A Review of Programming Paradigms Throughout the History

With a Suggestion Toward a Future Approach

ELAD SHALOM

A Review of Programming Paradigms Throughout the History

With a Suggestion Toward a Future Approach

ELAD SHALOM

A review of Programming paradigms throughout the history

WITH a suggestion toward a future approach

Elad Shalom

CONTENTS

What is a programming paradigm and Why do we need more than one

A programming paradigm is a fundamental style of computer programming, serving as a way of building the structure and elements of computer programs. Various programming languages have different capabilities and styles and they are defined by their supported programming paradigms. Some programming languages follow only one paradigm, while others support multiple paradigms.

1. Overview

Programming languages are defined by differing paradigms and some of them are designed to support multiple paradigms – C++, Java, C#, Scala, Python, Ruby… For example, programs written in C++ can be purely procedural, purely object-oriented, or they can contain elements of both or other paradigms. Deciding which elements are used is up to software designers and programmers. Some of the languages that support one particular paradigm are Smalltalk (object-oriented) and Haskell (functional).

Often distinguished programming paradigms include imperative, declarative, functional, object-oriented, procedural, and logic programming. In object-oriented programming, programmers can think of a program as a collection of

interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or system with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Many programming paradigms are as well known for what techniques they forbid as for what they enable. For example, pure functional programming disallows the use of side-effects, while structured programming disallows the use of the ***goto*** statement. This is the reason why new paradigms are often regarded as overly rigid by those accustomed to earlier styles.

Programming paradigms can also be compared with programming models that are abstractions of computer systems. For example, the ***von Neumann model*** is a programming model used in traditional sequential computers. For parallel computing, there are many possible models typically reflecting different ways processor can be interconnected.

Some programming language researchers criticise the notion of paradigms as a classification of programming languages. They argue that many programming languages cannot be strictly classified into one paradigms, but rather include features from several paradigms.

2. History of programming languages

A definition often advanced for a programming language is “a notation for communicating to a computer what we want it to do”, but this definition is inadequate. Before the middle of the 1940s, computer operators “hardwired” their programs. That is, they set switches to adjust the internal wiring of a computer to perform the requested task. This effectively communicated to the

computer what computations were desired, but programming consisted of the expensive and error-prone activity of taking down the hardware to restructure it.

2.1. Machine language

A major advance in computer design occurred in the late 1940s, when John von Neumann had the idea that a computer should be permanently hardwired with a small set of general-purpose operations. The operator could then input into the computer a series of binary codes that would organize the basic hardware operations to solve more-specific problems. Instead of turning off the computer to reconfigure its circuits, the operator could flip switches to enter these codes, expressed in ***machine language***, into computer memory. At this point, computer operators became the first true programmers, who developed software – the machine code – to solve problems with computers.

```
0010001000000100
0010010000000100
0001011001000010
0011011000000011
1111000000100101
0000000000000101
0000000000000110
0000000000000000
```

The code above is a machine language program. In this program, each line contains 16 bits or binary digits. A line of 16 bits represents either a single machine language instruction or a single data value. Program execution begins with the first line of code, which is fetched from memory, interpreted, and executed. Control then moves to the next line of code, and the process is repeated, until a special halt instruction is reached.

To decode or interpret an instruction, the programmer and the hardware must recognize the first 4 bits of the line of code as an opcode, which indicates the type of operation to be performed. The remaining 12 bits contain codes for the instruction's operands. The operand codes are either the numbers of machine registers or relate to the addresses of other data or instructions stored in memory.

Despite the improvement on the earlier method of reconfiguring the hardware, programmers were still faced with the tedious and error-prone tasks of manually translating their designs for solutions to binary machine code and loading this code into computer memory.

2.2. Assembly language

The early programmers realized that it would be a tremendous help to use mnemonic symbols for the instruction codes and memory locations, so they developed ***assembly language*** for this purpose. This type of language relies on software tools to automate some of the tasks of the programmer. A program called an ***assembler*** translates the symbolic assembly language code to binary machine code.

Programmers also used a pair of new input devices – a keypunch machine to type their assembly language codes and a card reader to read the resulting

punched cards into memory for the assembler. These two devices were forerunners of today's software text editors. These new hardware and software tools made it much easier for programmers to develop and modify their programs.

To insert a new line of code between two existing lines, the programmer now could put a new card into the keypunch, enter the code, and insert the card into the stack of cards at the appropriate position. The assembler and loader would then update all of the address references in the program, a task that machine language programmers once had to perform manually. Moreover, the assembler was able to catch some errors, such as incorrect instruction formats and incorrect address calculations, which could not be discovered until run time before.

```
.ORIG x3000;

LD  R1, FIRST;

LD  R2, SECOND;

ADD R3, R2, R1;

ST  R3, SUM;

HALT;

FIRST  .FILL #5  ;

SECOND .FILL #6  ;

SUM    .BLKW #1  ;

.END  ;
```

Above is an assembly language program that adds two numbers.

Although the use of mnemonic symbols represents an advance on binary machine codes, assembly language still has some shortcomings. The code example allows the programmer to represent the abstract mathematical idea, "Let ***FIRST*** be 5, ***SECOND*** be 6, and ***SUM*** be ***FIRST + SECOND***" as a sequence of human-readable machine instructions. Many of these instructions must move data from variables/memory locations to machine registers and back again. Assembly language lacks more powerful abstraction capability of conventional mathematical notation. An abstraction is a notation or way of expressing ideas that makes them concise, simple, and easy for the human mind to grasp.

A second major shortcoming of assembly language is due to the fact that each particular type of computer hardware architecture has its own machine language instruction set, and thus requires its own dialect of assembly language. Therefore, any assembly language program has to be rewritten to port it to different types of machines.

The first assembly languages appeared in the 1950s. They are still used today, whenever very low-level system tools must be written, or whenever code segments must be optimized by hand for efficiency.

2.3. FORTRAN and algebraic notation

Unlike assembly language, high-level languages (Java, Python, C, …) support notations closer to the abstractions, such as algebraic expressions, used in mathematics and science. For example, the following code segment in C or Java is equivalent to the assembly language program for adding two number shown earlier:

```
int first = 5;

int second = 6;

int sum = first + second;
```

One of the precursors of these high-level languages was FORTRAN, an acronym for FORmula TRANslation language. John Backus developed FORTRAN in the early 1950s for a particular type of IBM computer. In some respects, early FORTRAN code was similar to assembly language. It reflected the architecture of a particular type of machine and lacked the structured control statements and data structures of later high-level languages.

2.4. Structured abstractions and machine independence

Soon after FORTRAN was introduced, programmers realized that languages with still higher levels of abstraction would improve their ability to write concise, understandable instructions. Moreover, they wished to write these high-level instructions for different machine architectures with no changes.

In the late 1950s, an international committee of computer scientists agreed on a definition of a new language whose purpose was to satisfy both of these requirements. This language became ALGOL. Its first incarnation, ALGOL-60, was released in 1960.

ALGOL provided first of all a standard notation for computer scientists to publish algorithms in journals. As such, the language included notations for structured control statements for sequencing, loops, and selection. These types of

statements have appeared in more or less the same form in every high-level language since. Likewise, elegant notations for expressing data of different numeric types as well as the array data structure were available. Finally, support for procedures, including recursive procedures, was provided.

The ALGOL committee also achieved machine independence for program execution on computers by requiring that each type of hardware provide an ALGOL compiler. This program translated standard ALGOL programs to the machine code of a particular machine.

A very large number of high-level languages are descended from ALGOL like Pascal and Ada. The designers of ALGOL's descendants typically added features for further structuring data and large units of code, as well as support for controlling access to these resources within a large program.

2.5. Computation without the von Neumann architecture

Although programs written in high-level languages became independent or particular makes and models of computers, these languages still echoed the underlying architecture of the von Neumann model of a machine. This model consists of an area of memory where both programs and data are stored and a separate central processing unit that sequentially executes instructions fetched from memory.

For the first five decades of computing, the improvements in processor speed and the increasing abstraction in programming languages supported the conversion of the industrial age into the information age. However, this progress in language abstraction and hardware performance eventually ran into separate roadblocks.

On the hardware side, engineers had increased processor performance by

shortening the distance between processor components, but as components were packed more tightly onto a processor chip, the amount of heat generated during execution increased. This was solved by factoring some computations, such as floating-point arithmetic and graphics/image processing, out to dedicated processors.

Within the last few years, most desktop and laptop computers have been built with multicore architectures. A multicore architecture divides the central processing unit into two or more general-purpose processors, each with its own specialized memory, as well as memory that is shared among them.

