LingBaW

Linguistics Beyond And Within 7 (2021), 5-18

Spreading digital literacy via Minimal English. The concept of 'class' in Swift programming language

Bartłomiej Biegajło
Siedlce University of Natural Sciences and Humanities, Poland

Abstract
The article aims at providing explications of the concept of a class, as it is implemented in the Swift programming language offered by Apple. The explications are framed in Minimal English, which is based on the theory of Natural Semantic Metalanguage. Detailed analysis of the Swift concept of class leads to four distinct core explications of the programming construct in question and the related feature that Swift classes possess, namely the concept of property. The article's primary purpose is to offer a more smooth experience with programming, especially with beginners in mind. Their initial exposure to programming might face several challenges due to the complicated digital jargon of the documentation. Minimal English is implemented to ease the learning curve and promote digital literacy as one of the most fundamental skills in today's world.

Keywords: programming, Swift language, Natural Semantic Metalanguage, Minimal English

1. Introduction

If literacy today might be taken for granted, digital literacy still has a long way to be considered a widespread skill available to everyone. In a well-researched book, *How Was Life? Global Well-being since 1820*, which is a selection of commentaries on world literacy seen from several alternative viewpoints and time frames, the authors try to pin down the actual gain which goes together with language competence and state the following:

> 'Literacy and education are crucial variables determining well-being, since they not only directly influence a person's intrinsic agency, but also indirectly affect well-being in material (e.g. per capita income and wages) and immaterial terms (e.g. lower crime rates, higher life expectancy).'
>
> van Leeuwen and van Leeuwen-Li (2014: 98)

It is particularly interesting to situate literacy in this combined context of gains and their intrinsic opposition. While oscillating between the two poles, material gains vs immaterial gains, it is tempting to conclude that the very skill of becoming literate, which is essentially yet another area of knowledge one can take the effort to master, can lead to tangible results. Not only do they provide change to the general well-being of the humankind but also have the

potential to enrich the physical reality as they translate literacy into a better social status of an individual. Investigating different planes of literacy in ancient Greece, Rosalind Thomas seems to give special attention to the tangible results on which the promotion of literacy had to necessarily concentrate: "for literacy to take root in a society, it has to have meaning, it needs to have obvious and valuable uses, to be 'relevant' or empowering in some way" (Thomas 2009: 13). It would not be far-off to presume that, historically, literacy has been the type of skill that allowed people to enjoy being 'upwardly mobile'. Educators' role in promoting the immediate advantages that go together with acquiring the ability to read and write is beyond dispute. David Olson, discussing the 'literacy hypothesis', which he understands as: "the bold claim that the invention, adoption, and application of a new mode and technology of communication, namely writing, altered the social practices of the society as well as the cognitive processes of those so affected" (Olson 2009: 386), appears to reiterate the underlying role played by educators: "the literacy hypothesis received a ringing endorsement from educators. It confirmed the long-held belief that early education, centered on learning to read and write, was a universally valid goal" (Olson 2009: 387).

Few would argue that the contemporary idea of literacy assumes a far broader meaning and is hardly limited to acquiring an essential skill set allowing one to communicate with words. Rijpma observes that "in respect to basic education, the world has progressed from low to near-universal literacy attainment" (Rijpma 2014: 251), and, echoing van Leeuwen and van Leeuwen-Li, declares that "education is important for well-being because improved access to information is of intrinsic importance, but also because there are indirect effects through the impact of education on other well-being indicators, such as income, health and political stability" (Rijpma 2014: 251). The contemporary world offers other stimuli for 'upward mobility'. Becoming a digitally literate participant of life creates many more opportunities for an individual, not only from the perspective of sheer income capability but also from the viewpoint of challenging the traditional modes of cognitive processes attached to reasoning and categorizing the digital data described with the use of a natural language. Both the seasoned programmers and everyday users of digital applications can make sense of the digital information that is open to manipulation – it is part and parcel of their everyday digital experience. However, readability, considered from the vantage point of a programmer, as opposed to a user, initiates an entirely different set of cognitive processes. Wierzbicka remarks that "every language has lexically encoded some scenarios involving both thoughts and feelings and serving as a reference point for the identification of what the speakers of this language see as distinct kinds of feelings" (Wierzbicka 1999: 15). The assumption is that as long as one agrees that there is a scenario for conceptualizing and expressing feelings in natural languages, an equivalent scenario has to exist for conceptualizing and expressing the specific type of information encoded in a meaningful line of code to a programmer and a user. They, however, involve two distinct planes of 'meaningfulness' – the user enjoys the performance of an application, its usefulness together with a visual appeal is what matters the most. At the same time, a programmer is most typically concerned with the logic of an application, its natural 'flow' encoded in lines of code that he/she conceptualizes before translating it into a user-friendly expression, i.e. the performance. The scenario does not change. It is the reference point that is different. It can be said that a different kind of focus is at play for the respective groups of participants of the digital literacy

