論　文

First Programming Language for Humanities Majors - A Comparison of Java and Swift

有　賀　妙　子
同志社女子大学・学芸学部・メディア創造学科・特別任用教授

ARIGA Taeko
Department of Media, Faculty of Liberal Arts,
Doshisha Women's College of Liberal Arts, Special appointment professor

Abstract

While Java, C family, and Python are the top three languages used in introductory programming courses in higher education, Swift is seldom used. This study considers whether the Swift language can be used as a first language for introductory programming for humanities majors. Learning programming is important for students regardless of major to foster computational thinking and utilize programming for their study. Our department conducted two courses to learn programming in Java and Swift for the creation of digital contents and media studies. This paper presents the contents of both courses, results of examinations, and students' self-reflections. The result indicates that students can learn the basic programming concepts in Swift in the same way as in Java, but the effect of the Swift course for the GUI and OO parts was inferior to that of the Java course. The finding from students' self-reflections concerning programming shows that students perceived that their confidence was enhanced in both the Swift and the Java courses.

Keywords: Introductory programming, Java, Swift, Programming for non-majors, Programming language

1　Introduction

This study examines whether the Swift language can be used as a first language for teaching introductory programming and at the same time to develop a real-world application for students whose major is humanities, especially media design and information design. The author conducted two courses to learn programming for first learners in Java and Swift, and analysed the results.

1.1 Intention of teaching programming to humanities majors in Swift

There is a large literature examining which languages are used for introductory programming, and how a language choice makes an impact on learning. Studies (Davies et al. 2011; Murphy et al. 2017) reported that Java, C family, and Python are the top three languages used in introductory programming courses in higher education. On the other hand, Swift is seldom used. In the surveys by Simon et al. (2018), Swift was just one of 11

languages in the "other" category, none of which was used in more than 2% of the surveyed courses. Most courses using Swift are positioned as intermediate or advanced courses in departments of computer science or engineering, and aim to foster real-world coding skills for Apple's devices.

Nevertheless, this study aimed to investigate the feasibility of using Swift as a first language for humanities majors for two reasons: enhancing motivation and fostering the ability to create digital contents. The digital contents here include Web applications, games, mobile applications, interactive animations, and so on. For developing those contents, engagement by people having humanities backgrounds in planning and designing those contents is essential. For students in humanities to be involved in creation, learning programming is inevitable to foster computational thinking and to learn how to create those contents. Humanities majors themselves recognize the mounting importance and relevance of computing in their own fields. Camp et al. (2017) reported the large increase of non-majors taking computing courses, because computing plays a role in a wide range of disciplines and jobs.

Since iOS devices are widely used and familiar to students as a platform of digital media, creating applications for iOS devices attracts students' interest and raises their motivation. Additionally, it is desirable for students to be able to take the course without the prerequisite of programming experience. Examining the syllabi of iOS programming courses offered in the US and Canada, the author found that most of them set prerequisites at a basic knowledge of programming at least, because those are constructed as advanced courses on a step-by-step curriculum to learn the whole knowledge and skills of software development in computing disciplines. However, the requirement of prior programming experience may make students hesitate to start learning iOS application development. If it is possible to learn application development by Swift as an introductory course, it will be beneficial for humanities majors who are interested in developing iOS applications, but not so interested in matters of computer science.

1.2 Two introductory programming courses in Java and Swift for comparison

Our department targets the creation of digital contents and media studies, and our students want to learn programming for creating games, interactive animations, and Web applications. We have two courses to learn programming for first learners: Introductory programming in Java (Prog A), and Introductory iOS Development in Swift (Prog B). The former course intends to teach basic computer programming concepts: types, variables arrays, loops, conditionals, class definitions in Java, and development of applications with a graphical user interface (GUI) by JavaFX. In the latter course, students learn the same basics in Swift and development of a simple application using Xcode, an integrated development environment for iOS applications. Both courses are designed for students with no prior programming experience.

If our Swift course required knowledge of programming as a prerequisite, it would narrow students' opportunities to learn development of iOS applications, because a considerable proportion of those students is not necessarily so eager to learn programming itself, but is interested in iOS development. We want to make iOS programming courses open to them, and target learning both fundamental concepts and development of applications with GUI. This study will explore whether learning of introductory programming in Swift is achieved or not.

2 Literature review

2.1 Impact of different teaching languages

Many studies have been conducted for exploring which language is suitable for introductory programming courses in universities.

Several recent studies are reviewed. Kunken and Allen (2016) developed a test to assess learning of programming concepts, and used it to investigate the impact of different teaching languages: C++, Java, and Visual Basic. Students' learning to program in Java and C++ consistently performed better than those learning to program in Visual Basic.

