

**Deanship of Graduate Studies
Al-Quds University**



Syntactic Sugar Programming Languages' Constructs

Mustafa Sudqi Abed Al-Tamim

M.Sc. Thesis

Jerusalem – Palestine

1432 / 2011

**Deanship of Graduate Studies
Al-Quds University**



Syntactic Sugar Programming Languages' Constructs

*Prepared By:
Mustafa Sudqi Abed Al-Tamim*

B.SC from Bir-Zeit University, Palestine

*Supervisor:
Dr. Rashid Jayousi*

A thesis Submitted in Partial fulfillment of requirements for the Master degree of Computer Science from Computer Science department of Al-Quds University

Jerusalem – Palestine

1432 / 2011

**Al-Quds University
Deanship of Graduate Studies
Computer Science Department**



Thesis Approval

Syntactic Sugar Programming Languages' Constructs

***Prepared By: Mustafa Sudqi Abed Al-Tamim
Registration No: 20714027***

Supervisor: Dr. Rashid Jayousi

Master thesis submitted and accepted. Date: / / 2011

The names and signatures of the examining committee members are as follows:

1- Head of Committee: Dr. Rashid Jayousi	Signature:
2- Internal Examiner: Dr. Raid Al-Zaghal	Signature:
3- External Examiner: Dr. Mahmoud Saheb	Signature:

Jerusalem – Palestine

1432 / 2011

Dedication

*To my parents, who guided me to success with their wisdom and guidance,
To my wife, the woman who supports and stands beside me,
To my brothers and sisters,
To my colleagues and friends at Al-Quds University*

Mustafa Al-Tamim

Declaration

I certify that this thesis submitted for the degree of Master is the result of my own research, except where otherwise acknowledged, and that this thesis (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed

Mustafa Sudqi Abed Al-Tamim

Date: / /2011

Acknowledgement

First of all, Praises and thanks always to Allah, The creator, who gave me the power to complete this work.

Then, I'm heartily thankful for my supervisor Dr. Rashid Jayousi, without his guidance, support, and encouragement, this thesis will not have been possible.

Also, I owe my deepest gratitude to all professors and teachers at Al-Quds University/Computer Science department for their support. And I want to thank Dr. Raid Zaghal, Mr. Saeed Salah, and the students for their help in executing the experiment.

I deeply express my sincere thanks to Professor Younis Amro, president of Al-Quds Open University, who encouraged and helped me to continue my study.

It's my pleasure to thank Dr. Elias Dabit for his help in the questionnaire design.

And I will not forget my employer "gSoft company" for their cooperation.

Lastly, I express my deepest thanks and blessing for my father, mother, brothers and sister. Special thanks for my brother Riyad Al-Tamim for his help. And I will not forget my wife and want to thank her for the inspiration and patience.

Mustafa Al-Tamim

Abstract

Software application development is a daily task done by developers and code writers all over the world. A valuable portion of developers' time is spent in writing repetitive keywords, debugging code, trying to understand its semantic, and fixing syntax errors. These tasks become harder when no integrated development environment (IDE) is available or developers use remote access terminals like UNIX and simple text editors for code writing. Syntactic sugar constructs in programming languages are found to offer simple and easy syntax constructs to make developers' lives easier and smoother.

In this study we propose a new set of syntactic sugar constructs, and try to find if they really can help developers in reducing syntax errors, make code shorter, more readable, easier to write, and can help in debugging and semantic understanding.

Our methodology was to construct a new syntactic sugar constructs set extracted from existing programming languages' syntax in addition to other syntactic enhancements proposed by us, then we verified the efficiency of the new syntactic sugar constructs set through executing an exploratory case study with students and professional programmers.

The exploratory case study results showed positive indicators for using the new proposed syntactic sugar constructs set to write programs' syntax. They helped in reducing syntax errors, making the code more readable, easier to write, and to understand.

المحسنات النحوية للجمل البنائية في لغات البرمجة

الملخص

تطوير البرمجيات التطبيقية هي مهمة يومية يقوم بها المطورون والمبرمجون في كافة انحاء العالم، ويهدر جزء لا بأس به من وقت المبرمجين في كتابة كلمات مفتاحية بشكل متكرر في الجمل التركيبية للبرامج وتصحيح الأخطاء في بناء الجمل التركيبية، ومحاولة فهم دلالات البرامج. هذه المهام تصبح أكثر صعوبة إذا لم تكن هناك بيئة تطوير متكاملة متاحة للإستخدام، أو عندما يقوم المطورون بكتابة البرامج بإستخدام محررات نصوص بسيطة، وكذلك في حالة تطوير البرامج عن بعد بإستخدام برمجيات الاتصال الطرفي كما في نظام التشغيل يونيكس مثلاً.

لقد أوجدت محسنات بناء الجمل التركيبية في لغات البرمجة لتقدم تركيبات نصية بسيطة وسهلة وجعل حياة المطورين أسهل. بناءً على ذلك، نقترح في هذه الدراسة مجموعة جديدة من محسنات بناء الجمل التركيبية، ونحاول معرفة ما إذا كانت هذه المحسنات تساهم في التقليل من الأخطاء النصية وجعل تركيب الجمل في لغات البرمجة أبسط وأكثر وضوحاً وأسهل للقراءة والكتابة والتتبع وفهم دلالات البرامج.

منهجية البحث المتبعة في هذه الدراسة تقوم على إيجاد مجموعة من محسنات بناء الجمل التركيبية والمستخرجة من بعض لغات البرمجة المستخدمة ، إضافة الى عدد من التحسينات المقترحة، ومن ثم محاولة التحقق من فعالية هذه المحسنات من خلال إجراء دراسة حالة استكشافية مع عدد من الدارسين والمطورين ذوي الخبرة.

ولقد اظهرت نتائج الدراسة مؤشرات ايجابية واضحة حول استخدام محسنات بناء الجمل التركيبية في كتابة البرمجيات، ولقد ساعدت هذه المحسنات في الحد من الأخطاء النصية وجعل تركيب الجمل في لغات البرمجة أكثر وضوحاً وأسهل للقراءة والكتابة والتتبع وفهم الدلالات.

Table of Contents

DEDICATION	
DECLARATION	I
ACKNOWLEDGEMENT	II
ABSTRACT	III
TABLE OF CONTENTS	V
LIST OF TABLES	VI
LIST OF FIGURES	VII
LIST OF APPENDICES	VIII
CHAPTER ONE	1
INTRODUCTION	1
1.1 MOTIVATION	3
1.2 ORGANIZATION	4
CHAPTER TWO	5
LITERATURE REVIEW	5
2.1 INTRODUCTION	5
2.2 HISTORY OF PROGRAMMING LANGUAGES	5
2.3 SYNTACTIC SUGARS CONSTRUCTS REVIEW	16
2.4 SYNTAX ERRORS REDUCTION	29
2.5 CONTRIBUTION	31
CHAPTER THREE	33
CONSTRUCTS SELECTION AND ENHANCEMENTS	33
3.1 INTRODUCTION	33
3.2 CONSTRUCTS SELECTION METHODOLOGY	33
3.3 ABSTRACT CONSTRUCTS SELECTION	34
3.4 PROGRAMMING LANGUAGES STUDY AND CONSTRUCTS EXTRACTION	37
3.5 QUESTIONNAIRE DESIGN, DISTRIBUTION, COLLECTION, AND RESULTS	38
3.6 SYNTACTIC SUGAR CONSTRUCTS SET AND ENHANCEMENTS SELECTION	43
CHAPTER FOUR	46
EXPLORATORY CASE STUDY	46
4.1 INTRODUCTION	46
4.2 EXPLORATORY CASE STUDY EXPERIMENT	46
4.3 CASE STUDY DESIGN	47
4.4 NEW SYNTAX PARSER AND IDE	48
4.5 STUDENTS' CASE STUDY TRACK	50
4.6 PROFESSIONALS' CASE STUDY TRACK	52
4.7 DIFFICULTIES FACED DURING CASE STUDY EXECUTION	53
CHAPTER FIVE	55
EXPLORATORY CASE STUDY RESULTS	55
5.1 INTRODUCTION	55
5.2 STUDENTS' INTERVIEW RESULTS	55
5.3 STUDENTS PRACTICAL CASE STUDY RESULTS (PROGRAM WRITING)	57
5.4 PROFESSIONALS' SEMANTIC EXTRACTION CASE STUDY RESULTS	60
5.5 PROFESSIONALS' INTERVIEW RESULTS	61
5.6 OBSERVATIONS AND NOTES	63
CHAPTER SIX	65
CONCLUSION AND FUTURE WORK	65
6.1 FUTURE WORK	66
REFERENCES	67
APPENDICES	73
APPENDIX 1: PROGRAMMING LANGUAGES' EXTRACTED CONSTRUCTS SHEET	73
APPENDIX 2: THE DESIGNED AND DISTRIBUTED QUESTIONNAIRE	80
APPENDIX 3: SELECTED PROGRAMMING LANGUAGES' BRIEF DESCRIPTION	89
APPENDIX 4: ICT PROFESSIONAL POPULATION CALCULATION	91
APPENDIX 5: USERS' FEEDBACK INTERVIEW QUESTIONS	92
APPENDIX 6: STUDENTS' CASE STUDY EXPERIMENT PROGRAMS	93
APPENDIX 7: PROFESSIONALS' CASE STUDY EXPERIMENT PROGRAMS	94
APPENDIX 8: SELECTED AND ENHANCED CONSTRUCTS SET DETAILED DESCRIPTION	100

List of Tables

Table. No.	Table Name	Page
Table 2.1:	Summary of syntax error reduction strategies	31
Table 3.2-A:	The selected and enhanced syntactic sugar constructs – part 1	44
Table 3.2-B:	The selected and enhanced syntactic sugar constructs – part 2	45
Table 5.3:	Students answers on interview questions	55
Table 5.4:	Semantic extraction results	61
Table 5.5:	Professionals answers on interview questions	62

List of Figures

Fig. No.	Figure Name	Page
Figure 2.1:	Important programming languages summary	7
Figure 4.2:	New syntax IDE and parser tool	50
Figure 5.3:	Students interview answers distribution	56
Figure 5.4:	% of errors generated by using old and new constructs in each program ..	57
Figure 5.5:	% of old and new constructs used in each program	58
Figure 5.6:	Professionals interview answers distribution	62

List of Appendices

Appendix No.	Appendix Name	Page
Appendix 1:	Programming Languages' Extracted Constructs Sheet	73
Appendix 2:	The Designed And Distributed Questionnaire	80
Appendix 3:	Selected Programming Languages' Brief Description	89
Appendix 4:	ICT Professional Population Calculation	91
Appendix 5:	Users' Feedback Interview Questions	92
Appendix 6:	Students' Case Study Experiment Programs	93
Appendix 7:	Professionals' Case Study Experiment Programs	94
Appendix 8:	Selected And Enhanced Constructs Set Detailed Description	100

Chapter One

Introduction

Developing and writing software applications is a common daily activity done by thousands or even hundreds of thousands of developers and programmers as the demand on software applications is increasing to meet the technical revolution needs, which is involved in most trends in life.

Enterprise software applications development using programming languages (PL) requires a lot of code writing. Such applications have a lot of functionality and business logic for developers to focus on: *functions, actions, use cases, and data processing that form the core of the application which offers the functionality to the users through application graphical interface or APIs (Application Programming Interfaces)* (Mitchell, 2002). A valuable portion of developers' time is spent writing repetitive keywords and determining classes, methods, and code building blocks' scopes that can be ambiguous for them to follow up on and debug, which also may generate many of syntax errors that require extra effort to find and fix. Source code reading and semantic extraction by developers is not an easy task, especially when the code is huge and moved from one team to another, or is bought from 3rd party providers and the developers want to continue working on and customizing it.

Students who learn programming languages in universities and schools face similar issues in code ambiguity and syntax errors (Russell et al., 2009). Their issues with code sometimes cost them hours to fix certain syntax errors or to find logical ones because their experience is not mature enough to help them in code debugging and memorizing syntax keywords and complex structures. This enforces students to spend a portion of their time on syntax issues which can be saved and used to focus on application

functionality and logic as mentioned in (Kummerfeld and Kay, 2002): "Our own experience and observations of students indicates that students using an unfamiliar or new programming language waste considerable time correcting syntax errors."

Syntactic sugar enhancements on programming languages' syntax constructs is one of the approaches used to enhance syntax and help in making it more readable, easier to write, with less ambiguity. In this research, we study many programming languages through analyzing their syntax; studying it, and then offering a new suggested syntactic sugar constructs set where applicable. The set is composed of a mix from existing constructs obtained from the analyzed programming languages with a set of syntactic enhancement suggested by us. The purpose of the new syntactic sugar constructs set is to make programming language users' work easier, smoother, and less ambiguous, with fewer errors and more focus on application core functionality and less focus on code syntax and syntactic errors.

The questions we try to answer in this research: Do syntactic sugar constructs help in development with fewer syntax errors? Do they help with semantic extraction? Can they make code more readable and easier to write?

The research focuses on programming languages' syntax and how to enhance it by adding new syntactic sugar constructs for easier coding and compiling these enhancements to form a set of recommendations for programming language designers to make use of them while designing programming languages' syntax.

Research results showed positive indicators of using syntactic sugar in writing application source code.

1.1. Motivation

Through our work as software developers, team leaders, and guiding many students in their projects, we noticed that developers usually write a lot of repetitive keywords in specific parts of code like packages calling keywords, code segments and building blocks' scopes determination symbols, and many others, for example, the use of the curly braces “{ }” symbols to determine classes, methods, expressions, and control statement (IF, FOR, WHILE...etc.) scopes in the same program building block. Using the same symbols can make it difficult to distinguish the method scope from its internal control statement scopes, especially in the case of missed opening or closing symbol.

These kinds of repeated keywords and ambiguous scopes consume part of developers' efforts and time especially when using a text mode development environment. The most important is that it can cause many syntax errors and make it hard to debug and understand the semantic.

This motivated us to search for syntactic constructs that help enhance programming language syntax to use fewer repetitive keywords, better scope determination symbols, better exception handling, more readable code with less writing efforts, and many other properties that make developers' work easier with more focus on business logic implementation.

Many researches were done and tools created to help generate source code automatically such as macros, annotations, IDE (Integrated Development Environments), and reverse engineering. These tools help save syntax writing efforts and minimize syntactic errors with hints to solve them. These tools eliminate part of the problem, but it is not useful in development environments that are dependent on text mode, where no visual user interface is available for the developers and they cannot use

IDEs and tools, as the case of the Linux or Unix command-line remote terminals or SSH command-line tools used for remote access.

We are targeting both novice and professional programming language users with focus on users who use remote or simple text mode editors, where no advanced IDE and code wizards are available.

1.2. Organization

Our methodology of finding a new syntactic sugar constructs set is explained in details throughout this thesis with all theoretical and technical details. An introduction to the subject and motivation with problem statement was introduced above; the rest of this thesis is organized as follows:

Chapter two is a literature review for the history of programming languages, and a review for syntactic sugars and their implementations. Chapter three discusses and explains our methodology in details, and the work done to obtain the syntactic sugar constructs set. Details of the exploratory case study done to validate the new constructs set are explained in chapter four. Chapter five presents and discusses the results obtained from executing the case study. Finally we conclude and discuss the future work in chapter six.

Chapter Two

Literature Review

2.1 Introduction

This chapter presents a review of the research areas related to the work done in this thesis.

We outline a historical review for the major and most popular programming languages development. In section two we present the done work related to syntactic sugars and form them in simple survey. Section three through light on work done to reduce syntax errors.

2.2 History of Programming Languages

Programming language is a tool that includes a set of instructions and commands expressed through a well defined syntax that are programmers familiar with and used to form programs that can be executed by the computer in logical way producing efficient work proposed by the written program semantic. In other words, it is a medium of expression in the world of computer programming (Mitchell, 2002) (Collberg, 2005).

Each programming language has a syntax that is the form of the program and how it is written by programmer and parsed by the computer. The syntax is composed of declarations, expressions, commands, and constructs that are used to compose the program (Watt and Findlay, 2004) (Collberg, 2005).

The semantic in programming languages is the meaning of the program and the desired functionality of it when it is executed. Semantic is used to determine the programmer's desired functionality and how it is understood by the computer at execution time (Watt and Findlay, 2004) (Collberg, 2005).

Programming language paradigm expresses how the languages is designed to be used and in which domains. This also affects how programmers design their programs to solve certain problem. Each programming language can support one or more paradigms.

The most well known paradigms are: 1) Functional paradigm which depends on functions and their calls as main building construct. 2) Imperative paradigm which uses procedures, commands and variables. 3) Concurrent paradigm which supports the concurrent execution of commands and processes. 4) Logical paradigm which depends on facts and relations. 5) Object oriented paradigm where the object and class concepts are the core items in this paradigm, in addition to relations such as inheritance, composition, and aggregation. 6) Scripting paradigm, programming in this paradigm is simple, complete program is not needed and it uses high level commands, scripts can be executed and interpreted in simple and primitive environment like UNIX shell, DOS, or internet browsers (Watt and Findlay, 2004).

In this research, we focus on high level programming languages summarized in Figure 2.2. A high level programming language is a language that its programs are executed independently from machine. High level programming languages use compilers to convert programs to machine language or use interpreters (Watt and Findlay, 2004).

In the mean time, we have hundreds of high level programming languages that were developed over time since the first high level programming language was developed (Mitchell, 2002). In this chapter we review the most common programming languages over the last 60 years.

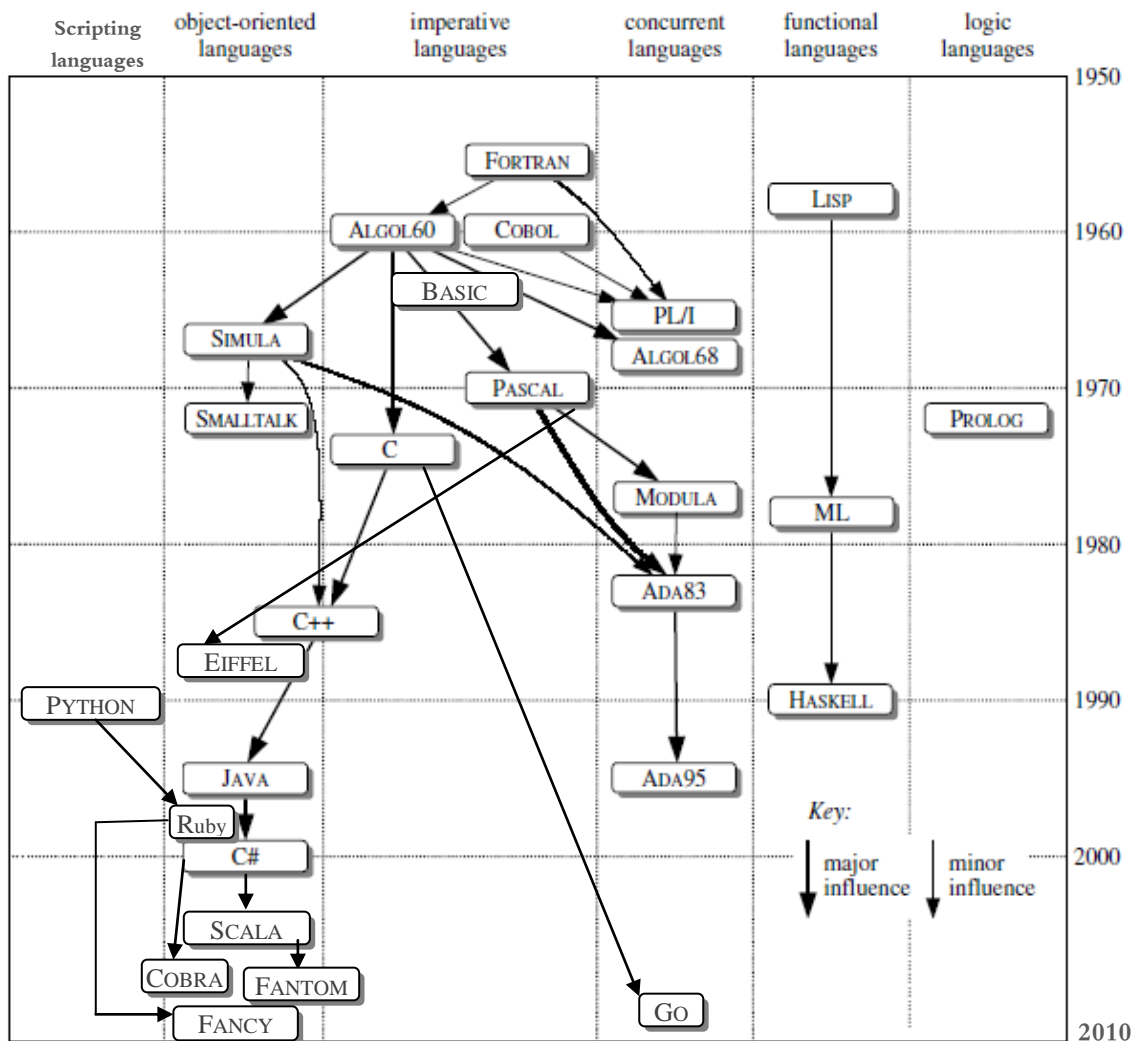


Figure 2.3: Important programming languages summary, adapted and updated from Watt and Findlay, 2004

The first popular high level programming language was FORTRAN. FORTRAN was developed at IBM around 1957 by a John Backus team and help from Peter Sheridan (Mitchell, 2002) (Sammet, 1996). FORTRAN was the first language used arrays, ordinal mathematical notation expressions, procedures, symbolic names for variables, and formatted inputs and outputs (Mitchell, 2002) (Watt and Findlay, 2004). It was good for mathematical and economical calculations (Nerlove, 2004) (Watt and Findlay, 2004). FORTRAN contains many limitations such as no recursion support, numbers storing in the memory was weak, and programmer may change a value by mistake if he was not careful (Mitchell, 2002). FORTRAN considered as imperative language. It was

developed over time and many new versions were delivered, for instance FORTRAN77 at 1977, FORTRAN for Microsoft Dos operating system at 1982, and FORTRAN90 at 1990 which support object oriented paradigm (Nerlove, 2004).

