Irish Journal of Technology Enhanced Learning
Vol 5, Issue 1, 2020

Hybrid Java: The creation of a hybrid programming environment

Mark Noone*[a], Aidan Mooney*[a] and Keith Nolan[b]

[a] Maynooth University

[b] Technological University Dublin, Tallaght Campus

(Received April 2020; final version received July 2020)

Abstract

This article details the creation of a hybrid computer programming environment combining the power of the text-based Java language with the visual features of the Snap! language. It has been well documented that there exists a gap in the education of computing students in their mid-to-late teenage years, where perhaps visual programming languages are no longer suitable and textual programming languages may involve too steep of a learning curve. There is an increasing need for programming environments that combine the benefits of both languages into one.

Snap! is a visual programming language which employs "blocks" to allow users to build programs, similar to the functionality offered by Scratch. One added benefit of Snap! is that it offers the ability to create one's own blocks and extend the functionality of those blocks to create more complex and powerful programs. This will be utilised to create the Hybrid Java environment. The development of this tool will be detailed in the article, along with the motivation and use cases for it.

Initial testing conducted will be discussed including one phase that gathered feedback from a pool of 174 first year Computer Science students. These participants were given instructions to work with the hybrid programming language and evaluate their experience of using it. The analysis of the findings along with future improvements to the language will also be presented.

* Corresponding author. Email: mark.noone@mu.ie, Irish Journal of Technology Enhanced Learning Ireland, 2020. © 2020 Noone, M. The Irish Journal of Technology Enhanced Learning Ireland is the journal of the Irish Learning Technology Association, an Irish-based professional and scholarly society and membership organization (CRO# 520231) http://www.ilta.ie/ . This is an Open Access article distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0), allowing third parties to copy and redistribute the material in any medium or format and to remix, transform, and build upon the material for any purpose, even commercially, provided the original work is properly cited and states its license.

1. Introduction

Teaching (and learning) a first computer programming language is not an easy task (Lahtinen, Ala-Mutka, & Järvinen, 2005). Regardless of student age or language choice, difficulties will arise for some students. It is with this in mind that researchers are looking for answers to the question "What is the best First Programming Language (FPL) to teach students?" (Gupta, 2004; Krpan & Bilobrk, 2011) while also examining alternative approaches to teaching novice programmers (Price & Barnes, 2015; Wester, Sint, & Kluit, 1997).

Visual programming languages (such as Scratch and Alice) are often used to introduce younger students to computer programming. It is well documented that these young students, generally under the age of sixteen tend to favour visual programming environments (Cheung, Ngai, Chan, & Lau, 2009). Some visual programming languages (such as ScratchJr and Snap!) have been used by children as young as five. Based on the official Scratch statistics (MIT Media Lab, 2020) which show that there were over 550,000 monthly active users of Scratch in March 2020, 1.57 million new projects in March 2020 and over 52 million projects shared overall, the popularity of Scratch is ever growing.

Despite these facts, nearly all CS1 (the first programming module on an undergraduate degree, usually in first year) modules, and follow up modules, at tertiary level are taught using a traditional text-based programming language such as Java, C++ or Python (Davies, Polack-Wahl, & Anewalt, 2011). Why is this the case? When was it decided that primarily text-based languages were suitable despite much evidence that students have difficulties with them (Lahtinen et al., 2005; Watson & Li, 2014). We need to be asking ourselves whether visual languages could be a tangible alternative option as a first programming language. Some researchers and educators have had good success using a visual FPL as their language of choice (Aktunc, 2013; Asamoah, 2006; Noone & Mooney, 2019a).

Research has shown that there is a certain age group (13-16 year olds) 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 (Cheung et al., 2009). There is an increasing need for languages that combine the power 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 in a more welcoming and more suitable interface. A need for a hybrid language is growing alongside the increasing interest among young people in computer programming. To satisfy this need, we have created a programming environment that combines the effectiveness of a text-based programming language with the ease of use of a visual programming language. We call this programming environment "Hybrid Java". The goal for this environment is to combine the power of a text-based language with the ease of use of a visual language which may be used as a first programming language or as a bridge when moving from a visual language to a text-based language.

2. Motivation and Background

There have been numerous studies (Boshernitsan & Downes, 2004; Kiper, Howard, & Ames, 1997; Sáez-López, Román-González, & Vázquez-Cano, 2016; Weintrop & Wilensky, 2015; Whitley, 1997) undertaken to evaluate the benefits and disadvantages related to teaching programming using visual programming languages. One such study (Weintrop & Wilensky, 2015) 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 differences 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, 2015). 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 (2017).
- 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 (2003) and Kölling has developed the Greenfoot programming environment (2010). 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?”. With the ever changing landscape of literature in the area of FPL, and with some of it often being contradictory, the authors undertook a systematic literature review to examine the prevalence of visual and textual programming languages in education, with the aim of determining the best FPL approaches (Noone & Mooney, 2018). This review centred around two research questions:

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

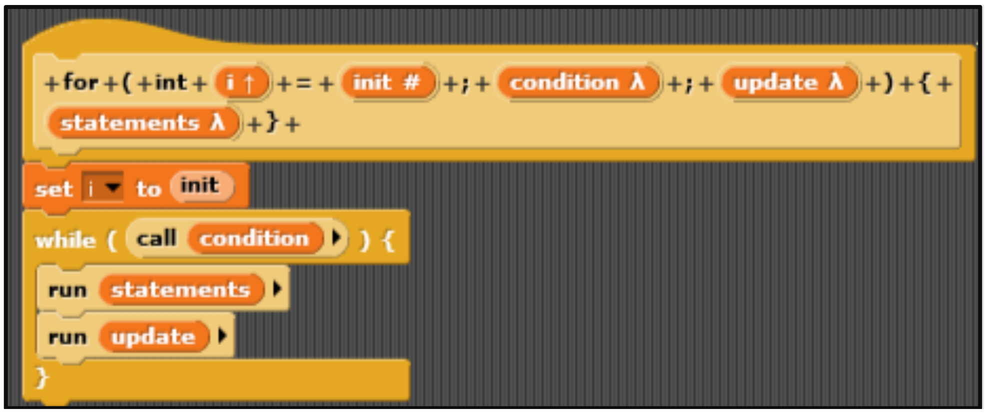
A very quick summary of the results of this study would be to say that visual programming languages are extremely beneficial when taught to the “right age group”. The study also further pointed to the existence of this “educational gap” around the ages of 14-17 where neither language type is ideal. Therefore, we have discerned that text-based languages have their weaknesses, but visual block-based languages also have their issues. This led us 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, we wanted to focus on a hybrid blocks-text environment.