As Python became popular as a first language for learning programming, the reports on comparing Python to Java and C increased. Koulouri et al. (2014) revealed that using a syntactically simple language (Python) instead of a more complex one (Java) facilitated students' learning of programming concepts. Wang et al. (2017) selected Python to teach programming to their students who possessed little prior experience. They explained that the reasons for this are simplicity, versatility, and flexibility of Python. Wainer and Xavier (2018) compared an introductory programming course in C to one in Python and found that the course in Python yielded better student outcomes than the course in C. The main concern of introducing programming by Python is that students learn using too simple a language, which causes them to have difficulty when having to handle a more complex one later on. Many studies concluded that a programming course in Python improved student grades and reduced failure rates, and educators who changed their learning language from C or Java to Python could rest easy.

However, there are studies that show that no substantial difference occurs regardless of the language used in introductory programming courses. Tew et al. (2005) investigated how the outcomes differ depending on the students' alternative first programming courses. Their study revealed that the post-test at the end of the second programming course indicated no significant differences in students' understanding of programming concepts between students who learned in Python and those who learned in Java and MATLAB in the introductory courses. Enbody et al. (2009, 2010) compared two groups of students' grades for subsequent programming courses after learning in Python or C++ in introductory programming courses. They found there was no significant difference between the two groups.

Since Swift is relatively new and mainly used for professional sectors, there are few studies on Swift for learning programming. Rogers and Siever (2015) showed advantages of using Swift compared to Objective-C. However, little study could be found comparing Swift to other popular languages. As Tew et al. (2005) and Enbody et al. (2009, 2010) reported, if it does not affect the future learning no matter which programming language students learn as the first one, Swift could be one of our choices for the first learning language with the advantage of facilitating students' motivation, though it is limited to the Apple environment.

2.2 Programming course for non-majors

Humanities majors do not always have a keen interest in a deep level of understanding of software development. Dawson et al. (2018) proposed an introductory programming course CS0.5 (computer science 0.5) for non-majors regardless of their academic area of interest. Dawson et al. reduced the overall number of learning goals compared to CS1. It made the pass rates of students improve considerably in CS0.5 over CS1; however, it did not address how to capture students' interest. Bishop-Clark et al. (2007), and Ali and Smith (2014) also dealt with the fact that taking a first programming course is considered difficult for most non-major students. Their solution is to teach Alice in introductory programming courses. The Alice environment makes it easier for students to create animation and/or games. Their studies showed that working with Alice helped to dispel the notion that programming is "boring," and it also enhanced motivation. On the other hand, Alice is strictly a teaching/learning tool, and is not used in

developing real-world applications. As a more general environment, Fernandez et al. (2017) conducted Android programming as a first course that was open to all students with no prerequisites, and used familiar Android mobile devices. They reported that it was useful to motivate students, and can awaken their interest in programming.

2.3 Features as a learning programming language

At first, this section considers features of Swift as a learning programming language. Mannila and Raadt (2006) compared several languages by 17 criteria for learning programming (Table 1). Each criterion is drawn from the design decisions made by four language creators as they described their

Table 1 Comparison of Language by Features for Learning Programming

Features	C	C++	Eiffel	Java	JavaScript	LOGO	Pascal	Python	VB	Swift
Learning										
(1) Is suitable for teaching			✓			✓	✓	✓		
(2) Can be used to apply physical analogies			✓	✓	✓	✓		✓	✓	✓
(3) Offers a general framework	✓	✓	✓	✓	✓		✓	✓	✓	✓
(4) Promotes a design-driven approach for teaching software			✓	1[a]		✓				✓
Design and Environment										
(5) Is interactive and facilitates rapid code development						✓		✓		✓
(6) Promotes writing correct programs		2[b]	✓	2[b]				2[b]		✓
(7) Allows problems to be solved in "bite-sized chunks"	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
(8) Provides a seamless development environment			✓	1[a]						✓
Support and Availability										
(9) Has a supportive user community	✓	✓	✓	✓	✓			✓	✓	✓
(10) Is open source, so anyone can contribute to its development								✓		
(11) Is consistently supported across environments	✓	✓	✓	✓	✓	✓	✓	✓	✓	
(12) Is freely and easily available	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
(13) Is supported with good teaching material		✓	✓	✓		✓	✓	✓	✓	
Beyond Introductory Programming										
(14) Is not used only in education	✓	✓	✓	✓	✓			✓		✓
(15) Is extensible	✓	✓	✓	✓				✓		✓
(16) Is reliable and efficient	✓	✓	✓	✓	✓		✓	✓		✓
(17) Is not an example of the QWERTY phenomena		✓	✓	✓	✓	✓		✓	✓	
Mannila and Raddt's score (except Swift)	8	11	15	14	9	9	7	15	9	12

Note: Table reprinted from Mannila and Raddt (2006) with addition of last column for Swift.
[a]: Possibly with some IDE (Integrated Development Environment).
[b]: Possibly with unit testing.

languages: LOGO, Pascal, Python, and Eiffel. Criteria are grouped into four related subsections: learning, design and environment, support and availability, and beyond introductory programming. The author added Swift's evaluation to their comparison in Table 1, and numbered the features from one to 17.

