing occurred with the tool. The first phase of testing occurred with a first year undergraduate Computer Science class. The second phase took place at the annual Computer Science Summer Camp at Maynooth University for students aged 12-17 years old. The third phase of testing occurred during a workshop delivered by the authors at the UKICER conference. At the end of these phases, we had semi-experienced programmers' feedback, novice programmers' feedback and researchers' feedback. This was an invaluable suite of testing to have completed in a short period of time and allowed us to sample a variety of people who would engage with the tool.

The tool was always seen as a multi-purpose tool, in that it could be used as a scaffolding tool for undergraduate students, but also as a tool that could be used with learners who had moved beyond the pure block-based programming languages. Having access to these groupings in a first year university class and a group attending a summer camp allowed us to survey these key users. An overview of the phases of testing will be provided in the following sections along with the discussion of iterative development of the tool after these testing phases. The overall purpose of this testing was to obtain feedback on the tool to help improve it while considering the effectiveness of the developed tool.

4.4.1 TEST 1: UNDERGRADUATE SURVEY

Once a full prototype of Hybrid Java was built, plans for testing the efficacy of the environment were put in place. In May 2019, at the end of semester two of the 2018/2019 academic year, a system test and survey was undertaken with the CS1 undergraduates (in module CS162). These students use Java as their programming language, hence they were the

perfect candidates to test the new environment. The primary goal for this test was to verify the ease of use of the user interface and to see if users with some Java knowledge could easily transition to using Hybrid Java.

For this test, 174 students were asked to solve a question, presented in Figure 4.4, using the new Hybrid Java programming environment. The students were only given a brief overview of the system and a quick description of how to create a program before being left to work through it themselves. This allowed us to observe if the system was intuitive to use for the students. As the students were already competent in Java, the main variable was the Hybrid Java system itself. The students were only given ten minutes to complete this task due to time constraints.

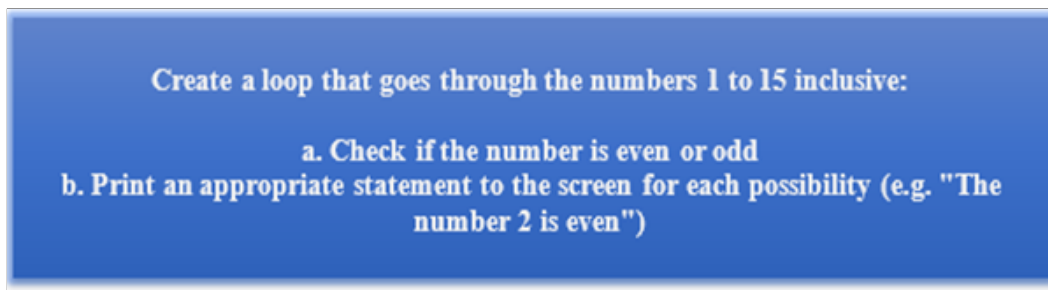


Figure 4.4: Undergraduate System Test Question

After attempting to solve the problem, the students were then asked to complete a short survey detailing their experiences of using the programming environment. The questions asked in the survey were as follows (note, for all "how difficult" questions (Q3-Q5), the response was a number between 1 and 10, where 1 represented "not difficult" and 10 represented "extremely difficult"):

1. How many years of programming experience do you have?
2. Did you complete the question?
3. How difficult was it for you to find the blocks you needed?
4. How difficult was it for you to understand the functionality of the blocks?
5. How difficult was it for you to complete the question?

6. Do you have any other comments about your experience?

For Q1, 79% of respondents said they had less than one year of programming experience, creating a good baseline for the knowledge level of the participants. Only 5% of participants claimed to have more than two years of experience. In total, 46% of the participants completed the question in the allotted time. This was likely due to the overhead of getting used to the system, coupled with the limited time they were given to complete the task. With more time, it is expected that more people would have completed the task. Some participants also completed the task very quickly, and these students stated that they had used Scratch or another visual programming language before, reducing their learning overhead.

The most interesting data was produced by the replies to Q3, Q4, and Q5. Figure 4.5 presents a summary of the results of all three questions, broken down by the rating band. For all three questions, most participants answers aligned with the lowest band (1-2 in difficulty).

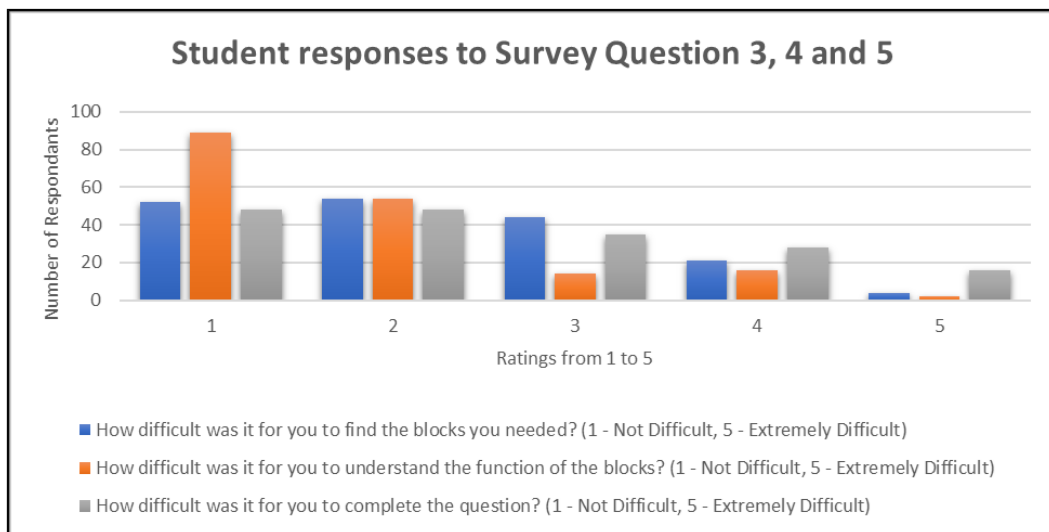


Figure 4.5: Graph of responses to Q3, Q4 and Q5 respectively

- For Q3, 60% of participants found it easy to find the required blocks.
- For Q4, 82% of participants understood the functionality of the blocks which makes sense given that all the students had been study-

ing Java for at least two semesters already, and should be familiar with Java code.

- Finally, for Q5, 55% of participants reported low difficulty in answering the question. This number may have been skewed upwards given that the time constraint may have been a factor in the "difficulty" of completing the question.

Some of the key comments that participants made included discussion on the ease of use of the tool ("Overall very easy to understand and use"), the usefulness of using the tool to introduce Java to young or novice programmers ("Seems like this could be a very innovative and novel way to teach Java") but also on the reduced usefulness for experienced programmers ("I would have been able to complete this task quicker in text"). These comments mostly align with the author's own thoughts on the placement of this programming environment in the pedagogy.

Overall, the results from this first phase of testing were very promising. They showed that there was an ease of transitioning from Java to Hybrid Java, with very few participants having issues with the system. There is an overhead to learning any new tool, so with more time given to practice, the participants would likely have all completed the task. Even with some students not completing the task, given the average difficulty rating of approximately 2 out of 5, it can be said that for semi-experienced programmers, Hybrid Java was a relatively easy tool to use. In test phase two, we will examine if the tool remains as intuitive for first time programmers.

4.4.2 TEST 2: COMPUTER SCIENCE SUMMER CAMP

In July 2019, the Computer Science department at Maynooth University ran their annual Summer Camp for 12-17 year old participants. This summer camp offers short 90-minute sessions in numerous topics related to Computer Science. This camp was the ideal place to test Hybrid Java in

a more detailed learning session with participants who had little to no programming experience. This would run very similarly to how the initial test sessions ran in Section 3.2.

This Summer Camp runs over a three-week period, with participants able to attend one, two or three weeks. During week 1 of the camp, a session on Java was ran. This session was the same one from Section 3.2.5. This session used BlueJ [43] as its Integrated Development Environment (IDE). During week 2 of the camp, a session on "Hybrid Java" was run which closely aligned with the material from the Java session but utilising the new tool. Both sessions covered an introduction to the language, language boilerplate, a "Hello World" program, Variables, Operators and Selection. After going through these topics, the students were tasked with creating a very simple calculator which performs some elementary calculation on two numeric variables based on the operator sign inside a third variable. They received support in this from a team of demonstrators who were working at the camp. The expected output for the code in both Java and Hybrid Java is presented in Figure 4.6.

Thirty-nine participants completed the Hybrid Java session at the Summer Camp. Of these thirty-nine participants, seventeen had attended the Java session. Once again, after completing the Hybrid Java session, the participants were asked to fill out a quick survey to collect their opinions on both the session and the environment itself. Some of the key results from this survey were that:

- Thirty-eight out of the thirty-nine participants either enjoyed or somewhat enjoyed the session, leaving only one participant who did not enjoy the session.
- Thirty-eight participants found the Hybrid Java system "approachable". An interesting comment on this was that "It began confusing but eventually I got the hang of it and realised how straight forward it was".