The authors decided to investigate the creation of a hybrid form of Java (the FPL of choice at the authors institution). Other researchers have done similar testing using other languages with promising results (Weintrop, 2015). The premise is that, by creating and teaching with a hybrid programming environment we may see a curve where a visual language proves the easiest to learn for students, a text-based language appears the most difficult to learn but the hybrid “language” is placed between these two. This would suggest that a hybrid programming environment could be used within a CS1 module as an intervention tool for students who might be struggling with the text-based approach or additionally to provide a challenge to younger learners who might be bored with the visual programming approach.

3. Environment Development

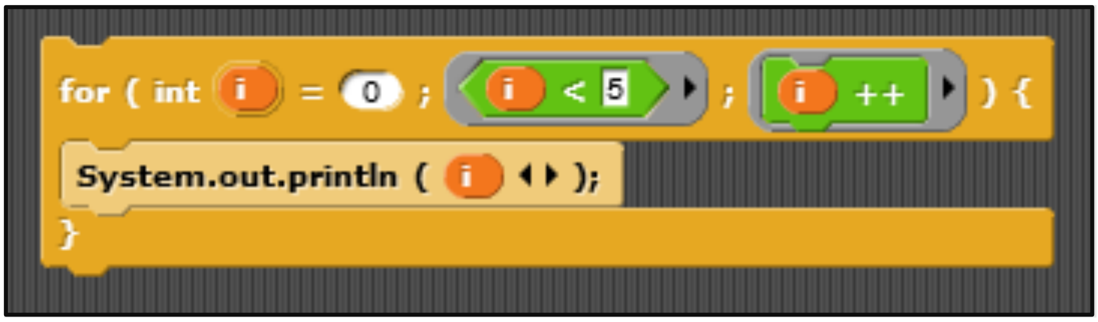
Hybrid Java is built using the Snap! Platform (Harvey & Mönig, 2016). Snap! was developed as an extensible reimplementation of Scratch (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), with the added benefit of being able to create one’s own custom visual blocks. This allows one to create some “backend” 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 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 words *“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 2 presents the user facing display of the code block from Figure 1. In terms of functionality, it is equivalent to a Java *for loop* which runs a piece of code several times.



```
+for+(+int+ i ↑ +=+ init # +;+ condition λ +;+ update λ +)+{+
statements λ +}+
set i to init
while ( call condition ) {
  run statements
  run update
}
```

Figure 1: For Loop backend in Snap!



```
for ( int i = 0 ; i < 5 ; i ++ ) {
  System.out.println ( i );
}
```

Figure 2: For Loop frontend 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 following list of concepts, which would need to be replicated for Hybrid Java:

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 these concepts were included in the first iteration of the environment. A full set of all the finished code blocks are provided in the Appendix. 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.

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 our programming environment, we hid all of the original blocks (as we did not want our users getting confused between Snap! blocks and Hybrid Java blocks). We then placed the Hybrid Java blocks into the following categories, linking back in with the previously discussed list of concepts:

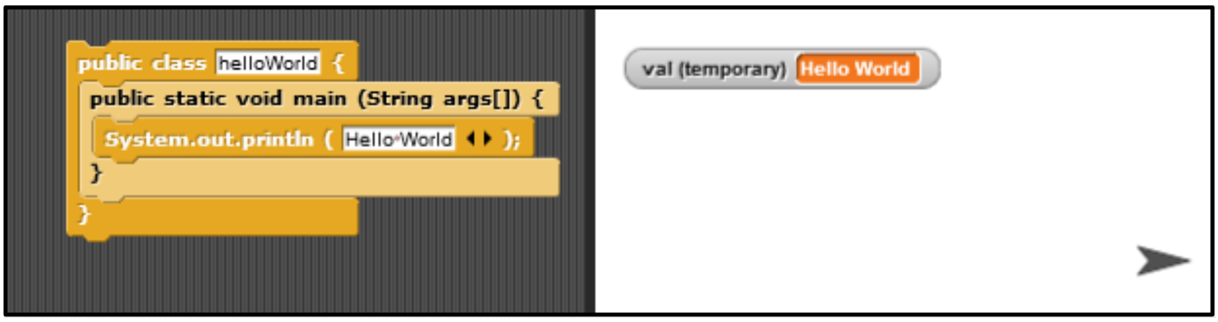
- *Control* – All of the Java structure (1), imports, printing (3), selection (6) and looping (7) blocks,
- *Sensing* – Blocks for commenting code (5),
- *Operators* – All operator (4) blocks (simple operations, incrementing, logical operators),
- *Variables* – All remaining variable blocks (basic types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

These blocks can be seen in full in the Appendix, and 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 3.