A suitable language for teaching in feature (1) means to have simple syntax and natural semantics, avoiding cryptic symbols, abbreviations, and other sources of confusion. Swift does not fit this feature; neither does Java or C family. For example, an optional type that is one of Swift's original elements is difficult for beginners to understand. Swift is apparently more confusing than Java for beginning learners. As regards the other three features in the "learning" group, Swift has the same functionalities as Java.

Since the Playground function in Xcode provides interactive and immediate feedback, Swift has feature (5). This is an advantage of Swift compared to Java and C family. Feature (8) means whether or not a language has an intuitive GUI for design and implementation that provides access to libraries for basic and advanced programming. Xcode has those functions. Swift is superior to other languages with respect to the features in the "design and environment" group.

On the contrary, Swift has drawbacks in terms of the features in the "support and availability" group. While everybody can use Xcode freely and there is a web site to post questions and get comments from fellow developers, Swift and Xcode have been developed by Apple Inc. for its own devices; therefore, the whole decision is made by Apple. For this reason, the answers to features (10) and (11) are "no".

Considering the features in the fourth group, since Swift is a professional programming language, not for learning, it is extensible for real world applications (15) and reliable in creating applications (16). However, regarding feature (17), Swift cannot say its usefulness now and into the future, because users have to follow Apple's future plans and strategy.

Summarizing the comparison, Swift's characteristics are similar to Java with regard to learning. An inevitable development environment for programmers, Xcode, is a quite complex tool for beginners; they are probably confounded by what happens behind their manipulation. To compensate for that, it provides the Playground function that allows students to test snippets of code. Additionally, since Swift programs can work only on Apple's devices, it can be said to be an exclusive environment. This exclusiveness is one reason for its seldom being chosen to learn fundamental programming for beginners in higher education in many cases.

3 Method in course practices and results

3.1 Course structure

The department established two courses to teach programming for beginners: Introductory programming in Java, and Introductory iOS Development in Swift, in 2018 and 2019. Table 2 shows the syllabi for the two programming courses, which took place in a 15-week semester. Both courses have the same structure that starts with basic concepts of programming, moves on to the object-oriented (OO) concepts, and then proceeds to development applications with GUI. The course in Java (Prog A) used an editor and a console window in class, not IDE (integrated development environment) such as Eclipse. For learning the basic concepts and the OO concepts in Prog A, we used the Turtle Graphic library that we developed to function the same as LOGO (Ariga & Tsuiki 2001). It provides students an intuitive understanding of fundamental procedures and the object-oriented principles. In the Swift course (Prog B), students learned those parts by Playground where students used the print function to see how a code works in a read-evaluate-print-loop (REPL). In the GUI programming part, Prog

Table 2 Syllabus of courses in Java and Swift

	Course in Java (Prog A)	Course in Swift (Prog B)
1	Introducing Java, an editor and compiler	Introducing Swift, Playground and Xcode
2	(TG[a]) Data Type, variable, assignment, arithmetic operation, using methods	(Playground) Data Type, variable, assignment, arithmetic operation
3	(TG) Creating methods, control flow (loop)	(Playground) Control flow (loop and if)
4	(TG) Control flow (if)	(Playground) Creating function
5	(TG) Array	(Playground) Array
6	(TG) Creating class	(Playground) Creating class
7	(TG) Creating class	(Playground) Creating class
8	(JavaFX) UI component and layout	(Xcode) An app with UILabel, UIButton, and touch interaction (Hello World)
9	(JavaFX) A program with Label, Button, and click event (Hello World)	(Xcode) An app with calculation of an inputted value in UITextField
10	(JavaFX) A program with calculation of an inputted value in TextField	(Xcode) An app with calculation of an inputted value in UITextField
11	(JavaFX) A program with Button and CheckBox	(Xcode) A tally counter app with UILabel and UIButton,
12	(JavaFX) A program with Button and CheckBox	(Xcode) A tally counter app with UILabel and UIButton,
13	(JavaFX) A rock-paper-scissors game with click interaction	(Xcode) A smash game app with touch
14	(JavaFX) A drawing program with mouse interaction	(Xcode) A smash game app with touch
15	Examination	Examination

[a]: TG is Turtle Graphics library.

A used the JavaFX library, and Prog B used the Cocoa Touch framework in Xcode environment.