phenomenon. It is conditioned by different needs which the respective groups are interested in pursuing. On the one hand, users seek functionality and performance that would meet their individual needs. On the other hand, programmers pursue the aim of exploring diversified ways in which the individual needs of users can be satisfied. To achieve this result, programmers are conventionally expected to employ the tools available to them in the form of knowledge about how to make one line of code communicate with other lines of code in a given digital application. It is the scenario that programmers are required to comprehend, unlike in the case of users who, very frequently, rely on their intuition while working with a digital application.

In one of her books, Wierzbicka is seen to be keen on addressing the 'native speakers' intuitions' as both the first step and the final step for testing the hypotheses she puts forward: "objective data, such as those that occur in contemporary linguistic corpora, cannot interpret themselves, and to make sense of them one still needs to consult ones' semantic intuitions" (Wierzbicka 2010: 20), or elsewhere: "although the figures involved are small, these results are consistent with native speakers' intuitions, which […], allow us to formulate two generalizations […]" (Wierzbicka 2010: 305). Apart from showing Wierzbicka's approach to her research agenda, these and many other passages appearing in the book can indeed shed some light on the similar experience that digital literacy has to offer. As long as linguists are customarily expected to study languages from the 'under-the-hood' perspective, native speakers are often unable to explain why a language rule they apply intuitively and, more importantly, correctly takes a specific form of a given kind. It seems that the phenomenon of digital literacy can be considered from a similar context – programmers acquire knowledge about what occurs 'under the hood' once a digital application is initiated and can read and add new lines of code to change the behaviour of a programme. In contrast, users accept the performance of an application without having to understand the nuts and bolts or the technical configuration of the application. What connects these two distinct types of digital literacies (programmers' vs users') is their direct usefulness which translates into potential immediate gains, including financial benefits. It is the same example of usefulness mentioned by Thomas in the context of traditional literacy developed in ancient Greece – both offer true promise for a different, better life, as the new skill is directly relevant to humankind's condition.

The pressure on usefulness is also voiced by Tariq Rashid – an ardent advocate of computer literacy: "many education curricula have been updated to ensure that children are digitally literate, equipped to participate in a digital economy, able to develop their own technology ideas, and be better-informed consumers and citizens" (Rashid 2019: 6). What Rashid means is not only confined to transforming oneself into a proficient user of a plethora of digital applications available on the market but rather being able to develop the crucial skills that would significantly enhance the understanding of how this very specific digital market is arranged. According to Rashid, "coding is considered by many to be as essential as reading and writing" (Rashid 2019: 6), and one may be tempted to add one of the essential types of literacies that one should contemplate mastering in the 21st century.

Apart from school curricula, there are nowadays several various private market initiatives aimed at spreading knowledge about programming. Various coding boot camps (e.g. Coder Academy, General Assembly, FireBootCamp, Le Wagon, to name just a handful of similar enterprises in Australia alone) are currently becoming extremely attractive to programming

enthusiasts, which testifies that the importance of digital literacy is growing at an unprecedented rate. Boot camps are often successful in demystifying the complex idea of programming and can be very effective in terms of future employability, they are often overly expensive and, at many times, fail to be tailored to the cognitive needs of school children who are less concerned with market competitiveness, and more enticed to pursue the creative aspect of programming.

This market void has recently been filled with coding coursebooks aimed directly at schoolchildren who only take their first steps in this area (Prottsman 2019; Vorderman et al. 2014; Woodcock 2016a, 2016b). They, however, rely heavily on visual content which accompanies the explanations related mainly to the basic logic behind the projects which are being discussed. At this stage of a learning path, the idea of entertaining a thought of covering a set of more complex programming constructs, including, for example, Object-Oriented Programming, is hardly justifiable. However, since illustrations of programming concepts, which due to their very nature are highly artificial, take place with the use of a natural language such as English, the prerequisite for these linguistic illustrations is that a language employed to discuss these artificial concepts (i.e. the logic of programming as well as programming concepts) has to be clear, unambiguous and comprehensible. Object-Oriented Programming, with which the concept of class is commonly associated, is most often omitted in programming coursebooks for children, and there is a good reason for it – it is very abstract and poses a genuine challenge even to more experienced programmers. On the other hand, 'classes' provide far broader functionality to a digital application and mastering the intricacies of 'classes' and other Object-Oriented Programming concepts can significantly benefit future job prospects in a digital market.