In 1960, COBOL was founded by Grace Murray Hopper (Mitchell, 2002). It was used for business applications and commercial data processing (Mitchell, 2002) (Watt and Findlay, 2004). The designers tried to make its syntax English like (Mitchell, 2002). COBOL introduced data description concept that was used to build data types in successor languages (Watt and Findlay, 2004) and used the record data structure. COBOL considered to be imperative programming language, it suffered from low level flow control (Mitchell, 2002) (Watt and Findlay, 2004), and it did not support local variables, recursion, and dynamic memory allocation (COBOL, n.d.). COBOL gained many enhancements and improvements over time, the last version was COBOL2002 at 2002 which added support for object oriented features, user' defined functions, pointers, Boolean support, floating point support, XML manipulation, and many others (COBOL, n.d.).

The first functional programming language was Lisp; it was developed by the end of 1950s at MIT for artificial intelligence and symbolic computations (Mitchell, 2002). Lisp is considered as simple and flexible language for expressing logical expressions. Its main data structure is lists and it support recursive calls. Lisp continued developing over time and used widely: in 1960 Maclisp was developed at MAC MIT project, another version was released in 1970s by Guy Steele and Gerald Sussman adding new features to Lisp. The current Lisp that called common Lisp is a new version that offers complex object oriented primitives (Mitchell, 2002).

The general purpose imperative programming language ALGOL60 was designed by Alan Perlis, John Backus, and John McCarthy in the period 1958 to 1963 (Mitchell, 2002). ALGOL60 supported functions, recursion, block structure that allow declaring procedures and variables anywhere in the programs, and it offered better data structures representations (Mitchell, 2002) (Watt and Findlay, 2004). A new version of ALGOL was developed at 1968 called ALGOL68; this version supported declaring arrays to be of type integer, array, or procedure. Procedures in ALGOL68 can accept parameters and return values of type integer, array, or other procedures (Watt and Findlay, 2004). Many programming languages were developed on top of ALGOL, ALGOL formed programming languages family where all successors languages called ALGOL-LIKE languages (i.e. Pascal, C, ML) (Mitchell, 2002) (Watt and Findlay, 2004).

The simple imperative programming language BASIC was design in 1963 by John G. Kemeny and Thomas E. Kurtz at Dartmouth College (Nerlove, 2004) to offer simple, easy, and quick programming language for students. BASIC supported sound playing and graphics (Nerlove, 2004). Apple II was sold with BASIC support in 1977. BASIC was used in IBM PC DOS operating system in 1981-1982 and called at that time Quick BASIC. It was extended and used by Microsoft through Visual Basic (Sureau, 2010).

In 1964, George Radin at IBM, tried to combine ALGOL60, FORTRAN, and COBOL best features in one general purpose programming language called PL/1 (Nerlove, 2004). PL/1 considered being both imperative and concurrent programming languages. It introduced new concepts in programming like concurrency and exceptions low level forms. Because of the combination of many language features and paradigms, PL/1 was complex, difficult to implement, and huge. It didn't success on the long term (Watt and Findlay, 2004).

The first object oriented programming languages was SIMULA67 that design by O.-J.Dahl and K. Nygaard in 1967 (Mitchell, 2002). SIMULA67 introduced the concepts of o classes, inheritance, objects, dynamic lookup, and sub typing, SIMULA67 didn't supported encapsulation and abstraction (Mitchell, 2002) (Watt and Findlay, 2004). Many object oriented languages were developed depending on SIMULA67 and used its object oriented conceptual inspiration like C++ and SMALTALK (Mitchell, 2002) (Watt and Findlay, 2004).

PASCAL, the ALGOL-like imperative programming language, was developed around 1970 as successor for ALGOL60 (Watt and Findlay, 2004). PASCAL was simple and efficiently implemented and used in system development and programming languages curricula classes to teach programming languages (Nerlove, 2004). PASCAL offered rich simple set of control structures, data types (i.e. arrays, recursive types, records, Booleans, characters, enumerations, files) pointers, and procedures (Watt and Findlay, 2004).

SMALLTALK was developed as the first pure object oriented programming language at Xerox PARC in the 1970s (Mitchell, 2002). Everything in SMALLTALK was considered as object (values, control structures such as IF statement, commands) (Watt and Findlay, 2004). SMALLTALK was successor from SIMULA67, but it added many new features in object oriented such as message passing to objects, abstraction, access modifiers (public methods, and private instance variables) (Mitchell, 2002).

The first well-known programming language that design based on logic paradigm was PROLOG at 1973 by Philippe Roussel (Watt and Findlay, 2004) (Nerlove, 2004) (Mitchell, 2002). PROLOG at its beginning was weak and inefficient until it was

supported with new extra logical features and it still used in logic programming (Watt and Findlay, 2004).

Between 1972 and 1974, the evolutionary imperative programming language C was developed by Dennis Ritchie at AT&T Bell Labs (Nerlove, 2004) (Mitchell, 2002). C was created to be used in writing UNIX operating system (Watt and Findlay, 2004). In 1980, C became famous programming language because of its efficiency, simplicity, and flexibility, and it was used in developing software applications rather than UNIX operating system (Nerlove, 2004). C is ALGOL-like language and support blocks, recursive functions, and local variables declaration. But it is more restricted as it doesn't allow declaration of functions within nested building block; they must be outside the main program (Mitchell, 2002).

MODULA was developed lately 1970s by Niklaus Wirth as successor for PASCAL (Mitchell, 2002). MODULA considered as concurrent programming language (Watt and Findlay, 2004). The main feature MODULA offered over PASCAL was the module system which used to group related declaration sets in programming units (Mitchell, 2002).

The Meta language ML was designed by Robin Milner as part of developing a *Logic for Computable Functions* LCF project by the end of 1970s (Mitchell, 2002). ML mainly is a functional programming language but it support the imperative paradigm too as its syntax is ALGOL like (Mitchell, 2002) (Watt and Findlay, 2004). It allows inline functions creation within expressions, pass them to other functions as parameters and return them as results (Mitchell, 2002).

Another well know PASCAL successor was ADA, it was developed at early 1980s as an initiative by U.S. Department of Defense (DoD) to standardize their software around one language that support specific features such as real-time programming and usage of concurrent programming paradigm (Mitchell, 2002) (Watt and Findlay, 2004). ADA introduced packages, generic units, high level exceptions, and offered wide variety of data types (Booleans, characters, enumerands, integers, real numbers, records, arrays, discriminated records, objects (tagged records), strings, pointers to data, and pointers to procedures.) (Watt and Findlay, 2004). ADA was known by its reliability, robustness, and efficiency; it used in the development of critical systems such as aerospace (Watt and Findlay, 2004). Because of its high compiler cost, it was little used in universities, research, and civilian software market (Mitchell, 2002). ADA continued in development and the latest version was ADA95 in 1995 (Watt and Findlay, 2004).

An extension to C was implemented around 1984 by Bjarne Stroustrup at A Bell Laboratories to offer object oriented support in it, the extension formed new programming language called C++ and it inherit most of C language features and shortcomings (Mitchell, 2002) (Nerlove, 2004). C++ considered as imperative and object oriented language (Watt and Findlay, 2004). C++ became very popular and widely used language in many platforms applications such as UNIX, Apple MAC, and Microsoft Windows (Mitchell, 2002).

Eiffel is an object-oriented programming language designed by Bertrand Meyer in 1985 (Sureau, 2010) to produce robust software. Its syntax is keyword-oriented like ALGOL and Pascal tradition and it is strongly statically typed with automatic memory management (typically implemented by garbage collection) (Meyer, 2001). Eiffel

support programming by contract (usage of pre and post conditions in functions) (Sureau, 2010).

The pure functional programming language HASKELL was design around early 1990 by a large committee led by Simon Peyton Jones and John Hughes (Watt and Findlay, 2004). HASKELL was affected by ML. And it allows passing functions to other functions as parameters and returns them as results. It uses polymorphic functions, support algebraic types and very close to the mathematical disjoint-union notation (Watt and Findlay, 2004).

PYTHON is well designed and famous scripting languages design in 1991 by Guido Van Rossum (Sureau, 2010) (Watt and Findlay, 2004). PYTHON can support C libraries and object oriented feature. It can run within Java JVM (Sureau, 2010). Python was known as dynamic programming language with very clear readable syntax, intuitive object orientation, natural expression of procedural code, full modularity, supporting hierarchical packages, exception-based error handling, very high level dynamic data types, and extensive standard libraries and third party modules. The extensions and modules are easily written in C, C++ (or Java for Jython, or .NET languages for IronPython), and embeddable within applications as a scripting interface (About Python, n.d.).

Java (its first name was Oak) is a well known object oriented programming language developed at Sun Microsystems by James Gosling and others between 1990 and 1995 (Mitchell, 2002). Java is also suitable for concurrent programming and distributed environment. Its power appears in web development and writing applets which are small programs run within web pages (Watt and Findlay, 2004). Java came to simplify C++ and solve a number of problems in modern programming practices (Watt and

Findlay, 2004) (The Java Language, n.d.). Java Virtual Machine helped it to be platform independent and Java can run on any operating systems, hardware, and even small, portable, and imbedded devices (Watt and Findlay, 2004) (Mitchell, 2002). Java offered many new features to object oriented such as interfaces, abstract classes, and run time class loading, and it focused on security, efficacy, simplicity, high-performance, multithreading, and portability in its design (Mitchell, 2002) (The Java Language, n.d.).

The object oriented language C# was developed at 2000 by Microsoft systems; it is close to Java with minor changes that made it suitable for desktop applications (Watt and Findlay, 2004). C# is a type-safe, object-oriented language that is simple yet powerful, allowing programmers to build a breadth of applications. Combined with the .NET Framework (Getting Started with Visual C#, n.d.), a new version called Visual C# 2008 produced that enabled the creation of Windows applications, Web services, database tools, components, controls, and more (Getting Started with Visual C#, n.d.).

RUBY is a modern object oriented programming languages design as successor of PYTHON and PERL (Sureau, 2010) to offer clearer and more object oriented support (Sureau, 2010). RUBY design started in late 1990s by Yukihiro Matsumoto (Sureau, 2010). Yukihiro Matsumoto blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balance functional programming with imperative and object oriented programming (About Ruby, n.d.).

The New modern programming language SCALA was developed in the programming methods laboratory at EPFL and released in 2004 (Odersky et al., 2006). SCALA considered as general purpose programming language that support the common programming patterns in type safe and elegant way (The Scala Programming Language, 2008). SCALA is JAVA-like languages. The code size produced by SCALA is

relatively short but hard to read as it uses a lot of symbols and inline functions declaration (Odersky et al., 2006). SCALA integrate the features of object oriented and functional paradigm (The Scala Programming Language, 2008).

In the literature, many programming languages were developed in the period between 2004 and 2010. These languages were not widely used and known. Most of them were extensions on top of exist languages like Java and Python to form frameworks for special needs, or create new high level general purpose programming languages to simplify the previous programming languages and offer new features (Timeline of programming languages, 2011).

In 2006, the object oriented programming languages COBRA was developed by Chuck Esterbrook to collect many features from many exist programming languages (PYTHON, C#, EIFFEL) and offer them in one language to support development by contract, static and dynamic binding, quality control, runtime performance and quick coding (Why Cobra, 2010).

The general purpose programming language FANTOM was developed in 2007 by Brian Frank and Andy Frank (Why Fantom, 2011). FANTOM is influenced by C#, JAVA, SCALA. It supports functional, concurrent, declarative, and object oriented programming. FANTOM offers static / dynamic binding, elegant programming APIs, and runtime portability on both Java and .NET platforms (Why Fantom, 2011).

GO is concurrent programming languages that developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2009 (The Go Programming Language, 2011). GO main purpose is to develop concurrent programs that make use of multi-core machines. GO offers runtime reflection and garbage collection in addition to a fast static

typing that affect program execution performance (The Go Programming Language, 2011).

In 2010, the work started on designing and developing a new pure object oriented programming language called FANCY by Christopher Bertels (About Fancy, 2011). FANCY is still under development and influenced by RUBY, SMALTALK, and others. Its current versions works under LINUX and MAC-OS, it supports dynamic typing and garbage collection (About Fancy, 2011).

2.3 Syntactic Sugars Constructs Review

During our work in this thesis, we conducted a review for syntactic sugar construct previous work, this showed that no dedicated research area or surveys available in this field. We found out that the work done in this area was discrete efforts related to enhance certain languages syntax, or mentioning syntactic sugars as minor part of work done in other researches. In this review, we tried to collect the available related work done in syntactic sugar constructs and put them in a form of survey.

The term "Syntactic Sugar" was found by the British computer scientist Peter J. Landin (Mageed, 2010), this term describes making programming languages syntax user friendly and offer alternative syntactic expressions to language common constructs to be sweeter and written in simpler way without affecting the semantic (Mageed, 2010) (Golbreich and Wallace, 2008).

In 1985 Andrew Koenig proposed a new language called "Snocone" as an extension to the "SNOBOL4" text processing and pattern matching language (Koenig, 1985). Andrew introduced the new language by adding syntactic sugars to SNOBOL4 in order to make it easier to implement as SNOBOL4 control structure is old and complex, in

addition to the usage of blank character as operator which caused many syntactic troubles to programmers.

TEX language is unfamiliar formatter programming language that has complex and low level encoding construct syntax that make it difficult for usage (Laan, 1992), In 1992, the authors of TEX provided syntactic sugars to make the syntactic constructs about the loop, switch, array addressing, and keyword parameters closer to high level programming languages constructs such as Pascal in order to be easier for users.

In 1996, Roberto Ierusalimschy and his team introduced "Lua", the extendable language that offers ability to build extending application (Ierusalimschy et al., 1996). Extending applications are application that can be reconfigured and extended in production time. Lua offered the syntax and application program interface (API) needed for configuration. Lua provided set of syntactic sugars constructs for users to make code writing simpler. Examples (Ierusalimschy et al., 1996):

Method definitions using syntactic sugars:

```
function object:method (params)
    ...
End
```

This is equivalent to the un-sugared syntax:

```
function dummy_name (self, params)
    ...
end
object.method = dummy_name
```

Method call sugared constructs written as

```
receiver:method(params)
```

This is equivalent to the un-sugared syntax:

```
receiver.method(receiver,params)
```

In (Chiba, 1996), Shigeru Chiba introduced *OpenC++* which is C++ extension offers meta-level program features interpreted by compiler at compile time. *OpenC++* provided syntax sugar for matrix manipulation library to define matrix as an array with initialized values which was not possible in regular C++ (Chiba, 1996). A new kind of loop statement using "*forall*" notation to loop over all matrix entries in shorthand way was introduced too. Examples (Chiba, 1996):

To initialize array with double values, C++ use the following construct:

```
double tmp[] = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };  
Matrix r = tmp;
```

In *OpenC++* syntactic sugar construct, it can be:

```
Matrix r = { 0.5, -0.86, 0, 0.86, 0.5, 0, 0, 0, 1 };
```

To loop over matrix and initialize its entries, *OpenC++* use:

```
r.forall(e){ e = 0.0; }
```

While in C++ it must be:

```
for(int i = 0; i < N; ++i){  
double& e = r.element[i];  
e = 0.0;  
}
```

Referring to the examples above, the matrix definition and initialization constructs in *OpenC++* are shorter than and C++.

RhoStratego is a programming language developed by Eelco Dolstra in 2001 as part of his thesis work (Dolstra, 2001). *RhoStratego* is a language used to implement program transformation. This language used syntactic sugars to code un-ambiguity as described in (Dolstra, 2001) by replacing parentheses with angle brackets. Also, it provided syntactic sugar for congruencies, lists, and tuples. Examples on *RhoStratego* syntactic sugars (Dolstra, 2001):

Lists and tuples can be defined simply using the following syntactic sugar constructs:

```
numbers = [1, 2, 3];
stuff = <42, "Foo", C 23>;
```

The unary tuples unambiguous notation was solved by using angle brackets instead of parenthesis (Dolstra, 2001). In addition, *RhoStratego* offered syntactic sugar constructs for complex expressions and patterns to be written in simpler and shorter way, for example:

The following construct

```
foo = C 123 -> "foo";
```

Is syntactic sugar form for this complex one:

```
foo = Decomp(x, y) -> (C -> 123 -> "foo") x y;
```

We can notice how sugars helped making the code shorter, simpler, and clearer.

FC++ is an extension library added to C++ to support functional programming (McNamara and Smaragdakis, 2003). In 2003, Brian and Yannis added new features to FC++ related to support lambda sublanguage. The authors used syntactic sugar in FCC++ to simplify functional notation and lambda calling constructs, for example (McNamara and Smaragdakis, 2003):

The de-sugared version of code:

```
[1,2,3] 'bind' (\x ->
  [2,3] 'bind' (\y ->
    if not (x<y) then zero
    else unit (x+y) ))
```

Can be replaced by the following syntactic sugar:

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
-- results in [3,4,5]
```

Usage of syntactic sugar in FC++ made the constructs shorter and simpler for users to write and read.

In Scott Lystig Fritchie study (at 2003) of the performance of "in memory databases" implemented using ETS (Fritchie, 2003), and by comparing the results with 4 different data structures, he explained that a character representation was used as syntactic sugars to replace the ASCII characters representation to be more readable (Fritchie, 2003). For example, word "scott" is syntactic sugar representation for the ASCII characters list [115,99,111,116,116].

In 2004, Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes in their paper Freeman et al., 2004, investigated using mock object in test driven development (TDD). Test driver development is a methodology where programmers define tests for their code functionality before implementing it, and start development until the test pass (Freeman et al., 2004). In their research, they used an open source framework called jMock (Freeman et al., 2004) that provides API for creating mock objects and specify how to invoke tests and to verify results with pass or fail criteria. jMock API itself considered as syntactic sugar implementation that can be used to create testing APIs and make test suite construction simpler. In this paper, the syntactic sugar concept was applied on framework API level and not only on simple syntactic constructs.

Christian Kirkegaard with his partners in (Kirkegaard et al., 2004) made a trial to make XML document manipulation easier, high level, and faster based on XPATH, they introduced new extension to Java called XACT (Kirkegaard et al., 2004). XACT was equipped with many syntactic sugar constructs that made XML manipulation easier for people who use XACT in order to use simple functions instead of writing complex constructs to achieve the same operations (Kirkegaard et al., 2004). The following

examples show some of syntactic sugars in XACT where the left operand is the syntactic sugar form and the right one is the de-sugared code needed to perform the same operation:

```
smash(xs) = xs.length>0 ? group(xs,.[false()])[0] : [[]]
x.roots() = x.select(*)
x.text() = smash(x.select(text())).toString()
x.attribute(a) = smash(x.select(@a)).toString()
x.has(p) = x.select(p).length>0
x.size() = x.roots().length
x.delete(p) = x.gapify(p,g)
x.apply(p,f) = x.gapify(p,g).plug(g,[]f(x.select(p)))
```

The "XQuery" language is used to query XML documents (Hidders et al., 2005). This language is powerful and its popularity was growing up. It suffered from complex syntax that made it hard to be used in research and education (Hidders et al., 2005). In 2005, the authors of (Hidders et al., 2005) created a new sub language based on "XQuery" called "LiXQuery". The new language offered simpler syntax using syntactic sugar to offer shorter constructs for common and certain expressions (*i.e. The Empty Function, Quanti_ed Formula, FLWOR Expressions, Coercion*), and to replace the complicated syntax in "XQuery" with "LiXQuery" constructs and make it suitable for research and education.

In Thomas Largillier work in (Largillier, 2005), two new syntactic sugars were added to C++ to simplify writing efficient and generic code transformers code in LRDE project (LRDE, n.d.). The transformation expression can be expressed in C++ syntax but they need clever programmers to be able to write them which costs time and produce complex, heavy and unnatural code. Because of these issues, Thomas offered new syntactic sugar constructs to help the developers. The first constructs was the usage of meta-tags with variables and classes to tell the compiler that values must be calculated

at compile time. The other one was virtual "typedefs" used to overwrite the statically typed variables in sub classes.

The Object Constraint Language (*OCL*) language used to describe UML rules (Süß, 2006), was extended by Jörn Guy Süß in 2006 to have syntactic sugars as its concrete syntax is verbose and hard to be read (Süß, 2006). Jörn extracted the new syntactic sugars from math and logic depending on positive results he got from using the syntactic sugars within workshop notes and formalized due to UML 1.4.2 standard (OMG, 2005). The new syntactic sugars were Unicode and Latex symbols that can be used within MS-word documents, HTML and other UML representation document format and tools.

Following is an example shows how collections in OCL are represented using the new sugar Latex and Unicode syntactic symbols (Süß, 2006):

Example one

```
Originals OCL Set: Set(X)
Unicode syntactic Sugar: {X}
Latex syntactic sugar: \{/ \}
```

Example two