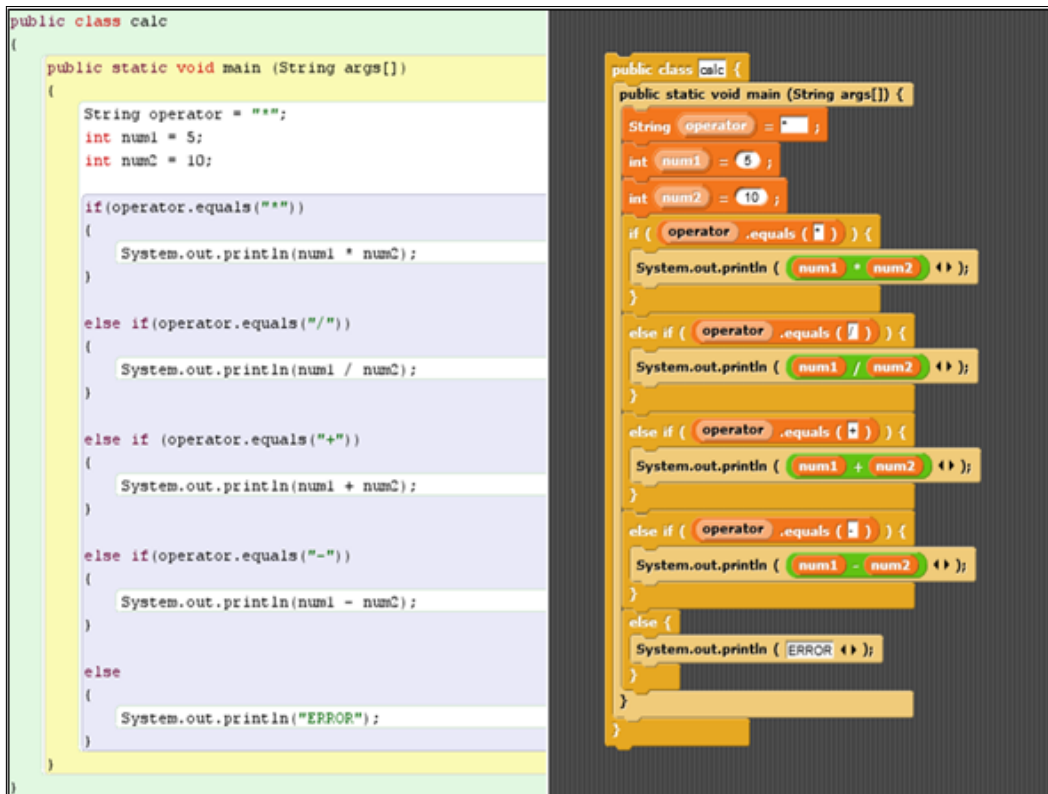


Figure 4.6: Simple Calculator program in both Java and Hybrid Java

- Of the seventeen participants who had attended the Java session the week prior, seven of them preferred the Java session, five of them preferred the Hybrid Java session and five of them thought they were about the same enjoyment level.

The primary questions of interest asked all participants (n=39) to rate the difficulty of the Hybrid Java session on a scale of 1-10 (1 representing "not at all" to 10 representing "very difficult"), and, for those who were present the previous week (n=17) to rate the difficulty of the Java session on a scale of 1-10. The results were that these questions showed a difficulty rating of 4.69 for Hybrid Java, and 5.35 for Java. While this difference is not statistically significant ($p=0.366$, $p>.05$), it follows the expected trend. Despite the identical content of the two sessions, the Hybrid Java session was considered somewhat easier by these participants. It is important to note that given that the Java session was introduced first, there is a

possibility of some recency bias in that the 17 students could have rated the Hybrid Java session "easier" due to having already seen the material in the Java session. When these seventeen participant ratings are removed from the average calculation a rating of 4.82 resulted. While this result is slightly higher than the 4.69 reported above, it is still much lower than the corresponding rating for Java. This suggests that Hybrid Java could have some ease of use that Java doesn't.

4.4.3 TEST 3: UKICER WORKSHOP

In September 2019, a workshop relating to the Hybrid Java programming environment was delivered at the UK and Ireland Computing Education Research (UKICER) conference in Canterbury, England [34]. This was less about testing the system but more about collecting feedback from other academics in the area prior to further development of the tool. The goal of this phase was to gauge the interest in the Computer Science Education community for the tool, and to give the tool a thorough debugging.

The workshop was a two-hour session, of which the first thirty minutes consisted of delivering a presentation on the tool. Subsequently, a booklet of information and sample programs was provided to the attendees along with the tool itself. They were encouraged to practice constructing programs that they might write themselves in their CS1 lectures. They were also encouraged to try some difficult code sequences to vigorously test the system. Finally, the last thirty minutes of the workshop were dedicated to collecting feedback. This was done via a survey and via a Padlet wall. Questions related to bug detection in the tool, general feedback on the tool and feedback on the workshop session itself. Some of these can be found as part of the Padlet linked in the final paragraph of Appendix C.

Three participants attended the workshop, which limited the amount of feedback that was received. However, the consensus in the room was that

it was an enjoyable workshop and an interesting idea for a tool but not ideal for the set of participant's student groups. One of the key comments that arose during this workshop was that "It takes a small learning curve to understand how to use the blocks in the environment and how to interconnect them. However, I see it with the eyes of a developer, I am curious how new learners grab it." This is an interesting statement given what we have seen in the summer camp test that new learners do engage with the tool when delivered in an instructor-led manner.

Some bugs and issues were also discovered, as was the hope. Some examples of these bugs included being able to use length operators and String operators on integers, not being able to assign true or false values to Booleans after their initial creation and some suggestions of missing blocks. Some of these issues were addressed immediately after returning from the conference, as discussed in Section 4.4.4, while others are noted as part of the future work in Section 5.

Overall, the workshop was a success and some strong feedback was provided for the tool from the researchers perspective.

4.4.4 ITERATIVE DEVELOPMENT

Following the completion of all three phases of testing, as well as some minor changes in between testing phases, a round of iterative development was undertaken on the Hybrid Java programming environment. Some of the main changes that were made included:

1. Reordering of all the blocks into a more logical order. The original ordering of the blocks was based on the order of block creation. This new block order (which is the same as the version shown in Appendix C) is focused more on the order of the main teaching concepts. This makes it easy for the educator to hide the blocks that they don't need until a later phase of their teaching.

2. Redevelopment of the 2D Arrays blocks. During testing, it was discovered that the original 2D array blocks did not function as expected. They were creating a long list of elements rather than a "list of lists". This led to issues when accessing elements. This was rectified in this change.
3. Inclusion of more versions of each Array type. In the original version, there were only arrays for *"int"*, *"double"* and *"String"*. This version also provides for *"float"*, *"long"*, *"char"* and *"Boolean"* Array functionality.
4. Moving the comments blocks to their own section. Originally the comments blocks were on top of the "Control" section. These were moved to the, as yet empty, "Sensing" section as feedback pointed out that they didn't make sense where they were.
5. Creating an external version of the tool which can be used externally to the Snap! website. Snapp! [32] is an external tool which allows for the creation of executable projects. This tool allowed for export of the Hybrid Java project into an external file for ease of use with classes.

The version of the tool which is referenced in Section 4.7 includes all of these discussed enhancements. Some other minor changes were made along the way such as the fixing of typos and very minor functionality changes etc. All of these changes are also present in the current iteration. This version of Hybrid Java is fully usable for teaching early introductory programming concepts.

4.5 BUILDING THE HYBRID JAVA CURRICULUM

In Chapter 3, the development of a Java curriculum and a Snap! curriculum was discussed in detail. Once the efficacy of Hybrid Java had been verified through all aforementioned testing phases and iterative develop-

ment phases, work began on creating the Hybrid Java curriculum. This curriculum fully mirrors the other two, with all material being the same. The full curricula can be seen in Appendix B.

As with the initial course development, the material includes slide decks, class plans, assessment, examples and more. Due to time constraints in the project, this course was never delivered to a class but will be discussed in the future work Section 5. With this course material created, and Hybrid Java developed to a high standard, the "final product" of this thesis could be created. This we will call the "Hybrid Java Package" (HJP), which will be discussed more in Section 4.7.

4.6 OTHER USES FOR HYBRID JAVA

The benefit of the Hybrid Java approach is that if one wanted to transition to a fully text-based language, like Java, there is little to no overhead to doing so. This is for several reasons. Firstly, the text on the blocks is the same as the "text" in the Java language. Secondly, it has been shown in the literature that starting with a visual programming language is a good option for introducing programming concepts before going to a full text-based language [5, 84]. It would be perfectly feasible to have a second programming course (Object-Oriented Programming or Algorithms and Data Structures) run in Java after initially learning Hybrid Java. The students would also have learned the same threshold concepts while undertaking a Hybrid Java CS1 course as would be expected from a text based CS1 course, so there will be no gap in the knowledge.

It has been shown that a focus on concepts and teaching methodologies is often more important than the choice of FPL [47, 68]. In particular, one interesting result came from Giordano and Maiorana [26] where they taught a course that started with Scratch and progressed to using the C language. This was done to allow a focus on concepts, and the result was

that when the students transitioned to C, they made less errors than would normally occur upon first exposure to the C language (during their normal teaching routine).

While it is non-traditional to use a visual or hybrid programming environment in a third-level CS₁ environment, one area where visual languages are commonly used is with teaching a programming language to young learners. As discussed in Section 4.1 there is a point where younger students disconnect from pure visual programming languages but aren't quite ready for a text-based programming language yet [13]. The Hybrid Java environment perfectly fills this educational gap.

Hybrid Java can certainly be used as a First Programming Language on its own, but it is possible that one of its greatest strengths lies in the area of intervention and student support. I have been involved in the running of the "Computer Science Centre" at Maynooth University [65] for a number of years. This is a centre that provides support for students who are struggling with the CS₁ course material or are looking for some extra challenges. In the centre we tend to find one of the most common issues a Java CS₁ student faces are the verbosity of the language, which opposes what a good FPL should be [29]. By giving the student the Hybrid Java tool, while they are learning a Java course, we would expect to see an ease in the complexity for them. The browsable nature of the commands [93] and the ease of use of drag-and-dropping the blocks in to place [76] combined with an already existing knowledge of how the Java programming language works provides a student with as much opportunity to understand as possible. This is something that needs to be tested fully as part of future work to see just how strong of an intervention tool Hybrid Java is.

4.7 THE HYBRID JAVA PACKAGE

The HJP is something that can be offered to individual educators, institutions and anyone else interested in using Hybrid Java for teaching or intervention purposes. They will be provided with all of the course materials, the Hybrid Java tool itself, instructions to set it up and instructions on how to delete blocks (temporarily) where necessary for a problem. This gives a level of control to the educator to utilise the tool how they see fit.

With the Hybrid Java curriculum, the educator can teach a short course in programming just like they can with the Java course and the Snap! course. They will be able to make their own judgements on it as a FPL, as an alternative to VPL courses they might otherwise teach or as a bridge between VPL and TPL skillsets.

The other major usecase possible for the HJP is as an intervention for third-level programming students. The idea here is that students who are struggling with the verbosity of Java can be given a version of Hybrid Java with the appropriate amount of blocks present for their current progression is CS₁ and solve their exercises as normal but using Hybrid Java to lessen the difficulty. A sample of this package (in its full format, with all blocks present) can be found as part of the full materials in Appendix B.