On the language side, despite the efforts of designers to provide higher levels of abstraction for von Neumann computing, two problems remained. First, the model of computation, which relied upon changes to the values of variables, continued to make very large programs difficult to debug and correct. Second, the single-processor model of computation, which assumes a sequence of instructions that share a single processor and memory space, cannot be easily mapped to the new hardware architectures, whose multiple CPUs execute in parallel. The solution to these problems is the insight that programming languages need not be based on any particular model of hardware, but need only support models of computation suitable for various styles of problem solving.

The mathematician Alonzo Church developed one such model of computation in the late 1930s. This model, called ***lambda calculus***, was based on the theory of recursive functions. In the late 1950s, John McCarthy created the programming language Lisp to construct programs using the functional model of computation. Although a Lisp interpreter translated Lisp code to machine code that actually ran on a von Neumann machine, there was nothing about the Lisp notation that entailed a von Neumann model of computation.

Researchers have developed languages modelled on other non-von Neumann models of computing. One such model is formal logic with automatic theorem proving. Another involves the interaction of objects via message passing.[1]

3. Language categories

Programming languages are often categorized into four bins: imperative, functional, logic, and object-oriented. However, languages that support object-oriented programming are not always considered to form a separate category since most of them grew out of imperative languages.
Although the object-oriented paradigm differs significantly from the procedure-oriented paradigm usually used with imperative languages, the extensions to an imperative language required to support object-oriented programming are not intensive. For example, the expressions, assignment statements, and control statements of C and Java are nearly identical. Similar statements can be made for functional languages that support object-oriented programming.
Another kind of language, the visual language, is a subcategory of the imperative languages. The most popular visual languages are the .NET languages. These languages include capabilities for drag-and-drop generation of code segments. Such languages were once called fourth-generation languages, although that name has fallen out of use.

The visual languages provide a simple way to generate graphical user interfaces to programs. For example, using Visual Studio to develop software in the .Net languages, the code to produce a display of a form control, such as a button or text box, can be created with a single keystroke. These capabilities are now available in all of the .NET languages.

Some refer to scripting languages as a separate category of programming

languages. However, languages in this category are bound together more by their implementation method, partial or full interpretation, than by a common language design. The languages that are typically called scripting languages, among them Perl, JavaScript, and Ruby, are imperative languages in every sense.

A logic programming language is an example of a rule-based language. In an imperative language, an algorithm is specified in great detail, and the specific order of execution of the instructions or statements must be included. In a rule-based language, however, rules are specified in no particular order, and the language implementation system must choose an order in which the rules are used to produce the desired result. This approach to software development is radically different from those used with the other two categories of languages and clearly requires a completely different kind of language.

Recently, a new category of languages has emerged, the markup/programming hybrid languages. Markup languages are not programming languages. For instance, HTML, the most widely used markup language, is used to specify the layout of information in Web documents. However, some programming capability has crept into some extensions to HTML and XML. Among these are the Java Server Pages Standard Tag Library (JSTL) and eXtensible Stylesheet Language Transformations (XSTL).

A host of special-purpose languages have appeared over the past 50 years. These range from Report Program Generator (RPG), which is used to produce business reports; to Automatically Programmed Tools (APT), which is used for instructing programmable machine tools; to General Purpose Simulation System (GPSS), which is used for systems simulation.[2]

4. Importance of various programming concepts

Knowing different concepts of programming languages has many benefits.

- ***Increased capacity to express ideas***

It is widely believed that the depth at which people can think is influenced by the expressive power of the language in which they communicate their thoughts. Those with only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction. In other words, it is difficult for people to conceptualize structures they cannot describe, verbally or in writing.

Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited.
Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

It might be argued that learning the capabilities of other languages does not help a programmer who is forced to use a language that lacks those capabilities. That argument does not hold up, however, because often, language constructs can be simulated in other languages that do not support those constructs directly. For

example, a C programmer who had learned the structure and uses of associative arrays in Perl might design strictures that simulate associative arrays in that language.

In other words, the study of programming language concepts builds an appreciation for valuable language features and constructs and encourages programmers to use them even when the language they are using does not directly support such features and constructs.

- ***Improved background for choosing appropriate languages***

Many professional programmers have had little formal education in computer science. Their training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization. Many other programmers received their formal training years ago. The languages they learned then are no longer used, and many features now available in programming languages were not widely known at the time.

The result is that many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem.

Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often

less elegant, more cumbersome, and less safe.

- ***Increased ability to learn new languages***

Computer programming is still a relatively young discipline, and design methodologies, software development tools, and programming languages are still in a state of continuous evolution. This makes software development an exciting profession, but it also means that continuous learning is essential.

The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general.

Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned. For example, programmers who understand the concepts of object-oriented programming will have a much easier time learning Java than those who have never used those concepts.

Finally, it is essential that practicing programmers know the vocabulary and fundamental concepts of programming languages so they can read and understand programming language descriptions and evaluations, as well as promotional literature for languages and compilers. These are the sources of information needed in order to choose and learn a language.

- ***Better understanding of the significance of implementation***

In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.

Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program.

For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called be a highly inefficient design choice.

- ***Better use of languages that are already known***

Many contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and

unused parts of the languages they already use and begin to use those features.

- ***Overall advancement of computing***

Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available.

In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts. For example, many people believe it would have been better if ALGOL 60 had displaced FORTRAN in the early 1960s, because it was more elegant and had much better control statements, among other reasons.

The reason it did not is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60. They found its description difficult to read and even more difficult to understand. They did not appreciate the benefits of block structure, recursion, and well-structured control statements, so they failed to see the benefits of ALGOL 60 over FORTRAN.[3]

Bibliography

1. ***LOUDEN, K. C., LAMBERT, K. A. (2011) Programming Languages Principles and Practice. 3rd Edition. Cengage Learning***

2. ***SEBESTA, R. W. (2012) Concepts of Programming Languages. 10th Edition. Addison-Wesley***

Imperative Programming

A computer program stores data in variables which represent storage locations in the computer’s memory. The contents of these locations at any given point in the program’s execution are called the program’s state. Imperative programming is characterized by programming with state and commands which modify the state. Imperative programming is probably the first programming paradigm.
1. History and languages

1.1. Early imperative languages

Imperative languages have a rich and varied history. The first imperative programming languages were machine languages. Machine code or machine language is a set of instructions executed directly by a computer's central processing unit (CPU). Instructions are patterns of bits that by physical design correspond to different commands to the machine. Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory.
For example:

- ***rs***, ***rt***, and ***rd*** indicate register operands
- ***shamt*** gives a shift amount
- the ***address*** or ***immediate*** fields contain an operand directly

So, adding the registers 1 and 2 and placing the result in register 6 is encoded:

[op | rs | rt | rd |shamt| funct]

0 1 2 6 0 32 decimal

000000 00001 00010 00110 00000 100000 binary

Machine instructions were soon replaced with assembly languages which are essentially transliterations of machine code. An assembly language is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions.

Let’s take the following array of bits as an example:

10110000 01100001

This binary computer code can be made more human-readable by expressing it in hexadecimal:

B0 61

Here, B0 means ‘Move a copy of the following value into ***AL***’ (***AL*** is a register), and 61 is a hexadecimal representation of the value 01100001, which is 97

in decimal representation. Intel assembly language provides the mnemonic ***MOV*** (an abbreviation of ***move***) for instructions such as this, so the machine code above can be written as follows in assembly language, completed with an explanatory comment if required, after the semicolon. This is much easier to read and to remember than some binary computer code.

```
MOV AL, 61h       ; Load AL with 97 decimal (61 hex)
```

FORTRAN (FORmula TRANslation) was the first high level language to gain wide acceptance. It was designed for scientific applications and featured an algebraic notation, types, subprograms, and formatted input/output. It was implemented in 1956 by John Backus at IBM specifically for the IBM 704 machine, which is the first mass-produced computer with floating point arithmetic hardware. Efficient execution was a major concern. Consequently, its structure and commands have much in common with assembly languages. FORTRAN won wide acceptance and continues to be in wide use in the scientific computing community.[4]

```
program summation

implicit none

integer :: sum, a


print*, "This program performs summations. Enter 0 to stop."

open(unit=10, file="SumData.DAT")
```

```
sum = 0


do
print*, "Add:"
read*, a
if (a == 0) then
exit
else
sum = sum + a
end if
write(10,*) a
end do


print*, "Summation =", sum
write(10,*) "Summation =", sum
close(10)
```

```
end
```

When executed, the console will display the following:

```
This program performs summations.  Enter 0 to stop.
Add:
1
Add:
2
Add:
3
Add:
0
Summation = 6
```

And the file SumData.DAT would contain:

```
1
2
3
```

Summation = 6

COBOL (COmmon Business Oriented Language) was designed at the initiative of the U. S. Department of Defence in 1959 and implemented in 1960 to meet the need for business data processing applications. COBOL featured records, files and fixed decimal data. It also provided a “natural language” like syntax so that programs would be able to be read and understood by non-programmers. COBOL won wide acceptance in the business data processing community and continues to be in wide use in business, finance, and administrative systems for companies and governments. In 1997, Gartner Group estimated that there were a total of 200 billion lines of COBOL code in existence which ran 80% of all business programs. He was in great use because COBOL has an English-like syntax which was designed to be self-documenting and highly readable.