```
Originals OCL Sequence: Sequence (X)
Unicode syntactic Sugar: [X]
Latex syntactic sugar: [ ]
```

Other examples for other constructs are available in (Süß, 2006).

In 2006, following to their work in (Freeman et al., 2004), Steve Freeman and Nat Pryce used syntactic sugars in Java based embedded domain specific languages (EDSL) to implement sugar methods to replace the Java noisy syntax and non domain related code

that used to create and set up domain specific objects (Freeman and Pryce, 2006). The idea was that the authors wanted to embed domain specific language within a general language to use its implementation, capabilities, and tools and not to build the domain specific language on top of the language. The authors used syntactic sugars to implement methods that provide domain specific functionality; these methods hide and encapsulate the complex general programming language code used to implement the desired functionality and used within the EDSL code.

In Java like languages (*Java, Scala and C#*), Philippe Altherr and Vincent Cremet described in their paper (Altherr and Cremet, 2006) many simple syntactic sugars used to reduce syntax complexity, make code shorter and cleaner such as omitting empty type parameters list in classes and methods, omitting empty arguments lists, and using special identifiers (`_`) for un-referenced parameters.

Example removing square brackets:

```
class A { val x: A val y: List[A] }
```

The above construct is a syntactic sugar for the following:

```
class A[] { val x: A[] val y: List[A[]] }
```

In (Fruhworth, 2007), Clemens Fruhwirth introduced *Liskell*, a new syntax for Haskell that provides programmers with a set of syntactic sugars to ease programming (*Simple List, The Dispatcher Namespace, syntax sugar for defining macros "defmacro" and others*). Using syntactic sugars provided, the code was shortened and became more readable. The following example shows how syntactic sugars affected the syntax complexity to be simpler (Fruhworth, 2007):

The following un-sugared block:

```
(SList ([] (SSym "if ")guard action (trf - cond rest )))
```

Can be written in Liskell syntactic sugars as follows:

```
(if ,guard ,action ,(trf-cond rest))
```

The same for macro definition, it can be done using a simple construct in Liskell:

```
(defmacro (macro-name pts) body)
```

Thaís Batista and Maurício Vieira used syntactic sugars in their work (Batista and Vieira, 2007) within Aspect Oriented Programming language called *RE-AspectLua*, a new version of *AspectLua*. Syntactic sugar constructs were used to reduce number of code lines needed to define aspect interface and associating it with aspect points in the code, and make interface definition simpler (Batista and Vieira, 2007). Example shows how code becomes shorter:

```
aspectA = Aspect:new( {name = "Aspect A"} )
aspectB = Aspect:new( {name = "Aspect B"} )

ai1 = AspectInterface:new()
ai1:refinement( {name = 'interfacel'},
{refine = 'abstractpointA',
action = advice1} )

ai2 = AspectInterface:new()
ai2:refinement( {name = 'interface2'},
{refine = 'abstractpointB',
action = advice2} )

aspectA:interface(ai1)
aspectA:interface(ai2)

aspectB:interface(ai1)
```

The above code using new syntactic sugars aspect interfaces will be as follows:

```
ai1 = AspectInterface:new_refinement( {name = 'interfacel'},
{refine = 'abstractPointA',action = advice1} )
ai2 = AspectInterface:new_refinement( {name = 'interface2'},
{refine = 'abstractPointB',action = advice2} )

aspectA = aspect ({ai1, ai2})
```

In (Liu et al., 2007), the bidirectional transformation language Bi-X was used in bidirectional input and output data transformations and synchronization. Bi-X can be used to define new functional languages with syntactic sugars for ease of use (Liu et al., 2007). For example, the usage of curly-braces "{}" is optional in some constructs like Boolean-value functions, the programmer can omit them. The syntax of any new language based on Bi-X is composed of set of core Bi-X syntax constructs and functions, the new language syntax is a simplified form of constructs that make it easy for user to define transformation rules using simpler and shorter constructs than using only Bi-X core functions.

Java programming language adds new features in each release to make programmers life easier (Kominetz, 2007). In Java 5, John Kominetz explained that Java offered the "FOR-EACH" looping construct that is useful to iterate over lists and collections instead of "ITERATE-WHILE" loop constructs. The "FOR-EACH" loop is much simpler and shorter than the traditional loop construct (Kominetz, 2007). According to Raja Kannappan, the latest release from Java "Java7" offered new set of features such as multi exception catch (Kannappan, 2010). The new constructs help programmers to write one catch block for many exception types like:

The old multi-catch form (Kannappan, 2010):

```
try {
    // Say some file parser code here...
} catch (IOException ex) {
    // log and rethrow exception
} catch (ParseException ex) {
    // log and rethrow exception
} catch (ClassNotFoundException ex) {
    // log and rethrow exception
}
```

Can be replaced by:

```
try {
    // Say some file parser code here...
} catch (IOException ex | ParseException ex |
ClassNotFoundException ex) {
    // log and rethrow exception
}
```

Many other features were added such as supporting strings in selection statement cases, allow underscores in telephone numbers, credit cards, and social number, the ability to parse binary number from string, in addition to many other features mentioned in (Kannappan, 2010).

The *W3C Web Ontology Language (OWL)* is used by applications to process document data and present it for human benefit and readability (Golbreich and Wallace, 2008). *OWL* was extended with a new set of features provided in a new release called *OWL 2* (Golbreich and Wallace, 2008). Two of the new features in *OWL 2* were extra syntactic sugar added to make the common *DisjointUnion* and *DisjointClasses* patterns in *OWL* easier to write (Golbreich and Wallace, 2008). The usage of new syntactic sugar patterns *DisjointUnion* and *DisjointClasses* hides the complex *DisjointWith* patterns used to perform the same functionality.

Chieri and Atsushi work in (Saito and Igarashi, 2008) was to solve the problem of recursive classes-subclasses as they lost referencing and type safety because they referenced by name in the object tree. They solved the problem by proposing new lightweight polymorphism that support type safe recursive classes and added this solution to *JAVA 5* as extension called *Featherweight Java (.FJ)*. *FJ* extension related to nested classes, family-polymorphic methods, and relative path types were considered as syntactic sugars added to *Java 5*.

In (Haskell syntactic sugar, 2010), Haskell functional language used syntactic sugars for simplifying expressions and making syntax more readable and shorter. Example: For functions used in numbers manipulations, Haskell offered the following syntactic sugar constructs:

The original form: `\x -> x + 2`

The sugared from: `(+2)`

For lists:

The original form: `1:2:3:[]`

The sugared from: `[1,2,3]`

List comprehensions original form:

```
let ok (x,y) = if x < 2 then [x] else [] in concatMap ok foos
```

The sugared from:

```
[ x | (x,y) <- foos, x < 2 ]
```

Many other examples are available in (Haskell syntactic sugar, 2010). The syntactic sugar constructs used changed HASKELL constructs to be more readable, easier to write, shorter, and well-formed.

In (Mageed, 2010), *C# 3.0* was provided with new features to support LINQ functional paradigm. These features were classified as syntactic sugars that help in cutting down the repetitive code tediousness. The authors explained the new syntactic sugar features exist in C# 3.0 and how they can help supporting LINQ. The explained syntactic sugar features were implicitly typed local variables, automatic properties, object and collection initializing, anonymous types, extension methods, and lambda expressions (Mageed, 2010). These new syntactic sugar constructs aimed to make code length and objects initializing constructs shorter. The following examples shown in (Mageed, 2010):

In order to define a class called Person and create instance from it, the old C# code to achieve this is:

```

public class Person
{
    private int _id; // Id property's private backing field
    // Id property
    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }

    private string _name; // Name property's private backing field
    // Name property
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Person() {} // default constructor with no parameters

    // overloaded constructor that accepts parameters
    public Person(int id, string name)
    {
        _id = id;
        _name = name;
    }
}

// set properties via parameters
Person ahmad = new Person(42, "Ahmad");

// set properties individually
Person emad = new Person();
emad.Id = 1;
emad.Name = "Emad";

```

Using new C # 3.0 syntactic sugars, this can be done as follows:

```

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}

var ahmad = new Person { Id = 42, Name = "Ahmad" };
Person emad = new Person { Name = "Emad", Id = 1 }; // order does not matter

```

Another example, to fill a list with instances of Person class using old C# syntax, user need to do the following:

```

List<int> list = new List<int>();
list.Add(1);
list.Add(2);

Dictionary<int, Person> dictionary = new Dictionary<int, Person>();
Person p = new Person(42, "Ahmad");
dictionary.Add(1, p);
p = new Person(1, "Emad");
dictionary.Add(2, p);

```


While in new C# 3.0 syntactic sugar, this can be done as follows:

```
var list = new List<int> { 1, 2 };

var dictionary = new Dictionary<int, Person>
{
    { 1, new Person { Id = 42, Name = "Ahmad" } }, // object initializer
    { 2, new Person(1, "Emad") } // constructor
};
```

C# 3.0 constructs are shorter than old one but they appear harder to read and understand (Mageed, 2010). Other examples exist in (Mageed, 2010).

2.4 Syntax Errors Reduction

Efforts to reduce and eliminate syntax errors were considered in many researches. The majority of these researches focused on students and novice programmers.

Teitelbaum and McIlrow in (Teitelbaum and McIlrow, 1981), proposed The "Cornell Program Synthesizer" text editor to help in reducing syntax errors through providing programmers with syntax templates for the constructs they want to write using command code. After the editor print the desired template (i.e. IF statement template), programmer has to fill the template with values and complementary statements (i.e. conditions and body statements). The programmer has to memorize the command codes for all constructs.

Sarah K. Kummerfeld and Judy Kay work in (Kummerfeld and Kay, 2002) offer simple way to help students in solving syntax compile-time errors in C/C++. They build a simple web based guide reference that explains the common compiler syntax errors with details on error description, expected reason, code examples, and how to correct it using an example. The authors validate this approach though preliminary experiment with simple size of students. They asked students to correct syntax errors in certain written programs. The results showed that the online reference guide helped novice

programmers in solving their errors more than experienced one. The same results were noticed for experienced programmers when they face errors they were not familiar with.

Another trend the authors in (Kelleher et al., 2002) followed to reduce syntax errors for novice programming students was by offering 3D visual programming language called "Alice2". The main propose of this language was to help learners to focus on program logic and execution and not on syntax writing, errors, and to minimize the set of syntax they need to memorize. Learners can easily build their programs using drag and drop tiles, objects, and relation, then execute the program and check results. They don't need to write the underlying code, it will be generated by Alice2 in the background.

The study done in (Chinchani et al., 2003) show that the popular programming languages syntactic features may cause logical errors and security breaches in program execution while the program syntax grammar is written correctly. For example, the usage of semicolon as complete statement in C can cause FOR loop to complete successfully but will not affect the loop body and cause logical error:

```
int x;
For(int i=0; i<100; 1++); // the ";" will prevent x from
    x = x + i;           // increment and cause wrong value

For(int i=0; i<100; 1++) // this is the correct code form
    x = x + i;
```

Another example is the usage of symbolic operators in expression:

```
int i , j, v ;