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 clear old temporary variables as seen in Figure 3.



```
public class helloWorld {
  public static void main (String args[]) {
    System.out.println ( Hello World ◂▸ );
  }
}
```

```
val (temporary) Hello World
```

Figure 3: Sample Hybrid Java program with its corresponding output

4. Testing

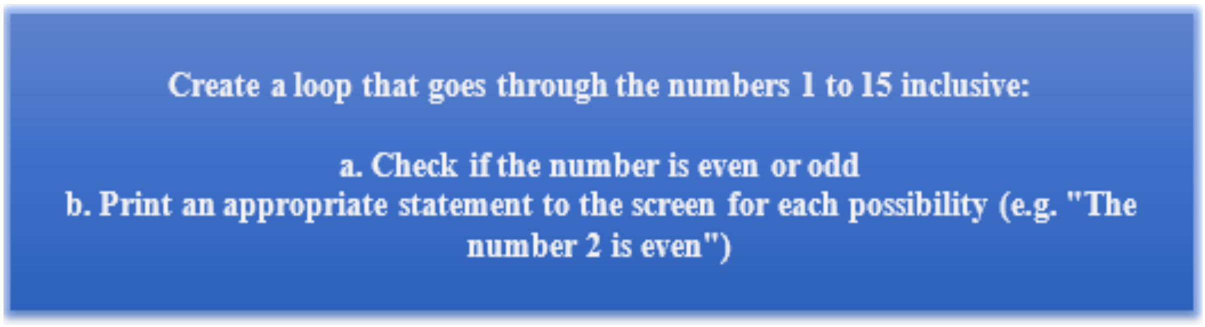
Multiple phases of testing occurred with the tool. The first phase of testing occurred with a first-year undergraduate Computer Science class at Maynooth University. The second phase took place at the annual Computer Science summer camp at Maynooth University for students aged 12-17 years old. The third 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 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 plan to perform some 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.

4.1 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 our institution. 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, 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.



Create a loop that goes through the numbers 1 to 15 inclusive:

a. Check if the number is even or odd
b. Print an appropriate statement to the screen for each possibility (e.g. "The number 2 is even")

Figure 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 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).

- 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 studying Java for 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.