ALGOL 60 (ALGorithmic Oriented Language) was designed in 1960 by an international committee for use in scientific problem solving. Unlike FORTRAN, it was designed independently of an implementation, a choice which lead to an elegant language. The description of ALGOL 60 introduced the BNF notation for the definition of syntax and is a model of clarity and completeness. Although ALGOL 60 failed to win wide acceptance because the languages it produced did not become the universal algorithmic programming language in industry. Although it was used in Europe more than in the USA, it could not beat FORTRAN. ALGOL 60 introduced block structure, structured control statements and recursive procedures. Despite its quality ALGOL was not received well in industry.

ALGOL 68 (short for ALGOrithmic Language 1968) is an imperative computer programming language that was conceived as a successor to the ALGOL 60 programming language, designed with the goal of a

much wider scope of application and more rigorously defined syntax and semantics.[5]

1.2. Evolutionary developments

Pascal was developed by Niklaus Wirth[6] partly as a reaction to the problems encountered with ALGOL 68 and as an attempt to provide a small and efficient implementation of a language suitable for teaching good programming style. C, which was developed about the same time, was an attempt to provide an efficient language for systems programming.
PL/1 (Programming Language I) was developed at IBM in the mid 1960's. It was designed for scientific, engineering, business and systems programming application, like FORTRAN, ALGOL 60, COBOL, LISP, and APL. PL/I incorporated the following from ALGOL 60:

- block structure - a block is a section of code which is grouped together
- structured control statements - refer to the order in which the individual statements, instructions or function calls are executed od evaluated
- the possibility that the same process calls itself, also named recursion

From FORTRAN, PL/I incorporated:

- subprograms, which represent a sequence of program instructions that perform a specific task, packaged as a unit.

- formatted input/output

From COBOL, PL/I incorporated:

- file manipulation
- the record

From LISP, PL/I incorporated:

- dynamic storage allocation, allocation and reservation of memory is done during program execution.
- linked structures, in the form of data structure which consists of a set of data records (nodes) linked together and organized by references (links or pointers).

Some array operations, like possibility that an element of an array is another array, PL/I incorporated from APL. PL/I introduced exception handling and multitasking for concurrent programming, it means that if during the execution of the program some error occurs you can handle him in some way. PL/I was complex, difficult to learn, and difficult to implement. For these reasons PL/I failed to win wide acceptance.[7]

2. Overview

Imperative programming is the most dominant paradigm of all the others. It is used in Assembly, Java, C, Python, Ruby, JavaScript and many others. (Note: Some programming languages have multiple paradigms) While using the imperative way of programming, the programmer is telling the "machine" how to do something and gets the result the way he wanted it.

Since it is the most dominant programming paradigm, it's what most people are familiar with. Therefore, it just seems like the natural way to program.[8]

When imperative programming is combined with subprograms it is called procedural programming. In either case the implication is clear. Programs are directions or orders for performing an action. Keywords and phrases used are: Assignment, goto commands, structured programming, command, statement, procedure, control-flow, imperative language, assertions, axiomatic semantics, state, variables, instructions, control structures.

The imperative programming paradigm is an abstraction of real computers which in turn are based on the Turing machine and the Von Neumann machine with its registers and store (memory). At the heart of these machines is the concept of a modifiable store. Variables and assignments are the programming language analog of the modifiable store. The store is the object that is manipulated by the program. Imperative programming languages provide a variety of commands to provide structure to code and to manipulate the store. Each imperative programming language defines a particular view of hardware. These views are so distinct that it is common to speak of a Pascal machine, C machine or a Java machine. A compiler implements the virtual machine defined by the

programming language in the language supported by the actual hardware and operating system.

In imperative programming, a name may be assigned to a value and later reassigned to another value. The collection of names and the associated values and the location of control in the program constitute the state. The state is a logical model of storage which is an association between memory locations and values. A program in execution generates a sequence of states. The transition from one state to the next is determined by assignment operations and sequencing commands.

State Sequence
S_0 -- O_0 -> S_1 - ... -> S_{n-1} -- O_{n-1}-> S_n

Unless carefully written, an imperative program can only be understood in terms of its execution behaviour. The reasons for that are the facts that during the execution of the code, any variable may be referenced, control may be transferred to any arbitrary point, and any variable binding may be changed.[9]

3. Variables and Assignment (quick overview)

Imperative programs are characterized by sequences of bindings (state changes) in which a name may be bound to a value at one point in the program and bound to a different value along the way. Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings.

When speaking of hardware, terms like bit pattern, storage cell, and storage address are used. Somewhat analogous terms in programming languages

are value, variable, and name. This means that what the CPU sees as a bit pattern, programming languages consider a value. In the same way, storage cell is interpreted as a variable and storage address is interpreted as a name. Since variables are usually bound to a name and to a value, the word variable is often used to mean the name of a value.

Most descriptions of imperative programming languages are tied to hardware and implementation considerations where a name is bound to an address, a variable to a storage cell, and a value to a bit pattern. Thus, a name is tied to two bindings – a binding to a location and to a value. The location is called the ***l-value*** and the value is called the ***r-value***. The necessity for this distinction comes from the implementation of the assignment. For example,

```
x = x + 2;
```

where the ***x*** on the left of the assignment denotes a location while the ***x*** on the right side denotes the value. Assignment changes the value at the location.

A variable may be bound to a hardware location at various times. It may be bound at compile time (rarely), at load time (for languages with static allocation) or at run time (for languages with dynamic allocation). Static memory allocation refers to the process of reserving memory at compile-time before the associated program is executed, unlike dynamic memory allocation where memory is allocated as required at run-time. From the implementation point of view, variable declarations are used to determine the amount of storage required by the program.[10]

4. Components

4.1. Assignment

In imperative programming, a name may be assigned to a value and later reassigned to another value. The collection of names and the associated values and the location of control in the program constitute the state. The state is a logical model of storage which is an association between memory locations and values.

The assignment construct allows the creation of a statement with a variable ***x*** and an expression ***t***. In Java, this statement is written as

```
x = t;
```

Variables are identifiers which are written as one or more letters. Expressions are composed of variables and constants with operators such as + (addition), - (subtraction), * (multiplication), / (division) and % (modulo). Therefore, the following statements

```
x = y % 3;
```

```
x = y;
```

```
y = 3;
```

```
x = x + 1;
```

are all proper Java statements, while

```
y + 3 = x;

x + 2 = y + 5;
```

are not.

To understand what happens when you execute the statement ***x = t*** supposes that within the recesses of your computer's memory, there is a compartment labelled ***x***. Executing the statement ***x = t*** consists of filling this compartment with the value of the expression ***t***. The value previously contained in compartment ***x*** is erased. If the expression ***t*** is a constant, for example 3, its value is the same constant. If it is an expression with no variables, such as ***3 + 4***, its value is obtained by carrying out mathematical operations, in this case, addition. If expression t contains variables, the values of these variables must be looked up in the computer's memory. The value (contents) of variables is called a state in computer's memory.

Let us consider, initially, that expressions such as ***x + 3***, and statements such as ***y = x + 3*** form two disjoint categories. Later, however, we shall be brought to revise this premise.

In these examples, the values of expressions are integers. Computers can only store integers within a finite interval. In Java, integers must be between -2^{31} and $2^{31} - 1$, so there are 2^{32} possible values. When a mathematical operation produces a value outside of this interval, the result is kept within the interval by taking its modulo 2^{32} remainder. Thus, by adding 1 to $2^{31} - 1$, that is to say

2147483647, we leave the interval and then return to it by removing 2^{32}, which gives -2^{31} or -2147483648.[11]

It is legal in Java to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value.

```
int x = 5;
System.out.print(x);
int x = 7;
System.out.print(x);
```

The output of this program is 57 because the first time ***x*** is printed, his value is 5, and the second time it is 7.