v = i > j ? i : j; // v will have int value which is correct
v = i > j ; i : j; // v will has Boolean value and cause program
// fault because "?" is replaced with ";"
```

In Languages design, the authors advice to avoid ambiguous syntax constructs that depend on symbolic operators, short expression statements, weak typing, similar

variable names with case sensitive, the similar or close key words and white spaces as they can cause logical errors and security breaches.

The following table summarizes approaches used to reduce syntax errors:

Table 2.1: Summary of syntax error reduction strategies

Reference	Strategy	Description
(Teitelbaum and McIlrow, 1981)	Syntax Template	Provide programmers with syntax templates to fill. "Cornell Program Synthesizer" text editor is used.
(Kummerfeld and Kay, 2002)	Web based guide reference	Offer web based guide reference for errors in C/C++ with description for reasons and how to fix with examples
(Kelleher et al., 2002)	Visual Programming	Use 3D visual programming language called "Alice2" to build programs using drag/drop of program building blocks (objects). No code is written by programmers.
(Chinchani et al., 2003)	Ambiguous Syntax	Avoiding ambiguous syntax constructs like symbolic operators, close keywords, short expression statements and others

2.5 Contribution

By referring to programming languages review in section 2.2 (Figure 2.4); we noticed that the mainly common programming languages paradigm used were the object oriented and imperative paradigms. Most of the modern programming languages focus on object oriented. Based on this we decided to focus on these two paradigms in our study.

All the related work described in syntactic sugars review (section 2.3) was focusing on enhancing certain syntax constructs partially in order to add support for specific concepts such as functional paradigm support in object oriented programming language, methods shorthand, interfaces, decrease code verbose and un-ambiguity, and adding new functionality or domain support to a programming language "DSL" (Altherr and Cremet, 2006) (Freeman et al., 2004). There was no proposal or work done to enhance the general programming languages common abstract syntax constructs using syntactic

sugars, and we didn't find previous study to measure syntactic sugars efficiency and if they achieve their proposed goals.

The trends used to reduce syntax errors focused on novice programmers. These trends provided code templates and error guide references (Kummerfeld and Kay, 2002) (Teitelbaum and McIlrow, 1981) to help in reducing the errors. Other trends changed the programming paradigm and offered 3D visual programming environment (Kelleher et al., 2002) to solve the problem, but 3D visual programming prevent programmers from practicing the real life programming experience and paradigms. In addition, most of the proposed trends used to reduce syntax errors don't work in simple and remote development environments, for example, the code template need IDE or rich editor to insert templates it in the code, the Web reference need internet connection, and the visual 3D need visualized environment and rich IDE to manipulate the objects, while syntax sugars can be useful and work in simple text and pure command line environment.

In our work, we tried to make enhancements using syntactic sugars on general level for the most common abstract constructs that are used in both object oriented and imperative programming languages. We propose using syntactic sugar to reduce syntax errors, reduction of repetitive keywords, better semantic extraction, code debugging, and make text based and remote development and development using simple editor easier for novice and expert programmers, which was not discussed previously. And we conducted an exploratory case study to measure the proposed syntactic sugar construct efficiency.

Chapter Three

Constructs Selection and Enhancements

3.1. Introduction

This chapter explains the technical and practical methodology we used to select and enhance the set of syntactic sugar constructs.

Constructs selection methodology depends on two factors: usability frequency and object oriented programming (OOP) Relevance. We determined the common abstract programming constructs (Mosses, 2005) that are classified under these factors. The abstract constructs are supported by set of actual syntactic constructs we extracted from 5 programming languages (PL).

We used a questionnaire designed using the extracted constructs to validate and select the syntactic constructs. Questionnaire results helped us in realizing new facts and improving our methodology to achieve our goals.

The following sections explain all of these steps in details.

3.2. Constructs Selection Methodology

This section explains the factors used in our methodology to select the constructs included within the research and to determine which of them to study, these factors are:

- **Usability frequency:** by usability frequency we mean how much these constructs are used in writing programs. We tried to focus and select the most common and frequently used constructs which most programs, even simple programs with few lines, will use a set of them, for example control constructs (IF and SELECTION constructs), looping, methods, functions...etc.

- **OOP Relevance:** OOP paradigm is now one of the common programming paradigms used by real application developers and in most university curriculums. This research aspired to cover the most common OOP abstract constructs: inheritance, collection iteration, modules and packaging, access modifiers, method overloading and riding, object instantiation and initialization, message passing, and exception handling.

We tried to cover the more common and most widely used programming languages constructs used by both professionals and students. This range of users makes the benefit of the research reach most programming language users. The final effect is most beneficial for programmers based on their professional level.

3.3. Abstract Constructs Selection

Programming syntactic constructs are built based on abstract constructs that are common between the majority of programming languages, but differ in their syntax and implementation as each programming language has its own different syntactic constructs which are obtained based on the abstract constructs.

For example, in any program, we need control constructs (conditional branching, looping...etc.). Looping and conditional branching show the abstract constructs as abstracted representation, while their implementation is different between PLs and expressed using syntactic constructs like the IF statement, FOR loop, WHILE loop and many others.

Below, we show the abstract constructs categories we selected for each concept depending on construct selection methodology factors described in the previous section:

Usability frequency constructs:

1. ***Method (function) definition construct***: this construct study shows the way of defining a method, parameters, return type and value, and scope determination.
2. ***Looping construct***: this construct study describes the definition of looping control construct over a range of numbers or array entries, and how to determine looping block scope in a better way.
3. ***Selection construct***: this construct study describes the selection construct over a set of values.
4. ***Exception handling variables scope construct***: this construct study describes the scope of variables defined within exception handling block and tries to enhance it. This construct is semantic-related and has nothing to do with syntax.
5. ***Building blocks scopes determination***: this covers more than one construct, in this area we tried to enhance the syntax scope determination for many constructs like looping, methods, and others to make the code more readable, less ambiguous, and easier for semantic understanding.

Object oriented programming constructs:

We studied the main constructs used in OOP development especially constructs that represent the main OOP concepts like inheritance, encapsulation, polymorphism...etc., as described below:

1. ***Class***: In this construct we studied the main class building blocks that are explained in the other constructs below. We focused on attributes (states) and method (behavior) definitions, access modifiers, constructor replacement, class definition constructs. As a result, many class building blocks are affected by this study.

2. **Inheritance**: in this construct we studied the inheritance relation syntactic writing form.
3. **Methods as Constructors**: in this construct we studied how to use any method as a constructor to be executed at instance creation.
4. **Libraries and packages**: here we studied the syntax used to call certain libraries or modules and how we enhanced them.
5. **Attributes access modifiers**: this construct shows how to define attribute-access modifiers and enhancements we've suggested.
6. **Methods access modifiers**: this construct shows how to define method-access modifiers and enhancements we've suggested.
7. **Iteration**: in this construct we studied how to iterate over object lists in easier and more readable way.
8. **Object instantiation**: here we studied how objects are instantiated from classes and how to make them easier.
9. **Object / Method messages passing (calling) format**: in this construct we investigated the syntax used in calling object-instance methods, pass messages (values) to them, and how to enhance it wherever possible.

From the abstract construct categories mentioned above, the study covered the following common OOP properties in a direct or indirect way as follows:

1. **Inheritance**: the inheritance construct is affected directly by considering the syntax used to express inheritance relation and enhancing it.
2. **Polymorphism**: there is no direct construct to represent polymorphism; it is a set of concepts more than direct syntactic construct.

Polymorphism can be achieved through method definition (overloading, overriding), method invoking and message passing.

3. **Encapsulation**: This concept is achieved in an indirect way through using the construct mentioned previously: attribute-access modifiers, method definitions, and access modifier.
4. **Relations** (Is A, Kind off...etc. Association and aggregation): no direct construct for these relations, this can be achieved using inheritance and/or the class attributes definition and access modifiers.

3.4. Programming Languages Study and Constructs Extraction

Depending on the previous section output, we had the abstract constructs set which we used to extract the actual syntactic constructs from a set of programming languages as described in this section.

Constructs included in this study were obtained from two main sources: the first by extracting the actual syntactic constructs that represent the abstract constructs explained in section (3.3) through studying a set of programming languages, and the second source by brain storming we did to suggest a set of syntactic sugar enhancements on the abstract and syntactic constructs.

The final result was a mix of syntactic constructs extracted from programming languages, which are considered to be widely used, in addition to a set of enhancements we suggested.

We selected 5 programming languages to extract the syntactic constructs. The PL selection criteria were based on:

1. Language usage and spreading.

2. Language families and development; we focused on languages based on their families.

We tried to select programming languages that are widely used previously and currently, and considered selecting languages that are developed on top of other languages or languages whose syntax is a mix of other older languages. This is to create balance between old and new programming languages, in addition to covering a large set of syntactic constructs that are used in a large set of programming languages (Lévénez, 2009).

Selected programming languages are: Eiffel, Python, Java, C#, and Ruby. A brief overview about each language can be found in Appendix 3.

After studying these programming languages, we started analyzing the syntax that exists in them and extracted all syntactic constructs that match our abstract constructs and filled them in a matrix (sheet) of constructs showing how they syntactically exist in each of the 5 languages. The matrix is shown in Appendix 1.

To determine which syntactic constructs from the extracted set are suitable and meet our goals, we need to get the opinion of people who use programming languages which we did in the following section.

3.5. Questionnaire Design, Distribution, Collection, and Results

To select and form the new syntax constructs set from the extracted and enhanced constructs, we followed a questionnaire approach to get the opinion of people who use programming languages (programmers, developers, students, tutors), and get their recommendation for which constructs are better based on their experience and expectations.

The questionnaire was designed on top of the extracted constructs sheet introduced in Appendix 1 which includes 23 different constructs for each of the 5 languages as follows: class definitions symbols, class scope, method signature definition, method scope symbols, inheritance construct, constructor definition, super / parent class calling, package / module definition, libraries and packages calls construct, libraries and packages repetition, attribute access modifiers, method access modifiers, methods and attributes organization in class, object instantiation, object message / method calling, exception handling constructs used, exception handling variables scope, control statements scope symbols, object collections iteration, FOR loop, WHILE loop, IF statement, and SWITCH / CASE statement.

The questionnaire design process and questions' construction was based on standard scientific guidelines (Clarke, 2001) (Cheah, 2005) (Borgatti, 1996) (Quick MBA, n.d.) (Arsham,1994) (Survey Design, n.d.) (Galloway, 1997). We consulted with a statistical expert in the design process who verified the questionnaire and questions whether they are measurable and achieve desired goals.

A pilot version of the questionnaire with 36 questions in 20 pages was reviewed by the statistical expert and distributed on 3 different developers in 2 companies to test its validity and questions' correctness. The feedback obtained from the pilot version showed that the questions were clear and understandable with measurable answers, but the questionnaire was too long and took a long time to be answered, which can cause a problem in collecting answers as people usually don't like spending a lot of time answering questionnaires, especially if they are long. To solve the length problem, we—including the research supervisor and the statistical expert—discussed minimizing the size of the questionnaire and including the key questions that represent the most

important constructs and cover similar or less important constructs. After deep brainstorming and review for constructs and their usage, we were able to minimize the questionnaire to include 19 questions in 9 pages.

This also affected the covered construct to be as follows:

- Constructor definition
- Method signature definition
- Method scope symbols
- Inheritance construct
- Libraries and packages calls construct
- Libraries and packages repetition
- Attributes access modifiers
- Methods access modifiers
- Methods and attributes organization in class
- Object instantiation
- Object message / method calling
- Exception handling variables scope
- Object Collections iteration
- FOR loop
- IF statement
- SWITCH / CASE statement

The excluded constructs were:

- Class definitions symbols: these constructs are not as vital as most languages use similar constructs for class definition

- Class scope: most languages use similar constructs for class definition; we considered the most common form that uses ({}).
- Super / Parent class calling: not common case in all selected languages, preferred to go with the C# or Java construct as it is the simplest and most similar to other languages.
- Package / module definition: not a major construct for students and they rarely use it; we preferred to go with the Java construct as we deduced that it is the simplest.
- Exception handling constructs used: we selected the simplest and most common construct that is used in many other languages (C#, Python, and Java)
- Control statement scope symbols: this construct is covered implicitly when enhancements suggested over some constructs described in section 3.6.
- WHILE loop: similar to FOR loop in concept.

The final version of the questionnaire is in Appendix 2.

We distributed the questionnaire to programming professionals and students in Palestinian universities and companies in the West Bank, Palestine¹. The population and sample size were calculated depending on a report of ICT working forces in Palestine we got from "Palestinian IT Association of Companies (PITA)" (PITA, 2008). Details of population size calculation are explained in Appendix 4.

We used confidence interval 10, and confidence level 95%, the sample size was: ICT Professionals: 77, ICT Students: 93. The number of distributed copies was 600, and the

¹ Universities: Al-Najah University, Ber-zeit University, Al-Quds University
 ICT Companies: Information and communication technology center at Alquds Open University, Information and communication technology center at Alquds University, Hulul, Safad, Asal, Exalt, Al-Andalus, NoorSoft, Al-Watanya Mobile, Jawwal, GSSI, Bisan

number of collected copies was 251, distributed as follows: ICT Professionals: 79, ICT Students: 172

After analyzing the results obtained from the collected questionnaire, we found that 14 out of 15 construct questions were with Java construct selections, and only 1 construct was from another language (Ruby), which was the method definition construct. The results helped us in realizing the fact that people usually prefer what they know while resisting change (change management); they answered in a way that didn't nominate a new easier constructs set.

We were able to explain the results by referring to PITA working forces report (PITA, 2008). In the report, we found statistics about "Percentage Distribution of ICT Professionals According to Technical Skills" that shows 48% of people have C++ experience, and 39.2% have Java experience. Both C++ and Java were on the top of the languages list with which ICT professionals have skilled experience. This lead to a fact that these 2 languages are the most commonly used programming languages in the Palestinian companies and universities. In addition, companies have started using C#, which has the same syntax as Java. As a conclusion, people answered the questionnaire by selecting constructs they are familiar with, not necessarily what is better.

Results directed us to modify our methodology by nominating a set of syntactic sugar constructs from the extracted and enhanced set, which we assumed helps in improving code and program writing, minimizing syntax errors and ambiguity, create clearer semantic, and achieve all the objectives we try to approve. Then we request people practice them, and give their feedback as explained in the coming sections and chapters.

3.6. Syntactic Sugar Constructs Set and Enhancements Selection

Based on the modification done in the methodology, we took the responsibility to select and nominate a set of syntactic constructs from the extracted programming languages' constructs, in addition to a set of syntactic sugar enhancements added to these constructs to produce new syntax for programming languages' common constructs.

The criteria we followed in selecting the new constructs and formulating enhancements was as follows:

1. ***Reduce repetitive keywords***: that is to not repeat the same keyword many times if it can be replaced with one keyword, whenever possible.
2. ***Shorter constructs***: try to make constructs shorter to write where possible, like looping constructs when looping is sequential, whereas in such cases conditions are not needed.
3. ***Close to natural language***: try to select constructs and modify them to be closer to natural language like the selection statement.
4. ***Make constructs closer to standards***: make some constructs derived from well known standards for developers as the case of inheritance constructs: the symbol used is derived from UML inheritance relation symbol.
5. ***Offer many alternatives for the same construct***: in some constructs, try to offer the same semantic through many syntax alternatives like method calling.
6. ***Enhance constructs scope identifiers***: make code more readable and less ambiguous by modifying some constructs scope identifiers like loop constructs and methods.

The selected constructs and syntactic sugar enhancements, suggested based on the described criteria, are supposed to make the code shorter, less error-prone, semantically clearer, easier to write, and more readable.

The following table summarizes all selected and enhanced constructs, their description, and gives a simple example for each one. A detailed description for these constructs is available in Appendix 8.

Table 3.2-A: The selected and enhanced syntactic sugar constructs – part 1

Enhanced Constructs	Suggested Syntax	Comments
Class Inheritance Construct	class ChildClass -> ParentClass // UML notation class ChildClass:ParentClass	Offers code reusability, shorthand, and maintenance
Class Instantiation Construct	myInstance = MyClass ();	Keyword reduction
Method Definition Construct	def methodName(int size, Object obj) int x = 5 + size; return x; endef	Used simple construct to define a method where the return type is not needed.
Method Calling Construct	instanceName.methodName ; // calling method without parenthesis instanceName.methodName() ; // calling method with parenthesis instanceName.methodName2(5, objInst) ; //calling method with parameters and parenthesis instanceName.methodName2 5, objInst ; //calling method with parameters and without parenthesis	Many alternatives to call a method from class instance and message passing
Method Execution on Class Construction Construct	class MyClass create executeMeMethod { def executeMeMethod() system.out.println("I'm executed on instance"); endef }	Used to execute a method on class instantiation without using constructors or if no constructors / defaults constructor is available.
Looping Construct	5:times do ref // loop 5 times System.out.println("Val: "+ref+" in: "+arr[ref-1]); end	Used to Loop a block of statements or array entries number of times in simple way. "ref" is optional.

Table 3.2-B: The selected and enhanced syntactic sugar construct – part 2

Enhanced Constructs	Suggested Syntax	Comments
Object Collection Iteration Construct	myCollection:each do ref // iterate myCollection System.out.println("Hi, I'm looping..." + ref); endEach	Iterates over collection of objects or any type derived from collection type in easy way
Selection Construct	choose(a){ case 1: System.out.println("One..."); }	The keyword used to be more close to human natural language
Packages / Modules Calling Construct	import: java.io.*; // write import only once for all java.util.*; java.lang.*;	This to reduce repetitive "import" keywords
Variables Access Modifier	class ClassName{ private: // private attributes int a = 1; String b; public: // public attributes File file = new File(); double length; // the same for other access modifier }	An enhancement to define many attributes with the same access modifier. <i>Close to C++.</i>
Method's Access Modifier	def __privateMeth() //2 underscores : private def _protectedMeth() //1 underscores: protected def publicMeth() //no underscores: public method	The access modifiers for methods are specified in simple way by using underscore(s) "_" at the beginning of method name to define it access modifier.
Exception Handling Variables Scope	try{ int nm = Integer.parseInt(br1.readLine()); } catch (Exception e){ System.out.println("num="+ nm);//nm is accessible } System.out.println("num="+ nm);//num is accessible	We modified the scope (accessibility) of variables defined within the exception try block to be accessible outside the try block

The output of this phase was a new syntax constructs set that we assume will achieve our goals. This assumption of the new syntax set needs verification. The verification was done through designing and executing an exploratory case study on these constructs as described in chapters 4 and 5.

Chapter Four

Exploratory Case Study

4.1. Introduction

This chapter describes the exploratory case study used to verify the assumption of the suggested syntactic constructs set obtained based on methodology modification.

In this case study, we implemented the new constructs on top of Java 1.5 syntax parser with simple IDE to be used within the experiment where participants use the parser to write programs using the new syntax constructs set.

The case study is done with a small sample of computer science students and professional developers in order to get feedback from different programming language users with different experience levels.

The following sections explain the experiment design, how it is executed, and types of data collected.

4.2. Exploratory Case Study Experiment

We conducted an exploratory case study with a small sample size as we used this experiment as an indicator to know if our assumption regarding the new constructs set was valid.

The work we did was an indicator and represented a start for more advanced research. We don't claim that the results in this research are final, they are indicators. In future work, we need to extend the constructs set and increase the experiment sample size, and perform the experiment within a longer timeline, for instance teaching the constructs in a university course for multiple semesters.

We were unable to make the experiment with a large set of users due to many difficulties and issues explained in section 4.7.

4.3. Case Study Design

In this section, we describe the methodology used in the case study design and execution. Questions we need to answer through the case study are:

- Do syntactic sugars help in development with fewer errors?
- Do syntactic sugars help in semantic extraction?
- Do syntactic sugars make code more readable, easier to write?
- What is its effect on students and professionals?

Answers to these questions can judge and validate whether the new syntactic sugar constructs set achieved the desired goals.

The way to measure the constructs' efficiency is to let users practice them, especially the novice users like students. The chance for errors and difficulty in writing programs with students is much higher than with experienced professional users as their skills help them.

Semantic extraction and debugging are core tasks the professionals do throughout their work in the case of huge programs and logical error debugging. We need to verify whether or not the new constructs can help in semantic extraction and debugging. We covered this area through the case study we did with professionals.

The case study was designed into two tracks in order to achieve what is described above: The first track was with students. We introduced the new constructs to them, got feedback of their initial impression through an interview, then asked them to write programs using the new constructs set. The other track was with programming

professionals. We gave them a set of programs written using the new constructs and asked them to debug and extract their semantic. Then we introduced the new constructs set to them, and did an interview to get their feedback. Details of these two tracks and how they were executed is explained in details in the following sections (4.5, 4.6).

We prepared the material needed in the case study which: detailed tutorial contains all details for the new constructs including sample programs written using these constructs, a summarized presentation including the constructs set presented for users, and a parser used in writing the code and checking syntax error as described in section 4.4. All material and resources were prepared on CDs and distributed on the case study participants.

Finally, a set of questions in interview form were prepared to get feedback from participants. Details of the interview questions are in Appendix 5.

4.4. New Syntax Parser and IDE

Syntax parser was implemented to validate programs written within the case study for any syntax errors including the new syntactic sugar constructs set.

The new constructs set we proposed is not a complete programming language. It is a new subset of syntactic sugar constructs and not enough to write complete programs.

We used existing programming language syntax and added the new constructs set to it. We used Java 1.5 syntax grammar and modified its grammar rules (BNF) by adding the new syntactic constructs set without eliminating any of Java 1.5 syntax. This way, users can still write code and programs' using the existing Java 1.5 code in addition to the new constructs set. It matches the concept of syntactic sugars constructs as they added to an

existing language, and the syntax became more user-friendly, requires less writing efforts, and is easier while users can still use the old syntax.

We implemented the parser using parser generator tool called JavaCC (JavaCC, 2010) in addition to a very simple integrated development environment (IDE) that offer users with a very simple text editor to write their programs and a simple set of visual commands to manipulate source files and invoke the parser on the written syntax. The IDE shows all syntactic errors that have occurred in the code.

Having a very simple IDE without any kind of wizards or tools that help users, such as code completion or suggestions, was done intentionally to make the experiment real and force users to write their code completely on their own. This was to match the environment we want to study and to identify the effect of syntactic sugar constructs where users usually don't have a rich IDE and need to write code in command line mode using simple tools.

We must clarify that the IDE and parser are only used to write and parse syntax. No compiler or program execution was done as we focused on the syntax writing and correctness. Compiler can be done in future work.

The IDE tracks and logs user syntactic errors generated during writing any program. It saves all errors, their line numbers, and timestamp for each error, in addition to the program itself. This kind of logged data helped us to determine the count of errors occurred, the source of such errors (old syntax, new syntax), and the time consumed in writing the program. All of this data will be explained with results analysis in chapter 5.

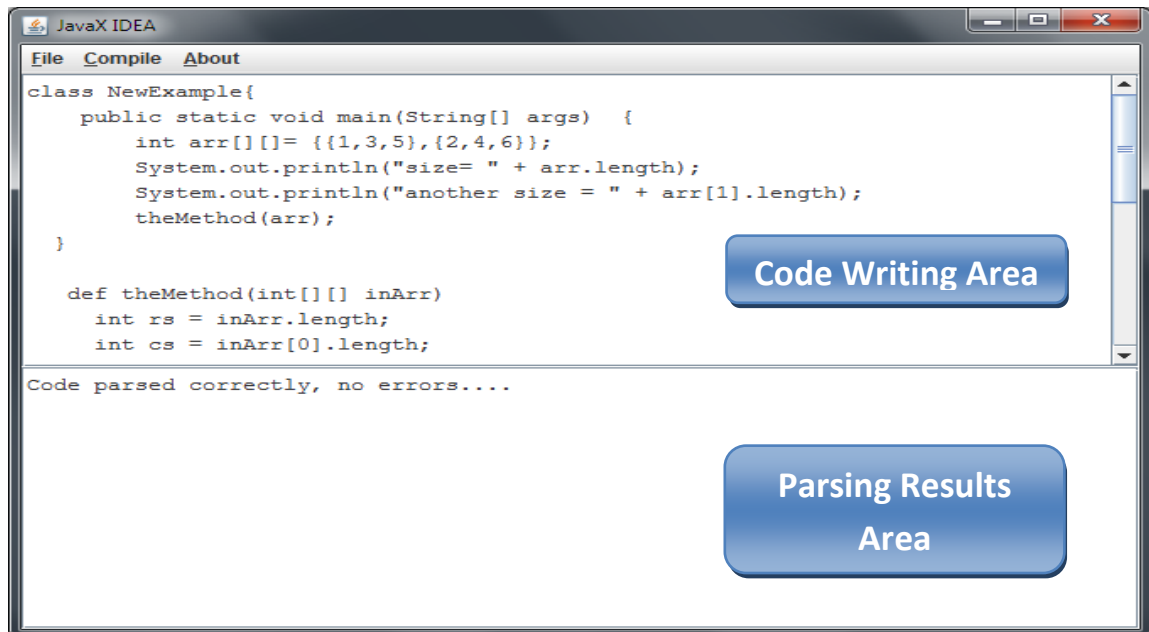


Figure 4.2: New syntax IDE and parser tool

4.5. Students' Case Study Track