This paper assumes that Object-Oriented Programming can be explained using a natural language, e.g. English, Chinese, Russian, etc., which, additionally, can be simple, transparent and not off-putting to school children. The help can come from a reduced version of English, Chinese, Russian, etc. The idea of language reductionism has been promoted by researchers working within the framework of Natural Semantic Metalanguage, especially Anna Wierzbicka and Cliff Goddard. Their relatively recent project is known under the name of Minimal English (just as there can be Minimal Chinese, Minimal Russian, etc.), and the following study relies on the theoretical tenets proposed by researchers involved in this particular project. The case study is focused on one of the Object-Oriented Programming concepts, namely a 'class', as it is implemented in Apple's Swift programming language.

2. Minimal English revisited

Minimal English stems from the theory of Natural Semantic Metalanguage.[1] NSM has received wide recognition and has been discussed extensively throughout the past four decades (most recent NSM and Minimal English studies include Goddard 2018a, 2018b; Goddard and Wierzbicka 2014; Wierzbicka 2014, 2010). Minimal English is based on the assumption that each natural language possesses a set of words and, consequently, a set of related concepts these

[1] Hereafter referred to as NSM.

words encapsulate, which have a universal or near-universal meaning. In one of her recent books, Wierzbicka calls it "a neutral framework for comparing meanings across cultures" (Wierzbicka 2014: 16) and, indeed, neutrality and translatability are probably the two keywords that highlight the core part of hypothesis the researchers working within the framework of Minimal English are seen to formulate. Goddard and Wierzbicka imply precisely this, stating the following: "since Minimal English has its counterparts in Minimal Chinese, Minimal Russian, Minimal Finnish, and so on, expressing oneself in Minimal English facilitates translatability into one's home language, if that is a language other than English" (Goddard and Wierzbicka 2018: 23). Therefore, the underlying assumption behind the theory is the idea of possible cross-translatability between any number of natural languages without a loss of meaning which is typically associated with a transfer of meaning from one language to another. According to the theory of NSM and its superset version, Minimal English, this can be achieved with the help of any natural language (not limited to English only) suitably adapted to operate on a tested number of more simple concepts, separated from the vast pool of vocabulary available in a given natural language. Thus, the dream of faithful translation becomes a reality as it can be supported with a viable theory. A theory based on extensive research has involved continuous testing and the subsequent modifications that the theory underwent along the way as new data kept emerging.

It is beyond the scope of this paper to address in detail the stages of the development of NSM, especially the history behind the long and arduous task of establishing the final and definite version of NSM. Of historical note is what Goddard shares in one of his publications: "the Minimal English project has emerged from, and in a sense rests upon, the findings of a program of linguistic research known as NSM (Natural Semantic Metalanguage)" (Goddard 2018c: 29). Additionally, Goddard highlights the fact that it is a highly systematic study of meaning that "places words and meaning at the very centre of language study" (Goddard 2018c: 29). NSM is essentially a reduction-oriented analysis of meaning where a fixed set of 65 words are regarded as universal concepts which, if translated to other natural languages, retain the same meaning as the words they were translated from. A complete list of updated 65 semantic primes, as they are sometimes alternatively referred to, can be conveniently accessed via the official Natural Semantic Metalanguage website.

Biegajło notes that NSM "is simply a tool made of any natural language (the assumption being that semantic primes connote the same meaning, regardless of the language they are translated into) which is used to talk about less simple concepts found in those languages" (Biegajło 2019a: 10). It convincingly recapitulates the underlying tenets of the framework. The problem with NSM, however, lies elsewhere and was succinctly pointed out by another Polish scholar, Roman Kalisz, who observed that: "the explications that rely solely on primes are vague, which is the opposite of what they are meant to achieve" (Kalisz 1998: 56), where explications are meant to be understood as the vital instrument in defining the meaning of a given concept. Over the years, as more voluminous amounts of data became available to the NSM community, the project evolved to embrace this setback. In order to secure the readability of the explications and also in an attempt to address the intrinsic and natural cognitive expectations of the human mind, which, most typically, strives to receive meaning which is unconvoluted and transparent, the NSM researchers proposed what is called 'semantic molecules'. According to Goddard, "the

principle was clear enough: certain complex terms were needed as 'concept-building' elements" (Goddard 2018c: 51), thus partly eradicating the frequently striking vagueness of the NSM explications. Simultaneously, trying not to compromise the core assumptions of the NSM theory, it was fundamental that in search of the actual list of semantic molecules, they have to be able to be explicated into primes, "so there is no danger of circularity and no compromise of the reductive principle" (Goddard 2018c: 50). In other words, semantic molecules are regarded as near-universals, 'near-primes', although, technically, they do not belong to the selected category of the 65 semantic primes which are part of NSM. However, they are considered necessary, firstly, to write explications of more complex concepts and, secondly, to complement the readability of the explications. A complete account of the developments involving testing and selecting the final, tentative version of the list of semantic molecules can be found primarily in Goddard (2018c).