This kind of multiple assignment is the reason variables were described as a ***container*** for values. When a value is assigned to a variable, the contents of the container are changed as shown in the figure:

Figure 1 – Value assignment

When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because Java uses the = symbol for assignment, it is tempting to interpret a statement like ***a = b*** as a statement of equality. This is not true.

First of all, equality is commutative while assignment is not. For example, in mathematics if ***a = 7*** then ***7 = a***. But in Java ***a = 7*** is a legal assignment statement and 7 = a is not. Furthermore, in mathematics, a statement of equality is always true. If ***a = b*** is true in the giving moment, then ***a*** will always equal ***b***. In Java, an assignment statement can make two variables equal, but they don't have to stay that way.

```
int a = 5;

int b = 5; //a and b are now equal

a = 3; //a and b are no longer equal
```

The third line changes the value of ***a*** but it does not change the value of ***b*** so they are no longer equal. In many programming languages an alternate symbol is used for assignment, such as <- or :=, in order to avoid this confusion.

Although multiple assignment is frequently used, it should be used with caution. If the values of variables are changing constantly in different parts of the program, it can make the code difficult to read and debug.[12]

4.2. Variables

Since imperative programming is a list of steps that need to be executed, there needs to be some way to keep track of everything computed to that point. That’s where variables come in. They are keeping the state the program is at, which can control where the program should go next as it continues to modify the state.

Before being able to assign values to a variable ***x***, it must be declared. Variable declaration associates the name ***x*** with a location in the computer's memory. This is a construct that allows the creation of a statement composed of a variable, an expression, and a statement. In Java, this statement is written as ***int x = t; p*** where ***p*** is a statement. For example

```
int x = 4;

x = x + 1;
```

The variable ***x*** can then be used in the statement ***p***, which is called the scope of variable ***x***. It is also possible to declare a variable without giving it an initial value:

```
int x;

x = y + 4;
```

We must, of course, be careful not to use a variable which has been declared without an initial value and that doesn't yet have a value assigned. This results in an error.[13]
Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;

float sum = 0.0, product;
```

```
char ch = '7';
```

which assigns the value 0 to the integer variable count and the value 0.0 to the real variable sum. The character variable ***ch*** is initialised with the character 7. ***i***, ***j***, and product have no initial value specified, so the program should make no assumption about their contents. And now, if during the execution of the programs you want to know the result of ***i + j***, you can get some random values from memory and thus cause a bad result.[14]

Apart from the ***int*** type, Java has three other integer types that have different intervals. The integers are of type ***byte***, ***short***, ***int*** or ***long*** corresponding to the intervals $[-2^7, 2^7 - 1]$, $[-2^{15}, 2^{15} - 1]$, $[-2^{31}, 2^{31} - 1]$ and $[-2^{63}, 2^{63} - 1]$, respectively.

Constants are written in base 10, for example, 666. When a mathematical operation produces a value outside of these intervals, the result is returned to the interval by taking its remainder, modulo the size of the interval. In Java, there are also other scalar types for decimal numbers, Booleans, and characters.

Decimal numbers are of type ***float*** or ***double***. Constants are written in scientific notation, for example 3.14159, 666 or 6.02E23. The operations for decimal numbers are +, -, *, /, along with some transcendental functions: sine, cosine, ... (available with methods ***Math.sin(...)***, ***Math.cos(...)***).

Booleans are of logical type ***Boolean***. The operations allowed in Boolean expressions are == (equal), != (different, not equal), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), & (binary and), && (logical and), | (binary or), || (logical or), ! (not, negation). Constants are written as ***false*** and ***true***.

Characters are of type ***char***. Constants are written between apostrophes, for example ‘z’. Variables can also contain objects that are of composite types, like arrays and character strings. Character strings are of type ***String***. Constants are written inside quotation marks, for example "Principles of Programming Languages".

For all data types, the expression ***(b) ? t : u*** results in the value of ***t*** if the Boolean expression ***b*** has the value ***true***, and results in the value of ***u*** if the Boolean expression ***b*** has the value ***false***.

In Java and in C, it is impossible to declare the same variable twice and the following program is not valid.

```
int y = 4;

int x = 5;

int x = 6; //error

y = x;
```

Java and C allow the creation of variables with an initial value that can never be changed. This type of variable is called a constant variable. A variable that is not constant is called a mutable variable. Java assumes that all variables are mutable unless you specify otherwise. To declare a constant variable in Java, you precede the variable type with the keyword final, for example:

```
final int x = 4;
```

```
y = x + 1;
```

The following statement is not valid, because an attempt is made to alter the value of a constant variable

```
final int x = 4;

x = 5; //error
```

4.3. Sequence

A sequence is a construct that allows a single statement to be created out of two statements ***p1*** and ***p2***. In Java, a sequence is written as ***{p1 p2}***. The statement ***{p1 {p2 { ... pn} ...}}*** can also be written as ***{p1 p2 ... pn}.***

To execute the statement ***{p1 p2}***, the statement ***p1*** is executed first which produces a new state. Then the statement ***p2*** starts executing.[15]

5. Representation of controls in Java

5.1. Glossary

Before going deeper into the matter, let’s define several terms that will be used.

A ***local variable*** is a variable that is declared inside a method and exists only within that method. It cannot be accessed from outside its home method and does not interfere with any other methods. To ***encapsulate*** means to divide a

large and complex program into smaller components (like methods) and isolate the components from each other (i.e. by using local variables). Replacing something unnecessary specific (such as a constant value) with something appropriately general (such as a parameter) is ***generalization***. This makes the code more likely to be reused.

A ***loop*** is a group of statements that executes repeatedly while or until some condition is met. When the condition is always ***true***, the loop is called an ***infinite loop***. The group of statements inside the loop or some other function is called ***body***. One pass through (execution of) the body of the loop, including the evaluation of the condition, is called ***iteration***.

5.2. Test, if and else

A test is a construct that allows the creation of a statement composed of a Boolean expression ***b*** and two statements ***p1*** and ***p2***. In Java, this statement is written as follows:

```
if (b) p1 else p2
```

To execute this statement in a state ***s***, the value of expression ***b*** is first computed in this state, and depending on whether or not its value is ***true*** or ***false***, the statement ***p1*** or ***p2*** is executed in the state ***s***.

5.3. Loop, simple definition

A loop is a construct that allows the creation of a statement composed of a Boolean expression ***b*** and a statement ***p***. In Java, this statement is written:

```
while (b) {
    p
}
```

To execute this statement while in the state ***s***, the value of ***b*** is first computed in this state. If this value is ***false***, execution of this statement is terminated. If the value is ***true***, the statement ***p*** is executed, and the value of ***b*** is recomputed in the new state. If this value is now ***false***, execution of this statement is terminated. If the value is still ***true***, the statement ***p*** is executed and the value of ***b*** is recomputed in the new state and so on... This process continues until ***b*** evaluates to ***false***. This construct introduces a new possible behaviour: non-termination. Indeed, if the Boolean value ***b*** always evaluates to ***true***, the statement ***p*** will continue being executed forever, and the statement ***while (b) p*** will never terminate.

This is the case with the following code snippet:

```
int x = 1;
while (x >= 0) {
    x = 3;
}
```

To understand what is happening, imagine a fictional statement called ***skip*** that performs no action when executed. You can then define the mentioned statement as shorthand for the statement

```
if (b) {
        p
        if (b) {
                p
                if (b) {
                        p ...
                else
                        skip;}
        else
                skip;}
else
        skip;}
```

So, a loop is one of the ways in which you can express an infinite object using a finite expression. And the fact that a loop may fail to terminate is a consequence of the fact that it is an infinite object.

The while, repeat and for commands are the most common representatives of iteration.[16]

5.4. While, do while

The while statement continually executes a block of statements while a particular condition is ***true***. Its syntax can be expressed as:

```
while (expression) {
statement(s)
}
```

The while statement evaluates expression, which must return a ***Boolean*** value. If the expression evaluates to ***true,*** the while statement executes the statement(s) in the while block. Therefore, statement continues testing the expression and executing its block until the expression evaluates to ***false***.

You can implement an infinite loop using the while statement as follows:

```
while (true){
//statements
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the end of the loop instead of the beginning. Therefore, the statements within the do block are always executed at least once.

```
public class MyClass {

public static void main(String[] args) {
int x = 1;
while(x > 1){
System.out.println("While loop: " + x);
x--;
}
x = 1;
```

```
do {
System.out.println("Do-while loop: " + x);
x--;
} while(x > 1);
}
}
```