3.2 Examination results

At the end of the course, examinations with the same contents were conducted for both courses. The examination consisted of two parts: a basic concept part and a GUI part including object-oriented concepts. The former included data types, arithmetic operators, control flow, and function (or method). For those grammatical elements, the examinations asked the exact same questions for both Java and Swift except for the differences of their syntax. The latter asked about the basic usage of UI component class, event handing by showing a sample code. It included questions on the basic OO such as class inheritance and instantiation in the latter, since the UI components are formed on the OO concept. The program codes that showed in the questions were different because they were based on the application that each course addressed in class, but the questions were intended to confirm the same points.

Forty students, who had not taken a programming course before, participated in Prog A. Twenty students took Prog B, divided into two groups based on their previous learning experience: ten students had no prior programming experience (G1), and ten had taken Prog A in the preceding semester (G2). The G2 students took Prog B for aiming to learn development of iOS application. All were the second- or third-year undergraduate students, and females aged 20-22.

Table 3 and Fig. 1 show the examination

Table 3 Comparing examination result of Prog A and each group of ProgB

Course		Basic Concepts[a]				GUI and OO[a]			
		Median	SD	U-test[b]	R[c]	Median	SD	U-test[b]	R[c]
Prog A (Java, n=40)		27.7	9.46	-		30.6	11.7	-	
Prog B (Swift)	All (n=20)	29.3	9.7	U: 343.0 z: 0.89 p: 0.37	0.12	31.4	11.4	U: 373.5 z: 0.42 p: 0.68	0.05
	G1 (n=10)	28.2	7.9	U: 183.0 z: 0.41 p: 0.68	0.06	20.0	7.7	U: 98.0 z: 2.48 p: 0.01	0.35
	G2 (n=10)	34.5	9.6	U: 126.0 z: 1.80 p: 0.07	0.26	40.0	10.4	U: 141.0 z: 1.43 p: 0.15	0.20

G1: Students without Java (Prog A) experience.
G2: Students who had taken Prog A.
[a]: Each test is 50-point scale.
[b]: Mann-Whitney U-test between ProgA and each group in ProgB.
U: U-value, z: standardized value, p: two-sided p-value.
[c]: Effect size (z/sqrt(N)).

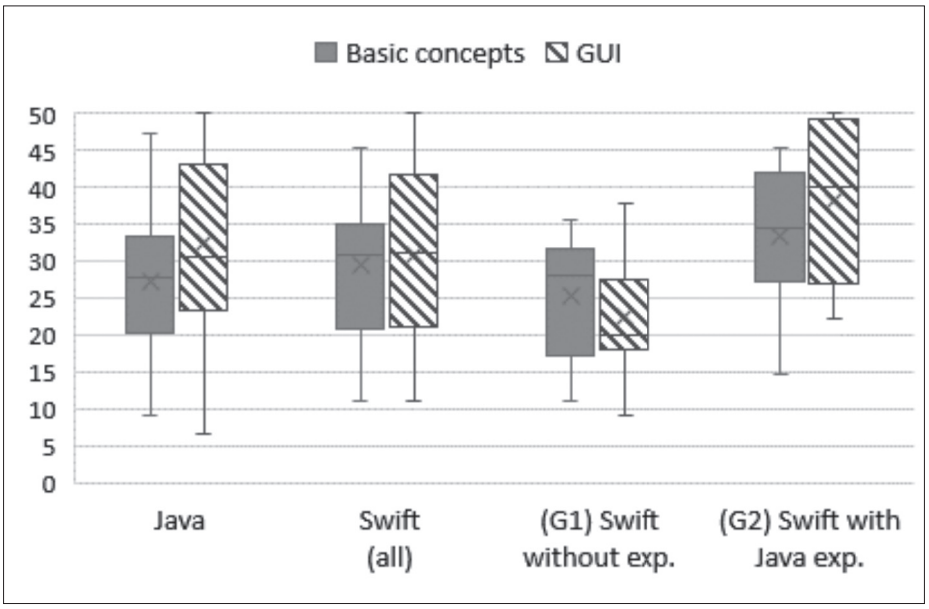

Fig.1 Box plot of exam scores for the basic concepts and GUI part

results of both courses. Each part of the examination (basic concept part and GUI+OO part) was on a 50-point scale. Table 3 presents median and standard deviations for Prog A and each group of students classified by prior programming experience in Prog B. The author ran a Mann-Whitney U test, and calculated an effect size for comparing the examination result of Prog A and each group of Prog B. Fig. 1 presents the distribution of students' scores for Prog A and Prog B. Three box plots for the Swift course (Prog B) respectively show from the left: the whole class, the group without programming experience (G1), and the group with Java experience (G2).