Eventually, as Goddard admits, "the Minimal English project began to take shape in 2013" (Goddard 2018c: 61), and the key assumption that goes together with the inception of Minimal English is that "the NSM research community had accumulated enough knowledge and experience about semantic variation and cross-translatability that it was now practical to adapt NSM into a user-friendly tool for thinking and communicating outside the confines of Anglo English" (Goddard 2018c: 61). Strictly speaking, it is turning the forty-year research within the NSM framework into a more practical and less theoretical endeavour that would potentially serve the needs of a wider audience, unlimited to scholars and the world of academia. A complete set of lists grouped into selected thematic categories, including words and the related concepts 'allowed' in Minimal English, can be found in Goddard and Wierzbicka (2018). This paper rigorously follows the proposed vocabulary sets, and all the explications that follow are based on a collection of words presented there.

3. The documentation of Object-Oriented Programming vs a natural language

According to the updated version of the Swift documentation provided by Apple, "*structures* and *classes* are general-purpose, flexible constructs that become the building blocks of your program's code" (Apple 2020: 345, original emphasis). In essence, Swift's structures and classes share a standard set of features that, from the perspective of their core application, are intended to represent one of the critical components of a computer program. It is the building blocks into which structures and classes are transformed that make them essential components of an application because when the code of a program is executed, whether it is a building block or a single line of code, it triggers a series of specific instructions that an electronic device should execute. Incorrect code or no code at all means that a device cannot make sense of the instructions at hand or that it remains idle because there are no instructions to be interpreted by a device interpreter. Common sense suggests that the internal arrangement of Swift structures and classes must not only be understandable to an electronic device in order for them to be fully usable but must also usually occupy a clearly specified place within a body of code if they are intended to be constructs that are 'general-purpose' and 'flexible'. Programmers often refer to such a collection of rules as the syntax of a programming language, and Apple's

documentation is no different in acknowledging a fundamental significance to the syntax of both structures and classes: "you define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions" (Apple 2020: 345). Biegajło notes that "any given app is essentially a collection of data that can be stored in various types of containers whose contents can be freely manipulated" (Biegajło 2019b: 246), and structures together with classes, but also properties, methods, constants, variables and functions, to name just a few of the most common programming concepts, are no exception in this respect. A syntax error or an inappropriate distribution of a building block leads to an app crashing. Only extensive trial-and-error practice can lead an aspiring developer to integrate various programming concepts into a unified and functioning program. This is why novice programmers often fail at the beginning of their programming experience and eventually often give up the challenge to learn the tricks of the trade too early. It seems hardly helpful for them to read passages of the following kind:

> 'Structures and classes in Swift have many things in common. Both can: define properties to store values, define methods to provide functionality, define subscripts to provide access to their values using subscript syntax, define initializers to set up their initial state, be extended to expand their functionality beyond a default implementation, conform to protocols to provide standard functionality of a certain kind.'
>
> Apple (2020: 346)

As long as an experienced programmer can easily translate the list of capabilities inherent in structures and classes into a meaningful piece of valid information, beginners would most likely be confused by the overwhelming jargon they are forced to make sense of.

The following example from Apple's guide proves that this practice is not just occasional:

> 'Classes have additional capabilities that structures don't have: inheritance enables one class to inherit the characteristics of another. Type casting enables you to check and interpret the type of a class instance at runtime. Deinitializers enable an instance of a class to free up any resources it has assigned. Reference counting allows more than one reference to a class instance.'
>
> Apple (2020: 346)

Being precise and thus avoiding syntax errors when writing code in any programming language is equally important as providing clear-cut definitions or explanations about what a selected piece of code is set to do. These, however, seem to be two entirely different areas of activity. If syntax comprehension is about language competence, documentation of the code's behaviour needs to have a certain didactic angle attached to it, which, most typically, rarely go hand in hand, as evidenced by the two passages above. Donald Knuth, who, according to a research profile available at the Stanford University website, is widely credited as the father of the analysis of algorithms, noted the following as early as 1984:[2]

> 'The past ten years have witnessed substantial improvements in programming methodology. This advance, carried out under the banner of "structured programming", has led to programs that are more reliable and easier to comprehend; yet the results are not entirely satisfactory. My purpose in the present paper is to propose another motto that may be appropriate for the next decade, as we attempt to make further progress

[2] Under the title "Literate programming", the paper appeared in 1984 and was later reprinted in the collection of papers quoted in this article.