When executed, the console displays the following:

```
Do-while loop: 1
```

Both terms are incorrect but as we said, do while loop will execute statements first and than will check the value of term. While loop first evaluates expression, that will evaluate to false and statements will not be executed.[17]

5.5. For

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The initialization expression initializes the loop; it's executed once, when the loop begins.
- When the termination expression evaluates to false, the loop terminates.
- The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.[18]

5.6. Input and Output

An input construct allows a language to read values from the keyboard and other input devices, such as a mouse, disk, a network interface card, etc. An output construct allows values to be displayed on a screen and outputted to other peripherals, such as a printer, disk, a network interface card, etc.[19]

6. Reasoning about Imperative Programs

Imperative constructs jeopardize many of the fundamental techniques for

reasoning about mathematical objects. For example, the assignment axiom of axiomatic semantics is valid only for languages without aliasing and side effects. Much of the work on the theory of programming languages is an attempt to explain the “referentially opaque” features of programming languages in terms of well-defined mathematical constructs. By providing descriptions of programming language features in terms of standard mathematical concepts, programming language theory makes it possible to manipulate programs and reason about them using precise and rigorous techniques. Unfortunately, the resulting descriptions are complex and the notational machinery is difficult to use in all but small examples. It is this complexity that provides a strong motivation to provide functional and logic programming as alternatives to the imperative programming paradigm.[20]

Bibliography

1. DOWEK, G. (2009) ***Principles of Programming Languages.*** Springer

2. HOFSTEDT, P. (2011) ***Multiparadigm Constraint Programming Languages.*** Springer

3. LIANG, Y. D. (2011) ***Introduction to Java Programming***. 8th Edition. Prentice Hall

4. MALIK, D. S. (2012) ***Java Programming: From Problem Analysis to Program Design***. 5th Edition. Cengage Learning

5. EASTERN MEDITERRANEAN UNIVERSITY (2015) ***The Imperative Programming Paradigm*** [Online] Available at: http://www.emu.edu.tr/aelci/Courses/D-318/D-318-

Files/plbook/imperati.htm [Accessed: 1st June 2015]

6. WIKIBOOKS (2015) ***Fortran/Fortran Examples*** [Online] Available at: http://en.wikibooks.org/wiki/Fortran/Fortran_examples [Accessed: 1st June 2015]

7. OPEN BOOK PROJECT (2015) ***How to Think Like a Computer Scientist – Iteration*** [Online] Available at: http://www.openbookproject.net/thinkcs/archive/java/english/chap06.htm [Accessed: 1st June 2015]

8. ORACLE (2015) ***The for Statement*** [Online] Available at: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html [Accessed: 1st June 2015]

Structured Programming

S tructured programming is a paradigm that is based on improving clarity and quality of programs by using subroutines, block structures and loops (***for*** and ***while***) and discouraging the use of ***goto*** statement. ***Goto*** statement allows unconditional jumping from ***goto*** to a labelled statement. In C, the syntax for this statement is:

```
goto label;

...

label : statement;
```
[21]

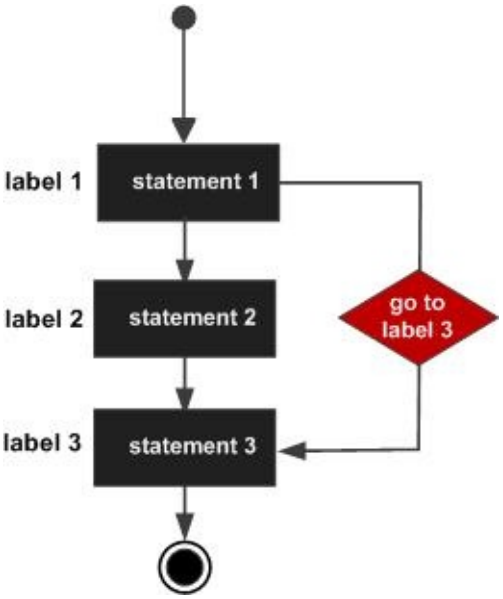


Figure 1 – goto presented with a flowchart (in this example, statement 2 will be skipped)

1. History and foundation

Before structured programming was introduced, programs were written mostly the way people would provide instructions – off the top of their heads. Thus, when a program did not work, it had to be patched error by error. This approach was tolerated until the development of third generation of computers in the 1960s.

During this period, hardware became more powerful than the software that ran it so non-systematic programming could not keep up and it became hard to write vast programs. The history of structured programming began with Corrado Böhm and Giuseppe Jacopini in 1964 when they proved in their presented paper[22] that only three control structures were required to write any program. The paper was originally written in Italian (they were both professors at the University of Rome) and presented in Israel at an international colloquium. The theorem made the ***goto*** statement unnecessary even though no programmer during the mid-1960s could write a code without it. As a result, the theorem was not widely accepted even when it was translated to English, but it served as a

base in further research.[23]

The ***Böhm-Jacopini theorem***, also called ***structured program theorem***, stated that working out a function is possible by combining subprograms in only three manners:

1. Executing one subprogram, and the other subprogram (***sequence***)
2. Executing one of two subprograms according to the value of a Boolean expression (***selection***)
3. Executing a subprogram until a Boolean expression is ***true*** (***iteration***)

This observation did not originate with the structured programming movement. Instead, these three structures were sufficient to describe the instruction cycle of a CPU and the operation of a Turing machine. In this sense, the processor always executes a “structured program” even if the read instructions are not part of a structured program. The theorem itself does not address how to write a structured program. This issue was dealt with during the late 1960’s and early 1970’s.

Even though the theorem was even ignored at first, the turning point was Edsger Dijkstra’s letter entitled “Go To Statement Considered Harmful”[24]. He advocated organizing programs in a systematic way, called ***structured programming***. Dijsktra was Dutch and along with Italians Böhm and Jacopini that started the discussion in the first place, it becomes clear that Europe was working on a new approach to programming.

After the successful outcome of the first project to use structured programming,

developed by the ***New York Times*** and completed in 1972, developers began paying attention to Dijkstra's work. The mentioned project was a system to automate the newspaper's clipping file. Using a list of index terms, users could browse through abstracts of all the paper's articles and then retrieve the chosen full-length articles from microfiche for display on a screen. This project was probably the sole reason structured programming became widely accepted.

The concept of structured programming became widespread but it is important to mention Edward Yourdon as the person most responsible for popularizing the concept in the United States. He was giving seminars on the subject in the mid-1970s. By late-1970s, structured programming was used for everything and was later called a "revolution in programming".[25]

Some of the languages that initially used structured approach are ***ALGOL***, ***Pascal***, ***PL/I*** and ***Ada***. By the end of the 20th century, concepts of structured programming were widely applied so programming languages that originally lacked structure now have it (***FORTRAN***, ***COBOL*** and ***BASIC***). Now, it is possible to do structured programming in any programming language (***Java***, ***C++***, ***Python ...)***.

2. Overview

Most programs that serve a purpose are long and the longer they are, the harder it is to follow the code instructions. Structured programming was defined as a method used to minimize complexity that uses:

1. top-down analysis for problem solving
2. modularization for program structure and organization
3. structured code for the individual modules

Top-down analysis includes solving the problem and providing instructions for every step. When developing a solution is complicated, the right approach is to divide a large problem into several smaller problems and tasks.

Modular programming is a method of organizing the instructions of a program. Large programs are divided into smaller sections called ***modules***, ***subroutines***, or ***subprograms***. Each subroutine is in charge of a specific job.

Structured coding relates to division of modules into set of instructions organized within ***control structures***. A control structure defines the order in which a set of instructions are executed. The statements within a specific control structure are executed:

- sequentially – denotes in which order the controls are executed. They are executed one after the other in the exact order they are listed in the code.
- conditionally – allows choosing which set of controls is executed. Based on the needed condition, one set of commands is chosen while others aren't executed.
- repetitively – allows the same controls to be performed over and over again. Repetition stops when it meets certain terms or performs a defined number of iterations.