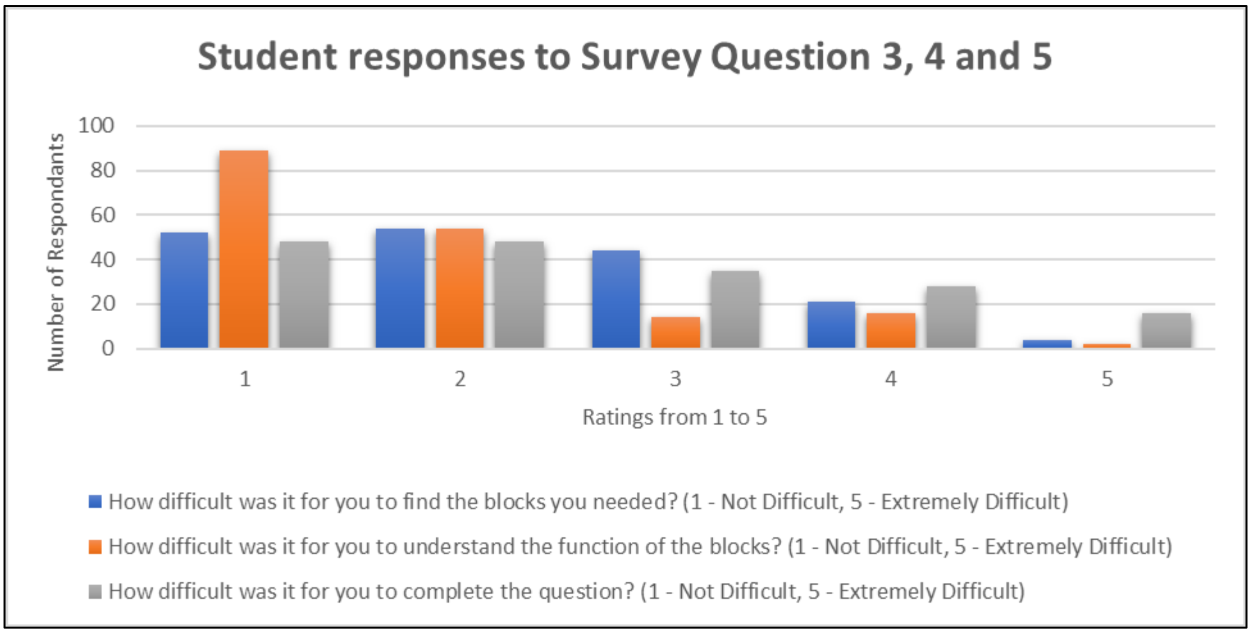


Figure 5: Graphs of responses to Q3, Q4 and Q5 respectively

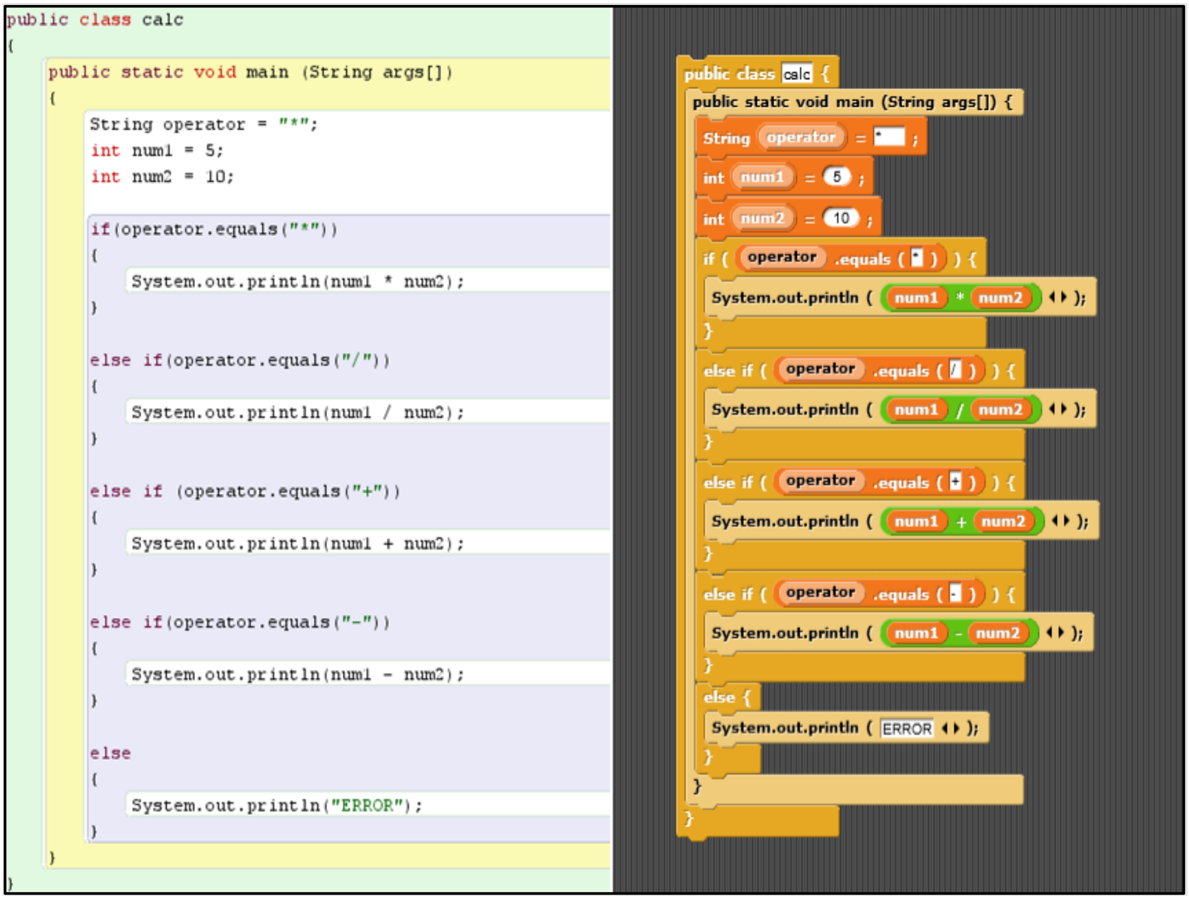
Some of the key comments that participants made included discussion on the ease of use of the tool ("Overall very easy to understand and use"), the usefulness of using the tool to introduce Java to young or novice programmers ("Seems like this could be a very innovative and novel way to teach Java") 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 authors 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 difficultly rating of approximately 2 out of 5, it can be said that for semi-experienced programmers, Hybrid Java a relatively easy tool to use. Will this be the same for true beginners to programming? In test phase two, this is what we aimed to examine.

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 ninety 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 Summer Camp runs over a three-week period, with participants able to attend one, two or three weeks. During week one of the camp, a session on "Java" was ran. This session was based on a short 90-minute session constructed by the authors and detailed in the following paper (Noone & Mooney, 2017). This session used BlueJ (Kölling et al., 2003) as its Integrated Development Environment (IDE). During week two 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 6.



```
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");
        }
    }
}
```

```
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 );
    }
  }
}
```

Figure 6: Simple Calculator program in both Java (left) and Hybrid Java (right)

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".
- 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. This will be discussed further in Section 5.

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 Conference (UKICER) in Canterbury, England. 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.

Only 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 participants 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 5. Others are noted as part of the future work for this project.

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

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 the Appendix) 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 module.
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! (Hintze & Romagosa, 2015) is an external tool which allows for the creation of executable projects. This tool allowed us to export our Hybrid Java project into an external file for ease of use with classes.

The version of the tool which is referenced in Section 7 includes all of these discussed enhancements. Some other minor changes were made along the way such as the fixing of typos, 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.

5. Potential Uses of Hybrid Java

The Hybrid Java programming environment has several potential places that it can be used. The first of these is as an introductory programming language. The blocks within Hybrid Java have been designed and based on the authors' CS1 module. It would be relatively straightforward to convert an existing CS1 module in Java to its equivalent Hybrid Java format.

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 (Armoni, Meerbaum-Salant, & Ben-Ari, 2015; da Silva Ribeiro, de Oliveira Brandão, Faria, & Brandão, 2014). 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 (Lahtinen et al., 2005; Noone & Mooney, 2018). In particular, one interesting result came from Giordano and Maiorana (2014) 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 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 (Cheung et al., 2009). The Hybrid Java environment perfectly fills this educational gap. More testing will be undertaken by the authors of this paper in the near future on using Hybrid Java with school students. We are confident that they will engage well with it, particularly given the history of Scratch usage in Irish schools (O'Rourke, 2017).

Hybrid Java can certainly be used as a First Programming Language on its own, but we believe one of its greatest strengths lies in the area of intervention and student support. The authors are involved in the running of the "Computer Science Centre" at Maynooth University (Nolan, Mooney, & Bergin, 2015). 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 (Gupta, 2004). 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 (Weintrop & Wilensky, 2015) and the ease of use of drag-and-dropping the blocks in to place (Price & Barnes, 2015) 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 our future work to see just how strong of an intervention tool Hybrid Java is.

6. Conclusions and Future Work

In this paper, we have presented a new hybrid programming environment called "Hybrid Java". Between the existing literature in the area of hybrid programming languages, and the multi-phase testing performed by the authors, 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. There is more testing to be done soon to validate this fact.

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 we plan to both run the workshop again in the future and gather a pool of testers who are willing to use Hybrid Java in their own teaching, even for one session.

On top of this, the authors have developed their own 10-week short course aimed at 14 to 16 year-old students in schools. This short course exactly mirrors courses they have developed in both Java and Snap!, and have been taught in schools (Noone & Mooney, 2019a). The next major phase of comparison testing will involve teaching the Java and Snap! curricula in some schools, and Hybrid Java in others. From the results of a final exam, as well as anecdotal feedback, we will be able to determine the "difficulty" of Hybrid Java over a longitudinal timeframe. It is expected that Hybrid Java will be easier than Java but more difficult than Snap!