Students who learn programming languages usually face a lot of syntax errors in their code, and can become lost while tracking and debugging the code, especially when syntax and/or logical errors occur. It is not easy for them to remember the syntax keywords and deal with long constructs. We thought that this kind of user is a good candidate to verify the new syntactic sugar constructs with them as they are suffering from most of the issues we've tried to solve using the new syntactic sugar constructs (Kummerfeld and Kay, 2002).

The case study was done with 6 volunteer students with the cooperation and support of the computer science department in Al-Quds University.

We had a short meeting with the students and explained to them the idea, what we were trying to do, and their role in the experiment.

In the second meeting, we did a presentation for them explaining the new constructs, their objective, what benefit we expect to get, and presented some examples that were

written using the new syntax. After the presentation, we did a quick interview with them in order to answer the interview questions, which reflected their impression regarding the new constructs. By the end of the meeting, students got a CD that contains the presentation material, a detailed tutorial of the constructs, and an executable version of the parser. The purpose of this CD was to give students the ability to read more about the new constructs and practice them using the parser in their free time at home, and before executing the practical part.

In the third (final) meeting, we prepared the PCs in lab with Java and the parser IDE and provided students with a description of 4 programs to write using the new syntax. These programs included: a program to find the sum of numbers, a second program to find the minimum and maximum numbers in an array, a third to demonstrate bubble sort, and a final program to extend the Java String class with methods to return tokens in a string and their count. Details of these programs are in Appendix 6.

We designed and choose these programs on purpose by taking into consideration the need for student to practice the new syntax using simple programs and then use them in harder ones. The first program is very simple so that they can practice the new syntax and the parser. The second and third programs were from courses they've learned and already understood well with increasing difficulty level. This helped to check the effect of new constructs in writing such programs, especially students who practiced them in other languages, and what kind of differences they noticed when using the new syntax.

The last program was new and the most difficult one. We wanted to know if new constructs help them in solving new problems efficiently with less effort and fewer errors.

At the end, log files, saved data, and programs were collected to be used in results analysis.

4.6. Professionals' Case Study Track

The second track of the case study was set out to check the new syntactic sugar constructs set with professional programming language users (programmers and developers) and to verify whether or not the new syntactic sugar constructs set helped in reducing errors, debugging, and with semantic extraction.

Professional programmers were the participants in this track, as their experience allows them to judge whether or not the new syntax helped in program debugging and semantic. They worked on many cases to debug certain code or to continue working on others' code where they have to understand its semantic.

This track was done with 6 professional volunteer programmers from 2 companies.

The first phase was that we gave programmers 10 programs divided into two groups: 5 of them written in ordinal Java code, the other 5 using the new constructs. We shuffled them all in a document while taking into consideration that those programs should be similar in concept and difficulty in order to be fair and unbiased. Supervisor consultation and review is done before distributing them to the participant. Details of these programs are in Appendix 7.

The aim of distributing these programs was to see whether users can extract and understand the semantic of programs written using the new syntax without knowing it previously, compared to their ability to do the same with programs written in ordinal Java syntax, which they were already familiar with. Programs were distributed and answers collected.

Next step, we did a presentation for the same programmers explaining the new constructs set with examples, and then got their impression and feedback by answering interview questions.

All data and answers were collected and analyzed as described in chapter 5.

4.7. Difficulties Faced During Case Study Execution

We faced many difficulties and limitations during the case study that prevented us from extending the sample size. These difficulties were summarized as follows:

- Students didn't show interest in participating in such types of research and experiments. We posted an announcement leaflet for students to register in order to participate in the experiment. The announcement included some kind of encouraging rewards (small amount of money) that will have been given for participating and committed students. The result was no students registered at all.
- As students were not interested, we decided to talk to them in classes and encourage them to participate. As a result, we were able to register 6 students only.
- Students' time and availability: we were forced to change the time of the meeting many times because of students' special circumstances or because they were busy with exams. This caused some latency and we attended some meetings without getting any output.
- Students' self learning: most students who participated in the experiment didn't read the full tutorial nor practice the code during their free time. This required them to spend some time before the experiment to review the constructs, affecting the experiment time negatively.

- We faced similar issues with programmers and developers in companies, such kinds of users don't prefer to spend time doing work outside of their paid tasks. Most of the time they were stressed and busy carrying out their duties. In addition, they need management permission to participate in such kinds of activities which made things difficult sometimes as they considered it a minor activity along with the fact that employees must focus on work and delivering on time. Based on this, we worked with a small set of developers (6) whom we were able to get permission from their employers.
- Professional developers' availability: it was hard to manage meetings with participating developers as they were busy most of the time in their tasks. And when some of them were available, the rest were in meetings, outside the company, at a customer site, or on vacation. It was hard to gather them all at once.
- Financial issues: work with a large set of users who will be reward formed a slight financial load we couldn't afford.

Chapter Five

Exploratory Case Study Results

5.1. Introduction

This chapter shows and analyzes the results of the exploratory case study done to check the suggested syntactic constructs set validity.

Results were related to the interviews with students and professionals, the practical work done by students through writing programs using the new constructs set, and the work done by professionals in extracting the semantic from set of programs.

Analyzing the data collected in all mentioned areas showed encouraging results that support our assumptions as explained in the following sections.

5.2. Students' Interview Results

Students' answers on interview questions showed a positive indicator for new syntactic sugar constructs set validity. Results are summarized in the following table¹:

Table 5.3: Students answers on interview questions

Question	Result
1- Do you believe that using the new constructs will save efforts in writing code especially in case of repetitive keywords (import, access modifiers...etc.) and shorter looping constructs?	Agree
2- Do you think that using new constructs will help in decreasing syntax errors as result from saving repetitive keywords and distinguish scope using different identifiers?	Totally Agree
3- Do you agree that using new constructs will make the code debugging easier?	Agree
4- Do you think that the code will be more readable using the new constructs?	Totally Agree
5- Are the new constructs can help in extracting the program semantic from just reading it with minimal execution efforts and without the need for executing it many times and debug it to understand its functionality?	Agree with Reservation
6- Is it true that the new construct can help in producing programs with less number of code lines (shorter syntax)?	Totally Agree

¹ (Answering scale: Totally Agree, Agree, Agree with Reservation, No Answer, Poor, Disagree, Totally Disagree)

The distributions of students' answers are shown in the following chart:

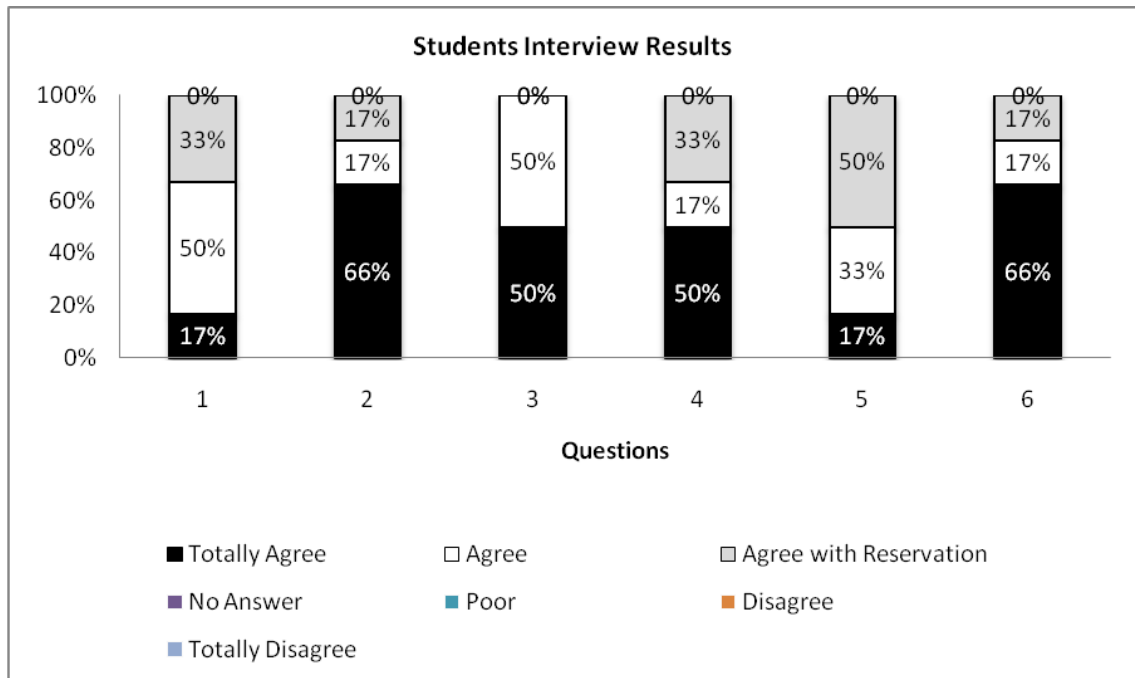


Figure 5.3: Students interview answers distribution

Answers distribution in Figure 5.3 showed that there were no negative answers. This means the new syntax constructs set indicates an enhancement in programming style and students were able to understand these constructs and their effect on syntax.

Summarized answers showed that the new syntax constructs set effect on reducing the syntactic errors and making the code more readable and shorter is high. They help in a reasonable way in saving code writing efforts, using less repetitive keywords, and making code debugging easier.

Students believe that new constructs set helps in semantic extraction with reservation. This result was the lowest. We expect the reason for this is: usually in students' assignments and work, they write programs more than debug, and their need to understand written code and extract its semantic is less than professionals. They write their own programs from scratch (which are mostly short programs) and don't need to

continue on somebody else's code as in the case of professional developers when huge code is moved from one developer to another.

5.3. Students Practical Case Study Results (Program Writing)

We collected the log data generated by the parser IDE which recorded errors that occurred while students were writing the syntax of the programs.

From the log data, we extracted errors that occurred in each program for each student. We classified these errors into two types: errors that occurred in old existing Java syntax, and the errors that occurred in the new suggested constructs. The objective of this classification was to measure the percentage of errors that occurred by each type and to check if there was any improvement achieved using the new constructs.

The following chart shows the percentage of errors that occurred by each construct type in each program (average summary for all students):

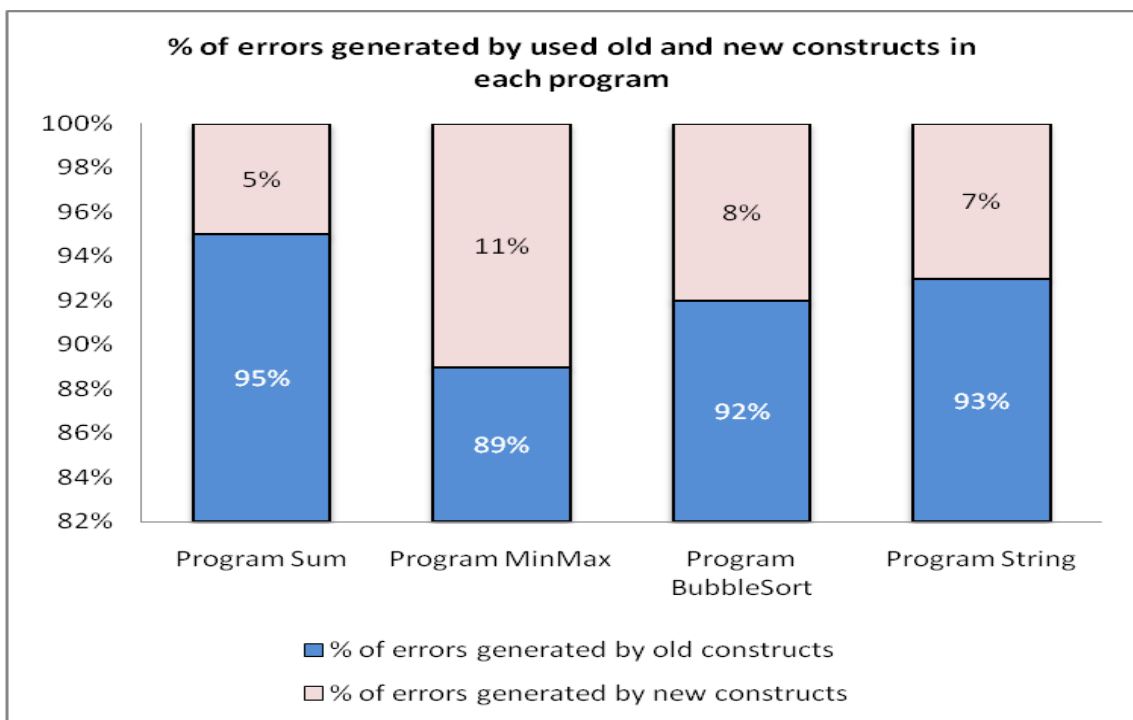


Figure 5.4: % of errors generated by using old and new constructs in each program

Prior to analyzing the results in Figure 5.4, we counted the number of each construct type used in each program and summarized them. This was done in order to check if the ratio of generated errors from each type was reasonable to the number of constructs used, and to make sure that there was no gap between the number of used constructs from each type and number of errors (i.e. it is not reasonable to say that new constructs caused no errors when it was used only 1 time in a program).

The following chart shows the percentage of old and new constructs used in each program (average for all students in all programs):

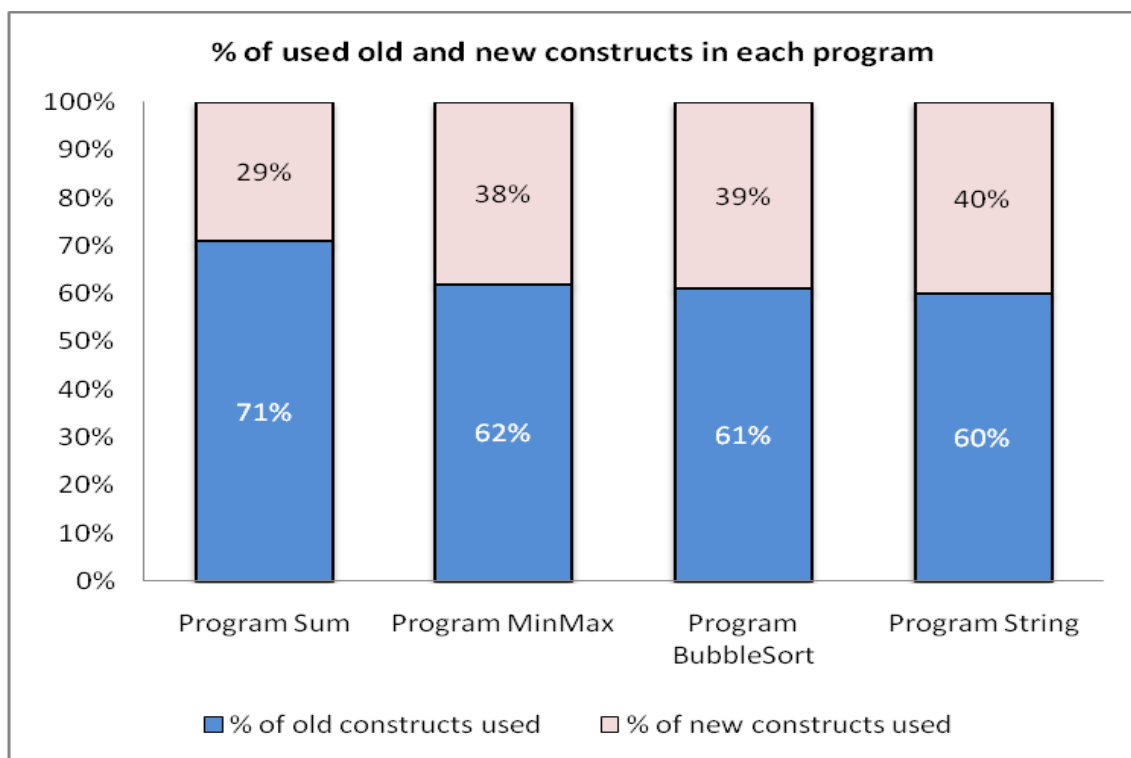


Figure 5.5: % of old and new constructs used in each program

Using the 2 previous charts, we observed that the number of errors occurred by old constructs was much higher than the number of errors occurred because of the new constructs set. By referring to Figure 5.5, and if we look at bubble sort program, for example, we notice that the used new constructs form 39% of the whole program constructs while old constructs form 61%, however if we looked at the percentage of

errors in Figure 5.4, we find that new constructs caused 8% of total errors in this program while the old constructs caused 92%. The same observation can be noticed for the other programs.

It was noticed that whenever the program becomes longer (number of syntax lines is higher), new constructs are used more and in higher percentage within the syntax. This can be observed from Figure 5.5, when we move to right on the program's axis, the percentage of the new used constructs increases as programs become longer and more complicated. New constructs usage percentage starts with 29% in the Sum program and ends with 40% in the String program.

This leads to another observation: the new constructs were used in higher percentage in longer programs while they generated fewer errors, meaning the total error count in the programs decreased due to new constructs usage. Figure 5.4 shows that the error percentage generated by new constructs decreased starting from MinMax program which had 11% of errors to the String program, which had 7% only. If we match this to the new constructs usage percentage in Figure 5.5, then it is clear that new constructs will decrease the errors as their usage becomes higher and generates fewer errors. This will cause the total error count in the whole program to decrease.

Results in Figure 5.4 show another indicator for error reduction: while students practice the new constructs more and more and become familiar with them through writing more programs, the errors generated by these constructs become less. This is clear from the generated error percentage in the last 3 programs (MinMax, BubbleSort, and String), that the errors started from 11% and decreased to only 7%.

Students were able to understand and use the new constructs in a short period time (1 week). These new constructs helped them reduce syntax errors in their programs

compared to their previous basic, but familiar knowledge in Java syntax, which they used in many classes but which generated a higher error percentage for them.

The final result from this part of the case study was that new constructs helped in reducing syntax errors in program writing.

5.4. Professionals' Semantic Extraction Case Study Results

In the part of semantic extraction executed within the professionals' case study set out to measure the new constructs set efficiency in semantic extraction, answers are collected and graded on a scale from one to three (1-3). One means the extracted semantic is far from the correct answer, three means the semantic is correct and accurate, values in this range vary based on answer accuracy.

We calculated the average answer grade for each program for all answers using the following equation:

$$A_{avg.} = (\sum A(1...n))/n$$

Where

A: The professional answer grade

n: number of participant professionals

To measure how accurate the answers are, we calculated the “*Accuracy Ratio*”. This ratio was used to show how close the $A_{avg.}$ for each program was to the complete accurate answer grade, which is 3. The equation used:

$$\text{Accuracy Ratio} = A_{avg.} / 3$$

The following table shows the results and accuracy ratio for each program (data is sorted based on program syntax construct type and accuracy ratio):

Table 5.4: Semantic extraction results

Programs	Constructs Type	A avg.	Accuracy Ratio
P8	Old Syntax	1.6	53%
P9	Old Syntax	2.5	83%
P3	Old Syntax	2.5	83%
P1	Old Syntax	2.6	87%
P5	Old Syntax	2.8	93%
P10	New Syntax	2.5	83%
P7	New Syntax	2.6	87%
P6	New Syntax	2.8	93%
P2	New Syntax	2.8	93%
P5	New Syntax	3	100%

From the results in Table 5.4, we concluded that the new constructs helped in extracting a more accurate program semantic than the old constructs. The new constructs results show the lowest accuracy ratio was 83%, and the highest was 100% with two programs having an accuracy of 93%. In the old constructs' program results, the lowest accuracy was 53%, which is much less than the lowest new constructs accuracy result, and the highest was 93%, not 100% as with the new construct highest accuracy.

It is important to note that in this part of the case study, professionals asked to extract the semantic of programs without any previous knowledge or overview about the new constructs while they had enough knowledge about the old constructs as all participants were Java developers. This is considered as a positive indicator for the effect of new constructs in semantic extraction, code understanding, and making the code more readable and less ambiguous.

5.5. Professionals' Interview Results

Results obtained from interview with professionals were a positive indicator that supported assumptions regarding the new syntactic constructs set proposed.

Results are summarized in the following table¹:

¹ (Answering scale: Totally Agree, Agree, Agree with Reservation, No Answer, Poor, Disagree, Totally Disagree)

Table 5.5: Professionals answers on interview questions

Question	Result
1- Do you believe that using the new constructs will save efforts in writing code especially in case of repetitive keywords (import, access modifiers...etc.) and shorter looping constructs?	Agree
2- Do you think that using new constructs will help in decreasing syntax errors as result from saving repetitive keywords and distinguish scope using different identifiers?	Agree
3- Do you agree that using new constructs will make the code debugging easier?	Totally Agree
4- Do you think that the code will be more readable using the new constructs?	Totally Agree
5- Are the new constructs can help in extracting the program semantic from just reading it with minimal execution efforts and without the need for executing it many times and debug it to understand its functionality?	Agree
6- Is it true that the new construct can help in producing programs with less number of code lines (shorter syntax)?	Totally Agree

The distributions of professionals' answers are shown in the following chart:

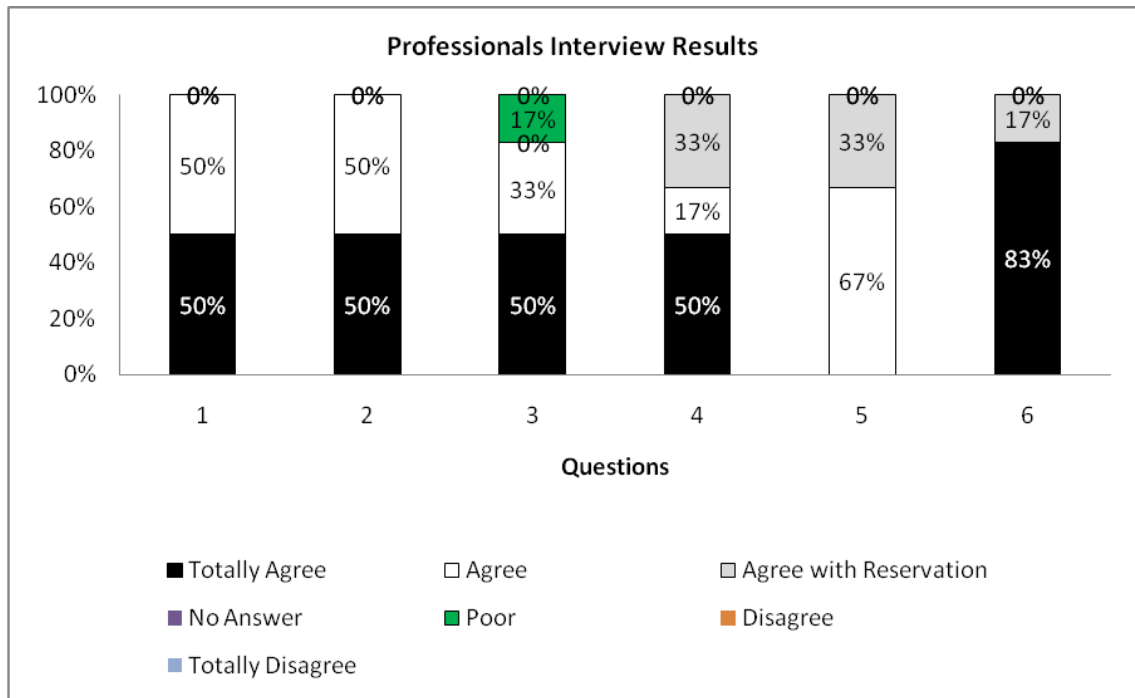


Figure 5.6: Professionals interview answers distribution

Results show that professionals expect that the new constructs set will help in making the code easier to debug, more readable, and with a fewer number of lines in a very high percentage.

In addition, the new constructs will help in a reasonable way in saving code writing efforts by reducing repetitive keywords, decreasing syntax errors, and extracting the program semantic in an easier and faster way.

5.6. Observations and Notes

Through the case study execution and results analysis, we concluded the following observations and notes:

- Using different scope determination symbols for each construct will help in reducing errors and making code more readable and less ambiguous. This observation was obtained from the results and from reviewing student errors, many of which were in scope determination symbols used in old syntax (`{}`, `()`).
- The effect of new syntactic sugar constructs on semantic extraction was higher from professionals' perspectives as they usually focus on semantic extraction more than students do.
- The new syntactic sugar constructs can be used to teach novice students how to write programs as these constructs help in reducing errors and code lines, and help students focus on the program's semantic and logic.
- The new syntactic sugar constructs help in making the constructs memorable and closer to natural language. They help in avoiding symbolic operators that can cause code ambiguity and errors.
- All participants recommended extending the new constructs set to include additional constructs.
- Feedback from participants on two of the new constructs raised issues that we answer below:

- Looping Construct "**times do ... end**": participants commented that there is no condition to exit the loop. As we explained, the purpose of this new loop constructs was to use it when the number of looping times is known and starts from 1, or looping over array entries. In other cases, the old for loop constructs can be used were it has a loop condition.
- Object instantiation construct "**myInstance = MyClass()**";": the feedback remark was that using this construct may cause mixing and ambiguity with the constructs used to call methods that return a value. Also, this construct will not work in the case of polymorphism (interface instance instantiated from implementing class) as the user cannot specify the parent interface. This point is correct and valid; we will try to enhance it to avoid these negative notes in future work. Users can still use the old constructs, especially in case of objects instantiation with polymorphism.

The conclusion from all results: using the new constructs set including syntactic sugar constructs in programming languages syntax can help in producing fewer syntactic errors and repetitive keywords, and in producing more readable, shorter (less number of code lines), easier to write and to debug code, with clearer code scope determination, and better semantic extraction and understanding. The new constructs set can be applied to any object oriented or imperative programming language even if it is general or domain specific. We recommend considering these results in the design of new programming languages' syntax as all results were positive indicators for enhancing syntax and code development.

Chapter Six

Conclusion and Future Work

Syntactic sugars were used in many programming languages to help programmers by offering coding simplicity and avoiding ambiguity, but were limited in specific areas and special needs in programming languages as obtained from the literature review. The approaches that followed to reduce syntax errors didn't consider using syntactic sugars for this purpose; instead they offered some kind of coding helpers and templates.

Our research was based on the idea of constructing new syntactic sugar constructs set and checking whether or not it is efficient in reducing syntax errors, reducing repetitive keywords, making the code more readable, easier to write and debug, and more understandable. The research focused on measuring the new set efficiency in simple and remote development environments where no IDE or coding helpers like templates and auto complete are available for both novice (students) and professionals programmers.

The methodology we used in this work based on forming a new syntactic sugar constructs set for the common abstract constructs. The set was extracted from the syntax of some well known programming languages in addition to a set of enhancements and constructs proposed by us. We conducted an exploratory case study with students and professional programmers to measure the efficiency of the new constructs set. The case study included program writing, semantic extraction, and interviews.

The results collected and analyzed from the case study demonstrated that the new constructs set with syntactic sugar showed positive, encouraging indicators that can help in reducing syntactic errors and repetitive keywords, making more readable, shorter, easier to write, easier to debug, and clear code, in addition to a better scope

determination, and more accurate semantic understanding. We recommend considering these results in the design of new programming languages' syntax.

6.1 Future Work

This work is an exploratory case study used to measure syntactic sugars' efficiency in code simplicity and error reduction. It can be considered as a starting point for more advanced research. In future work, we need to extend the set of analyzed programming languages and the constructs set to include new constructs because the current set is not enough to form complete programming language syntax. The enhancements can be applied on constructs other than the common abstracted constructs. This work can be extended to develop a completely new programming language with new syntax and a compiler based on results obtained from this work.

We need to improve the case study and make it more accurate and realistic, we need to increase experiment sample size to be larger and for a longer period of time. One way we intend to do this is to teach the constructs during a university course for several semesters in order to get more representative evaluation.

References

1. About Fancy, 2011, "*About Fancy*." (2011). Retrieved from Python official site: <https://github.com/bakkdoor/fancy/wiki>
2. About Python, n.d.: "*About Python*." (n.d.). Retrieved from Python official site: <http://www.python.org/about/>
3. About Ruby, n.d.: "*About Ruby*." (n.d.). Retrieved from Ruby official site: <http://www.ruby-lang.org/en/about/>
4. Altherr and Cremet, 2006: Altherr, P. & Cremet, V. (2006). "*Abstract Type Constructors for Java-like Languages*", Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/similar?doi=10.1.1.90.6424&type=ab>
5. Arsham, 1994: Arsham, H. (1994). "*Questionnaire Design and Surveys Sampling*," Retrieved from <http://home.ubalt.edu/ntsbarsh/stat-data/Surveys.htm>
6. Batista and Vieira, 2007: Batista, T. & Vieira, M. (2007). "*RE-AspectLua - Achieving Reuse in AspectLua*," *Journal of Universal Computer Science*, 13 (6), 786-805, 2007.
7. Bezault, 1999: Bezault, E. (1999). "*Eiffel: The Syntax*," Retrieved from <http://www.gobosoft.com/eiffel/syntax/>
8. Bolton, n.d.: Bolton, D. (n.d.). "*All about the C# Programming Language*," Retrieved from: <http://cplusplus.about.com/od/introductiontoprogramming/p/profileofcsh.htm>
9. Borgatti, 1996: Borgatti, S. P. (1996). "*Principles of Questionnaire Construction*," Retrieved from <http://www.analytictech.com/mb313/princip1.htm>
10. Collberg, 2005: Collberg, C. (2005). Chapter one: Introduction. "*Principles of Programming Languages*," University of Arizona.
11. COBOL, n.d.: "*COBOL*". (n.d.). Retrieved January, 7, 2011 from <http://en.wikipedia.org/wiki/COBOL>
12. Cheah, 2005: Cheah, A. J. (2005). "*Snapshot of the State of Adoption of IT in the Profession*," Retrieved from: http://www.pam.org.my/library/pam_it_questionnaire.pdf
13. Chiba, 1996: Chiba, C. (1996). "*OpenC++ Programmer's Guide for Version 2*," Xerox PARC Technical Report.
14. Chinchani et al., 2003: Chinchani, R., Iyer, A., Jayaraman, B., & Upadhyaya, S. (2003). "*Insecure Programming: How Culpable is a Language's Syntax*,"

- Proceedings of the 2003 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY, 158-163.
15. Clarke, 2001: Clarke, S. (2001). "*Evaluating a new programming language*," In 13th Workshop of the Psychology of Programming Interest Group, 275–289.
 16. Dolstra, 2001: Dolstra, E. (2001). "*First Class Rules and Generic Traversals for Program Transformation Languages*," Thesis, Utrecht University, 2001.
 17. Eiffel Programming Language, n.d., a:
(<http://www.computernostalgia.net/articles/EiffelProgrammingLanguage.htm>)
 18. Freeman and Pryce, 2006: Freeman, S. & Pryce, N. (2006). "*Evolving an Embedded Domain-Specific Language in Java*," OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.
 19. Freeman et al., 2004: Freeman, S., Pryce, N., Mackinnon, T., & Walnes, J. (2004). "*Mock Roles, not Objects*," Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, October 24-28, 2004, Vancouver, BC, CANADA. doi: 10.1145/1028664.1028765
 20. Fritchie, 2003: Fritchie, S. L., (2003). "*A Study of Erlang ETS Table Implementations and Performance*," Proceedings of the 2003 ACM SIGPLAN workshop on Erlang, 43-55, August 29-29, 2003, Uppsala, Sweden. doi:10.1145/940880.940887
 21. Fruhwirth, 2007: Fruhwirth, C. (2007). "*Liskell Haskell Semantics with Lisp Syntax*", Unpublished research, Retrieved from <http://clemens.endorphin.org/ILC07-Liskell-draft.pdf>.
 22. Galloway, 1997: Galloway, A. (1997). "*Questionnaire Design & Analysis*," Retrieved from <http://www.tardis.ed.ac.uk/~kate/qmcweb/qcont.htm>
 23. Getting Started with Visual C#, n.d.: "Getting Started with Visual C#." (n.d.). Retrieved January, 7, 2011 from Microsoft official website: <http://msdn.microsoft.com/en-us/vcsharp/dd919145.aspx>
 24. Golbreich and Wallace, 2008: Golbreich, C. & Wallace, E. K. (2008). "*OWL 2 Web Ontology Language: New Features and Rationale*," W3C Working Draft.
 25. Haskell syntactic sugar, 2010, a:
(http://en.wikibooks.org/wiki/Haskell/Syntactic_sugar)

26. Hidders et al., 2005: Hidders, J., Michiels, P., Paredaens, J., & Vercammen, R. (2005). "*LiXQuery: A Formal Foundation for XQuery Research*", SIGMOD Record, 34 (4), 2005.
27. Ierusalimschy et al., 1996: Ierusalimschy, R., Figueiredo L. H. D., & Filho, W. C. (1996). "*Lua - an extensible extension language*," Journal of Practice & Experience, 26 (6), 635–652.
28. JavaCC, 2010: JavaCC (2010). Parser/scanner generator for java, a: (<https://javacc.dev.java.net>)
29. Kannappan, 2010: Kannappan, R. (2010). "*New Features in Java 7 (Dolphin)*," [Web Log Post]. Retrieved from: <http://rajakannappan.blogspot.com/2010/05/new-features-in-java-7-dolphin.html>
30. Kelleher et al., 2002: Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., & Pausch, R. (2002). "*Alice2: Programming without Syntax Errors*," in Proceedings of the 15th Annual Symposium on the User Interface Software and Technology, Paris, France, October 2002.
31. Kirkegaard et al., 2004: Kirkegaard, C., Møller, A., & Schwartzbach, M. I. (2004). "*Static Analysis of XML Transformations in Java*," IEEE Transactions on Software Engineering, 30 (3), 181–192.
32. Koenig, 1985: Koenig, A. (1985). "*The Snocone Programming Language*," This paper originally appeared in the proceedings of the USENIX conference in Portland, Oregon in June 1985. Published in UNIX Vol. II.
33. Kominetz, 2007: Kominetz, J. (2007). "*Java 1.5: Gimme Some Syntactic Sugar, Baby!*," Retrieved from <http://kominetz.com/2007/09/13/java-15-gimme-some-syntactic-sugar-baby/>
34. Kummerfeld and Kay, 2002: Kummerfeld, S. K., & Kay, J. (2002) "*The neglected battle fields of Syntax Errors*," ACE '03 Proceedings of the fifth Australasian conference on Computing education, 20, 2003.
35. Laan, 1992: Laan, K. V. D. (1992). "*Syntactic Sugar*," Dutch TEX Users Group (NTG), AJ Schagen, the Netherlands.
36. Largillier, 2005: Largillier, T. (2005). "*Syntactic Sugar Support for Advanced C++ Constructs*," Technical Report (no0515).
37. Lévénéz, 2009: Lévénéz, E. (2009). "*Computer Languages History*," Retrieved from http://www.levenez.com/lang/lang_a4.pdf

38. Liu et al., 2007: Liu, D., Nakano, K., Hayashi, Y., Hu, Z., Takeichi, M., Morihata, A. & Xiong, Y. (2007). "*Bi-X Core: A General-Purpose Bidirectional Transformation Language*," Proceedings of the 24th JSSST Conference, No. 2C-3, Nara, Japan, September 2007.
39. LRDE, n.d.: LRDE.VAUCANSON. (n.d.), a: (<http://vaucanson.lrde.epita.fr/>)
40. Mageed, 2010: Mageed, A. E. (2010). "*The Evolution of the C# Language: The Impact of Syntactic Sugar and Language Integrated Query on Performance*," A thesis submitted to the Graduate Faculty of Auburn University, Auburn, Alabama.
41. McNamara and Smaragdakis, 2003: McNamara, B. & Smaragdakis, Y. (2003). "*Syntax sugar for FC++: lambda, infix, monads, and more*," Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03).
42. Meyer, 2001: Meyer, B. (2001). "*An Eiffel Tutorial*." Retrieved March, 18, 2009 from Eiffel software Site:
<http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial.pdf>