Because the order of the instructions is determined, structured code cannot include ***goto*** statements since they alter the order and do not follow any pattern. [26]

It is essential to remember that control structures need to be implemented in a way that will result in them having single entry and single exit point. For example, if ***C2*** is a control that follows ***C1***, written like

C1;

C2

at the exit of ***C1*** command, control is passed to the entry point of ***C2***. Controls can internally contain branching and/or loops, but only entry and exit points should be visible externally. Statement like ***goto*** violates this property. Because it allows jumps forward and backwards, program execution often results in what is called “spaghetti code” since different program components interlace.[27]

Even though code is broken into independent modules, there is a main module which can call other modules. That is, the programmer programs one main module and when it is required to perform some other set of instructions a separate module is coded. The main module then simply calls for the execution of other modules or sections as needed.[28]

Besides providing rules about how programs should be structured, this paradigm also specifies some verbal guidelines. First, it implies usage of meaningful names for variables, procedures, etc. as it makes it easier to understand the code

and then apply changes to it. Secondly, it is suggested to use comments extensively. They are important for testing, reading, modification, etc., of the code. Without comments, longer programs easily become very hard to understand.

What is also important is to use structured data types such as records to group information even if it contains diverse types. Instead of using several variables to hold the information, it is preferred to write just one variable of type record where all the information will be stored.

To sum up, advantages of structured programming are:

- Programs are more easily and more quickly written.
- Programs have greater reliability.
- Programs require less time to debug and test.
- Programs are easier to maintain.[29]

3. Components

3.1. Structograms

Structogram or Nassi–Shneiderman diagram is a graphical representation of structured programming.[30]

Following the steps of top-down problem analysis, the problem is reduced into smaller and smaller parts of code until each part contains only simple statements

and control structures. Structograms reflect this by using nested boxes to represent subproblems. Nassi–Shneiderman diagrams have no representation for a ***goto*** statement.

Structograms can be compared to flowcharts:

- Everything that can be represented with a structogram can also be represented by a flowchart.
- Everything that can be represented with a flowchart can also be represented by a structogram with the exception of ***goto***, ***break*** and ***continue*** statements.

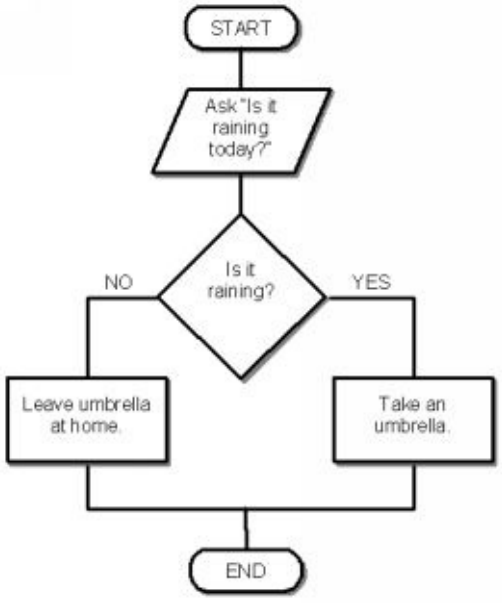


Figure 2 – Flowchart

Goto was already explained at the very beginning of the text. The example of ***break*** and ***continue*** statements in Java is demonstrated in the code below (in Java).

```
int [] numbers = {10, 20, 30, 40, 50};


//break will stop the execution of the loop
for(int x : numbers ) {
      if( x == 30 ) {
      break;
      }
System.out.print( x );
System.out.print("\n");
}
//printed numbers will be 10 and 20



//continue will cause the loop to jump to the next iteration
for(int x : numbers ) {
      if( x == 30 ) {
            continue;
}
```

```
System.out.print( x );

System.out.print("\n");

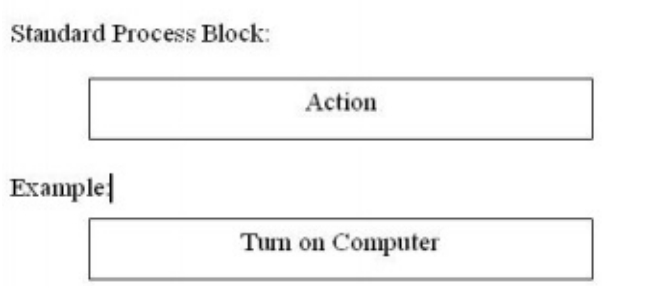
}

//printed numbers will be: 10, 20, 40, 50
```

Structograms use the following diagrams: process blocks, branching blocks, testing loops.

Process blocks represent the simplest actions and don't require analysis. Actions are performed block by block.

Figure 3 – Process block



Branching blocks are of two types – True/False or Yes/No block and multiple branching block. The simple block offers the program two options depending on whether or not the condition has been fulfilled. They can be used as a looping procedure that stops a program until the condition is fulfilled. Multiple branching blocks are used the program requires a select case and they provide an array of choices.

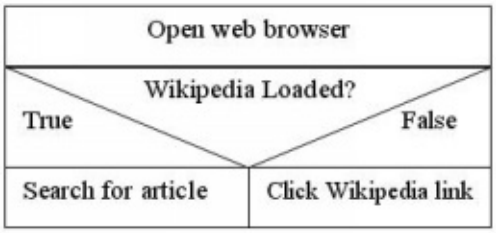


Figure 4 – True/False branching block

Testing loops allow the program to repeat one or many processes until a condition is fulfilled. There are two types of testing loops – test first and test last blocks – and the order in which the steps are performed is what makes them different. With test first, the program checks whether the condition is fulfilled. If it isn't, the program completes actions (process blocks), loops to the beginning and tests the condition again. When the condition is fulfilled, the program skips inner process blocks and moves onto the next block. The steps in test last loops are reversed – the process blocks are completed before the first test is performed. The important difference is that, in this case, actions in process blocks will be performed at least once.

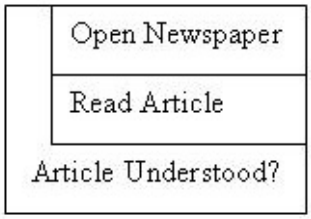
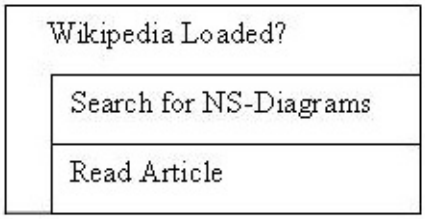


Figure 6 – Test last

Figure 5 – Test first



3.2. Subroutine

Subroutine is a sequence of program instructions packaged as a unit so that together they perform a specific task. The unit is used wherever that task needs to be performed. Subroutines can be defined within programs or separately (libraries used by multiple programs). Depending on the programming language,

subroutine can be called a ***procedure***, a ***function***, a ***routine***, a ***method***, or a ***subprogram***. ***Callable unit*** is sometimes used as a generic term.

Callable units behave in the same manner as a program but they are coded so they can be called several times. They can also be called from different places like from the program itself or other subroutines after which they return to the first instruction that comes after the call.

Subroutine's content is called its ***body*** and it contains code that is executed when the subroutine is called (invoked). They can acquire one or more values – ***parameters*** – from the part of the program that called it. During the call, actual values are provided and they are called ***arguments***. Besides inputting data, subroutines may return a value/result or change the value of the argument passed on to him and provide it as an output parameter. If a subroutine is coded so that it calls itself, it is ***recursive***.

Typical structured programming languages such as ***Pascal*** or ***Ada*** distinguish ***functions/function subprograms***, which return a value when called, and ***subroutines/procedures***, which do not. Functions are normally a part of an expression while procedure call is equal to a statement.

```
(*procedure call where input is the name and n is a parameter*)
input(n);


(*function call where f is the name and n is a parameter*)
x = f(n);
```

Subprograms can also have variables called ***local variables***. Every subprogram also stores an address that refers to where control should be passed after the subprogram finishes. Before mentioned structured programming languages also support nested subroutines. Those are subroutines callable only within the parent (outside) subroutine. They have access to the local variables of the outer subroutine.

Convention says that subprogram's name should be a verb associated with the task it does (i.e. ***calculate*** for a subprogram that performs calculations). Subroutines should also perform only one task and have minimal dependency on the rest of the code.

3.3. Block

Block is a section of code grouped together and it consists of one or more declarations and statements. The concept of blocks is fundamental to structured programming where blocks form control structures. Blocks enable groups of statements to be treated as a single statement while also narrowing the scope of variables, procedures and functions within a block so that they do not conflict with variables with the same name outside the block.

Blocks have different syntax in different languages but two approaches form two main families:

- the ALGOL family – blocks are limited by the keywords "begin" and "end"

- the C family – blocks are limited by curly braces “{“ and “}”

Two types of blocks can be distinguished based on their location within a program:

- block associated with a procedure – the block corresponds to the procedure’s body

```
//the following code is a procedure
//brackets determine the beginning and the end of the block
int square(int x)
{
int squareX;
squareX = x * x;
return squareX;
}
```

- in-line block – this block can appear wherever a command can appear

```
//the following code is a part of a class
```

```
//the class contains a method with its own variables
class ExampleClass {
...
public methodExample() {
int n;

//the following part is inside brackets
//it contains declaration of a new variable
//this variable is not visible outside of the block
{
int squareN = n * n;
System.out.println(squareN);
}
...
}
}
```

Since a block can be used in the same places a command can, a particular case of

using blocks would be to create a tree-like structure by nesting one block inside another.