4.8 SUMMARY

In this chapter, a new hybrid programming environment called "Hybrid Java" has been presented. Between the existing literature in the area of hybrid programming languages, and the multi-phase testing performed here, we have shown that this tool could fill a much needed educational gap and could provide an ease of learning for both struggling students and as a FPL. There is also evidence to suggest that the main strengths of the environment may lie in the realm of programming intervention.

Test one verified that students who are already studying Java could easily transition to Hybrid Java. There was very little overhead in learning how to use the tool, which was particularly positive since the students were only given 10 minutes to complete a task with little introduction. Test two showed that teaching Hybrid Java as a FPL had promise, particularly with school students. Despite covering the same material, the Hybrid Java course was considered marginally easier. Test three helped to point out some bugs with the tool and helped provide insight on how other educators perceive the tool. The number of participants at this workshop was low, and as such there is further work to be done.

Overall, the initial development and testing of the Hybrid Java tool has been very positive. Some strong feedback and initial opinions have been received. All the testing phases were successfully completed, and some encouraging results were received. This has allowed for the discovery of where the tool best lies in the pedagogy of programming. A strong plan can be laid out for future work and future development. Perhaps soon we will see more hybrid programming languages being used as either a FPL or as an intervention tool, as it certainly seems that using such a language can have positive outcomes for students.

Part IV

CONCLUSIONS & FUTURE WORK

CONCLUSIONS AND FUTURE WORK

This chapter will present the conclusions and future work components of this project. It will examine the research questions and discuss these in detail before suggesting possible future work that could be carried out with the project to further enhance Hybrid Java, and to further the testing and delivery of all three courses to further verify the "linear line" hypothesis, discussed below.

5.1 CONCLUSIONS

In this thesis, it has been shown that there are some differences when teaching a VPL versus teaching a TPL. Outcomes when learning both language can very much be the same, but this body of work has shown that a TPL is more complex to understand, particularly for younger learners. As discussed in Section 4.4.2, Hybrid Java appears to have some ease of use that Java doesn't, while not being considered quite as simple as Snap! was. This leads us to the "Linear Line Hypothesis".

The Linear Line Hypothesis is a hypothesis held by the author that Hybrid Java (and in general, hybrid programming languages) rests in the middle of a VPL (Snap!) and a TPL (Java) in terms of difficulty to understand and ease of use. The testing phase in Chapter 4 aligns with this hypothesis. A deeper analysis of Hybrid Java, and additional large-scale testing phases and data analysis would be beneficial to proving this point. However, the initial data very much aligns in this direction.

5.1.1 RESEARCH QUESTIONS

In Section 1.3, the research questions for this project were presented. In this section, an overview of the answers to these questions will be discussed. As a reminder, the questions were as follows:

1. Does the choice of language type affect the performance of learners as they learn programming for the first time? (RQ₁)
2. Do visual programming languages, given their close interconnection with mathematics in terms of delivery, result in more effective and higher performance outputs than textual languages? (RQ₂)
3. Is there an alternative approach to the traditional programming language types, and if so, how effective would this approach be? (RQ₃)

For RQ₁, it can be reasoned that yes, the choice of language can affect the performance of learners as they learn programming for the first time. Data presented in Chapter 3 has shown that Snap! is statistically significantly easier to learn for Transition Year students than Java in certain scenarios. This is an important result, particularly because most research on visual programming languages demonstrates the "traditional" use cases for these languages as discussed in Chapter 2. For example, designing simple games that interact with the sprites. This research examined the usage of Snap! when teaching first principles programming concepts. This means that we can use a VPL to teach core programming, while making the learning easier for our younger students. It would be valuable to see more educators taking this approach. For older learners (University age particularly), the data is a little less clear with much background research highlighting that TPL gave better outcomes.

For RQ₂, conclusions show that yes, VPLs can result in higher performance outputs compared to TPLs, for certain age groups or types of students. Students who took the assessment in the Snap! course in Section

3.3.2 not only did so with a higher average grade, but also did so in a faster average time. It is vital to remember that the programming exercises for both the Snap! course and the Java course were designed to be equivalent. The only differences in requirement were the language type used. Everything was taught by the same educator, delivered in the same manner, with the same topics and same examples. The only variables that changed was the language type and editor. This result shows that using a VPL as a first principles programming tool can be very effective with the correct group.

For RQ₃, it has been shown that a Hybrid model of programming is an effective one. In particular, Hybrid Java was developed and has had positive reactions from all members of the CS community (students and educators alike). This Hybrid model allows for a reduction in verbosity and overhead when teaching a TPL, without removing any of the features or styling of a TPL. This allows for easy transition from the Hybrid model to the Textual model, much easier than the leap from VPL straight to TPL. More research is needed to fully verify the "place" where Hybrid programming will be at its most effective. In the following sections, this will be discussed in detail.

5.1.2 PLACE FOR HYBRID JAVA

Hybrid Java is a tool that has potential for multiple different avenues of success. These include:

- As a First Programming Language (particularly for youth)
- As an intervention tool (for struggling novice programmers at University level)
- As a "simplification" tool by showing only a small subset of blocks necessary to solve a problem

A FPL is the most obvious function of the tool, given that this is what it was developed for. Initial testing of delivering this as a short course shows that there should be good outcomes when using Hybrid Java. It will effectively teach programming concepts while also making learning Java easier.

In Section 4.7, the Hybrid Java Package was discussed. This is one major output of the project and will help to further test Hybrid Java as a FPL. This package provides educators with an off-the-shelf tool which they can bring into their classroom setting to help students as they learn programming. They can use the provided instructions and materials to make it their own, and teach programming in their own style utilising it.

Intervention is an important element of third level programming support. The author is very familiar with supporting students, as will be discussed more in Section 5.3.2. Hybrid Java has great potential in this space. It can be used to support novice programmers through the obfuscation of blocks, easing of verbosity and other key elements. This could even be of great assistance to students with dyslexia or other reading difficulties, reducing the complexity for them too.

Further to the above point, by obfuscating blocks we can create a simplified version of the tool (which can be built upon each week) to provide to students. They can look at the list of blocks available to them and solve a problem using only this subset. This reduces the cognitive load required. Normally, with a TPL you would need to know all of the code and how to write it, and then solve the problem on top of that.

5.2 CHALLENGES

No project delivery comes without it's challenges. In this section, some of the challenges that arose in this project will be evaluated.

The primary challenge in this project was an extended deferral period. Due to work commitments, work on this project stopped in September 2019 and no re-registration occurred until October 2022. Even so, very little time was available during the academic year 2022-23. However, through some dedicated time commitment and some great support from my supervisor, my other colleagues and my family and friends alike, completion was eventually achieved.

Some smaller challenges arose in the teaching phase of the project. There were major differences in teaching style required in the public schools attended versus the private ones. It was essential that this did not effect the data, but it did prove challenging at the time.

Finally, a lack of time and lack of ease to get into schools in a post-COVID-19 world meant that Hybrid Java did not get as much testing time as would have been desired. This is something that is addressed in Section 5.3, and is something the author is passionate about continuing as part of ongoing research in the future.

5.3 FUTURE WORK

This body of work has many potential avenues for continuation. This section will discuss these ideas in detail. Some of this will be continued over time by the author, other elements are available for others to experiment with and improve upon.

5.3.1 FULL HYBRID JAVA DELIVERY

The next major phase of comparison testing would involve teaching the Java and Snap! curricula in some schools, and Hybrid Java in others. A large scale study producing much data from "similar" school environments will help conclusively determine the "difficulty" of Hybrid Java over a lon-

itudinal time-frame. It is expected that Hybrid Java will be easier than Java but more difficult than Snap!, as discussed in earlier chapters.

5.3.2 FURTHER TESTING PHASES

Another phase of planned testing could be in the Computer Science Centre (CSC) at Maynooth University, to see how students who are currently learning Java interact with it and how they perceive the benefits of the tool. This would help to further examine the benefits of Hybrid Java as a support tool, rather than a teaching tool. Is this where Hybrid Java particularly shines? Is it better suited as support for those who are otherwise struggling with learning text-based Java?

The author has experience with working in (and managing) the CSC for a number of years. There is definite scope for a tool such as this to assist struggling programmers with their work. One of the major issues that students have in our CS1 module is difficulty in typing. Minor typos or poor indentation have lasting effects on their learning and progress. More details on the CSC can be found in a number of publications which the author contributed on [71, 72].

It would be interesting to see if students who are currently struggling with the course material have a different outcome from interacting with the tool. Perhaps it will provide them a different perspective on the challenges they are facing. In this testing phase, a comparison would be conducted between students who only interact with Java, students who are briefly introduced to Hybrid Java but still primarily work in Java and students who heavily rely on Hybrid Java after their initial introduction. In particular, it would be interesting to see if a like-for-like pair of students (determined through some metrics such as Leaving Certificate results, similar background, similar first semester results etc.) perform differently in

the final examination with the only changing variable being the language tool that was used.

5.3.3 OBJECT ORIENTED PROGRAMMING

One of the strongest benefits of the Hybrid Java tool is its extensibility. This is one of the key factors in a good first programming language [55]. Future development of the tool could involve the creation of additional blocks. One area of focus that is not currently present is the ability to code in an object-oriented manner. Some research will need to be undertaken to determine the feasibility of this. Some other blocks could also be added based on feedback from educators and students alike on missing blocks. Similarly, as there is recognition for the requirement of new blocks, they can be added.

Another tie-in concept to this is to implement libraries or algorithms that "allow" more complex code to be ran in a simple fashion. Hybrid Java is perfectly suited to this given that you can abstract the code behind a visual block which the students simply need to place. For example, a student could drag-and-drop a "Bubble Sort" block which takes input and bubble sorts that input, all without needing to know the complexity of the code at first.

5.3.4 HYBRID JAVA TO JAVA TRANSLATION

Given the ease of extensibility of Snap!, another interesting task would be to create a tool add-on that could convert the visual Hybrid Java blocks into fully functional Java code. This code could then be taken and utilised in an external Java compiler and should work like standard Java programming. Alternatively, it could be examined whether a Java compiler could be built underneath Snap! so that running Hybrid Java is more authentic,

so that it has the ability to provide true errors and so that students would have more understanding if their code failed. This would be a difficult implementation, but it would provide major quality of life improvements if it were possible.