The comparison of the whole class indicates no differences (basic concepts: p=0.37, r=0.12, GUI +OO: p=0.68, r=0.05). However, when looking into the group without programming experience in Prog

B (G1), the median of the GUI+OO part score in G1 (Mdn=20.0) is significantly lower than the one in Prog A (Mdn=30.6, p=0.01, r=0.35). It indicates that Java is better than Swift in terms of learning GUI and OO programming. Whereas, comparing the median of the basic concept score of G1 (Mdn =28.2) to Prog A (Mdn=27.2), there is no significant difference (p=0.68, r=0.22). This shows that a first learning language, whether Java or Swift, does not affect learning of the basic concepts for beginning learners. The statistical difference for other comparisons was not determined from the U-test.

The data of the two groups in Prog B in Fig. 1 shows that the scores of students with the Java experience (G2) are higher than those of students without experience (G1). A Mann Whitney U-test was carried out to check the difference between the two groups (Basic: U=23.0, z=2.04, p=0.04, r=0.46, GUI+OO: U=12.0, z=2.88 p=0.004, r=0.64); the medians of G2 (Basic: 34.5, GUI+OO: 40.0) were significantly higher than those of G1 (Basic: 28.2, GUI+OO: 20.0). Thus, this finding shows that learning in the Java course naturally helps student understand programming in Swift.

3.3 Students' self-reflections

At the beginning and end of each course in 2019, I asked students to rate their own perception of programming ability by a self-efficacy scale that comprised six items (Appendix). Ramalingam and Widenbeck (1998) developed a thirty-two-item self-efficacy scale for C++ programming by a seven-point Likert style scale, and assessed its reliability. Their scale has been used for studies of influence from learning programming (Ramalingam et al. 2004; Sethuraman & Dee Medley 2009). The author selected six items from their scale that are not clearly affected by the learning contents and the language environment, and used them for the questionnaire.

Table 4 shows the median of the amount of the self-efficacy scores that students judged by themselves before and after the course, and the result of a Wilcoxon signed-rank test. The median of the self-efficacy scores increased significantly over both courses: in Prog A from 11 to 18 (p=0.002, r=-0.79), and in Prog B from 14.5 to 21.5 (p=0.004, r=-0.82). It indicates that both courses, in Java and Swift, equally affected an increase in self-efficacy. Looking at data for G1 and G2 in Prog B, the self-efficacy scores increased by seven points over the course for both groups; however the number of subjects was not large enough to calculate statistical significance.

Additionally, students reflected on what they learned from the course by filling out a free-description questionnaire at the end of the course. The author collected responses from 23 students in Prog A and 12 students in Prog B (G1 group: 5,

Table 4 Change in Self-efficacy for programming

Prog A (Java, n=23)	Prog B (Swift)		
	G1+G2 (n=12)	G1 (n=5)	G2 (n=7)
Pre[a]: 11 (SD: 5.0)	Pre[a]: 14.5 (SD: 6.4)	Pre[a]: 14 (SD: 2.9)	Pre[a]: 15 (SD: 4.2)
Post[b]: 18 (SD: 5.9)	Post[b]: 21.5 (SD: 6.1)	Post[b]: 21 (SD: 3.3)	Post[b]: 22 (SD: 8.0)
T[c]: 6.5, z: -3.8	T[c]: 1.0, z: -2.8	T[c]: 1.0, z: -1.62	T[c]: 0, z: -2.1
p: 0.002, r[d]: -0.79	p: 0.004, r[d]: -0.82	- [e]	- [e]

[a]: Median of pre-self-efficacy scores.
[b]: Median of post-self-efficacy scores.
[c]: Wilcoxon signed-rank test statistic, z: standardized value, p: two-sided p-value.
[d]: Effect size (z/sqrt(N)).
[e]: N is not large enough to calculate a p-value.

G2 group: 7), and qualitatively analysed them. 45 key sentences were extracted from students' comments and classified into five categories using SCAT (Steps for Coding and Theorization) open coding. Compared to Grounded Theory Approach, SCAT is a simplified qualitative data analysis method (Otani 2008). Table 5 shows sample comments and the number of comments for each category and each course. While the percentage of comments in the categories 1, 4, and 5 do not display considerable differences between the courses, category 2 (tool) appears only in the Swift course and category 3 (conceptual thinking) appears only in the Java course.