Some languages do not fully support declarations within a block. For example, some don’t allow a function definition/nested function within a block (C-derived languages) and Pascal does not permit the use of blocks with their declarations inside the begin and end of an existing block (only statements like if, while, repeat).[31]

3.4. Indentation

Another typical characteristic of structured programming is indent style applied to a block to display program structure. In most programming languages, indentation is not a requirement but when used, code is easier to read and follow.

Tab is usually used for indentation so the size of the indent varies since it depends on the editor and environment. There are also different styles used and each has specified rules but that won’t be covered in this topic.

3.5. Control structure

In structured programming, all programs are composed of three structures:

- sequence
- selection
- iteration

Instructions that are executed in the order in which they appear form a ***sequence***. Instructions are coded as a sequence when they are executed in order regardless of any existing condition. When looking at the flowchart, instructions are executed from top to bottom (Figure 7). If structograms are used, sequence is represented with process blocks (Figure 3).

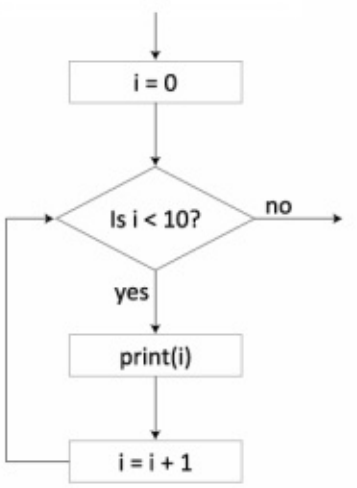


Figure 7 - Sequence

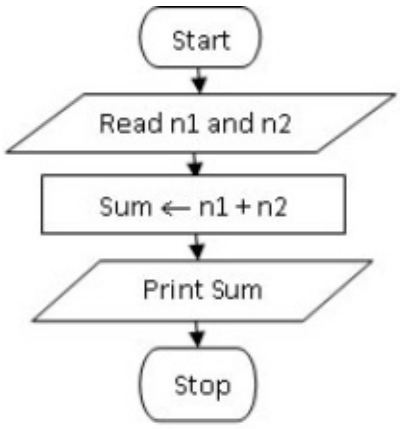


Figure 8 - Iteration

Selection is a construct with logical control and it executes instructions depending on the existence of a condition. As a logical control structure, it is called IF-THEN-ELSE. Usually, there are several sets of instructions that are executed when different condition is fulfilled. This means that not all of the instructions will be executed. Flowcharts representing selection contain a condition and two options, one on the right and one on the left where one side is chosen when the condition is true and the other when it is false (Figure 2). Structograms use branching blocks to portray this structure (Figure

4).

Iteration is a term for the logical control structure that enables executing a series of steps repeatedly until a specific condition becomes true. Depending on the programming language used, iteration is performed while condition is false or while it is true, it isn't the same everywhere. After the condition is met, the program continues execution in sequence starting after the iteration structure. This type of iteration is referred to as a loop. In order for the loop to terminate properly, it is important that the condition that is tested must at some point be true. If the instructions within iteration structure never change the condition, the process will repeat itself without programmed termination. This error is called an infinite loop and needs to be avoided in structured programming. Structograms use test loops (Figures 5 & 6) and a flowchart representation of iteration is shown below (Figure 8).[32]

4. Representations in different languages

4.1. Commands for explicit sequence control

The sequential command allows programmers to directly specify the sequential execution of two commands and it is represented in many languages with ";". The following line of a code indicates that ***C2*** starts executing immediately after ***C1*** terminates.

C1 ; C2

C1 ; C2 ; ... ; Cn

As written above, sequence of commands can contain more than two commands

where the operator “;” is left-associative. A sequence of commands can also be grouped as was mentioned in section 3.3. When talking about blocks, it was also mentioned that there are two typical ways of grouping code. So in ALGOL and C it would be:

```
begin
...
end
```

```
{
...
}
```

The operator “;” is not always a sequential command and its meaning depends on the language. In languages like C, C++ and Java, it is a command terminator and it is always required after a command, no matter where it is within a block.

```
{
x = 1;
x = x + 1;
```

```
}
```

On the other hand, languages like Pascal see ";" as a sequential command and the last command inside a block does not have to contain it at the end.

```
begin
x := 1;
x := x + 1
end
```

If there is a ";" at the end of the command and there is no command coming after it, compiler solves this problem by inserting an empty command between the last ";" and the ***end***. Empty command has no associated action and is not visible in the code.
For example, if two previous examples were combined and the code is as follows

```
begin
x := 1;
x := x + 1;
end
```

after executing ***x := x + 1***, there comes an empty command ; that has no effect on the rest of the code. After that, command is transferred to the part of the code that comes after this block. More information on the use of the command ";" in Pascal can be found here.[33]

4.2. Conditional commands

Conditional or selection commands are used for choosing which set of commands will be executed based on the value of a condition. Conditional commands can be divided into two groups:

- IF
- CASE

If is a command present in a lot of programming languages and its form can be generalized as

```
if Condition then C1 else C2
```

where ***Condition*** is a Boolean expression and ***C1*** and ***C2*** are commands. If ***condition*** is ***true***, ***C1*** is executed, otherwise ***C2*** is executed. ***Else*** branch is not mandatory so if ***condition*** is ***false***, ***C1*** is not executed and control passes to the command immediately after the conditional command.

```
if Condition then C1
```

If commands can be nested as well, but that requires more attention when writing. With nested commands, ***else*** branch belongs to the innermost if in almost every language. To avoid this problem, some languages use a terminator that indicates where the conditional command actually ends:

```
if Condition then C1 else C2 endif
```

In some cases, instead of using just ***if then else***, there is another branch ***elseif*** that replaces nesting:

```
if Condition1 then C1
elseif Condition2 then C2
...
elseif Condition then Cn
else Cn + 1
endif
```

Second type of conditional command is ***case*** which is a special case of the ***if*** command with more branches. Its simplest form would be as follows:

```
case Expression of
```

label1: C1;

label2: C2;

...

labeln: Cn;

else Cn + 1

where ***Expression*** is of the type compatible with the type of labels ***label1, label2, ..., labeln***. ***C1, ..., Cn + 1*** are commands. Each label is represented by one or more constants and different labels have different constants. The type permitted for labels and how they are written varies in different languages but they are usually simple types (their values aren't made out of combinations of other values) like integer, real, character...
The meaning of ***case*** is equal to multibranch ***if***. First, ***Expression*** is evaluated and then it is compared to values of labels. When the right label is found, command with that label is executed. If there is no label with the value of the expression, the command in the ***else*** branch is executed.

Whatever can be done using a case can also be written using nested ***ifs***. Many languages include some form of case since it can improve the readability of the code and compile more efficiently.

Languages like C or Java have a different syntax of the ***case*** command. Let's look at the code example first:

switch (Expression) {

```
case label1: C1 break;

case label2: C2 break;

...

case labeln: Cn break;

default: Cn + 1 break;

}
```

When ***Expression*** is evaluated, control is transferred to the appropriate label and its commands (as already explained), or to the label ***default*** if there is no right label. ***Default*** is not mandatory. This form of ***case*** uses ***break*** as an explicit control transfer at the end of each command so that control can be transferred to the command coming after the entire ***switch*** block instead of to the next label.

Ranges of values are allowed types for labels in the first type of ***case***. ***Switch*** does not permit this type but lists of values can be implemented using the fact that control passes from one branch (label) to the next if there is no ***break***. Consequently, if program needs to execute ***C1*** if ***Expression*** is 1, 2 or 3, this is how it will be coded:

```
switch (Expression) {

case 1:

case 2:

case 3: C1 break;
```

```
case 4: C2 break;

default: C3 break;

}
```

4.3. Iterative commands

The commands mentioned so far execute chosen commands only once. A program with only commands like that would not be efficient and could not implement all algorithms, like simple input of ***n*** elements, where ***n*** is not a value known beforehand.
In order to implement all algorithms, it is important to learn instructions that allow repetition of groups of instructions by jumping to the start of the code. It was already explained that ***goto*** commands need to be avoided. Instead, two basic mechanisms are employed to achieve the same effect:

- structured iteration
- recursion

Structured iteration allows formation of loops in which commands are repeated or iterated. It is also possible to distinguish ***unbounded iteration*** from ***bounded iteration***. In bounded iteration, commands are repeated a specific number of times. On the other hand, unbounded iteration repeats commands until a specific condition is met.