Similarly, there is potential for the reverse to be possible. Using Javascript (which already interfaces with Snap! in the background), a "block" could be built which reads in a Java file and converts it to equivalent Hybrid Java blocks. This would also be a challenging implementation, but would have many interesting use cases if possible. From students creating an "easier to modify" code sequence that they can then interface with in the drag-and-drop methodology, to students inputting sample code from their teacher with a task to then make modifications or fill in the gaps.

5.3.5 VISUAL FEEDBACK

Another key task for future development is providing better visual feedback to users. When a program is run, it is important to see what went wrong if a user doesn't get the expected output. This is an area that Hybrid Java is currently lacking in. Visual feedback has been shown to make learning hard concepts easier [40]. This is something that can be improved upon. As well as visual feedback, some functional "Java-like" error handling would be helpful. Right now, if you are missing the non-functional "main method" or "class" block, no errors occur. Implementation of Java errors such as "Main method not found in class", "At least one public class is required" and other such errors could be undertaken. This would help to align Hybrid Java more closely with Java and make the transition between the tools even more seamless. Meaningful error handling is important to the understanding and comprehension of a language, and in the ability to fix coding problems by oneself [47]. However, the option could be present

for the educator to turn these errors off to provide an "easier" introduction if desired.

5.3.6 OTHER LANGUAGE OPTIONS

With the framework for Hybrid Java created, it would now be much simpler to design other programming language using the same model. For example, Hybrid Python could be developed in much the same way, in many cases simply needing to change the label of the block to look like Python code, while the underlying Snap! code can remain the same. This can be augmented with (or in some cases replaced by) additional blocks that aren't ordinarily present in Java. For example, with Python, Lists would replace Arrays. Even functional languages could be modelled this way. The opportunities are vast in this sphere.

5.3.7 HJP SHARING

Finally, and perhaps most importantly, there is a plan to send on the completed "Hybrid Java Package" to second-level educators. This package contains everything a teacher needs to get started with Hybrid Java: instructions for the tool itself, class plans, in-class slides and exercises, homework sheets and a final assessment and feedback survey.

By sending this on to educators free of charge, with the only request being that they send back results of the survey and anonymised exam results (simply grades would suffice), we can see how a large range of students perform while using this course. This will be an element of ongoing large-scale data collection. From this data, we can analyse the position of Hybrid Java within the pedagogy, and confirm (or deny) the initial analysis that it fits in between a visual programming language and a textual programming language in terms of difficulty.

BIBLIOGRAPHY

- [1] Efthimia Aivaloglou and Felienne Hermans. “How kids code and how we know: An exploratory study on the scratch repository”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM. 2016, pp. 53–61.
- [2] Bedour Alshaigy, Samia Kamal, Faye Mitchell, Clare Martin, and Arantza Aldea. “Pilet: an interactive learning tool to teach python”. In: *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM. 2015, pp. 76–79.
- [3] Marvin Andujar, Luis Jimenez, Jugal Shah, and Patricia Morreale. “Evaluating visual programming environments to teach computing to minority high school students”. In: *Journal of Computing Sciences in Colleges* 29.2 (2013), pp. 140–148.
- [4] Jeraline Annirroot and M R de Villiers. “A study of Alice: A visual environment for teaching object-oriented programming”. In: *Proceedings of the IADIS International Conference on Information Systems 2012* (2012).
- [5] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. “From scratch to “real” programming”. In: *ACM Transactions on Computing Education (TOCE)* 14.4 (2015), p. 25.
- [6] Muhammad Ateeq, Hina Habib, Adnan Umer, and Muzammil Ul Rehman. “C++ or Python? Which One to Begin with: A Learner’s Perspective”. In: *Teaching and Learning in Computing and Engineering (LaTiCE), 2014 International Conference on*. IEEE. 2014, pp. 64–69.

- [7] Barbara P Benson. *How to meet standards, motivate students, and still enjoy teaching!: four practices that improve student learning*. Corwin Press, 2003.
- [8] Joe Bergin, Achla Agarwal, and Krishna Agarwal. “Some deficiencies of C++ in teaching CS₁ and CS₂”. In: *ACM SIGPlan Notices* 38.6 (2003), pp. 9–13.
- [9] Marat Boshernitsan and Michael Downes. “Visual Programming Languages: A Survey”. In: *Computer Science Division (EECS)* (2004).
- [10] Berkeley University of California. *Beauty and Joy of Computing: An AP CS Principles Curriculum*. [Online; accessed 2023-08-08]. 2023. URL: <https://bjc.edc.org/>.
- [11] Nora Ide McAuliffe Carl O’Brien Joe Humphreys. *Concern over drop-out rates in computer science courses*. [Online; accessed 2023-08-02]. 2016. URL: <http://www.irishtimes.com/news/education/concern-over-drop-out-rates-in-computer-science-courses-1.2491751>.
- [12] Martin C Carlisle, Terry A Wilson, Jeffrey W Humphries, and Steven M Hadfield. “RAPTOR: a visual programming environment for teaching algorithmic problem solving”. In: *Acm Sigcse Bulletin* 37.1 (2005), pp. 176–180.
- [13] Joey CY Cheung, Grace Ngai, Stephen CF Chan, and Winnie WY Lau. “Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students”. In: *ACM SIGCSE Bulletin*. Vol. 41. ACM. 2009, pp. 276–280.
- [14] Charmain Cilliers, André Calitz, and Jean Greyling. “The effect of integrating an Iconic programming notation into CS₁”. In: *ACM SIGCSE Bulletin*. Vol. 37. ACM. 2005, pp. 108–112.
- [15] Daniel C Cliburn. “Student opinions of Alice in CS₁”. In: *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*. IEEE. 2008, T3B–1.

- [16] Stephen Cooper. "The design of Alice". In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010), p. 15.
- [17] National Council for Curriculum and Assessment. *Leaving certificate computer science draft curriculum specification*. [Online; accessed 2023-08-02]. 2017. URL: <https://www.ncca.ie/media/3184/lc-computerscience.pdf>.
- [18] Tebring Daly. "Minimizing to maximize: an initial attempt at teaching introductory programming using Alice". In: *Journal of Computing Sciences in Colleges* 26.5 (2011), pp. 23–30.
- [19] Stephen Davies, Jennifer A Polack-Wahl, and Karen Anewalt. "A snapshot of current practices in teaching the introductory programming sequence". In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM. 2011, pp. 625–630.
- [20] Katherine Donnelly. *Computer Science finally on the way for Leaving Cert students*. [Online; accessed 2023-08-02]. 2016. URL: <http://www.independent.ie/irish-news/education/computer-science-finally-on-the-way-for-leaving-cert-students-34576921.html>.
- [21] Mark Dorling and Dave White. "Scratch: A way to logo and python". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM. 2015, pp. 191–196.
- [22] Wendy M DuBow, Beth A Quinn, Gloria Childress Townsend, Rosario Robinson, and Valerie Barr. "Efforts to make computer science more inclusive of women". In: *ACM Inroads* 7.4 (2016), pp. 74–80.
- [23] Chaker Eid and Richard Millham. "Which Introductory Programming Approach Is Most Suitable For Students: Procedural Or Visual Programming?" In: *American Journal of Business Education (Online)* 5.2 (2012), p. 173.

- [24] Stefano Federici. “A minimal, extensible, drag-and-drop implementation of the C programming language”. In: *Proceedings of the 2011 conference on Information technology education*. ACM. 2011, pp. 191–196.
- [25] Ryan Garlick and Ebru Celikel Cankaya. “Using alice in CS1: a quantitative experiment”. In: *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. ACM. 2010, pp. 165–168.
- [26] Daniela Giordano and Francesco Maiorana. “Use of cutting edge educational tools for an initial programming course”. In: *Global Engineering Education Conference (EDUCON), 2014 IEEE*. IEEE. 2014, pp. 556–563.
- [27] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. “Why complicate things?: introducing programming in high school using Python”. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc. 2006, pp. 71–80.
- [28] JH Greyling, CB Cilliers, and AP Calitz. “B#: The development and assessment of an iconic programming tool for novice programmers”. In: *Information Technology Based Higher Education and Training, 2006. ITHET’06. 7th International Conference on*. IEEE. 2006, pp. 367–375.
- [29] Diwaker Gupta. “What is a good first programming language?” In: *Crossroads* 10.4 (2004), pp. 7–7.
- [30] Alden H Harken. “To block or not to block? That is the question”. In: *The Journal of Thoracic and Cardiovascular Surgery* 149.4 (2015), pp. 1040–1041.
- [31] Brian Harvey and Jens Mönig. *Snap! (Build Your Own Blocks)*. [Online; accessed 2023-08-22]. 2023. URL: <https://snap.berkeley.edu/about>.

- [32] A Hintze and B Romagosa. *Snapp!* [Online; accessed 2023-08-23]. 2023. URL: <http://snapp.citilab.eu/>.
- [33] John M Hunt. "Python in CS1-not". In: *Journal of Computing Sciences in Colleges* 31.2 (2015), pp. 172–179.
- [34] *Hybrid Java Programming: A Visual-Textual Programming Language Workshop*. Accessed: 2023-07-31. URL: <https://www.cs.kent.ac.uk/events/2019/UKICER2019/workshopchoice.html>.
- [35] Mirjana Ivanović, Zoran Budimac, Miloš Radovanović, and Miloš Savić. "Does the choice of the first programming language influence students' grades?" In: *Proceedings of the 16th International Conference on Computer Systems and Technologies*. ACM. 2015, pp. 305–312.
- [36] Karin Johnsgard and James McDonald. "Using alice in overview courses to improve success rates in programming i". In: *Software Engineering Education and Training, 2008. CSEET'08. IEEE 21st Conference on*. IEEE. 2008, pp. 129–136.
- [37] Sara B Johnson, Robert W Blum, and Jay N Giedd. "Adolescent maturity and the brain: the promise and pitfalls of neuroscience research in adolescent health policy". In: *Journal of adolescent health* 45.3 (2009), pp. 216–221.
- [38] Staffs Keele. "Guidelines for performing systematic literature reviews in software engineering". In: *Technical report, Ver. 2.3 EBSE Technical Report*. EBSE. sn, 2007.
- [39] James D Kiper, Elizabeth Howard, and Chuck Ames. "Criteria for evaluation of visual programming languages". In: *Journal of Visual Languages & Computing* 8.2 (1997), pp. 175–192.
- [40] Myungsook Klassen. "Visual approach for teaching programming concepts". In: *9th International conference on Engineering Education*. Citeseer. 2006, pp. 23–28.