Another phase of planned testing will be in the Computer Science Centre at Maynooth University, to see how students who are currently learning Java interact with it and how they perceive the benefits of the tool. Particularly, it will 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 will 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 perform differently in the final examination with the only changing variable being the language tool that was used.

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 (Mannila & de Raadt, 2006). Future development of the tool will 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 will also be added based on feedback from educators and students alike on missing blocks. Similarly, as we recognise the requirement for new blocks ourselves, we will add them.

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 (Klassen, 2006). 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 will be undertaken. This will 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 (Lahtinen et al., 2005). However, the option will be present for the educator to turn these errors off to provide an "easier" introduction if desired.

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 has been 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.

Additionally, this tool could be used by legacy programmers who may wish to retrain after a hiatus from programming. Using a hybrid language will allow them to quickly refresh the main components of textual programming and make the transition back easier. Another factor that will be considered is the area of accessibility of the tool. We will endeavour to develop this tool in such a way that it does not disadvantage visually impaired users of the tool.

7. Trying it for yourself

If you would like to test out the Hybrid Java programming environment for yourself, please do not hesitate to contact us at *mark.noone@mu.ie* or *aidan.mooney@mu.ie* with any questions or for any guidance. We will be able to set you up with the tool and provide you with an introductory guide. You can also trial the version that was available at the time of publication of this paper (Noone & Mooney, 2019b). You are more than welcome to use the tool in your teaching or simply for trying something new with, as long as this paper has been referenced.

8. Acknowledgements

The authors would like to thank Emily O’Regan for her help in developing the Hybrid Java programming environment in its early stages. We would also like to thank all participants in every stage of the testing for helping with the iterative design of the user interface. Finally, we would like to thank anyone reading this paper who is interested in trying the tool out in their teachings in advance.

References

Aktunc, O. (2013). A teaching methodology for introductory programming courses using Alice. *International Journal of Modern Engineering Research (IJMER)*, *3*(1), 350–353.

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to "real" programming. *ACM Transactions on Computing Education (TOCE)*, *14*(4), 1–15.

Asamoah, A. (2006). Should We Be Using Visual Programming Languages Like Alice ©To Teach Programming? *6th Annual Multimedia Systems, Electronics and Computer Science*. Southampton.

Boshernitsan, M., & Downes, M. S. (2004). *Visual programming languages: A survey*. Citeseer.

Cheung, J., Ngai, G., Chan, S., & Lau, W. (2009). Filling the gap in programming instruction: A text-enhanced graphical programming environment for junior high students. *SIGCSE Bulletin Inroads*, *41*(1), 276–280. https://doi.org/10.1145/1539024.1508968

da Silva Ribeiro, R., de Oliveira Brandão, L., Faria, T. V. M., & Brandäo, A. A. F. (2014). Programming web-course analysis: how to introduce computer programming? *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, 1–8.

Davies, S., Polack-Wahl, J. A., & Anewalt, K. (2011). A snapshot of current practices in teaching the introductory programming sequence. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 625–630.

Giordano, D., & Maiorana, F. (2014). Use of cutting edge educational tools for an initial programming course. *2014 IEEE Global Engineering Education Conference (EDUCON)*, 556–563.

Gupta, D. (2004). What is a good first programming language? *Crossroads*, *10*(4), 7.

Harvey, B., & Mönig, J. (2016). Snap! (Build Your Own Blocks) 4.0. Retrieved January 17, 2020, from http://snap.berkeley.edu/

Hintze, A., & Romagosa, B. (2015). *Snapp!* Retrieved from http://snapp.citilab.eu/

Kiper, J. D., Howard, E., & Ames, C. (1997). Criteria for evaluation of visual programming languages. *Journal of Visual Languages & Computing*, *8*(2), 175–192.

Klassen, M. (2006). Visual approach for teaching programming concepts. *9th International Conference on Engineering Education*.

Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1–21.

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The bluej system and its pedagogy. *International Journal of Phytoremediation*, *21*(1), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

Krpan, D., & Bilobrk, I. (2011). Introductory programming languages in higher education. *2011 Proceedings of the 34th International Convention MIPRO*, 1331–1336.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *Acm Sigcse Bulletin*, *37*(3), 14–18.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 1–15.

Mannila, L., & de Raadt, M. (2006). An objective comparison of languages for teaching introductory programming. *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, 32–37.

MIT Media Lab. (2020). *Scratch Statistics*. Retrieved from https://scratch.mit.edu/statistics/

Nolan, K., Mooney, A., & Bergin, S. (2015). Facilitating student learning in Computer Science: large class sizes and interventions. *International Confernce on Engaging Pedagogy*.

Noone, M., & Mooney, A. (2017). First Programming Language : Visual or Textual ? *International Conference on Engaging Pedagogy (ICEP)*.

Noone, M., & Mooney, A. (2018). Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education*, *5*(2), 149–174. https://doi.org/10.1007/s40692-018-0101-5

Noone, M., & Mooney, A. (2019a). First Programming Language-Java or Snap? A Short Course Perspective. *10th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2019)*. https://doi.org/10.5176/2251-2195_CSEIT19.148

Noone, M., & Mooney, A. (2019b). *Hybrid Java Tool*. Retrieved from https://snap.berkeley.edu/snap/snap.html#present:Username=marknoone&ProjectName=HYBRID_JAVA_NEWORDER

O'Rourke, B. (2017). Coding in Irish Classrooms. Retrieved January 28, 2020, from RTE News website: https://www.rte.ie/lifestyle/living/2016/0810/808309-coding-in-irish-classrooms/

Preidel, C., Daum, S., & Borrmann, A. (2017). Data retrieval from building information models based on visual programming. *Visualization in Engineering*, *5*(1). https://doi.org/10.1186/s40327-017-0055-0

Price, T. W., & Barnes, T. (2015). Comparing textual and block interfaces in a novice programming environment. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, 91–99.

Sáez-López, J.-M., Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools. *Computers & Education*, *97*, 129–141.