> in the state of the art. I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming".'
>
> Knuth (1992: 99)

In other words, Knuth implies that computer programs[3] should be written in clean code that would be readily interpretable by electronic devices as well as human beings. The mathematician further elaborates on what literate programming points to: "the practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style" (Knuth 1992: 99). Knuth made his statement perhaps slightly too heavily laden in metaphor, nonetheless, it is an evocative illustration of the central assumption that is being suggested, i.e. a code needs documentation. It is an absolute must for it to be well-written and therefore readable to professionals and non-professionals alike.

Apple proudly boasts that Swift is "an industrial-quality programming language that's as expressive and enjoyable as a scripting language" (Apple 2020: 2), and it, indeed, belongs to a small group of programming languages that offer comparably more friendly experience than other common languages. Swift is devoid of many typical features that other languages contain, which makes it significantly more readable to humans, but at the same time, it does not lose the various functionality and can be employed to perform a number of complex tasks within a computer program and beyond. Unfortunately, the complex digital jargon found in the documentation seriously hinders potential programming enthusiasts from recognizing the full scope of Swift's applicability and, consequently, creates unnecessary barriers to understanding the concepts at play 'under the hood' of computer programs. Natural Semantic Metalanguage, combined with the functionality of Minimal English, can play a significant role in making visible advances in bridging the gap between the code's logic and the code's documentation. The ensuing discussion is primarily concerned with Swift's concepts of a class, as an exemplary concept of Object-Oriented Programming, and a directly related concept by which a class can be identified, namely the concept of property.

4. Documentation written in Minimal English?

To understand the functionality of a class code in Swift, it is necessary to re-emphasize that all programming activity involves traffic of data. Biegajło notes that "users can manipulate data – change their contents, add new items, delete unnecessary parts, or remove them altogether, and, essentially, store them in memory of a device" (Biegajło 2019a: 7) and, therefore, the opening question in the context of Swift classes would be to provide the most general characteristics for a class creation, with a clear implication that, once introduced into a code, it can be populated with data. One critical remark to make at this point refers to what has already been said about selected distinctive capabilities only Swift structures and classes are said to possess. No other programming concept in Swift can accept what Apple identifies as properties and methods. They are complex concepts. At least one of them would require further explanation, but simultaneously, their introduction to the explication of the general characteristics of a Swift

[3] The label, 'computer program', is understood here as an application launched on any electronic device.

class would help determine the preconditioned essence of the concept of class, i.e. its unique capacity to accept properties. Below is a proposed explication of the concept of a Swift class:

class (general characteristics):
a. something
b. someone can say many things about something else with this something
c. there can be/are (many) things (properties) inside this something

Component (a) ("something") verifies the fact that a class is unlike any animate object, it cannot make decisions, it is fundamentally a 'general-purpose', 'flexible construct' that is seen as a specific type of object created to store various types of data, as is further elaborated by component (b) ("someone can say many things about something else with this something"). The third component of the explication intends to differentiate classes from other programming constructs (e.g. variables, constants, functions, loops, etc.). It refers to the concept of property as distinctive programming construct that only classes and structures share in common. If the explication of a class is to be viable, the explication of the concept of property has to accompany the one above and is provided in later sections of this paper. Technically, if a programming concept offers data storage and, among many other things, it can accept properties, it is safe to assume that a Swift programmer works either with a class or with a structure.

Another essential feature that Swift classes are distinguished by is using a specific heading, otherwise technically labelled a keyword that indicates we are dealing with a class. Apple declares that "structures and classes have a similar definition syntax. You introduce structures with the 'struct' keyword and classes with the 'class' keyword" (Apple 2020: 347), and the intuitive denotation that the respective keywords carry greatly simplifies the overall experience of working with a Swift code. Apple offers an exemplary blueprint for both programming concepts:

```
struct SomeStructure {
// structure definition goes here
}
class SomeClass {
// class definition goes here
}
```

Apple (2020: 348)

Based on these remarks, below is a tentative version of the explication outlining the application of the 'class' keyword:

the *class* keyword
a. before all other things in this thing, there is the word "class"
b. because of this, this thing is a class

The explication consists of only two succinct components. It is readable and easy to follow, which is especially helpful for beginners. Component (a) simply postulates that to create a class, all that is required is a specific Swift keyword, i.e. a 'class' keyword. Component (b) stipulates

that once the keyword is introduced, one deals with a programming construct called a Swift class.