43. Mitchell, 2002: Mitchell, J. C. (2002). "*Concepts in Programming Languages*," 1st Ed., New York, NY: Cambridge University Press.
44. Mosses, 2005: Mosses, P. D. (2005). "*A Constructive Approach to Language Definition*", *Journal of Universal Computer Science*, 11(7), 1117-1134.
45. Nerlove, 2004: Nerlove, M. (2004). "*Programming Languages: A Short History for Economists*," *The Journal of Economic and Social Measurement*, 29, 189-203.
46. Odersky et al., 2006: Odersky, M., Altherr, P., Cremet, V., Dragos, L., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Spoon, L., & Zenger, M. (2006). "*An Overview of the Scala Programming Language*," (Technical Report LAMP-REPORT-2006-001). Retrieved from <http://www.scala-lang.org/docu/files/ScalaOverview.pdf>
47. OMG, 2005: Object Management Group (OMG) (2005). "*Unified Modeling Language Specification Version 1.4.2*," Retrieved from <http://www.omg.org/cgi-bin/doc?formal/05-04-01.pdf>
48. PITA, 2008: The Palestinian IT Association of Companies (PITA) (2008), "*Assessment of the Palestinian ICT Workforce*," Retrieved from http://www.pita.ps/newweb/pdfs/local_2008.pdf

49. Quick MBA, n.d.: Quick MBA (n.d.). "*Questionnaire Design*," Retrieved from <http://www.quickmba.com/marketing/research/qdesign/>
50. Russell et al., 2009: Russell, J., Russell, B., Pollacia, L., & Tastle, W. (2009). "*A Study of the Programming Languages Used in Information Systems and in Computer Science Curricula*," Proceedings of The Information Systems Education Conference (ISECON), 26, Washington DC.
51. Ruby from Other Languages, n.d., a: (<http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/>)
52. Saito and Igarashi, 2008: Saito, C. & Igarashi, A. (2008). "*The Essence of Lightweight Family Polymorphism*," *Journal of Object Technology*, 7(5), 67-99.
53. Sammet, 1996: Sammet, J. E., (1996). "*From HOPL to HOPL-II (1978 - 1993) 15Years of Programming Language Development, in History of Programming Languages*," Bergin, T. J., Gibson, R. G. (Eds), ACM Press, New York, NY, 16-23.
54. Süß, 2006: Süß, J. G. (2006). "*Sugar for OCL*," Proceedings of the 6th OCL Workshop at the UML/- MoDELS Conference, 240–251.
55. Sureau, 2010: Sureau, D. (2010). "*History of Programming Languages and Their Evolution*". Retrieved December, 17, 2010 from <http://www.scriptol.com/programming/history.php>
56. Survey Design, n.d., a: (<http://www.surveysystem.com/sdesign.htm>)
57. Teitelbaum and McIlrow, 1981: Teitelbaum, T. & McIlrow, M. (1981). "*The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*," *Communications of the ACM*, 24(9), 563-573, 1981.
58. The Go Programming Language, 2011, a: (<http://golang.org/>)
59. The Java Language, n.d.: "*The Java Language: An Overview*." (n.d.). Retrieved January, 7, 2011 from Java official website: <http://java.sun.com/docs/overviews/java/java-overview-1.html>
60. The Scala Programming Language, 2008: "*The Scala Programming Language*." (2008). Retrieved from Scala official site: <http://www.scala-lang.org/node/25>
61. Timeline of programming languages, 2011, a: (http://en.wikipedia.org/wiki/Timeline_of_programming_languages)
62. Watt and Findlay, 2004: Watt, D. A., & Findlay, W. (2004). "*Programming Languages Design Concepts*," 1st Ed., England: John Wiley & Sons Ltd.
63. Why Cobra, 2010, a: (<http://cobra-language.com/docs/why/>)

64. Why Fantom, 2011, a: (<http://fantom.org/doc/docIntro/WhyFantom.html>)

Appendices

Appendix 1: Programming languages' extracted constructs sheet.

Construct / language	Ruby	Eiffel	Java	Python	C#
Class definitions symbols	<code>class className</code>	<code>class className</code> <i>// other features special in Eiffel for class definition is discard</i>	<code>accessModifier class</code> <code>className</code>	<code>class ClassName:</code>	<code>accessModifier class</code> <code>className</code>
Class scope	<code>class className</code> <code>end</code>	<code>class className</code> <code>end</code>	<code>class className</code> { }	No Symbols used to determine the scope, instead it depend on indenting of class methods and variables, it must be indented like: <code>class ClassName:</code> <statement-1> . <statement-N>	<code>class className</code> { }
Method signature definitions	<code>def</code> methodName{([attribute[s]])}	<code>feature</code> [[([attribute/s:Type()]] methodName [:returnType] is	<code>accessModifier [static]</code> <code>returnType</code> methodName{([attribute e[s]]) [throwException]	<code>def</code> methodName{([attribut e[s]):	[accessModifier] [static] returnType methodName{([attribut e[s])]
Method scope symbols	<code>def</code> methodName{([attribu te[s]])} <code>end</code>	<code>feature</code> [[([attribute/s:Type()]] methodName [:returnType] is <code>do</code> <code>end</code>	<code>accessModifier [static]</code> <code>returnType</code> methodName{([attribut e[s]]) [throwException] { }	No Symbols used to determine the method scope, instead it depends on indenting of method body, it must be indented, to start new method, it go back the indent and repeat the same for the new method like: <code>def</code> methodName{([attribut e[s]): <statement-1> . <statement-N> <code>def</code> methodName2{([attribu te[s]): <statement-1> . <statement-N>	[accessModifier] [static] returnType methodName{([attribut e[s])] { }
Inheritance construct	<code>class className <</code> parentClassName	<code>class className inherit</code> parentClassA [parentClassB] <i>// multiple inheritance possible</i>	<code>class className</code> <code>extends</code> parentClassName	<code>class</code> className(parentClass Name):	<code>class className :</code> parentClassName
Constructor definition	In Ruby constructors are defined using a method called initialize , it is used to initialize the instance instead of construction it: <code>def</code> initialize{([attribute s]])}	<code>class classNameHELLO</code> <code>create</code> <i>methodName</i> <code>feature</code> <i>methodName</i> is <code>do</code> ... <code>end</code> <code>end</code>	<code>accessModifier</code> className{([attribute[s]])}	<code>def</code> <code>__init__(self[attributes, ...]):</code>	<code>accessModifier</code> className{([attribute[s]])} [:base()]
Super / Parent class calling	<code>def</code> initialize{([attribute s]])} <code>super</code> {([attribute[s])}; <code>End</code>	<code>precursor</code> {([attribute[s])]	<code>accessModifier</code> className{([attribute[s]])} { <code>super</code> {([attribute[s])}; }	<code>super</code> () [.method{([attri bute[s])}]	<code>accessModifier</code> className{([attribute[s]])} :base{([attribute[s])] { }

Package / module definition	<p>No direct package definition in classes, instead it depends on file system folder and packages and libraries defined externally. It used modules which declare all the classes in one module depending on namespaces. This could be a problem as all classes will be in one single file</p> <p>module P ... end.</p> <p>You can use modules and save them in file with .rb extension, and you can call them using require construct, also if the files are in directory structure, then the structure become part of package or module name.</p>	<p>There are no package declaration or calling in Eiffel, It depends on multiple inheritance to get access to other classes that form libraries and packages</p>	<p>package pack.age.name;</p>	<p>The module name automatically depend on the file name contains the module and code. If you have a file called mod1.py, the module name is mod1.</p> <p>For packages it is the same but it uses nested hierarchal directory structure like:</p> <p>/pack1/spack1/sub1/mod.py</p> <p>The package is pack1.spack1.sub1</p> <p>and module is mod</p> <p>each directory must have an empty file called __init__.py</p>	<p>namespace userDefinedNamespace { full class[es] body goes here }</p>
Libraries and packages calls construct	<p>require 'filename'</p>	<p>Using same multiple inheritance construct</p>	<p>import pack.age.name;</p>	<p>Load certain module: import pack.age.name.module</p> <p>Load set of module from the same package: from pack.age.name import module1, module2</p> <p>Load all modules from the same package: from pack.age.name import *</p>	<p>using pack.age.name;</p>
Libraries and packages repetition	<p>require 'filename.rb' require '././extensionName' require 'extensionName'</p>	<p>Multiple inheritance construct</p>	<p>import pack.age.name1; import pack.age.name2; import pack.age.name3;</p>	<p>Load set of modules from the same package: from pack.age.name import module1, module2</p> <p>Repeat to load from different packages import pack.age.name1.module1 import pack.age.name2.module2</p>	<p>using pack.age.name1; using pack.age.name2; using pack.age.name3;</p>
Attributes access modifiers	<ul style="list-style-type: none"> • A local variable (declared within an object) name consists of a lowercase letter (or an underscore) followed by name characters (sunil, _z, hit_and_run). • An instance variable (declared within an object always "belongs to" whatever object self refers to) name starts with sign ("@") followed by an upper- or lowercase letter, optionally followed by name characters (@sign, @_, @Counter). • A class variable (declared within a class) name starts with two signs 	<p>For methods, you can define a parameter to be local in that method by adding local key word in method declaration like</p> <p>feature deposit(sum: INTEGER) is -- Add sum to account. local new: AMOUNT do</p>	<p>private type attName; public type attName; protected type attName;</p>	<p>Everything in Python is public by default and accessible anywhere. To make an attribute or variable private use two underscores “__” before the varName. No protected access modifier in python.</p> <p>Example: def __attName</p>	<p>public Access is not restricted. protected Access is limited to the containing class or types derived from the containing class. internal Access is limited to the current assembly. protected internal Access is limited to the current assembly or types derived from the containing class. private Access is limited to the containing type.</p> <p>private type attName; public type attName; protected type attName; internal type attName; protected internal type attName</p>

	<p>("@" followed by an upper- or lowercase letter, optionally followed by name characters (@@sign, @@_, @@Counter). A class variable is shared among all objects of a class.</p> <ul style="list-style-type: none"> • A Global variables start with a dollar sign ("\$\$") followed by name characters. A global variable name can be formed using "\$-" followed by any single character (\$counter, \$COUNTER, \$-x). Ruby defines a number of global variables that include other punctuation characters, such as \$, and \$-K 				
Methods access modifiers	<p>By default, all methods in Ruby classes are public - accessible by anyone. If desired, this access can be restricted by public, private, protected object methods. It is interesting that these are not actually keywords, but actual methods that operate on the class, dynamically altering the visibility of the methods. As a result of that fact these 'keywords' influence the visibility of all following declarations until a new visibility is set or the end of the declaration-body is reached. Private can be in three ways:</p> <p>1- private/protected: all methods that follow will be made private: not accessible for outside objects</p> <pre>class className def m1 # this method is public end protected def m2 # this method is protected end private def m3 # this method is private end end</pre> <p>2- assign a method to be private after declaration (private keyword with an argument):</p>	N/A	<pre>private [static] returnType methodName public [static] returnType methodName protected [static] returnType methodName</pre>	<p>Methods are the same as attributes, default is public and private defined using “_” like:</p> <pre>def __methodName([attribute[s]]):</pre>	<pre>private [static] returnType methodName public [static] returnType methodName protected [static] returnType methodName internal [static] returnType methodName protected internal [static] returnType methodName</pre>

	<pre> class Example def methodA # public by default end def methodP end def m2 end def m3 end protected :m2, m3 # this method now is proteted private :methodP end now methodP became private 3- Note for class methods (those that are declared using def ClassName.method _name), you need to use another function: private_class_meth od: private_class_metho d :new Set new to private Protected: Private methods in Ruby are accessible from children (like protected in Java and C++). You can't have truly private methods in Ruby; you can't completely hide a method (no actual protected) </pre>				
Methods and attributes organization in class	<p>Methods can be organized in section of access modifiers specially for private level, when you set private and follow it with a set of methods, then all of these methods will be private until another modifier is declared or end of class (this is option, you can declare on method level). Attributes are not included in this. See previous constructs</p>		any organization	any organization	any organization
Object instantiation	<pre> [instanceName =] className.new ([[attributes]]) These attributes in initiaze method of exist </pre>	<pre> objInstName: ObjectName create objInstName.make ([[attribute[s]]) </pre>	<pre> ObjectType objInstanceName = new ObjectType ([[attribute[s]]) </pre>	<pre> objInstanceName = ClassName ([[attribute[s]]) </pre>	<pre> ObjectType objInstanceName = new ObjectType ([[attribute[s]]) </pre>
Object message / method calling	<p>Parentheses are usually optional with a method call. These calls are all valid:</p> <pre> instanceName.methodName instanceName.methodName{0} instanceName.methodName [[[attribute[s]]]] instanceName.methodName attribute[s] </pre>	<pre> objInstanceName.methodName [[[attribute[s]]]] // paranthese are removed on no attributes available </pre>	<pre> objInstanceName.methodName [[[attribute[s]]]]) </pre>	<pre> objInstanceName.methodName [[[attribute[s]]]]) </pre>	<pre> objInstanceName.methodName [[[attribute[s]]]]) </pre>

<p>Exception handling constructs used</p>	<pre> begin rescue OneTypeOfException [rescue AnotherTypeOfException else ] [ensure] end </pre> <p>The code in an else clause is executed if the code in the body of the begin statement runs to completion without exceptions. If an exception occurs, then the else clause will obviously not be executed.</p> <p>If you need the guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised then the ensure clause can be used. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. The ensure block will always run.</p>	<p>Exception handling done via using rescue keyword without exception type specification, it usually added at the end of method or other places. A retry instruction is only permitted in a rescue clause; its effect is to start again the execution of the routine, without repeating the initialization of local entities</p> <p>rescue Exception handling statements retry</p>	<pre> try { } catch(ExceptionType excepAlias) { } [more catches] [finally] { } } </pre>	<pre> try: ...statements except ExceptionType: ...handling statements [more except block for different exceptions types] [else: // executed when no exception happen ...statements] [finally: //always executed ...statements] </pre>	<pre> try { } catch(ExceptionType excepAlias) { } [more catches] [finally] { } } </pre>
<p>Exception handling variables scope</p>	<p>The variable are accessible in any scope even the were dealed within the begin rescue end block, they will be accessible in rescue, else, ensure and after end blocks. Slove Java problem</p>	<p>In rescue: all method attributed defined before are accessible in rescue block</p>	<p>In catch and finally blocks, attributes defined in try block are not accessible</p>	<p>Attributes defined in the try are accessible in except, else and finally sections</p>	<p>In try: all method and class predefined attributes, in catch and finally attributes defined in try block are not accessible</p>
<p>Control statements scope symbols</p>	<pre> controlStatementKeySignature (if, for,...) ... end </pre> <p>in case of each, upto, times... loops: The Ruby standard is to use braces for single-line blocks and do- end for multi-line blocks. Keep in mind that the braces syntax has a higher precedence than the do..end syntax</p>	<pre> controlStatementSignature ... end </pre> <p>where controlStatementSignature in { if, loop, ... }</p>	<pre> { } </pre>	<p>No Symbols used to determine the scope, instead it depend on indenting of statements and variables</p>	<pre> controlStatementSignature { } </pre>

Object Collections iteration	<p>Using do..end: method_call '['' expr...''] expr...end</p> <p>using Carle braces: method_call '['' '['' expr...''] expr...'']</p> <p>method_call: list name with .each construct, integerVal.times, integerVal.upto(integerVal) construct or any iteration construct used from the object defined</p>	<p>Iteration in Eiffel depend on agents:</p> <p>listName.do_all (agent (variableName: TYPE) do</p> <p>do_something_with variableName end)</p>	<p>Iterator object with while loop and casting: Iterator iter = list.iterator(); while(iter.hasNext()) { ObjectType obj = ObjectType iter.next(); }</p> <p>OR</p> <p>Generics for loop: List<ObjectType> objects = inst.getObjects(); for(ObjectType obj : objects) { }</p>	<p>for objRef in ObjectList: ...</p>	<p>Using for loop:</p> <p>for (IEnumeration ienInstName = ((CollectionType)collectionName).GetEnumerator(); ienInstName.MoveNext() ;) { ienInstName.Current; // current object in list }</p> <p>Using while loop:</p> <p>IEnumeration ienInstName = ((CollectionType)collectionName).GetEnumerator(); while (ienInstName.MoveNext()) { ObjectType objVar = ienInstName.Current; }</p> <p>Using foreach:</p> <p>foreach (ObjectType objVar in collectionName) { ... }</p>
For loop	<p>Looping can be done in many types:</p> <p>For loop: for var in collection # var refers to an element of the collection ... end</p> <p>for num in (4..6) puts num end</p> <p>Upto: 0.upto(10) do i ... end</p> <p>integerVal.times: 10.times do i ... end</p>	<p>from initialization until exit [invariant inv variant var] loop body end</p>	<p>for ({initialization}; {exit condition}; {control variable}) { }</p>	<p>for varName in [range(10, 0, -1)] list </p>	<p>for ({initialization}; {exit condition}; {control variable}) { }</p>
While loop	<p>while expr [do] ... end</p>		<p>while (condition) { }</p> <p>OR</p> <p>do { } while (condition);</p>	<p>while conditionalExpression: ... end</p>	<p>while (condition) { }</p> <p>OR</p> <p>do { } while (condition);</p>
If statement	<p>if expr [then] expr... [elseif expr [then] expr...] [else expr...] end</p> <p>You can use unless which is the opposite of if (if)</p>	<p>if condition then ... [else ...] end</p> <p>if condition then ... [elseif condition2 then ... end</p>	<p>if (conditionalExpression) { ... } [else if (conditionalExpression) { ... } end</p>	<p>if conditionalExpression: ... [elif conditionalExpression: ... else: ...] end</p>	<p>if (conditionalExpression) { ... } [elseif (conditionalExpression) { ... } [else if (conditionalExpression) { ... } end</p>

	<pre>unless expr [then] expr... [else expr...] end</pre>	<pre>... else ...] end</pre>			
swatch / case statement	<pre>case expr [when expr [, expr]...[then] expr..].. [else expr..] end</pre> <p>The case expressions are also for conditional execution. Comparisons are done by operator ==. Thus:</p> <pre>case expr0 when expr1, expr2 stmt1 when expr3, expr4 stmt2 else stmt3 end</pre>	<pre>inspect exp when v1 then inst1 when v2, v3 then inst2 ... else inst0 end</pre>	<pre>switch (expression) { case cond1: code_block_1 [break]; ... case condn: code_block_n [break]; default: code_block_default; }</pre>	<p>No switch statement exist in python, you can use some kind of dictionary data type for go around it</p>	<pre>switch (variable) { case val1: code_block_1 [break]; ... case valn: code_block_n [break]; default: code_block_default; }</pre>

Appendix 2: The designed and distributed questionnaire

Objective statement:

The purpose of this questionnaire is to find the best set of specific syntax constructs widely used in object oriented programming languages which developer consider easier to write and use in writing their programs and applications. Determining this set of constructs will offer syntactic constructs for programming languages designers and developers who use programming languages to reduce code writing syntax errors, and make the code more readable and less ambiguous which will lead to improve developers productivity.

So, we appreciate your cooperation in completing this questionnaire to help us in determining the best constructs set.

IMPORTANT: Please read the following notes before start answering the questionnaire:

- All your answers will be treated with confidentiality and used for research purpose only.
- This questionnaire consists of 19 questions distributed over 7 groups in 9 pages.
- Your answers will help us in our research and getting accurate results so please try to answer all questions carefully.
- Please read each question carefully and select the answer you see suitable.
- Each question can have only one single answer.
- It is preferable to be familiar with object oriented programming.
- Code syntax keywords and symbols used in the questionnaire are identified by **bold** font like “**class**” or “**{**”.
- *Italic* phrases like “*className*” are variables that can be any thing
- The “.....” or “<statement-N>” means any set of code syntax statements and constructs.
- The “[([attribute[s]])]” constructs and its similarities are used to express if any method / constructor can have attribute(s) or not.
- Constructs between “[...]” are optional.
- **accessModifier**: means specifying class/ method/ attribute access level as public, private, protected ... etc.

Group 1: Experience and skill

1.a) What is your gender?

- Male Female

1.b) What is your current profession?

- Student Member of academic staff Employee

1.c) What is your current resident location in Palestine?

- North Middle South

1.d) How long in years have you been in your profession?

- Less than 1 year 1 to 2 years 2 to 3 years 3 to 4 years
 4 to 5 years 5 years or more

Group 2: Class inheritance and instantiation:

2.a) Which of the following constructs you think is easier to write and more clear to express the single inheritance between two classes?

- `class className < parentClassName`
- `class className inherit parentClassName`
- `class className extends parentClassName`
- `class className(parentClassName):`
- `class className : parentClassName`

2.b) Suppose you want to define a constructor for a class, which syntax construct you prefer to used if all the following are available for you?

- Always use a method called **initialize** like:

initialize([attribute[s]])

- Select any method from the class to be the constructor by assigning its name (“*methodName*”) at the beginning of the class after **create** keyword like:

class className **create** methodName

- Use method with the same class name without return type:

accessModifier className([attribute[s]])

- Use a method in the following fixed format:

def __init__(self[attributes, ...]):

2.c) From your point of view, what is the best way to write object instantiation statement in order to create new class instance?

- [objectInstanceName =] `className.new`([attributes])
- `objectInstanceName: className create objectInstanceName.make`([attribute[s]])
- `objectInstanceName = className`([attribute[s]])
- `className objectInstanceName = new className`([attribute[s]])

Group 3: Methods definition, and calling (object messages):

3.a) Which one of the following constructs you think is easier to write method signature?

- `def methodName`([attribute[s]]):
 - ➔Note: def is reserved word here to declare the method
- `def methodName`([attribute[s]])
- `feature` [[([attribute/s:Type[])] `methodName[:returnType]`] **is**
 - ➔Note: feature is reserved word here to declare the method
- `accessModifier` [static] `returnType` `methodName`([attribute[s]])
- [accessModifier] [static] `returnType` `methodName`([attribute[s]])
[throwException]

3.b) What is the easier and simpler construct for you to call a method (send message to object) from the following?

- *objectInstanceName.methodName*
 →Note: here no parameters / attributes are passed to the method
- *objectInstanceName.methodName()*
 →Note: here no parameters / attributes are passed to the method and parentheses are optional.
- *objectInstanceName.methodName [(attribute[s])]*
 →Note: here parameters / attributes are passed to the method and parentheses are optional.
- *objectInstanceName.methodName attribute[s]*
 →Note: here parameters / attributes are passed to the method and NO parentheses are used at all
- *objectInstanceName.methodName([attribute[s]])*
 →Note: here parameters / attributes are passed as optional to the method but the parentheses are always exist

Group 4: Control statements (if, for, iteration,...etc.) :

(* NOTE *): Following is explanation to some words and expressions that you will notice through questions in this group:

- 1-) **conditionalExpression:** the expression that hold for a condition like the one used in if statements, i.e.: (a >= 5)
- 2-) **var:** is any variable hold a value.
- 3-) **collection:** is a collection (list) of numbers, characters, objects...etc.
- 4-) **initialization:** loop control variable initialization like (i=0).
- 5-) **exit condition:** the condition ends the loop (i<10).
- 6-) **control variable:** the variable used to control the loop (like i, or j) and its adjustment (like i=i+1, or i=i-2).
- 7-) [...].... means that statements are optional and can be repeated as needed

4.a) Suppose you want to write FOR LOOP in your code, which of the following you think is easier for you to write?

<input type="checkbox"/> Using: for var in collection <statement-1> ... <statement-N> end	<input type="checkbox"/> Using: from initialization until exit condition loop <statement-1> ... <statement-N> end
<input type="checkbox"/> Using Upto construct with the start number directly: 1.upto(10) do <statement-1> ... <statement-N> end	<input type="checkbox"/> Using ranges: for i in (1..10) <statement-1> ... <statement-N> end
<input type="checkbox"/> Using a number directly with times construct:	<input type="checkbox"/> Using: for (initialization; exit condition; control

<pre> 10.times do <statement-1> ... <statement-N> end </pre>	<pre> variable) { <statement-1> ... <statement-N> } </pre>
--	---

4.b) The easiest way to write IF statement is?

<pre> <input type="checkbox"/> if conditionalExpression [then] <statement-1> ... <statement-N> [elsif conditionalExpression [then] <statement-1> ... <statement-N>]... [else <statement-1> ... <statement-N>] end </pre> <p>// then is optional</p>	<pre> <input type="checkbox"/> if conditionalExpression then <statement-1> ... <statement-N> [elsif conditionalExpression then <statement-1> ... <statement-N>]... [else <statement-1> ... <statement-N>] end </pre>
<pre> <input type="checkbox"/> if conditionalExpression: <statement-1> ... <statement-N> [elif conditionalExpression: <statement-1> ... <statement-N>]... [else: <statement-1> ... <statement-N>] </pre>	<pre> <input type="checkbox"/> if (conditionalExpression) { <statement-1> ... <statement-N> } [else [if (conditionalExpression)] { <statement-1> ... <statement-N> }] </pre>

4.c) The simplest construct that can be used to write selection (switch, case...etc.) statement is ?

- case var inspect var switch (var)

4.d) To iterate over a collection with objects in it, which construct you prefer to write in your code?

<pre> <input type="checkbox"/> Using .each...do...end: collectionName.each do collItem <statement-1> ... <statement-N> end // collectionName is the name of collection, </pre>	<pre> <input type="checkbox"/> Using agents: collectionName.do_all (agent (collItem: TYPE) do <statement-1> ... <statement-N> end) </pre>
---	--

<p>while <i>collItem</i> is the variable that will hold the current collection item in the iteration to access it.</p>	
<p>□ Using Iterator object with while loop and casting:</p> <pre> Iterator iter = <i>collectionName.iterator()</i>; while(<i>iter.hasNext()</i>) { ObjectType collItem = (ObjectType) <i>iter.next()</i>; <statement-1> ... <statement-N> } </pre>	<p>□ Using for... loop</p> <pre> for(ObjectType collItem: <i>collectionName</i>) { <statement-1> ... <statement-N> } </pre>
<p>□ Using for...in construct:</p> <pre> for <i>collItem</i> in <i>collectionName</i>: <statement-1> ... <statement-N> </pre>	<p>□ Using for...Enumerator construct:</p> <pre> for (IEnumerator ienInstName = ((CollectionType)<i>collectionName</i>) .GetEnumerator();<i>ienInstName.MoveNext()</i>;) { ObjectType objVar = <i>ienInstName.Current</i>; <statement-1> ... <statement-N> } </pre>
<p>□ Using while...Enumerator construct:</p> <pre> IEnumerator ienInstName = ((CollectionType) <i>collectionName</i>) .GetEnumerator(); while (<i>ienInstName.MoveNext</i>) { ObjectType objVar = <i>ienInstName.Current</i>; <statement-1> ... <statement-N> } </pre>	<p>□ Using foreach construct:</p> <pre> foreach (ObjectType collItem in <i>collectionName</i>) { <statement-1> ... <statement-N> } </pre>

Group 5: Exception handling and variables scopes:

5.a) In some object oriented languages, when a programmer define an attribute or variable within the rescued block (try, begin, or what ever block), these variable remain accessible in the exception handling blocks (catch, rescue...etc.) and even after the whole exception handling block while other languages prevent accessing these variables outside the rescue block even in the handling blocks (catch, rescue...etc.), to have a access to these variables you have to define them before the whole exception handling block, so which form of these you prefer to use?

<p>□ Variables accessible any where:</p> <pre> begin int a = 5 print(a) // a is accessible rescue <i>OneTypeOfException</i> print(a) // a is accessible [rescue <i>AnotherTypeOfException</i> print(a) / a is / accessible [else print(a) // a is accessible]... [ensure print(a) // a is accessible] end print(a) // a is accessible </pre>	<p>□ Variables access limited by exception handling block:</p> <pre> try { int a = 5; // print(a) // a is accessible } catch(<i>OneTypeOfException</i> excepAlias) { print(a) //ERROR: a is not accessible } [finally { print(a) //ERROR: a is not accessible }] print(a) //ERROR: a is not accessible // you must declare a before try block to be // accessible like this: int a = 5; try { <statement-N> } </pre>
--	---

Group 6: Packing, modules calling:

In many of object oriented programming languages, classes can be grouped together in some kind of namespaces or packages where these classes can be reused or form a library to be used in other applications through calling and instantiating classes in these packages and libraries. Depending on this, try to answer the following questions.

6.a) What is the best and simplest way to write package / module calling construct in a class?

- Using a keyword like **import**, **using**, or **require** before package name like:

```

require 'moduleNameSpace'
import my.package.name;
using my.package.name;

```
- Using phrase: **from** my.package.name **import** *
- Use multiple inheritance: **class** myClass **inherit** classA, classB

6.b) In case you want to load a set of different classes from different packages / namespaces, what is the simplest construct to use?

➔ Suppose you want to call “*classA*” from package “**my.package.name1**” and “*classB, classC*” from **my.package.name2**

□ Using the same keyword (import, using, or require) in front of each package and repeat whenever new package is called like using import:

```
import my.package.name1;  
import my.package.name2;
```

□ Using one of **import/using/require** construct with repeating packages' names only without repeating the keyword:

Suppose we used import then:

```
import:  
my.package.name1;  
my.package.name2;
```

□ Using “**from...import**” selective constructs:

```
from my.package.name1 import classA  
from my.package.name2 import classB, classC
```

OR

```
from my.package.name2 import *; // load all classes in this package
```

Group 7: Methods and attributes (variables) access modifier definition:

In this section we try to investigate the way to define access modifier for attributes (variables) and methods in a class. We mean by access modifier that how attributes and methods in a certain class are reachable from other classes and modules. Access modifier can be private where only accessible within the same class, or protected so each class in the same package or inherits this class can access them, or it can be public where other classes (even classes not in the same package and without inheritance relation) can access them.

*** Attributes and variables:**

7.a) To define an attribute / variable access modifier (“private”, “protected”, “public”), which of the following is the easiest way to do that?

□ Depending on the programming language default when no access modifier keyword is specified before the attribute / variable name (i.e. no keyword means private), other access modifiers rather than the default must be specified using their keywords.

□ Depending on attribute / variable name letters and special characters:

```
varName      → lowercase letters means private  
_varName     → start with one or two underscore means private  
$varName     → start with dollar sign means public  
$-varName    → start with dollar sign followed by “-“ means public
```

□ Depending on specifying the access modifier keyword before the attribute / variable name like:

```
local varName      → means private  
private varName    → means private  
protected varName → means protected  
public varName     → means public
```

7.b) Suppose you want to define many attributes / variables with the same access modifier type like “private”, which of the following is the easiest way write these constructs?

□ Through repeating the access modifier key word with each attribute / variable:

```
private integer varName
```

```
private double varName2  
private string varName3
```

- Or using only one access modifier key word followed by all attributes / variables:

```
private:  
integer varName  
    double varName2  
string varName3
```

*** Methods:**

7.c) To assign method access modifier (“private”, “protected”, “public”), which of the following is the easiest way to do that?

- Depending on the programming language default when no access modifier keyword is specified before the method name (i.e. no keyword means private), other access modifiers rather than the default must be specified using their keywords.

- If method name start with **special characters** that define it access modifier like:

```
__methodName([attribute[s]])      → two under score means private  
_methodName([attribute[s]])      → one under score means protected  
methodName([attribute[s]])        → nothing means public
```

- Using access modifier keyword before method signature:

```
private methodName  
protected methodName  
public methodName
```

- At the end of the class, specify which methods to be private, protected, public:

```
class Example  
def methodA  
end  
  
def methodB  
end  
  
def methodC  
end  
// here define methods access modifier  
private: methodA  
protected: methodB  
public: methodC  
end
```

- Through dividing the class into zones for access modifiers where any method declared within that zone then it will have its access level:

```
class Example  
// this is public zone so any method follow is considered as public until  
another zone start or end of class reached  
    def methodAPublic  
    end  
  
// now public zone end and protected start, any method follow is considered protected
```

```
protected:  
    def methodCProtected  
    end
```

// now protected zone end and private start, any method follow is considered private

```
private:  
    def methodEPrivate  
    end  
end
```

- Thank You -

Appendix 3: Selected programming languages brief description

Following is a brief overview about the programming languages choose to extract syntax constructs from. Historical overview is available in chapter 2. In this appendix we try to focus on languages syntax, its origin, and features:

1- Eiffel:

Eiffel is an object-oriented programming language which emphasizes the production of robust software. Its syntax is keyword-oriented in the ALGOL and Pascal tradition. Eiffel is strongly statically typed, with automatic memory management (typically implemented by garbage collection).

With roots dating back to 1985, Eiffel is a mature language with development systems available from multiple suppliers. Despite this maturity and a generally excellent reputation among those who are familiar with it, Eiffel has failed to gain as large a following as some other object-oriented languages. The reasons for this lack of interest are unclear, and are a topic of frequent discussion within the Eiffel community (Meyer, 2001) (Bezault, 1999) (Eiffel Programming Language, n.d.).

2- Python:

Python is a remarkably powerful dynamic programming language that is used in a wide variety of application domains. Python is often compared to Tcl, Perl, Ruby, Scheme or Java. Some of its key distinguishing features include:

- very clear, readable syntax
- strong introspection capabilities
- intuitive object orientation
- natural expression of procedural code
- full modularity, supporting hierarchical packages
- exception-based error handling
- very high level dynamic data types
- extensive standard libraries and third party modules for virtually every task
- extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
- embeddable within applications as a scripting interface

Python lets you write the code you need, quickly. And, thanks to a highly optimized byte compiler and support libraries, Python code runs more than fast enough for most applications (About Python, n.d.).

3- Java:

The Java programming language and environment is designed to solve a number of problems in modern programming practice. Java started as a part of a larger project to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. When the project started, the authors of Java intended to use C++, but encountered a number of problems. Initially these were just compiler technology problems, but as time passed more problems emerged that were best solved by changing the language.

Java is simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language (The Java Language, n.d.).

Java syntax is mostly derived from C++, so we consider Java in the study to cover this PL family.

4- C#:

C# is a type-safe, object-oriented language that is simple yet powerful, allowing programmers to build a breadth of applications. Combined with the .NET Framework, Visual C# 2008 enables the creation

of Windows applications, web services, database tools, components, controls, and more (Getting Started with Visual C#, n.d.).

C# has a lot in common with java syntax, but it is much younger than Java and C++, in addition to many added new features and its star is rising quickly and starts to beat other famous languages like Java (Bolton, n.d.).

5- **Ruby:**

Ruby is a language of careful balance. Its creator, Yukihiro “matz” Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming. He has often said that he is “trying to make Ruby natural, not simple,” in a way that mirrors life.

Building on this, he adds:

 "Ruby is simple in appearance, but is very complex inside, just like our human body" (About Ruby, n.d.)

When you first look at some Ruby code, it will likely remind you of other programming languages you've used. This is on purpose. Much of the syntax is familiar to users of Perl, Python, and Java (among other languages), so if you've used those, learning Ruby will be a piece of cake (Ruby from Other Languages, n.d.).

Appendix 4: ICT professional population Calculation

(Work forces in Both PITA and non PITA companies)

Total ICT professionals = 3600

ICT professionals at PITA Companies = 633

ICT professionals at Non-PITA Companies = $3600 - 633 = 2967$

50.7% ICT professionals at PITA Companies work at software companies →

$50.7\% \times 633 = 320.9$ professionals at PITA Companies (This include not only programmer but other SW related jobs)

2.4% ICT professionals at Non-PITA Companies work at software companies →

$2.4\% \times 2967 = 71.2$ professionals at Non-PITA Companies (This include not only programmer but other SW related jobs)

→ Total population = $320.9 + 71.2 = 392.131$ (including nonprogrammers and Gaza which means larger population)

ICT Students Analysis from the Palestinian Higher Education System

In 2006/2007 the total number of bachelor's students enrolled in the traditional universities was 88,707 students, among them **5,678** students in the ICT field. (*NOT Including Al-Quds Open University*)

of ICT students in Gaza Universities = 1600

We will focus on students in west bank as we cannot reach Gaza:

→ Population = $5678 - 1600 = 4078$ student.

If we excluded Electrical engineering students (871 students) in west bank universities as they are not our target:

→ Population = $4078 - 871 = 3207$ students (in all ICT programs except electrical engineering).

Appendix 5: Users' feedback interview questions.

1- Do you believe that using the new constructs will save efforts in writing code especially in case of repetitive keywords (import, access modifiers...etc.) and shorter looping constructs?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

2- Do you think that using new constructs will help in decreasing syntax errors as result from saving repetitive keywords and distinguish scope using different identifiers?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

3- Do you agree that using new constructs will make the code debugging easier?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

4- Do you think that the code will be more readable using the new constructs?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

5- Are the new constructs can help in extracting the program semantic from just reading it with minimal execution efforts and without the need for executing it many times and debug it to understand its functionality?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

6- Is it true that the new construct can help in producing programs with less number of code lines (shorter syntax)?

- Totally Agree Agree Agree with Reservation No Answer
 Poor Disagree Totally Disagree

Appendix 6: Students' case study experiment programs.

1. Write a program to find the sum of numbers between 1 and 100, try to use new looping construct and new method definition if want to use different method.
2. Write a program that accept an array of integers in the constructor and do the following:
 - a. A method to find minimum number in this array.
 - b. A method to find maximum number in the array.
 - c. Save the array, minimum, maximum, and array in private class attributes.
 - d. Add 2 methods to return the minimum and maximum (encapsulation).
 - e. Try to use the new construct in looping, methods, attributes definition.
3. Write a program for Bubble sorting (swapping) where you have:
 - a. Main method that initialize the array with unsorted data, print it before sorting, call the sorting method, then print the results.
 - b. Method called bubbleSort that implement the sorting algorithm.
 - c. Private method that is used for swapping only.
 - d. Try to use new method and looping constructs.
4. The String in Java doesn't contain methods that return a list or tokens, print them, and show their count. So you are asked to extend the String object in new class and add the following:
 - a. A method get and parse the tokens from the string and save them in ArrayList of string.
 - b. A method to print the arraylist contains the tokens.
 - c. A method to print the count of tokens.
 - d. Notes: Use new inheritance construct, use new "each" construct to print the list of tokens. Use new method definitions, define the tokens list and the count as private attributes in the class. You can also use the try – catch construct with extended variable scope.

Appendix 7: Professionals' case study experiment programs.

Please read the following 10 short programs and try to conclude what they are doing (their semantic and functionality) and write your conclusion on the lines below each program.

1.