Watson, C., & Li, F. W. B. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39–44.

Weintrop, D. (2015). Blocks, text, and the space between: The role of representations in novice programming environments. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 301–302.

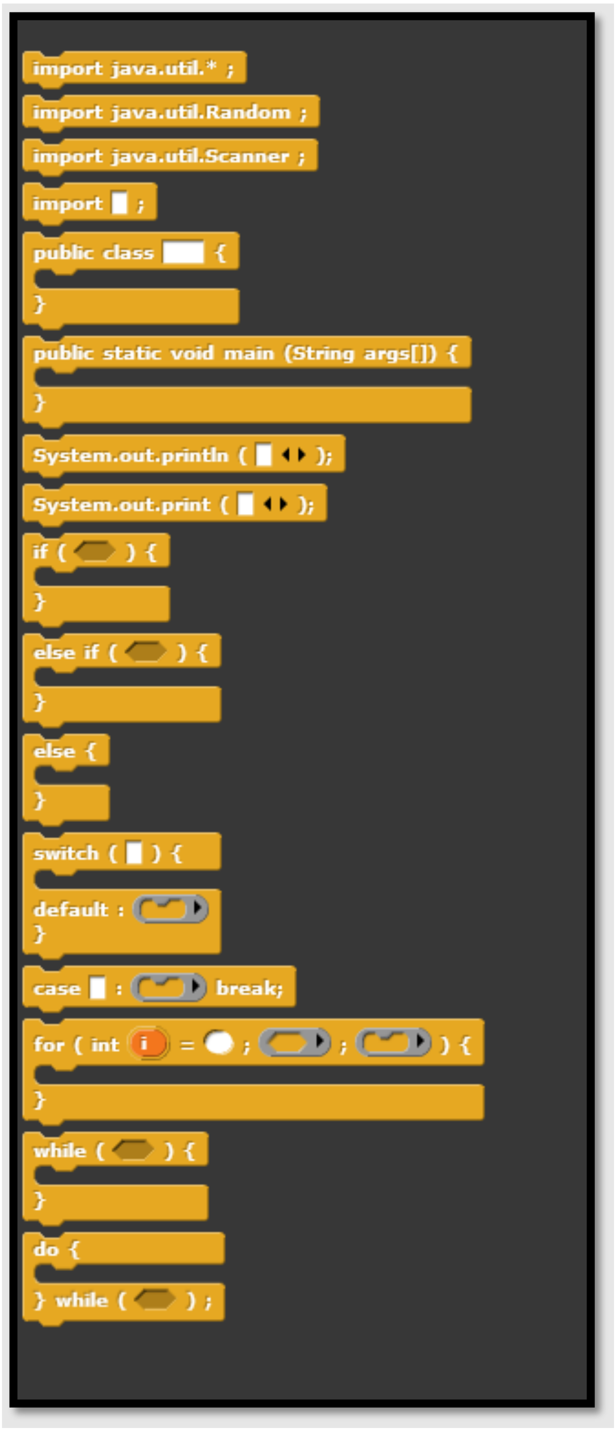
Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question: students' perceptions of blocks-based programming. *Proceedings of the 14th International Conference on Interaction Design and Children*, 199–208.

Wester, F., Sint, M., & Kluit, P. (1997). Visual programming with Java; an alternative approach to introductory programming. *Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education*, 57–58.

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, *8*(1), 109–142.

Appendix: Full List of All Hybrid Java Blocks

All blocks were placed under the “Control”, “Sensing”, “Operators”, and “Variables” sections of the Snap! User Interface. In this Appendix, all of the created blocks will be presented. Under the “Control” category the setup and section blocks can be seen; the full list is presented in Figure A1.



```
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 i =  ;  ;  ) {
}
while (  ) {
}
do {
} while (  ) ;
```

Figure A1: “Control Blocks”

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 A2.

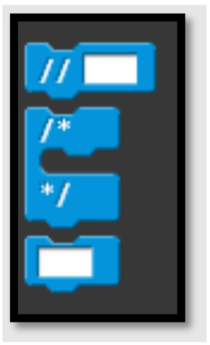


Figure A2: “Sensing” Blocks

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 A3.