Swift documentation also suggests that "whenever you define a new structure or class, you define a new Swift type" (Apple 2020: 348) which means that data can be encapsulated in several specific categories that are governed by a collection of syntactic rules. These and prior explanations collectively can serve as the basis for the explication, which outlines the consistent method of creating a specific type of Swift class.

Apple continues its commentary, pointing out that: "both [i.e. classes and structures] place their entire definition within a pair of braces" (Apple 2020: 347) which allows for a definition of the syntax of classes, as they are typically used in Swift:

defining a *class* type X
a. there is one word (X) after the word "class"
b. (it is before the "opening brace") if someone writes this word (X), this someone makes a class of kind X
c. after this, someone can do something with this class/someone can say something about this class
d. it is like this:
e. something is on two sides of a class
f. on one side, it is something like this: "{"
g. it is called "an opening brace" of a class
h. after this, there can be many things ("properties") that are part of this class
i. after these things ("properties"), it is something like this: "}"
j. it is called "a closing brace" of a class
k. after this, there cannot be things ("properties") that are part of this class

The opening component introduces the notion which is a part of an imprinted functionality of Swift classes and allows a programmer to create a specific instance of a class, which turns it into a specific type of a class ("if someone writes this word (X), this someone makes a class of kind X"). In most cases, the class type depends on the word that follows the 'class' keyword and can be composed of any number of characters that do not have to imply any meaning whatsoever. However, the advised practice widely shared among programmers is to give the type a specific, recognizable name that would greatly ease the navigation through the code, especially if the code requires (and, virtually, it almost always does) changes in the future. Thus, both components ("(a) there is one word (X) after the word "class" " and "(b) (it is before the "opening brace") if someone writes this word (X), this someone makes a class of kind X") are meant to satisfy this unwritten rule and implement the 'word' as a potential candidate to become a type of class, instead of a random set of characters which, as has been suggested, is also possible. The part "opening brace" does not belong either to the NSM set or to minimal language. Therefore it has been included in the quotation marks. Their meaning and significance for the composition of a Swift class need to be accounted for in the definition of a class. Once the class type is established, the class can be incorporated into a code and used and, more importantly, reused throughout the lifecycle of a programme (component (c)).

The braces mentioned in Apple's documentation illustrate that Swift class' scope is determined by an opening brace and a closing brace, respectively. Anything that falls beyond these confines is not part of the specific class in question; however, it can be part of another class that is located before or after the location of a given class. The three components ((d)–(g))

hint at this unique feature of Swift classes which programmers would read as the start of the abstract scope of a class.

Component (d) ("it is like this:") introduces the steps that need to be taken to create a class. Component (e) ("something is on two sides of a class") indicates that in order to create a class, according to the rules set by Swift, we are required to include something before it and after it. Components (f) ("on one side, it is something like this: "{" ") and (g) ("it is called 'an opening brace' of a class") demonstrate what is needed to start the scope of a class.

The four closing components ((h)–(k)) imply the end of the scope of a class. As classes can accept a number of programming constructs, as long as they can hold and manipulate data, for readability reasons, only one of them has been implemented into the explication, namely the concept of property. The word 'property' is not part of Minimal English and has been included in quotation marks. It is meant to indicate that a separate explication exists, namely that if 'property'. The concept of property has to be included in the explication of a class because it allows delineating the functionality of Swift classes from other constructs available in Swift. Finally, component (k) ("after this, there cannot be things ("properties") that are part of this class") shows that anything that falls beyond the class does not and cannot belong to the scope of a given class.

Before moving on to the discussion of the explication of a Swift concept of class, first, an explication of the concept of property has to considered. Property is seen here as one of the distinct members of Swift classes. It is therefore unique to classes only (although, technically, structures, mentioned earlier, copy the behaviour of classes in this respect; however, addressing this dual functionality of Swift properties is beyond the scope of this paper). The uniqueness of properties depends solely on whether they are inside a class or outside a class, as it critically conditions both the naming conventions and their scope. Once inside a class, property is accepted by a class as its member and is considered a fully-fledged example of a property. If, however, it is moved outside the scope of a class, a Swift class ignores it entirely, but its functionality is not lost, and the property turns either into a variable or a constant.[4] Apple explains that "a property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class" (Apple 2020: 27).

The concept of variable and the concept of constant in Swift were analyzed with the application of NSM by Biegajło (2019a). The study showed that Swift variables and constants need to have two distinct explications to illustrate the core difference between them regardless of their apparent similarity. Below are explications of the Swift concepts of variable and constant:

variable (var) of kind X:
a. there can be something inside it
b. this something inside is one thing of kind X
c. this something inside cannot be two things of kind X
d. many things of kind X can be inside it at many different times
e. one thing at one time, another thing at another time.