- [41] Roxane Koitz and Wolfgang Slany. “Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers”. In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM. 2014, pp. 21–30.
- [42] Michael Kölling. “The greenfoot programming environment”. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010), pp. 1–21.
- [43] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. “The BlueJ system and its pedagogy”. In: *Computer Science Education* 13.4 (2003), pp. 249–268. DOI: 10.1076/csed.13.4.249.17496.
- [44] Stephen D Krashen. “Principles and Practice in Second Language Acquisition”. In: *Learning* 46.2 (1982), pp. 327–69.
- [45] Charalampos Kyfonidis, Nektarios Moutoutzis, and Stavros Christodoulakis. “Block-C: A block-based visual environment for supporting the teaching of C programming language to novices”. In: (2015).
- [46] Mikko-Jussi Laakso, Erkki Kaila, Teemu Rajala, and Tapio Salakoski. “Define and visualize your first programming language”. In: *Advanced Learning Technologies, 2008. ICALT’08. Eighth IEEE International Conference on*. IEEE. 2008, p. 324.
- [47] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A study of the difficulties of novice programmers”. In: *Acm Sigcse Bulletin*. Vol. 37. ACM. 2005, pp. 14–18.
- [48] Vambola Leping, Marina Lepp, Margus Niitsoo, Eno Tõnisson, Varmo Vene, and Anne Villems. “Python prevails”. In: *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*. ACM. 2009, p. 87.

- [49] Colleen M Lewis. “How programming environment shapes perception, learning and goals: logo vs. scratch”. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM. 2010, pp. 346–350.
- [50] Andrew Luxton-Reilly. “Learning to program is easy”. In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 2016, pp. 284–289.
- [51] Lifelong Kindergarten Group at the MIT Media Lab. *Scratch Statistics*. [Online; accessed 2023-08-02]. 2023. URL: <https://scratch.mit.edu/statistics/>.
- [52] Louis Major, Theocharis Kyriacou, and O Pearl Brereton. “Systematic literature review: teaching novices programming using robots”. In: *IET software* 6.6 (2012), pp. 502–513.
- [53] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. “The scratch programming language and environment”. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010), p. 16.
- [54] Linda Mannila, Mia Peltomäki, and Tapio Salakoski. “What about a simple language? Analyzing the difficulties in learning to program”. In: *Computer Science Education* 16.3 (2006), pp. 211–227.
- [55] Linda Mannila and Michael de Raadt. “An objective comparison of languages for teaching introductory programming”. In: *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. ACM. 2006, pp. 32–37.
- [56] Yoshiaki Matsuzawa, Takashi Ohata, Manabu Sugiura, and Sanshiro Sakai. “Language migration in non-cs introductory programming through mutual language translation environment”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM. 2015, pp. 185–190.

- [57] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. "Habits of programming in scratch". In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM. 2011, pp. 168–172.
- [58] Mendeley. *Mendeley Reference Manager*. [Online; accessed 2023-08-02]. 2017. URL: <https://www.mendeley.com/>.
- [59] Susana Montero, Paloma Díaz, David Díez, and Ignacio Aedo. "Dual Instructional Support Materials for introductory object-oriented programming: classes vs. objects". In: *Education Engineering (EDUCON), 2010 IEEE*. IEEE. 2010, pp. 1929–1934.
- [60] Aidan Mooney, Joe Duffin, Thomas J Naughton, Rosemary Monahan, James F Power, and Phil Maguire. "PACT: An initiative to introduce computational thinking to second-level education in Ireland". In: (2014).
- [61] Aidan Mooney, Mark Noone, and Emily O'Regan. "Creation of a Hybrid Programming Language". In: (2019). Accessed: 2023-07-31. URL: <http://ilta.ie/wp-content/uploads/2019/05/Edtech-19-Book-of-Abstracts-1.pdf>.
- [62] Paul Mullins, Deborah Whitfield, and Michael Conlon. "Using Alice 2.0 as a first language". In: *Journal of Computing Sciences in Colleges* 24.3 (2009), pp. 136–143.
- [63] Uolevi Nikula, Jorma Sajaniemi, Matti Tedre, and Stuart Wray. "Python and roles of variables in introductory programming: experiences from three educational institutions". In: *Journal of Information Technology Education* 6 (2007), pp. 199–214.
- [64] Keith Nolan and Susan Bergin. "The role of anxiety when learning to program: a systematic review of the literature". In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. ACM. 2016, pp. 61–70.

- [65] Keith Nolan, Aidan Mooney, and Susan Bergin. "Facilitating student learning in Computer Science: large class sizes and interventions". In: 2015.
- [66] Mark Noone. *Hybrid Java Online Tool*. [Online; accessed 2023-08-21]. 2023. URL: https://snap.berkeley.edu/snap/snap.html#present:Username=marknoone&ProjectName=HYBRID_JAVA_NEWORDER.
- [67] Mark Noone and Aidan Mooney. "First Programming Language : Visual or Textual?" In: *International Conference on Engaging Pedagogy (ICEP)*. 2017. URL: http://icep.ie/wp-content/uploads/2018/01/ICEP_2017_paper_Mark_N.pdf.
- [68] Mark Noone and Aidan Mooney. "Visual and textual programming languages: a systematic review of the literature". In: *Journal of Computers in Education* 5 (2018), pp. 149–174.
- [69] Mark Noone and Aidan Mooney. "First Programming Language- Java or Snap? A Short Course Perspective". In: *10th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2019)*. Bangkok, 2019. DOI: 10.5176/2251-2195_CSEIT19.148.
- [70] Mark Noone, Aidan Mooney, and Keith Nolan. "Hybrid Java: The creation of a Hybrid Programming Environment". In: *Irish Journal of Technology Enhanced Learning* 5.1 (2020).
- [71] Mark Noone, Aidan Mooney, Amy Thompson, Frank Glavin, Monica Ward, Keith Nolan, Emer Thornbury, John Andrews, and David Williams. "A Review of the Supports Available to Third-Level Programming Students in Ireland". In: *All Ireland Journal of Higher Education* 14.2 (2022).
- [72] Mark Noone, Amy Thompson, and Aidan Mooney. "An Overview of the Redevelopment of a Computer Science Support Centre and

- the Associated Pedagogy Impacts". In: *All Ireland Journal of Higher Education* 13.2 (2021).
- [73] Brenda Parker. "Teaching experiences with Alice for high school students". In: *Journal of Computing Sciences in Colleges* 27.2 (2011), pp. 148–155.
- [74] Matthew Poole. "Design of a blocks-based environment for introductory programming in Python". In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE. 2015, pp. 31–34.
- [75] Cornelius Preidel, Simon Daum, and André Borrmann. "Data retrieval from building information models based on visual programming". In: *Visualization in Engineering* 5 (2017), pp. 1–14. DOI: 10.1186/s40327-017-0055-0.
- [76] Thomas W Price and Tiffany Barnes. "Comparing textual and block interfaces in a novice programming environment". In: *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM. 2015, pp. 91–99.
- [77] Keith Quille, Susan Bergin, and Aidan Mooney. "Programming: Factors that Influence Success Revisited and Expanded". In: 2015.
- [78] Abhiram G Ranade. "Introductory Programming: Let Us Cut through the Clutter!" In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 2016, pp. 278–283.
- [79] William Robinson. "From Scratch to Patch: Easing the Blocks-Text Transition". In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. ACM. 2016, pp. 96–99.
- [80] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. "Visual programming languages integrated across the curriculum in elementary school: A two year case study using

- “Scratch” in five schools”. In: *Computers & Education* 97 (2016), pp. 129–141.
- [81] Sergio Sandoval-Reyes, Pedro Galicia-Galicia, and Ivan Gutierrez-Sanchez. “Visual learning environments for computer programming”. In: *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2011 IEEE*. IEEE. 2011, pp. 439–444.
- [82] Cheryl Seals, Yolanda Mcmillian, Kenneth Rouse, Ravikant Agarwal, Andrea Williams Johnson, Juan E Gilbert, and Richard Chapman. “Computer Gaming At Every Age: A Comparative Evaluation Of Alice”. In: *i-Manager’s Journal of Educational Technology* 5.3 (2008), p. 1.
- [83] *Short Course Coding Specification for Junior Cycle*. Accessed: 2023-07-28. URL: <https://curriculumonline.ie/getmedia/cc254b82-1114-496e-bc4a-11f5b14a557f/NCCA-JC-Short-Course-Coding.pdf>.
- [84] Romenig da Silva Ribeiro, Leônidas de Oliveira Brandão, Tulio Vitor Machado Faria, and Anarosa Alves Franco Brandão. “Programming web-course analysis: how to introduce computer programming?” In: *Frontiers in Education Conference (FIE), 2014 IEEE*. IEEE. 2014, pp. 1–8.
- [85] Wolfgang Slany. “Catroid: a mobile visual programming system for children”. In: *Proceedings of the 11th International Conference on Interaction Design and Children*. ACM. 2012, pp. 300–303.
- [86] Neil Smith, Clare Sutcliffe, and Linda Sandvik. “Code club: bringing programming to UK primary schools through scratch”. In: *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM. 2014, pp. 517–522.
- [87] TIOBE Software. *TIOBE Index for July 2023*. [Online; accessed 2023-08-02]. 2023. URL: <http://www.tiobe.com/tiobe-index/>.
- [88] Elizabeth R Sowell, Paul M Thompson, Kevin D Tessner, and Arthur W Toga. “Mapping continued brain growth and gray matter density reduction in dorsal frontal cortex: Inverse relationships during post-

- dolescent brain maturation". In: *Journal of Neuroscience* 21.22 (2001), pp. 8819–8829.
- [89] Edward R Sykes. "Determining the effectiveness of the 3D Alice programming environment at the computer science I level". In: *Journal of Educational Computing Research* 36.2 (2007), pp. 223–244.
- [90] Brendan Tangney, Elizabeth Oldham, Claire Conneely, Stephen Barrett, and John Lawlor. "Pedagogy and processes for a computer programming outreach workshop—The bridge to college model". In: *IEEE Transactions on Education* 53.1 (2010), pp. 53–60.
- [91] Christopher Watson and Frederick WB Li. "Failure rates in introductory programming revisited". In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 2014, pp. 39–44.
- [92] David Weintrop. "Blocks, text, and the space between: The role of representations in novice programming environments". In: *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE. 2015, pp. 301–302.
- [93] David Weintrop and Uri Wilensky. "To block or not to block, that is the question: students' perceptions of blocks-based programming". In: *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM. 2015, pp. 199–208.
- [94] Kirsten N. Whitley. "Visual programming languages and the empirical evidence for and against". In: *Journal of Visual Languages & Computing* 8.1 (1997), pp. 109–142.