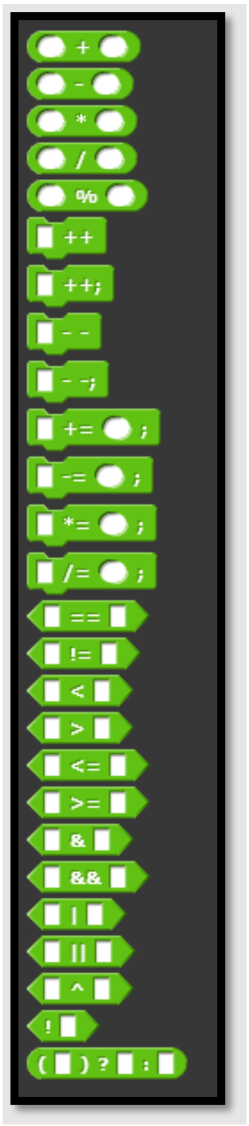
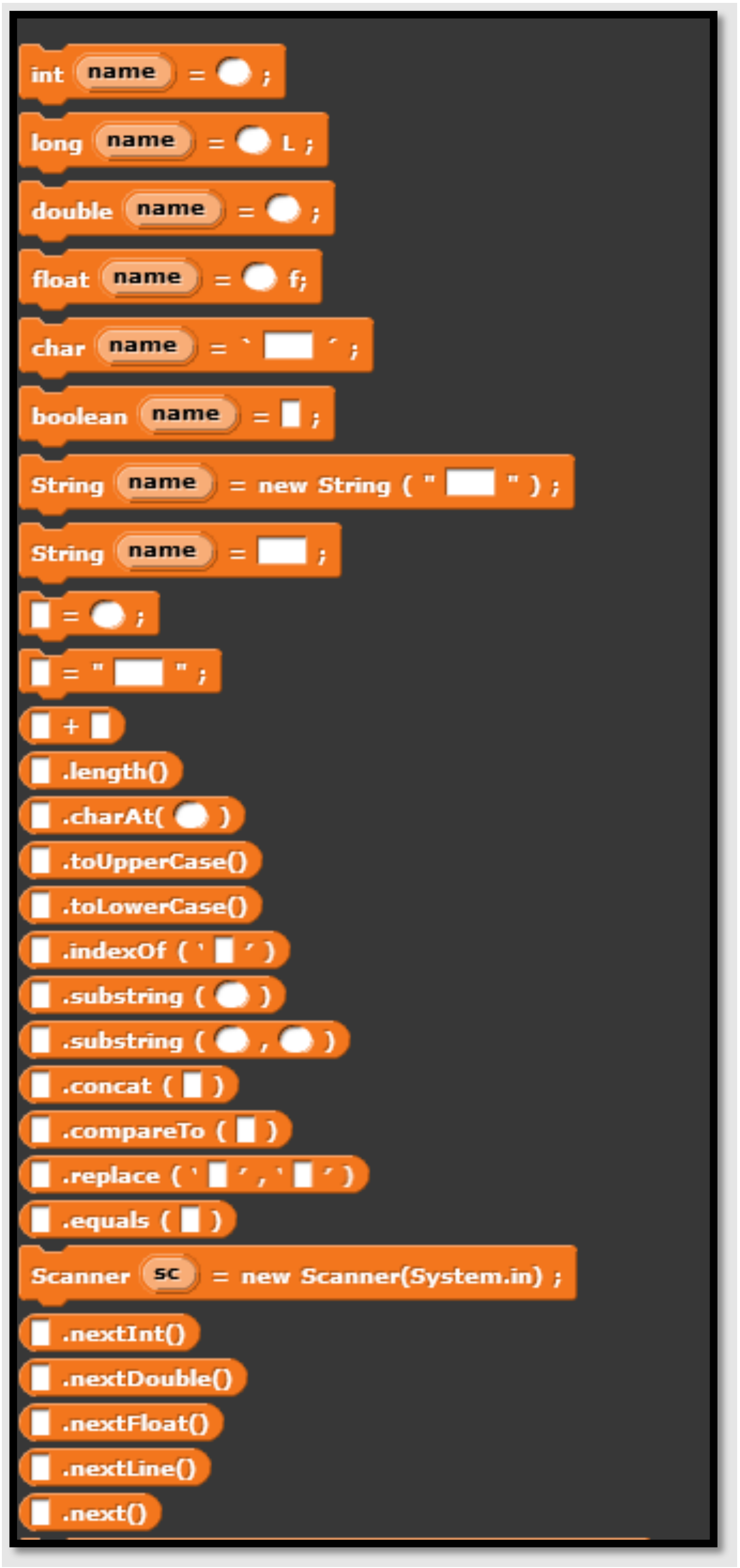


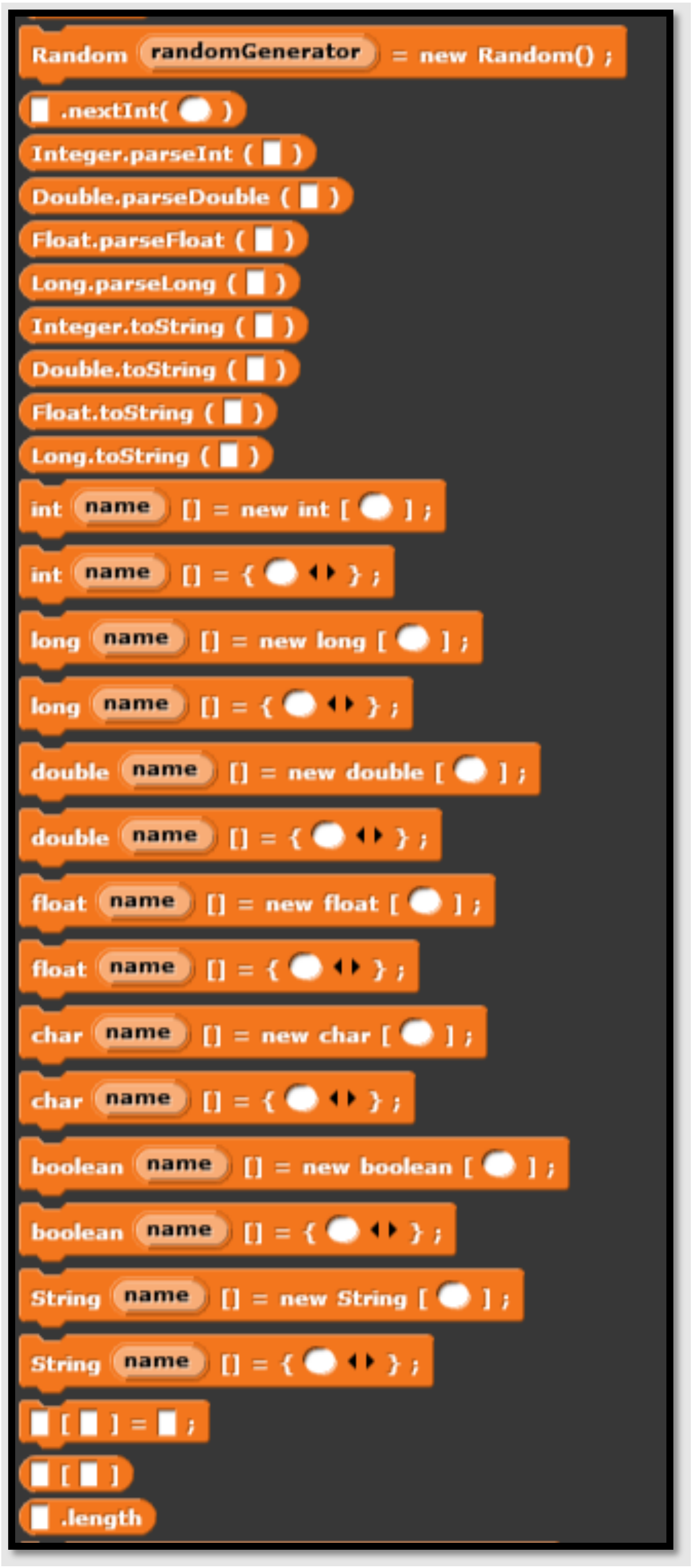
Figure A3: “Operators” Blocks

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 A4, Figure A5 and Figure A6.