Biegajło (2019a: 14)

[4] For a detailed discussion of Swift variables and constants from the perspective of NSM, see Biegajło 2019a.

constant (let) of kind X:
a. there can be something inside it
b. this something inside is one thing of kind X
c. this something inside cannot be two things of kind X
d. at all times this one thing is always the same thing.

Biegajło (2019a: 15)

As variables and constants are essentially "examples of unique labels, i.e. containers capable of storing data" (Biegajło 2019a: 12), both opening components (a) ("there can be something inside it") in the explications above point explicitly to that interpretation – variables and constants in Swift can hold data. Furthermore, both variables and constants "store precisely one value at a given time in the lifespan of an application" (Biegajło 2019a: 13) and, as Apple implies, "once you've declared a constant or variable of a certain type, you can't declare it again with the same name, or change it to store values of a different type" (Apple 2020: 59). Therefore, components (b) ("this something inside is one thing of kind X") and (c) ("this something inside cannot be two things of kind X") of the two explications contain a direct reference to the mentioned characteristics of variables and constants – they can accept only one value at a time, and, similarly to the behaviour shown by Swift classes, due to the in-built Swift functionality, these values, have to necessarily be of specific 'kind', i.e. they have to be ascribed a certain type (e.g. a string, a number, a Boolean value (i.e. a value which evaluates to true or false), etc.), depending on the type of value that is inside a variable or a constant. Eventually, the two closing components in the case of a variable ("(d) many things of kind X can be inside it at many times" and "(e) one thing at one time, another thing at another time"), and one closing component in the case of constant ("(d) at all times this one thing is always the same thing") is where the two concepts unmistakably differ. Variables in Swift are prone to change as long as they conform to the initial type they were declared with ("many things of kind X") and the Apple documentation clearly suggests that stating: "you can change the value of an existing variable to another value of a compatible type" (Apple 2020: 59). Once an application containing a variable with a value inside is run, it can simply accept an infinite number of other values of the same type. This is not the case with respect to constants in Swift, as is evidenced by a changed design of the respective explication ("at all times this one thing is always the same thing") because "a constant is principally an example of an immutable container" (Biegajło 2019a: 15). One final comment has to address the keywords used for introducing variables and constants into a Swift code – they are 'var' and 'let' respectively and, unlike in the case of a 'class' keyword, they were implemented into the very title of the explications, which goes on to show how flexible NSM and Minimal English can be in an attempt to suit personal preferences.

In light of these remarks, we can now try to propose an explication of property in Swift:

property (var/let) of kind X:
a. it can be like this:
b. there can be something inside a class
c. this something can be a property
d. a property is like a variable inside a class
e. if it is not a variable, it is a constant

Having adopted the blueprint used for the explications of the concept of variable and the concept of constant, the explication of the concept of property assumes some prior exposure to the dichotomy between variables and constants. The title of the explication of property suggests two keywords available that allow creating properties that can share the typical characteristics of variables and constants ('var' vs 'let', mutability vs immutability). Additionally, properties in Swift, just as is the case with variables and constants, need to be of a specific type ("of kind X"). Component (a) ("it can be like this:") points to the fact that properties might appear in a class, however, a class does not necessarily have to contain property to keep the characteristics of a class. It can accept other programming constructs typical to Swift classes and allow for significant variability in this respect. Component (b) ("there can be something inside a class") seems self-evident; a class can accept different data containers, one of which is, uniquely, a property (component (c) "this something can be a property"). The closing components (d)–(e) ("a property is like a variable inside a class" and "if it is not a variable, it is a constant") make us assume that properties copy the behaviour of variables and constants, i.e. one the one hand, they can store different values, they are open to accepting other values, as long as that other value is of a type compatible with the substituted value (behaviour characteristic to variables), on the other hand, they accept only one value, of one specific type which cannot be manipulated with after it was declared (behaviour characteristic to constants).

5. Concluding remarks

One of the underlying challenges that Object-Oriented Programming poses is that multiple complex concepts are in constant dialogue, creating a sophisticated arrangement where digital communication can take place. An echo of that is sent out by Apple stating that: "the additional capabilities that classes support come at the cost of increased complexity" (Apple 2020: 347) and, due to the constraints of this paper, only selected and most fundamental issues related to Swift classes are discussed. They include explications of general characteristics of Swift classes, the 'class' keyword, types of classes and how to create them, the syntax of classes, and the concept of property as one of the distinct features of Swift classes. The proposed explications are by no means fixed or final and are open to further amendments; however, they are a perfect starting point to pursue the project of rewriting selected programming documentation in Minimal English with the view of transforming it into more user-friendly explanations, especially with school children in mind. As has been mentioned in the opening paragraphs of this paper, classes share some of their core functionality with another Swift construct, namely structures, and to arrive at a comprehensive account of the usability of classes, structures would also have to be addressed with greater detail in the later stages of this project. To embrace digital literacy at a satisfactory level, this project could potentially develop into a self-contained reference book which would be focused on discussing selected programming concepts in a fashion that would emulate Anna Wierzbicka's retold Biblical Gospels, rewritten in Minimal Polish and Minimal English (Wierzbicka 2017, 2019). The obvious strength behind Wierzbicka's project is that the Gospels are not recounted exclusively in explications, but are a combination of explications and a narrative form that different minimal versions of natural languages can support.