APPENDIX 1 - SUMMER CAMP SESSIONS

A.1 INITIAL SESSION MATERIALS

This appendix section covers all material related to the initial Summer Camp tests. Included are slides for both sessions, the survey questions that were administered and a raw data sheet.

A.1.1 JAVA SESSION MATERIAL



How to use BlueJ



- ▶ Click "BlueJ" on the desktop
- ▶ Once it opens, click Project -> New Project...
- ▶ Give it a name, let's call it "helloWorld"
- ▶ Click the "New Class..." button on the side of the window
- ▶ Let's also call this "helloWorld".
- ▶ Now, on the main screen, there will be an orange button labelled helloWorld. Click into this and we will be at our code screen.

"Boilerplate"

- ▶ Delete all the text that's not on a green area
- ▶ Now add in the code shown in yellow below

```

helloWorld - helloWorld
Class Edit Tools Options
helloWorld X
Compile Undo Cut Copy Paste Find... Close

/**
 * Write a description of class helloWorld here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class helloWorld
{
    public static void main (String args [])
    {
    }
}

```

- ▶ This code section is called the "boilerplate" and it allows Java to work

Task 1: Your first program

- ▶ "Hello World" is a rite of passage in programming
- ▶ We already have most of the code we need from the last slide
- ▶ We just need to add one more line:

```
System.out.println("Hello World");
```

- ▶ This goes in the yellow area between the two { }
- ▶ This line allows us to "print" the statement "Hello World"
- ▶ The semi-colon shows that we have finished a statement
- ▶ Now we need to "run" our program so we can see this output

Compiling / Running

- ▶ First, we can click "Compile" in the top left of the menu bar.
- ▶ This lets the computer understand what we wrote.
- ▶ We now need to "run" the program to see it's output.
- ▶ Go back to the main BlueJ screen with the orange box. Right click on it and select "void main...." Hit OK on this box.



- ▶ You should now have a popup window that shows Hello World in it.
- ▶ Congratulations, you've now made your first program!

Variables

- ▶ We use "variables" as storage space in Java. They are usually of the form `type name = value;`
- ▶ Let's look at a few examples
- ▶ **int:**
 - ▶ This is for whole numbers, otherwise known as Integers (1,2,3,4,5...)
 - ▶ `int x = 10;`
- ▶ **double:**
 - ▶ This is for numbers with a decimal point (1.1, 1.5...)
 - ▶ `double y = 2.5;`

Variables

- ▶ **boolean:**
 - ▶ These are used to show if something is "true" or "false"
 - ▶ `boolean z = true;`
- ▶ **String:**
 - ▶ Strings hold sentences or words
 - ▶ They are formed a little differently to the other variables
 - ▶ `String name = new String("Hello World");`
- ▶ Task 2: Try declare one of each of these variables.

Selection

- ▶ Selection statements (also known as if/else statements) are used to check if something is true or not and perform actions based on that

```
int balance = 10;
if(balance == 0)
{
    System.out.println("You have no money!");
}
else
{
    System.out.println("You have money, yay!");
}
```

- ▶ Task 3: Write a simple if/else statement that checks if a number (int) is equal to 9 or not. Print a suitable message.
- ▶ You could also use "<" or ">" which are the greater than or less than operators to check if the number is greater than or less than 9.

Math with Variables

- ▶ The numeric variables can make use of all the math functions you're used to like addition, subtraction, multiplication and division.

- ▶ `num1 + num2`

- ▶ `num1 - num2`

- ▶ `num1 * num2`

- ▶ `num1 / num2`

Calculator (Putting it all together)

- ▶ Now that you know the basics of Java, let's put everything together, use the contents of these slides to help you!
- ▶ Step 1: Create a class named "calculator" in BlueJ
- ▶ Step 2: Create a variable called "operator" of type String that will tell our calculator what function to perform
- ▶ Step 3: Create 2 numeric variables (int's or double's, your choice!)
- ▶ Step 4: Create a sequence of if / else if statements that check what symbol is contained in "operator" (HINT: `operator.equals("**")`)
- ▶ Step 5: Inside each if statement, print the result of the function on the 2 variables. For example, if "operator" was +, then add num1 to num2
- ▶ Step 6: If the operator isn't valid, print "ERROR" to the screen

Calculator Code Example

```
public class calc
{
    public static void main (String args[])
    {
        String operator = "**";
        int num1 = 5;
        int num2 = 10;

        if (operator.equals("**"))
        {
            System.out.println(num1 * num2);
        }

        else if (operator.equals("/"))
        {
            System.out.println(num1 / num2);
        }

        else if (operator.equals("+"))
        {
            System.out.println(num1 + num2);
        }

        else if (operator.equals("-"))
        {
            System.out.println(num1 - num2);
        }

        else
        {
            System.out.println("ERROR");
        }
    }
}
```


Loops

- ▶ To go a little further, loops are used to continuously perform a task a number of times or until a condition is met.
- ▶ We will look at just one type, the “while” loop today
- ▶ While loop usually runs until a condition is met

```
int balance = 10;
while(balance != 0)
{
    if(balance == 0)
    {
        System.out.println("You have no money!");
    }
    else
    {
        System.out.println("You have money, yay!");
    }
    balance = balance - 1;
}
```

Loops

- ▶ As you could see on the previous slide, the “while” loop kept going until the balance variable contained 0.
- ▶ There are other types of loop that simply run a certain number of times
- ▶ Task 5: Write a while loop that counts from 1-10, and ends when the variable containing our counting number reaches 10

Advanced Calculator

- ▶ If you mastered the first calculator and loops, well done!
- ▶ If you want a challenge, let's try taking input from the user.
- ▶ The following lines need to be inserted into your code:
- ▶ We should print a line to the screen asking the user to enter input
- ▶ `import java.util.Scanner;` at the start of the program (before the class)
- ▶ `Scanner scan = new Scanner (System.in);` with the rest of your variables.
- ▶ These allow for "scanning" or user input to be used.
- ▶ In place of `int x = 10;` or `String name = new String("Hello World");`
- ▶ We will use `int x = scan.nextInt();` and `String name = scan.nextLine();`
- ▶ Use this information to update your calculator and test it!

Advanced Calculator Example

```
import java.util.Scanner;

public class inputCalc
{
    public static void main (String args[])
    {
        Scanner scan = new Scanner (System.in);

        System.out.println("Please enter an operator:");
        String operator = scan.nextLine();
        System.out.println("Please enter num1");
        int num1 = scan.nextInt();
        System.out.println("Please enter num2");
        int num2 = scan.nextInt();

        if(operator.equals("**"))
        {
            System.out.println(num1 * num2);
        }
        else if(operator.equals("/"))
        {
            System.out.println(num1 / num2);
        }
        else if (operator.equals("+"))
        {
            System.out.println(num1 + num2);
        }
        else if(operator.equals("-"))
        {
            System.out.println(num1 - num2);
        }
        else
        {
            System.out.println("ERROR");
        }
    }
}
```

Final Task

- ▶ This time, modify your program from the last example so that it:
- ▶ Loops and keeps taking input and printing results
- ▶ It should stop when the user enters "stop" as the operator

Hints:

- You will need a while loop with a Boolean condition
- `while (!operator.equals("stop"))`
- You will need to move the "scan's" inside the loop
- You still need to make your variable's outside the loop, just give them default values like 0 and ""
- `!=` is an operator that means "not equal to", opposite of `=`

Final Calculator Example

```
import java.util.Scanner;

public class advCalc
{
    public static void main (String args[])
    {
        Scanner scan = new Scanner (System.in);

        String operator = "";
        int num1 = 0;
        int num2 = 0;


        while(!operator.equals("stop"))
        {
            System.out.println("Please enter an operator:");
            operator = scan.next();
            System.out.println("Please enter num1:");
            num1 = scan.nextInt();
            System.out.println("Please enter num2:");
            num2 = scan.nextInt();

            if(operator.equals(""))
            {
                System.out.println(num1 + num2);
            }
            else if(operator.equals("/"))
            {
                System.out.println(num1 / num2);
            }
            else if (operator.equals("+"))
            {
                System.out.println(num1 + num2);
            }
            else if(operator.equals("-"))
            {
                System.out.println(num1 - num2);
            }
            else
            {
                System.out.println("ERROR");
            }
        }
    }
}
```

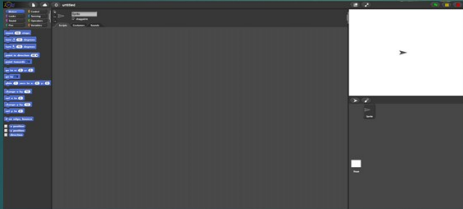
A.1.2 SNAP! SESSION MATERIAL



How to use Snap



- ▶ Go to [the snap website](#)
- ▶ You should see a screen like this:



- ▶ Create a new account on the website so you'll be able to save files later. To do this, click the cloud icon -> Sign up and complete the form.
- ▶ Now we can start working on our first pieces of Snap code!