Table 5 Comparing free description comments of Prog A and Prog B

Responses to a question of what you learn from the course		Prog A (Java)	Prog B (Swift)
Category	Sample comments		
1. Direct knowledge of programming	I learned basic knowledge of Java/Swift. I learned how to write a program. I understood how a program works. I understood how to create an application.	18 (66%)	11 (61%) [G1: 5, G2: 6]
2. Tool	I learned how to use Xcode.	-	3 (17%) [G1: 2, G2: 1]
3. Conceptual thinking	I obtained an ability to think how to solve a problem by myself. I learned a way to organize a problem to solve. I learned a way to think computationally.	5 (19%)	-
4. Joy	I found enjoyment of creating a program by myself. I felt achievement.	1 (3.7%)	2 (11%) [G1: 2]
5. Self-efficacy	I had confidence to create a simple program. I want to create a program by myself. I could predict how a program runs.	3 (11%)	2 (11%) [G1: 2]

Note: Total number of comment statements is 27 in Prog A and 18 in Prog B.
(%): Percentage of the number of students who wrote the comment.

4 Discussion

4.1 Basic concept part and language feature

Regarding Java, many educators noted that even a simple program in Java has a verbose and complex syntax overhead (Mannila et al. 2006; Bishop-Clark et al. 2007). In the case of the typical "Hello world" example to output a short phrase, Swift on Playground needs just one sentence: print ("Hello world"); while Java requires knowledge of a class, a main method, modifiers, and array. This feature seems to cause Swift's advantage of teaching the basic knowledge as the first language. Rogers and Siever (2015) mentioned that Swift is designed to avoid most of the simple, subtle logic errors that can be a significant impediment to relatively novice students. However, the comparison of the examination scores showed that there was no difference between Swift and Java to teach the basic concepts.

The author assessed Swift similar to Java for the features of syntax and semantics as a learning language (Table 1), though Java requires complex syntactic and semantic knowledge to simply display "Hello world" to the screen compared to Swift. One reason is that in the case of displaying a few words on a mobile device or a simulator in Swift, a program should include many grammatical elements, and they are not as simple as Java. Another reason is that Swift has peculiar

grammatical elements that make it difficult for beginners to understand: named parameters and optional types. Named parameters are supported in many languages, but not in Java. In Swift, each function parameter has both an argument label and a parameter name. An argument label is an external name, and used when calling a function. A parameter name is a local name, and used in the implementation of a function. Introduction of an argument label is intended to call a function in a natural English sentence manner, and improve readability of codes, as a book on Swift programming (e.g. Sahar & Clayton, 2020) explains the advantage of an argument label. However, it introduces a verbose element that novice students do not easily understand. In addition, the use of an argument label does not enhance readability when calling a function for non-native English speakers, rather it confuses them because of the two names of a parameter.

An optional type is another element that confuses students. The author explained to novice students that it can have a value of nil, which represents no value, but they cannot clearly understand in which case they should use an optional type. When they miss adding a trailing question mark for specifying an optional type in a case of necessity, Xcode suggests a correct code. It is helpful, but does not encourage understanding of why that mark is necessary.

Hence the author judged that Swift does not meet the criterion that the language has a simple syntax and natural semantics, avoiding cryptic symbols, abbreviations, and other sources of confusion, just as Mannila and Raadt (2006) assessed Java. The author was rather concerned that Swift had defects for learning the basic concepts as the first language compared to Java because of these peculiar grammatical elements, but the results of the examination scores show that the Swift course can teach the basic knowledge of introductory programming as well as the Java course.

4.2 GUI＋OO part and Xcode

The median of the examination score for the GUI and OO part in the Java course (30.6) was significantly higher than one in the Swift course (20.0, Table 3). It indicates that Java is better than Swift for beginners to learn the GUI and object-oriented program. There are three possible reasons; one is that Xcode hides the process of creating an object from an GUI component class in the Swift course. Students just manipulate Interface Builder in Xcode by the drag and drop operation to create an object and set it in a view that represents a monitor screen. However, in Java, students need to explicitly write codes to create an object from an GUI component class, and those procedures probably help to enhance understanding of the object-oriented programming concepts.

The second reason is the grammatical element for memory management. When students proceed to the stage of learning GUI in Swift, they have to understand modifiers for memory management, “strong” and “weak,” that specify how to keep data of a variable in memory. Swift does not require programmers to explicitly delete data, but they must correctly specify with a “strong” and “weak” modifier to organize memory. Xcode adds the proper declaration of a variable to store a reference to a GUI component when setting up the connection between a variable and an GUI component in Interface Builder; that is, students do not need to specify it manually. They can compile and run an application if they ignore the meaning. Consequently, they could not answer clearly the examination question about a definition of a variable including a modifier. In contrast, since Java has a garbage collection routine that automatically removes and reclaims memory for reuse when data are no longer necessary, students do not need to deal directly with the codes for memory allocation and recovery.

Additionally, an optional type is inevitable for GUI programming in Swift. Xcode automatically adds a mark for an optional type to a variable of

an UI component. It is very helpful as mentioned above, but students could not answer the meaning of it correctly. Those three issues probably prevented students from understanding the GUI and OO in the Swift course.