References

Apple. 2020. *The Swift programming language (Swift 5.3)*. Apple Inc.

Biegajło, B. 2019a. Explaining IT programming concepts using NSM explications: The case of 'variable 'and 'constant'. *Linguistics Beyond and Within 5*: 7–16.

Biegajło, B. 2019b. Harnessing the concept of an array in Swift programming language. Abstract concepts vs. Natural Semantic Metalanguage. *Language and Literary Studies of Warsaw 9*: 239–251.

Goddard, C. 2018a. *Minimal English for a global world. Improved communication using fewer words*. Cham: Palgrave Macmillan.

Goddard, C. 2018b. *Ten lectures on Natural Semantic Metalanguage. Exploring language, thought and culture using simple, translatable words*. Leiden: Brill.

Goddard, C. 2018c. Minimal English: The science behind it. In C. Goddard (ed.), *Minimal English for a global world. Improved communication using fewer words,* 29–70. Cham: Palgrave Macmillan.

Goddard, C., and A. Wierzbicka. 2014. *Words and meanings. Lexical semantics across domains, languages, and cultures*. Oxford: Oxford University Press.

Goddard, C., and A. Wierzbicka. 2018. Minimal English and how it can add to global English. In C. Goddard (ed.), *Minimal English for a global world. Improved communication using fewer words,* 5–27. Cham: Palgrave Macmillan.

Kalisz, R. 1998. Is it possible to operate with primitives in every explication?. In B. Lewandowska-Tomaszczyk (ed.), *Lexical semantics, cognition and philosophy*, 55–63. Łódź: Łódź University Press.

Knuth, D. 1992. *Literate programming*. Stanford, California: Center for the Study of Language and Information.

Olson, R. D. 2009. Why literacy matters, then and now. In A. W. Johnston and H. H. Parker (eds.), *Ancient literacies. The culture of reading in Greece and Rome*, 385–403. Oxford: Oxford University Press.

Prottsman, K. 2019. *How to be a coder*. London: Penguin Random House.

Rashid, T. 2019. *Creative coding for kids*. Scotts Valley, California: Create Space Independent Publishing Platform.

Rijpma, A. 2014. A composite view of well-being since 1820. In J. L. van Zanden, J. Baten, and M. M. d'Ercole (eds.), *How was life? Global well-being since 1820*, 249–269. OECD Publishing.

Thomas, R. 2009. Writing, reading, public and private 'literacies'. Functional literacy and democratic literacy in Greece. In A. W. Johnston and H. H. Parker (eds.), *Ancient literacies. The culture of reading in Greece and Rome*, 13–45. Oxford: Oxford University Press.

van Leeuwen, B. and J. van Leeuwen-Li. 2014. Education since 1820. In J. L. van Zanden, J. Baten, and M. M. d'Ercole (eds.), *How Was Life? Global Well-being since 1820*, 87–100. OECD Publishing.

Vorderman, C., J. Woodcock, and S. McManus. 2014. *Help your kids with computer coding. A unique step-by-step visual guide, from binary code to building games*. London: Penguin Random House.

Wierzbicka, A. 1999. *Emotions across languages and cultures: Diversity and universals*. Cambridge: Cambridge University Press.

Wierzbicka, A. 2010. *Experience, evidence, and sense. The hidden cultural legacy of English*. Oxford: Oxford University Press.

Wierzbicka, A. 2014. *Imprisoned in English. The hazards of English as a default language*. Oxford: Oxford University Press.

Wierzbicka, A. 2017. *W co wierzą chrześcijanie? Opowieść o Bogu i o ludziach*. Kraków: Wydawnictwo Znak.

Wierzbicka, A. 2019. *What Christians believe: The story of God and people in Minimal English*. Oxford: Oxford University Press.

Woodcock, J. 2016a. *Coding games in Scratch. A step-by-step visual guide to building your own computer games*. London: Penguin Random House.

Woodcock, J. 2016b. *Coding projects in Scratch. A step-by-step visual guide to coding your own animations, games, simulations, and more!* London: Penguin Random House.