The slide features a dark teal background with a red vertical bar in the top right. The title "How to use Snap" is in white. To the right of the text is the Snap! logo, which includes a yellow lambda symbol and the text "Snap! Build Your Own Blocks". Below the text is a screenshot of the Snap! IDE interface, showing a dark workspace with a script area on the left and a preview area on the right.

Snap Blocks

- ▶ You'll notice we have options on the left of the screen: Motion, Control, Looks, Sensing, Sound, Operators, Pen and Variables.
- ▶ **Motion** gives us options that allow us to move our "sprite"
- ▶ We will mainly use **Looks** to make our sprite speak
- ▶ **Pen** is used for drawing to the screen

Snap Blocks

- ▶ **Control** is used in many ways: to start the program, loops, selection, all of which we will talk about later
- ▶ **Operators** contains elements to help do basic math
- ▶ **Variables** lets us make and use storage of information
- ▶ We will not use **Sensing** or **Sound**

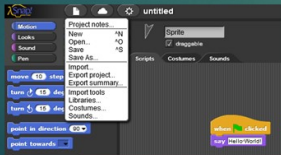
Your first program

- ▶ Making basic programs in Snap is simple
- ▶ Our "Hello World" only needs a "say" block from the Looks section
- ▶ We then need to connect it to something that will let it run
- ▶ We will use the "when green flag clicked" button from Control
- ▶ Let's make our sprite move as well
- ▶ For this, we need to use elements from the motion section
- ▶ For example, we might move a number of steps and turn
- ▶ Task 1: Make your sprite move in a square



Compiling / Running

- ▶ To run our programs, we simply have to do what the control block says (click the green flag, press space, etc.)
- ▶ Now that we've made a program, we need to save it
- ▶ Click the cloud file icon beside the cloud one and click save
- ▶ We can give it any name we want, call this one hello_world



Variables

- ▶ Variables can be used to store values to help us do an action with different values every time
- ▶ In the variables tab, click "make a variable", let's call it x
- ▶ We can now drag this into any box where a number would go
- ▶ Try the following code and see what it does
- ▶ We also need to "reset" our variable
- ▶ Task 2: Using a variable, make the sprite "say" a number every time you press space. Do this for 1-10.

```

when space key pressed
  move x steps
  change x by 1
  
```

```

when clicked
  set x to 0
  
```

```

when space key pressed
  say x
  change x by 1
  
```

Selection

- ▶ Selection statements (if / else) involve checking a condition and performing the appropriate action if the condition is true (or false)
- ▶ Let's modify the 1 – 10 program you just finished with an if statement
- ▶ We can also use an if / else block to print messages when we are done

```

when space key pressed
  if x < 10
    say x
    change x by 1
  
```

```

when space key pressed
  if x < 10
    say x
    change x by 1
  else
    say (done)
  
```

- ▶ Task 3: Using a set of if statements, print a different message when x is 2, 4, 6, 8 and 10. For example: "value is 2", "value is 4", etc.

Task 3 Code

```

when space key pressed
  if x = 2
    say 2 for 2 secs
  if x = 4
    say 4 for 2 secs
  if x = 6
    say 6 for 2 secs
  if x = 8
    say 8 for 2 secs
  if x = 10
    say 10 for 2 secs
  change x by 1
  
```

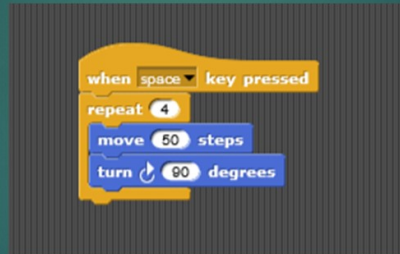
Loops

- ▶ So far, we have been pressing space to continuously repeat actions
- ▶ We can simplify this by using a loop
- ▶ There are 3 main loops in Snap. Repeat x times, repeat until <condition> and forever. They work exactly as you'd expect
- ▶ Let's update our 1 - 10 program once more so that we only need to press the spacebar once.
- ▶ We should also make a say block after the loop
- ▶ Task 4: Use the repeat x times block to make your sprite move in a square (like task 1)

```

when space key pressed
  set x to 0
  repeat until x = 10
  if x < 10
    say x for 1 secs
    change x by 1
  say Finished
  
```


Task 4 Code



Drawing

- ▶ Now that we are moving in a square again, lets use the pen
- ▶ We can use the pen elements to make the sprite draw the square
- ▶ Before we do anything else, go to the motion tab and click the "go to x: 0 y: 0" block to reset our sprite
- ▶ Now, go to the pen tab and click "pen down"
- ▶ Run your code that makes the sprite move in a square
- ▶ What happened now?
- ▶ We can click the "clear" block under pen to clear the screen

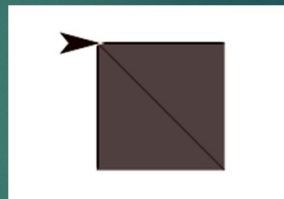


Putting it all together: Shapes

- ▶ Your final task is to make a square that is filled
- ▶ To do this, you need to draw increasingly bigger squares
- ▶ Step 1: Make a variable, let's call it *i*, and set it to 1
- ▶ Step 2: Make sure the sprite is in the centre of the screen
- ▶ Step 3: Put the pen down
- ▶ Step 4: Use a repeat loop to make 100 squares
- ▶ Step 5: Use your existing square code, but move *i* steps each time
- ▶ Step 6: Make sure to update *i* by 1 each run of the loop (1 -100)

Shapes Code Example

```
when clicked
set i to 1
go to x: 0 y: 0
pen down
repeat 100
  repeat 4
    move i steps
    turn 90 degrees
  change i by 1
```

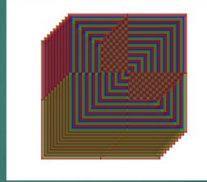


Advanced Shapes

- ▶ Now that you've made this square, it's time to experiment!

- ▶ **See if you can make:**

- ▶ A multi-coloured square
- ▶ A rectangle
- ▶ A file pattern



- ▶ You can also use this time to make anything else you want from what we've looked at, be sure to show a demonstrator if you made something really interesting, or ask us if you want help with anything!

A.1.3 SURVEY QUESTIONS

1. What age are you?
 - 9 or younger
 - 10 - 12
 - 13 - 15
 - 16 or Older
2. What gender are you? (Male, Female)
3. How would you rate the difficulty of Snap!?! (1-10)
4. How would you rate the difficulty of Java? (1-10)
5. What was the hardest aspect of Snap! to learn?
 - Variables
 - Selection (if / else)
 - Loops
 - Drawing Shapes

6. What was the hardest aspect of Java to learn?
 - Variables
 - Selection (if / else)
 - Loops
 - Drawing Shapes
7. Did you enjoy the Snap! course? (Yes / No / It was OK)
8. Did you enjoy the Java course? (Yes / No / It was OK)
9. Which course did you prefer? (Snap! / Java)
10. What was your favourite thing from either course?
11. What was your least favourite thing from either course?
12. Which style of programming do you prefer?
 - Text (Java)
 - Blocks (Snap!)
 - Both are equally good!
13. Would you like to learn more programming in the future? (Yes / No / Maybe)
14. Any other comments?

A.1.4 DATA SHEET

PROGRAMMING LANGUAGES SURVEY DATA (35 Participants):

- 68.6% 13-15 year olds (24)
- 22.9% 16+ year olds (8)
- 8.6% 10-12 year olds (3)

- 88.6% male (31)
- 11.4% female (4)

- Snap average difficulty rating: 3.57/10
- Snap average difficulty rating **13-15 year olds (24):** 3.75/10
- Snap average difficulty rating **16+ year olds (8):** 3.125/10
- Snap average difficulty rating **10-12 year olds (3):** 3.33/10

- Java average difficulty rating: 6.94/10
- Java average difficulty rating **13-15 year olds (24):** 7.08/10
- Java average difficulty rating **16+ year olds (8):** 6.875/10
- Java average difficulty rating **10-12 year olds (3):** 6/10

- Curves very nice on google forms

- Hardest Snap Aspects: Variables (34.3%), Drawing (28.6%), Loops (22.9%), Selection (14.3%)
- Hardest Java Aspects: Calculator (68.6%), Loops (17.1%), Selection (11.4%), Variables (2.9%)
- Mild mistake / bias here since calculator and drawing encompassed all previously learned steps, if we disregard those then Variables in Snap and Loops in Java were the hardest.

- Snap Approval: 60% (21), 34.3% (12) "It was OK", 5.7% (2) No
- Java Approval: 60% (21), 25.7% (9) "It was OK", 14.3% (5) No

- 51.4% preferred Snap (18), 48.6% preferred Java (17)
- **10-12 year olds:** 66.6% preferred Java (2), 33.3% preferred Snap (1)
- **13-15 year olds:** 54.2% preferred Snap (13), 45.8% preferred Java (11)
- **16+ year olds:** 50% preferred Java (4), 50% preferred Snap (4)

- Preferred Style: Text 37.1% (13), Blocks 31.4% (11), Both Good 31.4% (11)
- Preferred Style **10-12 year olds:** Text 33.3% (1), Blocks 33.3% (1), Both Good 33.3% (1)
- Preferred Style **13-15 year olds:** Text 29.2% (7), Blocks 41.6% (10), Both Good 29.2% (7)
- Preferred Style **16+ year olds:** Text 62.5% (5), Blocks 0% (0), Both Good 37.5% (3)

- 88.6% want to learn more programming, 11.4% maybe do

Other comments of note:

"Make the programming a bit more interesting."

"Spend more time on java"

"Overall it was really interesting and enjoyable!"

APPENDIX 2 - FULL COURSES

B.1 FULL COURSE MATERIALS

The materials provided to students for the Java and Snap! courses can be found on Padlet:

- https://padlet.com/mark_noone1/snap
- https://padlet.com/mark_noone1/java
- https://padlet.com/mark_noone/hybridjava

You can also find all material including class plans and some additional files here: <https://tinyurl.com/mncoursematerials>

APPENDIX 3 - HYBRID JAVA BLOCKS

C.1 FULL LIST OF ALL HYBRID JAVA BLOCKS

This appendix contains an overview of all of the blocks created for Hybrid Java. All blocks were placed under the "Control", "Sensing", "Operators", and "Variables" sections of the Snap! User Interface.