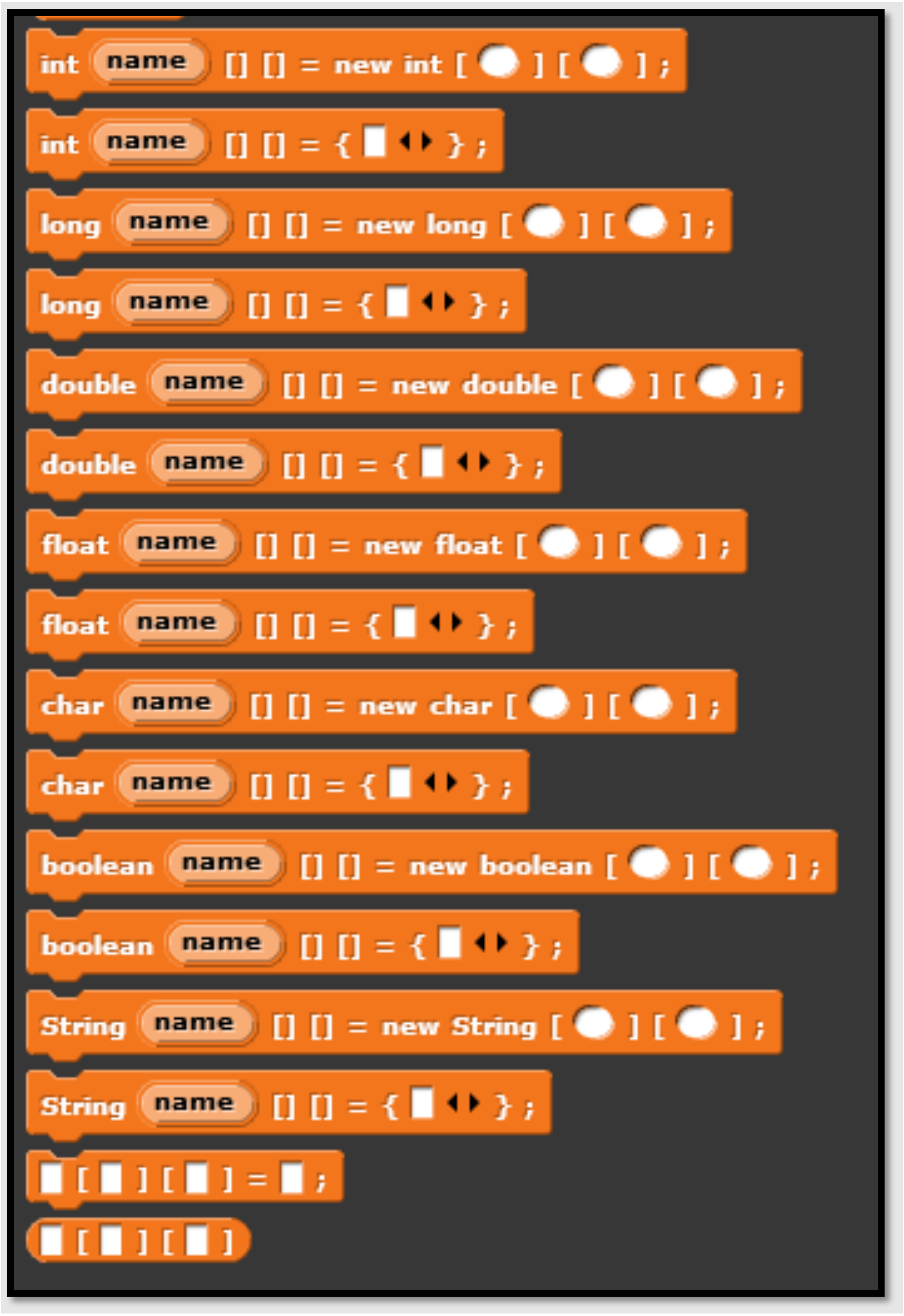
```
int name = ;
long name = L ;
double name = ;
float name = f;
char name = ` ´ ;
boolean name = ;
String name = new String ( " " ) ;
String name = ;
 = ;
 = " " ;
 + 
 .length()
 .charAt( )
 .toUpperCase()
 .toLowerCase()
 .indexOf ( ' ' )
 .substring ( )
 .substring ( , )
 .concat ( )
 .compareTo ( )
 .replace ( ' ' , ' ' )
 .equals ( )
Scanner sc = new Scanner(System.in) ;
 .nextInt()
 .nextDouble()
 .nextFloat()
 .nextLine()
 .next()
```

Figure A4: "Variables" Blocks – Variables, Strings, Scanner



```
Random randomGenerator = new Random() ;
.nextInt( )
Integer.parseInt ( )
Double.parseDouble ( )
Float.parseFloat ( )
Long.parseLong ( )
Integer.toString ( )
Double.toString ( )
Float.toString ( )
Long.toString ( )
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 [] = { } ;
[ ] = ;
[ ]
.length
```

Figure A5: "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 A6: "Variables" Blocks – 2D Arrays

Using all of these blocks, the majority of programs that would be used in a CS1 course can be created. Future development of the environment will look at mirroring even more blocks and styles of programming, with a particular focus on Object Oriented design.