```
public class MainClass {
    public static void main(String[] arg) {
        int j = 10;
        int s = 0;

        int i = 1;
        for (; i <= j;) {
            s += i++;
        }
        System.out.println(s);
    }
}
```

2.

```
import java.util.ArrayList;

public class ExampleClass {
    public static void main(String[] args) {

        ArrayList vls = new ArrayList();

        3:times do i
            vls.add(i);
        end

        vls:each do x

            choose(x)
            {
                case 0:
                    System.out.println("i is 0");
                    break;
                case 1:
                    System.out.println("i is 1");
                    break;
                case 2:
                    System.out.println("i is 2");
                    break;
                default:
                    System.out.println("i grater than 2");
            }
        endEach
    }
}
```

3.

```
class A {
    char doh(char c) {
        System.out.println("doh(char)")
        return 'd';
    }
    float doh(float f) {
```

```

        System.out.println("doh(float)");
        return 1.0f;
    }
}

class B {}

class C extends A {
    void doh(B m) {
        System.out.println("doh(B)");
    }
}

public class DriverClass {
    public static void main(String[])
    {
        C b = new C();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new B());
    }
}

```

4.

```

class A
{
    private:
        String tp;

    public A(String aTp)
    {
        tp = aTp;
    }

    def myMethod()
    {
        return "This is a " + tp;
    }
}

class D -> A {
    private:
        String nm;
        String brd;

    public D (String aNm, String aBrd)
    {
        super("D");
        name = aNm;
        brd = aBrd;
    }

    def myMethod()
    {
        return "It's " + nm + " the " + brd;
    }
}

class MainDriver {
    public static void main(String args)
    {
        D d = D("Pop", "Hop");
        System.out.println("The data in d: " + d.myMethod());
    }
}

```

5.

```
public class Main {

    private final int UL = 10000;

    public void executeMe() {

        int i = 0;
        int pnc = 0;

        while (++i <= UL) {

            int il = (int) Math.ceil(Math.sqrt(i));

            boolean isP = false;

            while (il > 1) {

                if ((i != il) && (i % il == 0)) {
                    isP = false;
                    break;
                } else if (!isP) {
                    isP = true;
                }

                --il;
            }

            if (isP) {
                System.out.println(i);
                ++pnc;
            }

            System.out.println("occurrences: " + pnc);
        }

        public static void main(String[] args) {
            new Main().executeMe();
        }
    }
}
```

6.

```
class NewExample{
    public static void main(String[] args) {
        int arr[][]= {{1,3,5},{2,4,6}};
        System.out.println("size= " + arr.length);
        System.out.println("another size = " + arr[1].length);
        theMethod(arr);
    }

    def theMethod(int[][] inArr)
        int rs = inArr.length;
        int cs = inArr[0].length;

        1:times do i
```

```

        System.out.print("[");

        2:times do j
            System.out.print(" " + inArr[i][j]);
        end

        System.out.println(" ]");
    end

    System.out.println();
endef
}

```

7.

```

public class MainClass {
    public static void main(String[] arg) {
        int s = 0;

        10:times do i
            s += i;
        end

        System.out.println(s);
    }
}

```

8.

```

import java.util.*;

public class ClassB {

    public void executionMethod() {

        long a1 = System.currentTimeMillis();
        ArrayList vls = new ArrayList();

        for (int i = 0; i < 10; i++) {
            vls.add(i);
        }

        for (Integer vl : vls) {
            try {

                Thread.sleep(60);

            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

        long a2 = System.currentTimeMillis();
        float ts = (a2 - a1) / 1000F;
        System.out.println("Result: "+ Float.toString(ts));
    }

    public static void main(String[] args) {
        new ClassB().executionMethod();
    }
}

```

9.

```
class ClassA{
    public static void main(String[] args) {
        int arr[][]= {{4,5,6},{6,8,9}};
        int arr2[][]= {{5,4,6},{5,6,7}};

        System.out.println("Size 1= " + arr.length);
        System.out.println("Size 2= " + arr[1].length);
        int l= arr.length;

        System.out.println("Data 1 : ");
        for(int i = 0; i < l; i++) {
            for(int j = 0; j <= l; j++) {
                System.out.print(" "+ arr[i][j]);
            }
            System.out.println();
        }

        int m= arr2.length;
        System.out.println("Data 2 : ");
        for(int i = 0; i < m; i++) {
            for(int j = 0; j <= m; j++) {
                System.out.print(" "+arr2[i][j]);
            }
            System.out.println();
        }

        System.out.println("Operation Result: ");
        for(int i = 0; i < m; i++) {
            for(int j = 0; j <= m; j++) {
                System.out.print(" "+(arr[i][j]+arr2[i][j]));
            }
            System.out.println();
        }
    }
}
```

10.

```
import:
java.io.BufferedReader;
java.io.IOException;
java.io.InputStreamReader;

public class WConMain create start{

def start()

    boolean inputOk = false;
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

    double pnd = 0;
    while (!inputOk) {
        System.out.println("Enter Value:");
        try {
            pnd = Double.parseDouble(reader.readLine().trim());
            inputOk = true;
        } catch (NumberFormatException e) {
```

```

        System.out.println("Invalid, try again.");
    }
}
System.out.println(pnd+" equals "+getPndToK(pnd)+" k & " + getPndToGr(pnd) + " gr");

inputOk = false;
double on = 0;
while (!inputOk) {
    System.out.println("Enter Value:");
    try {
        on = Double.parseDouble(reader.readLine().trim());
        inputOk = true;

    } catch (NumberFormatException e) {
        System.out.println("Invalid, try again.");
    }
}
System.out.println(on + " equals " + getOToK(on) + " k & " + getOToGr(on) + " gr");

endif

def getPToK(double pnd)
    double k = pnd * 0.45359237;
    return (int)Math.floor(k);
endif

def getPndToGr(double pnd)
    double k = pnd * 0.45359237;
    return (k - getPndToK(pnd)) * 1000;
endif

def getOToK(double on)
    double k = on * 0.0283495231;
    return (int)Math.floor(k);
endif

def getOToGr(double on)
    double k = on * 0.0283495231;
    return (k - getOToK(on)) * 1000;
endif

public static void main(String[] args) {
    try
    {
        wcon = WConMain();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}

```

Appendix 8: Selected and enhanced constructs set detailed description.

1. Class Inheritance Construct:

Inheritance is main concept in objects oriented and forms a power full mechanism used to enable reusability between classes when they share the same properties or behaviors.

Inheritance works on linking classes and let a class get his properties from the parent class it inherit. This offer code reusability and maintenance.

Inheritance occurs between parent / child classes. The parent class is inherited by the child. If we have a class called "**ParentClass**" contains some attributes and methods is inherited by child class called "**ChildClass**", all the attributes and methods exist in **ParentClass** will be available in **ChildClass** (except private attributes and methods).

In our research we tried to express the inheritance relation between classes in simple constructs that is derived from UML notation used to.

We used the "->" expression to define the inheritance between two classes

```
class ChildClass -> ParentClass
```

In this construct, ChildClass will inherit ParentClass. The user can write it another simpler way using the colon ":" like:

```
class ChildClass:ParentClass
```

The original java inheritance construct still available:

```
class ChildClass extends ParentClass
```

2. Class Instantiation Construct:

To create an instance of a class and use it, we considered Python convention. The user can do it using simple construct without the need to specify the object type before the instance variable and no need to use "new" keyword.

Suppose we have a class called "**MyClass**" and want to create an instance from it called "**myInstance**", this can be done using the following construct:

```
myInstance = MyClass();
```

This is equivalent to Java construct:

```
MyClass myInstance = new MyClass();
```

3. Method Definition Construct:

Method definition within a class can be done using simple constructs that help in making the method definition, signature, and scope (block) determination much simpler and shorter.

To define a method use "**def ...endef**" construct:

```
def methodName()  
    methodBody.....  
endef
```

If method has arguments to be passed then they added after the method name surrounded by parentheses:

```
def methodName(int size, Object obj)  
    methodBody.....  
Endef
```

User doesn't have to specify the return type (if method return value) in the method signature, just add return statement at the end of the method body:


```

def methodName(int size, Object obj)
    int x = 5;
    return x;
endef

```

Method's access modifier is assigned using method's name as described in (2.12).

Using the "**def ...endef**" construct makes the code more clear and easier to track, debug, and less ambiguous the developer as he can determine the scope of method inner block easier when find endef keyword. The developer can determine and distinguish methods block from each other, their internal blocks, and class block scope because the class and other building block (if, for, while...etc) use curly braces "{ }" to determine their scopes. Using the "**def ...endef**" construct let methods' scope be more visible and distinguishable easily and help in determining missed curly braces "{ }" of class inner building blocks:

```

class ClassName
{
    def methodName()
        int a = 0;
        if( a == 0)
        {
            system.out.println("a is 0");
        }
        else
        {
            system.out.println("a is not 0");
        }
    endef

    def methodName2(int size, Object obj)
        if(size != 0)
        {
            for(int i=0; i<size; i++)
            {
                system.out.println("Object: "+ obj.toString());
            }
        }
    endef
}

```

We obtained this construct from Ruby and Python. We enhanced it by using "**endef**" for closing the method scope to make it unique, better scope determination, and distinguish it from other constructs like if statement, loops...etc. We ignored the return type of method body to be dynamically specified at runtime if return keyword is used.

4. Method Calling Construct (message passing):

Using the new enhancements, calling a method from class instance is done in many fixable ways. If the class "**ClassName**" defined in previous section (3) is instantiated and the user wants to call its methods to pass some message to the object and the method has no parameters, then he can call it directly from object instance name with or without parenthesis (parenthesis are optional):

```

instanceName = ClassName();
instanceName.methodName; // calling the method without parenthesis
instanceName.methodName(); // calling the method with parenthesis

```

But if the method has attributes or parameters to be passed, then the user can do it in two ways: with or without parenthesis (i.e. list parameters after method name).

Using the same example mentioned previously, calling method "**methodName2**" with two passed parameters ("**5**" and "**objInst**") can be as follows:

```
instanceName = ClassName();  
instanceName.methodName2(5, objInst); // calling the method with parenthesis  
instanceName.methodName2 5, objInst; // calling the method without parenthesis
```

This flexibility is obtained from Eiffel and Ruby.

5. Method Execution on Class Construction Construct (Constructor Like):

Constructors are usually executed when an instance is created from a class. They are used to initialize the object instance. In case the user wants to execute another class method on object instantiation without using constructors, no constructors/defaults constructors are available or not accessible as the case of singleton pattern, he can use the "**create methodName**" construct. The user add the keyword "create" followed by the name of the method he want to execute at the end of class definition signature (after inheritance and interface implementation constructs if they are exist), the executed method must be defined within the class or inherited from parent class and it will be executed just when the instance is created the method must has no parameters).

Suppose we have class called "**MyClass**" inherits another class called "**ParentClass**", and want to execute method called "**executeMeMethod**" defined in the class when instance is created, then:

```
class MyClass -> ParentClass create executeMeMethod  
{  
    def methodName(int a)  
        if( a == 0 )  
        {  
            system.out.println("a is 0");  
        }  
    endef  
  
    def executeMeMethod()  
        system.out.println("I'm executed when instance is create");  
    endef  
}
```

This construct is obtained from Eiffel.

6. Looping Construct:

Looping a block of statements number of times is common programming procedure that is extensively used. To loop block of statements, array index, or code block, a simple construct is proposed that reduce the looping variable declaration and help in specifying the loop construct scope (begin ...end) which make the code more readable and less ambiguous.

This constructs is "**times do end**", to iterate a set of statements for 5 times then just write:

```
5:times do  
    System.out.println("Hi, I'm looping...");  
end
```

If the user interested in getting the current loop index to refer for an array entry or use the index in code block, then just set an alias for the index after "do" keyword and refer to it within loop body:

```
5:times do x
    System.out.println("Value "+x+" in the array is: "+arr[x-1]);
end
```

Here "x" is the loop index that is used to get an entry in the array of integers "arr".

If user have the array and want to loop over it without knowing its length or getting it in a variable, he do this directly from array instance name followed by times construct. To loop over the integers array called intArr then:

```
int[] intArr = new int[]{1,11,111,1111,11111,111111};
intArr:times do i
    System.out.println(arr[i]);
End
```

The user can use a predefined variable for looping. Suppose a method has attribute called "size" and wants to use it for loop, then:

```
def methodName2(int size)
    size:times do
        System.out.println("Hi, I'm looping...");
    end
endef
```

Notice that using "times do end" construct help in better scope determination of code building blocks as it doesn't use curly braces "{ }" which help is debugging, minimizing curly braces matching error, and clearer code.

The basic idea of this construct is obtained from Ruby. And we enhanced on it by offering ability to loop over array entries directly use the array name in the loop.

7. Objects Collection Iteration:

To iterate over a collection of objects (list, map, set...etc.), we offered new construct that minimize writing the iteration block with options to for reference the current object in the collection. This construct is "each do endEach" construct.

To iterate over a collection called "myCollection" passed to a method we use the following construct:

```
collectionInstancename:each do collectionCurrentReferencedItem
    iteration body
endEach
```

Where:

- **collectionInstancename**: is the collection instance name to be iterated.
- **:each do**: reserved construct follow the collection name.
- **collectionCurrentReferencedItem**: a reference variable points to the current object in the collection.
- **endEach**: the iteration block closing phrase.

```
def showCollection(Collection myCollection)
    myCollection:each do ref
        System.out.println("Hi, I'm looping..."ref);
    endEach
```

```
        endEach
    endif
```

Using "**each do endEach**" construct help in better scope determination of code building blocks as it doesn't use curly braces "{ }" which help is debugging, minimizing curly braces matching error, and clearer code.

This construct is not obtained from other languages; it is suggested as enhancement to make code more readable and close to natural language.

8. If Construct:

"if" statement is exactly the same as construct exist in Java, C#, or C++ without changes.

9. Selection Construct:

The new selection construct we propose is the same as Java "switch" construct with one simple change that is using the phrase "choose" instead of "switch" to make the code more understandable and closer to natural language.

```
    choose(a)
    {
        case 1:
            System.out.println("One...");
            break;
        case 2:
            System.out.println("Two...");
            break;
        case 3:
            System.out.println("Three...");
            break;
        default:
            System.out.println("No Number...");
    }
```

10. Multiple Packages / Modules Calling Construct:

Developers use predefined package in their code as reusable libraries using packing techniques and collect their classes in certain name spaces for future reusability as predefined packages and modules.

To call a package make is available within the application, the developer call it using certain construct, in Java this is done using "**import**" construct followed by package name needed. Each time the developer wants to load new package, he has to repeat the "**import**" keyword with each package. As enhancement, we redefine the import construct with ability to write the import keyword only once with a colon "**import:**" followed by all packages' names:

```
    import: java.io.*;
        java.util.*;
        java.lang.*;
```

OR

```
    import: java.awt.*; java.awt.event.*;
```

No need to repeat the import keyword in front of each package name as before:

```
    import java.io.*;
    import java.util.*;
    import java.lang.*;
```

This enhancement helps in not repeating the keyword to load a package and it format the code in clear block or section. It will save efforts to retype the same key word each time new package is needed and make code shorter.

11. Multiple Variables Access Modifiers Construct:

Access modifiers determine the visibility and accessibility of class instance variables and methods within the class, package, and other classes.

Access modifiers are defined in the following table:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

The developer can define the access modifier of each class member by setting the access modifier keyword in front of the attribute definition name (object or primitive types):

```
class ClassName
{
    private int a;
    private String b;
    public File file = new File();
    public double length;
}
```

This requires the developer to repeat the same access modifier keyword in front of each attribute. And if there are many attributes with the same access modifier, then keyword must be repeated in front of each one.

A new enhancement proposed to define many attributes with the same access modifier without the need to repeat the access modifier keyword. The developer just has to define the access modifier keyword followed by colon and list all attributes belong to this access modifier:

```
class ClassName
{
    private:
        int a = 1;
        String b;
    public:
        File file = new File();
        double length;
    protected:
        Object obj;
        float value = 0;
}
```

This enhancement helps in not repeating the keyword to define attributes' access modifiers and it format the code in clear block or section. It will save efforts to retype the same key word each time new package is needed and make code shorter. This is close to what exist in C++.

12. Methods Access Modifiers Constructs:

Methods access modifiers are used with the same conventions described in the previous section (11) and they in front of the method signature:

```
public void methodName()  
private int methodName2()
```

This exists in Java C# and others.

By referring to new method definition construct we presented in section 3 "**def ..endef**", the access modifiers are specified in different and simple way through adding underscore(s) "_" at the beginning of method name to define its access modifier:

Modifier	Number of underscores "_" in method name
public	None
protected	1
private	2

Examples:

```
def __privateMeth() //private method  
endef  
def _protectedMeth() //protected method  
endef  
def publicMeth() //public method  
endef
```

This reduces coding efforts. And the method access modifier can be determined from its name without the need to go the defining class and check its definition construct to know the access modifier.

The basic idea of this enhancement is obtained from Python for private methods, and we extend it protected and public methods' access modifiers.

13. Exception Handling Variables Scope Construct

No change on the exception handling construct (try ... catch) syntax is done. We modified the logic and scope (accessibility) of the attributes and variables defined within the try block. Modern languages like C# and Java prevent variable defined within try block to be accessible or reachable in the catch block(s), finally block or any code below the try catch construct as shown below:

```
try  
{  
    int num = Integer.parseInt(br1.readLine());  
}  
catch(Exception e)  
{  
    System.out.println("num =" + num); // will not work and cause error  
}  
System.out.println("num =" + num); // will not work and cause error
```

In the previous example, the two print statements in catch block and after it will cause errors because the variable "**num**" is not accessible in them. To fix this in C# or Java, user has to define "**num**" before the try block as follows:

```
int num;  
try  
{  
    num = Integer.parseInt(br1.readLine());  
}  
catch(Exception e)  
{  
    System.out.println("num =" + num); // will work now
```

```
}  
System.out.println("num =" + num); // will work Now
```

This costs new line before the try block and coder awareness. Our proposed enhancement modifies this so the variable defined within try block will be accessible within try, catch, finally, and the following code blocks. No need to define the variable outside try block to access it later as follows:

```
try  
{  
    int num = Integer.parseInt(br1.readLine());  
}  
catch(Exception e)  
{  
    System.out.println("num =" + num); // will work now  
}  
  
System.out.println("num =" + num); // will work Now
```

This enhancement has nothing to do with syntax; it affects the semantic of exception handling variables scope. The variables become accessible anywhere which removes the constraints on attributes definitions. Eiffel and Python offer something similar.