The author tried to use Swift as the first programming language, aiming to offer a course for humanities majors to be able to learn not only basic programming concepts but also development of iOS applications without the prerequisite of programming experience. Unfortunately, the findings showed that the Java course was better than the Swift course in the examination scores of the GUI and OO parts, and prior programming experience by a Java course was preferable for students to make learning in Swift effective.

4.3 Students' perception of programming

Self-efficacy is another instrument to measure effect from the courses. Students' self-efficacy would be expectedly increase as a result of learning programming. Students in 2019 answered the questionnaire for their programming self-efficacy. The post-self-efficacy scores were seven points higher than the pre-self-efficacy scores in both courses (Table 4). In each group (G1 and G2) in the Swift course, the self-efficacy scores also increased by seven points, but the result could not show the statistical significance, since the number of students in each group was small.

Students' frustration with Xcode in the Swift course was another concern, whether with or without prior programming experience. Xcode is the complex IDE for developing real applications, and it confounds students, especially when an error occurs. In the Swift course, students used Playground at the first phase of the course to acquire the basic programming concepts and Swift grammar, then the course proceeded to development of a GUI application by using Xcode. The interface and manipulation of Xcode is different from those in Playground, and there is a wide gap between testing snippets of code by Playground and creating an application by Xcode. The author assumed that students would be easily frustrated with programming because of Xcode's complexity and puzzling responses. However, the result showed that self-efficacy in the Swift course increased as in the Java course. This finding indicates that students felt that their confidence in programming was enhanced through the Swift course.

The qualitative analysis of students' responses to the question of what they learned from the course (Table 5) shows that around 60% of the total comments were commonly classified as category 1 for both courses. The students naturally perceived that the core of their learning was to gain the basic knowledge and understand how to create a program.

The comments about tools (category 2) appeared only in the Swift course, though the number of comments was small, and no comment in category 2 appeared in the Java course. It was a natural consequence, because the Java course did not use IDE as opposite to the Swift course using Xcode. On the other hand, the comments in category 3 were observed only in the Java course with the same percentage as category 2 in the Swift course. Those comments were related to the abstract perception on programming, not direct and practical knowledge. Students in the Swift course did not mention abstract things like the ability to think. They tended to focus on concrete things on skills such as a way how to use Xcode. The author assumes that what they learned in the Swift course remained practical matters, and was hardly recognized as abstract knowledge of programming.

5 Conclusion

This study explored how the department could teach introductory programming and simultaneously creation of entry level applications in Swift to humanities majors without a prerequisite by

comparing the introductory courses in Java and Swift. The result indicated that students can learn the basic programming concepts in Swift, similar to in Java, but the effect of the Swift course for the GUI and OO part was inferior to that of the Java course. From the finding that the previous learning in Java positively affected learning in the Swift course, Java is considered to be a better choice for an introductory programming rather than Swift.

Not only students with a CS background but also those with a humanities background will be expected to participate in planning and designing new digital contents based on information technology in the future. Fostering the ability of creating digital contents is therefore significant for students regardless of major. For this purpose, our Swift course also intended to capture the motivation of humanities majors in programming. The result showed an increase in self-efficacy for programming in the Swift course as well as in the Java course. This finding suggests that the Swift course enhances the perception of confidence of programming regardless of the level of gained knowledge over the course, and encourages humanities majors to continue learning programming.

Observing more students' self-reflections is necessary to discuss about motivation and satisfaction obtained from the course sufficiently. In future work, the author intends to collect more data of students' self-efficacy and reflections, and examine them.

References

Ali, A., & Smith, D. (2014). Teaching an introductory programming language in a general education course. *Journal of Information Technology Education: Innovations in Practice*. 13, 57-67. https://doi.org/10.28945/1992

Ariga, T., & Tsuiki, H. (2001). Programming for students of information design. *ACM SIGCSE Bulletin*, 33(4), 59-63. https://doi.org/10.1145/572139.572172

Bishop-Clark, C., Courte, J., Evans, D., & Howard, E. V. (2007). A Quantitative and Qualitative Investigation of Using Alice Programming to Improve Confidence, Enjoyment and Achievement among Non-Majors. *Journal of Educational Computing Research*, 37(2), 193-207. https://doi.org/10.2190/J8W3-74U6-Q064-12J5

Camp, T., Adrion, W. R., Bizot, B., Davidson, S., Hall, M., Hambrusch, S., Walker, E., & Zweben, S. (2017). Generation CS: The Growth of Computer Science. *ACM Inroads*, 8(2), 44-50. https://doi.orgs/10.1145/3084362

Davies, F., 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. https://doi.org/10.1145/1953163.1953339