Under the "Control" category the setup and section blocks can be seen; the full list is presented in Figure C.1.

Under the "Sensing" category, blocks that can be used to make code comments in the programs can be located. These blocks can be seen in Figure C.2.

Under the "Operators" category, all of the blocks that are related to operations (standard operators, increment / decrement, relational operators and logical operators) are located. These blocks can be seen in Figure C.3.

Finally, under the "Variables" category, all of the remaining blocks, those related to the creation of standard variables, Strings, arrays and all elements for manipulating these variables are located. These blocks can be seen in Figure C.4, Figure C.5 and Figure C.6.

Using all of these blocks, the majority of programs that would be used in a CS1 course can be created up until the point of Object Oriented Programming.

Further examples and background can be found on the Padlet Repository used during the UKICER conference: https://padlet.com/mark_noone1/hybrid-java-repository-nm96n0f3ar3a.

C.1 FULL LIST OF ALL HYBRID JAVA BLOCKS

```
import java.util.* ;
import java.util.Random ;
import java.util.Scanner ;
import ;
public class {
}
public static void main (String args[]) {
}
System.out.println ( );
System.out.print ( );
if ( ) {
}
else if ( ) {
}
else {
}
switch ( ) {
default :
}
case : break;
for ( int = ; ; ) {
}
while ( ) {
}
do {
} while ( );
```

Figure C.1: "Control" Blocks

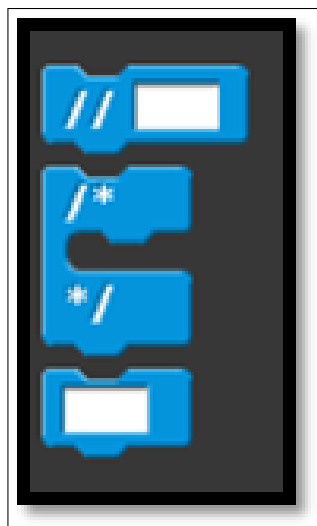


Figure C.2: "Sensing" Blocks

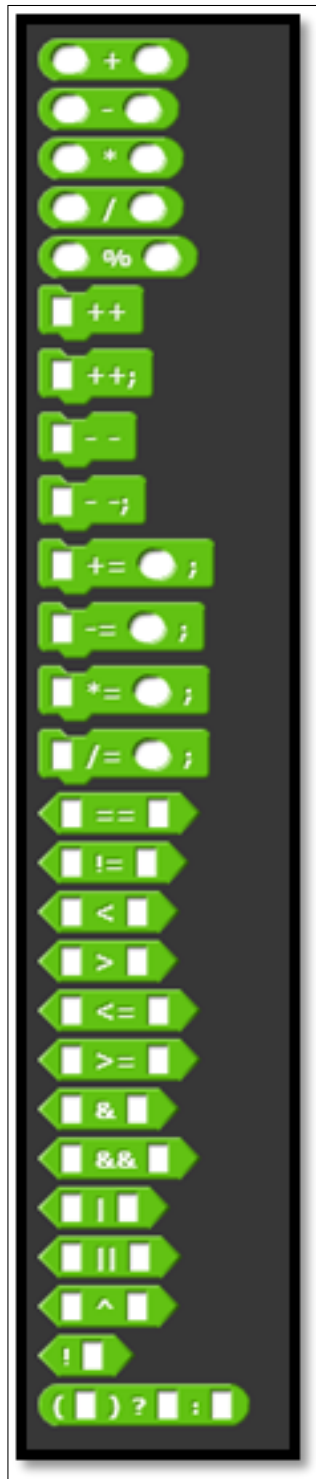


Figure C.3: "Operators" Blocks

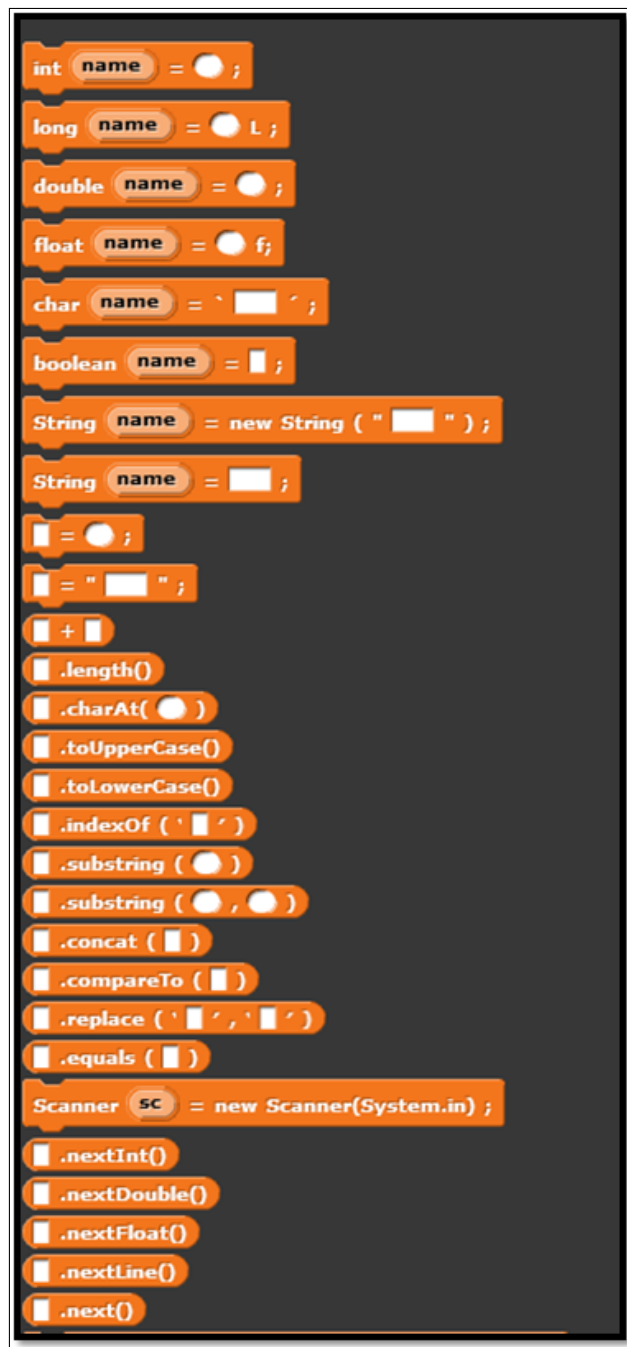


Figure C.4: "Variables" Blocks – Variables, Strings, Scanner

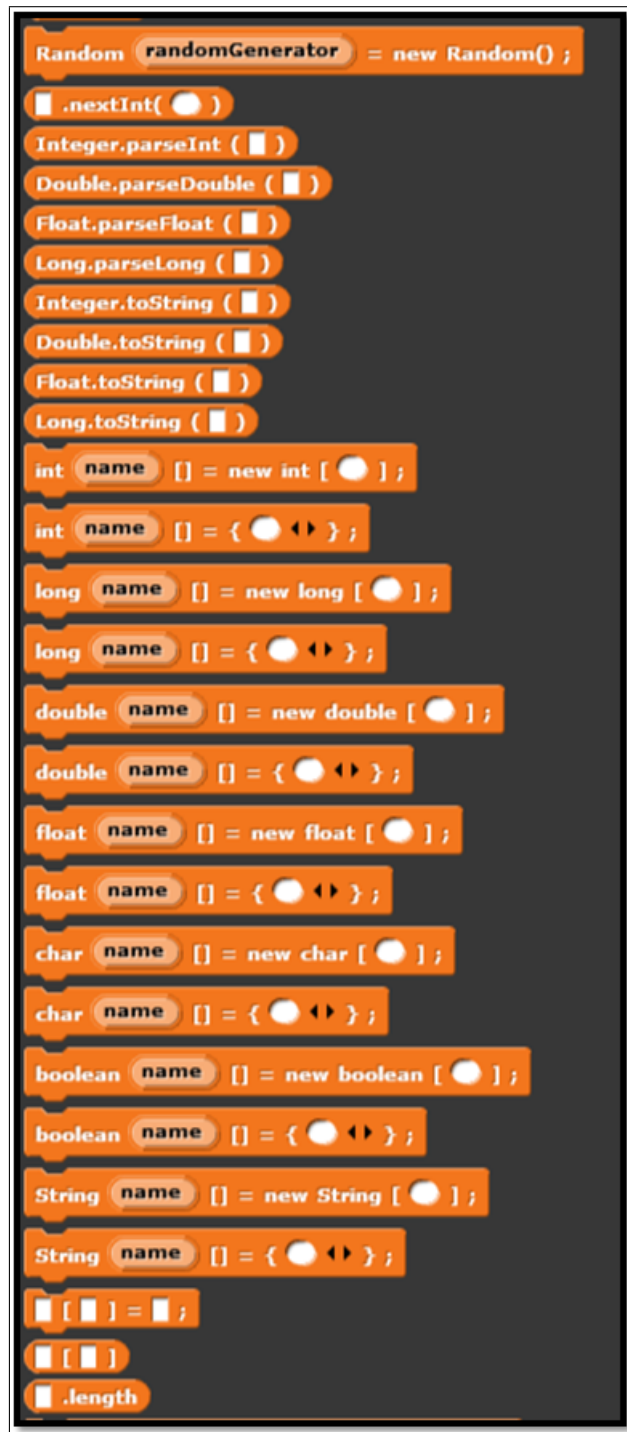


Figure C.5: "Variables" Blocks – Random, Parse, Arrays

```

int name [] [] = new int [ ] [ ];
int name [] [] = { };
long name [] [] = new long [ ] [ ];
long name [] [] = { };
double name [] [] = new double [ ] [ ];
double name [] [] = { };
float name [] [] = new float [ ] [ ];
float name [] [] = { };
char name [] [] = new char [ ] [ ];
char name [] [] = { };
boolean name [] [] = new boolean [ ] [ ];
boolean name [] [] = { };
String name [] [] = new String [ ] [ ];
String name [] [] = { };
[ ] [ ] [ ] = [ ];
[ ] [ ] [ ]

```

Figure C.6: "Variables" Blocks – 2D Arrays