Dawson, J. Q., Allen, M., Campbell, A., & Valair, A. (2018). Designing an Introductory Programming Course to Improve Non-Majors' Experiences. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 26-31. https://doi.org/10.1145/3159450.3159548

Enbody, R. J., Punch, W. F., & McCullen, M. (2009). Python CS1 as preparation for C++ CS2. *Proceedings of the 40th ACM technical symposium on Computer science education*, 116-120. https://doi.org/10.1145/1508865.1508907

Enbody, R. J., & Punch, W. F. (2010). Performance of python CS1 students in mid-level non-python CS courses. *Proceedings of the 41st ACM technical symposium on Computer science education*, 520-523. https://doi.org/10.1145/1734263.1734437

Fernández, C., Vicente, M. A., Galotto, M. M., Martinez-Rach, M., & Pomares, A. (2017). Improving student engagement on programming using app development with Android devices. *Computer Applications in Engineering Education*, 25(5), 659-668. https://doi.org/10.1002/cae.21827

Koulouri, T., Lauria, S., & Macredie, R. D. (2014).

Teaching introductory programming: A quantitative evaluation of different approaches. *ACM Transactions on Computing Education*, 14(4), Article 26, 28 pages. https://doi.org/10.1145/2662412

Kunkle, W. M., & Allen, R. B. (2016). The impact of different teaching approaches and languages on student learning of introductory programming concepts. *ACM Transactions on Computing Education*, 16(1), Article 3, 26 pages. https://doi.org/10.1145/2785807

Mannila, L., Peltomaki, M., & Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program. *Journal of Computer Science Education*, 16(3), 211-227. https://doi.org/10.1080/08993400600912384

Mannila, L., & Raadt, M. (2006). An objective comparison of languages for teaching introductory programming. *Proceedings of the 6th Baltic Sea conference on Computing education research*, 32-37. https://doi.org/10.1145/1315803.1315811

Murphy, E., Crick, T., & Davenport, J. H. (2017). An Analysis of Introductory Programming Courses at UK Universities. *The Art, Science, and Engineering of Programming*, 1(2), Article 18, 23 pages. https://doi.org/10.22152/programming-journal.org/2017/1/18

Otani, T. (2008). "SCAT" a qualitative data analysis method by four-step coding: Easy startable and small scale data-applicable process of theorization. *Bulletin of Nagoya University graduate school in education*, 54(2), 27-44.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy, *Journal of Educational Computing Research*, 19(4), 365-379. https://doi.org/10.2190/C670-Y3C8-LTJ1-CT3P

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program, *ACM SIGCSE Bulletin*, 36(3), 171-175. https://doi.org/10.1145/1026487.1008042

Rogers, M. P., & Siever, B. (2015). Switching to Swift: instructional issues and student sentiment. *Journal of Computing Sciences in Colleges*, 30(5), 144-150.

Sahar, A., & Clayton, C. (2020). *iOS 13 Programming for Beginners*, Packt Publishing Ltd.

Sethuraman, S., & Dee Medley, M. (2009). Age and self-efficacy in programming, *Journal of Computing Sciences in Colleges*, 25(2), 122-128.

Simon, Mason, R., Crick, T., Davenport, J. H., & Murphy, E. (2018). Language Choice in Introductory Programming Courses at Australasian and UK Universities. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 852-857. https://doi.org/10.1145/3159450.3159547

Tew, A. E., McCracken, W. M., & Guzdial, M. (2005). Impact of alternative introductory courses on programming concept understanding. *Proceedings of the first international workshop on Computing education research*, 25-35. https://doi.org/10.1145/ 1089786.1089789

Wainer, J., & Xavier, E. C. (2018). A Controlled Experiment on Python vs C for an Introductory Programming Course: Students' Outcomes. *ACM Transactions on Computing Education*, 18(3), Article 12, 16 pages. https://doi.org/10.1145/3152894

Wang, Y., Hill K. J., & Foley, E. C. (2017). Computer programming with Python for industrial and systems engineers: Perspectives from an instructor and students. *Computer Applications in Engineering Education*, 25(5), 800-811. https://doi.org/10.1002/cae.21837

Appendix:
Questionnaires for Self-efficacy of programming

Rate your confidence in doing following programming tasks using a scale of 1 (not at all confident), 2 (mostly not confident), 3 (slightly confident), 4 (50/50), 5 (fairly confident), 6 (mostly confident), or 7 (absolutely confident).

(1) I can write syntactically correct Java/Swift statements.
(2) I can debug (correct all errors) a long and complex program that I had written and make it work.
(3) I could complete a programming project if I had only the language reference manual for help.
(4) I could complete a programming project if I had a lot of time to complete the program.
(5) I could find ways of overcoming the problem if I got stuck at a point while working on a programming project.
(6) I could mentally trace through the execution of a program given to me.