Unbounded iteration is logically controlled iteration and is composed of two

parts: a loop condition and a body composed of commands. When executed, the body is executed until the condition becomes false (or true, depending on the concept). The form of the ***while*** command is the most common:

```
while (Condition) do C
```

Condition is of type Boolean (logical) and evaluating it is the first step. If this evaluation returns ***true***, command ***C*** is executed and returned back to the first step – condition evaluation. If ***Condition*** returns ***false***, the ***while*** command terminates.

In some languages, condition can be tested after execution of the command so it is always executed at least once. In Pascal, this would be coded as follows:

```
repeat C until Condition
```

This is nothing but a shorter way to write:

```
C;
while not Condition do C
```

Here, ***Condition*** is negated. The same code and its corresponding code is written like this in C:

do C while (Condition)

C;

while Condition do C;

While command makes it possible for a language to implement all computable functions. The same cannot be said for bounded iteration.

Bounded iteration is implemented by constructs more complex than those used for unbounded iteration. Command used is called ***for*** and it can be described, without using any specific syntax, as:

for i = start to end by step do

body

Here, ***i*** is a variable called index or counter; ***start*** and ***end*** are usually of integer type; ***step*** is a non-zero integer constant; ***body*** is the command that will be repeated. When using ***for***, counter cannot be modified during the execution of the body.

Assuming that ***step*** is positive, the semantics of ***for*** can be described as:

1. Expressions ***start*** and ***end*** are evaluated. The values are frozen and stored in dedicated variables which cannot be updated by the

programmer. They could be called ***start_save*** and ***end_save***.

2. ***I*** is initialized with the value of ***start_save***.
3. If the value of ***I*** is strictly greater than the value of ***end_save***, ***for*** is terminated.
4. ***Body*** is executed and ***I*** is incremented by the value of ***step***.
5. Control returns to step 3.

If ***step*** is negative, step 3 tests whether ***I*** is strictly less than ***end_save***.

Based on the fact that ***index*** is not changed within the execution of the ***body*** and the value of ***step***, the number of iteration can be determined with the following formula (where ***ic*** stands for iteration count and is either positive or 0):

$$ic = \left\lfloor \frac{end - start + step}{step} \right\rfloor$$

It is impossible to create an infinite cycle with this loop.

There are considerable differences between the versions of ***for*** in different languages, and here are the most important four:

- ***Number of iterations*** In the majority of languages, if ***step*** is positive and ***start*** is strictly greater than the value of ***end***, ***body*** is not executed at all.

//commands in the body will not be executed

for i = 5 to 2 by 1 do

command1;

...

commandN;

- ***Step*** It is necessary for ***step*** to be a non-constant zero so compiler can generate the appropriate code. To enable negative steps, languages like Pascal and Ada use special syntax like ***downto*** or ***reverse*** instead of ***to***. Other languages test ***ic*** and if it is negative, it is decremented by one until it reaches 0 while ***I*** and ***end*** are not used.

//code that successfully uses negative steps (in Pascal)

for i := 20 downto 1 do

...

- ***Final index value*** In many languages, ***I*** is a global variable (visible outside the loop) so it is important to know which value ***I*** has after loop has terminated. Naturally, this would be the value that was last assigned to ***I*** but ***for*** can cause an overflow. This is why some languages do not specify what the value of ***I*** should be.

- ***Jump into a loop*** Most languages forbid using ***goto*** command to jump into the middle of a ***for*** loop. But there are fewer restrictions regarding using ***goto*** for jumping out of a loop.

```
for i = 0 to 10 do
...
if (EndCondition) then
goto end;
...
end : EndCommand
```

It is most common for iterative constructions to perform sequential scanning of all the elements of the data structure. Implementing this function includes providing a lot of details that the compiler already knows (like indexes of an array). To make it easier to understand the code and harder to make a mistake, it is better to apply ***body*** execution to every element of the structure.

Some languages use a special construct for this kind of operation, called ***for-each***.

```
foreach (FormalParameter : Expression) Command
```

Command is applied to each element of ***Expression***. Besides arrays and vectors,

for-each can be used on collections. In Java, syntax is slightly different since only ***for*** is written. This language allows the usage of the loop on all subtypes of the library type ***Iterable***.[34]
The following codes are concrete examples of ***for-each*** in several languages.

Java:

```
List<int> numbers = new ArrayList<>();
...
for (int n : numbers) {
System.out.println(“” + n);
}
```

C#:

```
int[] numbers = new int[] {0, 1, 1, 2, 3, 5, 8, 13};
foreach (int n in numbers) {
System.Console.WriteLine(n);
}
```

Ruby:

```
array = [1,2,3,4,5]

array.each do |i|

puts i

end
```

5. Deviations and languages today

Even though the ***goto*** statement has been replaced by constructs of selection (if, case) and iteration (while, for), few languages remain purely structured today. In Java, ***goto*** is a reserved word but it is unusable while Python doesn't support ***goto***. Functional languages generally lack ***goto*** as well.

There are languages that support ***goto*** while a lot of others have an alternative approach to early exits and jumping between lines of the code. The typical example of ***goto*** usage today would be within the language C. It is used mostly in cases of implementing multi-level breaks. It is also thought that ***goto*** provides the most straightforward way to make the code more readable, make smaller programs without code duplications, etc. At machine code level, ***goto*** is used to implement the structured programming constructs.

Most common deviation in languages widely used today is early exit from a function or a loop. Statement ***return*** is used inside functions and statements ***break*** and ***continue*** are used inside loops. As a result, there are multiple exit point instead of a single exit point required by structured programming.

There have been arguments regarding throwing exceptions and violating the single-exit rule. In C++, this issue was resolved by adding the signature ***throw()*** to all functions. Exception handling is also supported in Java where the programmer has to declare that an exception will be thrown inside a specific method or surround the area that can lead to exceptions with ***try-catch*** block. Until one of these two things are done, the program won't compile.

Even though the rules of structured programming say otherwise, programming languages today require support for multiple exits. Besides single-exit rule being slightly ignored, introduction of structured programming revolutionized the way programs are written. Nevertheless, its concepts are part of languages that are based on other paradigms as well.

Bibliography

1. GABBRIELLI, M., MARTINI, S. (2010) ***Programming Languages: Principles and Paradigms.*** Springer.

2. STERN, N., STERN, R. A. (1991) ***Structured COBOL Programming.*** 6th Edition. Wiley

3. SCRIBD (2015) ***Structured Programming.*** [Online] Available at: https://www.scribd.com/doc/243311568/Structured-Programming [Accessed: 9th June 2015]

4. TUTORIALSPOINT (2015) ***goto statement in C*** [Online] Available at: http://www.tutorialspoint.com/cprogramming/c_goto_statement.htm [Accessed: 9th June 2015]

5. FILL, H. G. (2009) ***Visualisation for Semantic Information Systems***

Procedural Programming

1. History and Languages

Procedural languages are computer languages used to define the actions that a computer has to follow to solve a problem. Although it would be convenient for people to give computers instructions in a natural language, such as English, French, or Chinese, they cannot because computers are just too inflexible to understand the subtleties of human communication. Human intelligence can work out the ambiguities of a natural language, but a computer requires a rigid, mathematically precise communication system: each symbol, or group of symbols, must have the exact same meaning every time.

Computer scientists have created artificial languages that enable programmers to assemble a set of commands for the machine without dealing directly with strings of binary digits. The high-level form of a procedural language frees a programmer from the time-consuming chore of expressing algorithms in lower-level languages such as assembly and machine language. Additionally, procedural language instructions are expressed in a machine-independent form that makes it easier for the program to be portable, thus increasing the lifetime and usefulness of that program.

The idea of procedural programming was born around 1958, before structured programming and other paradigms. Procedural programming was necessary to enable breaking complex programs into smaller units. These units had a set of simple tasks called procedures and they are linguistically thought of as verbs.

Higher-level languages work for people because they are closer to natural language, but a computer cannot carry out instructions until that communication has been translated into zeros and ones. This translation may be done by compilers or interpreters. A compiler reads the entire program, makes a translation, and produces a complete binary code version, which is then loaded into the computer and executed. Once the program is compiled, neither the original program nor the compiler is needed. On the other hand, an interpreter translates and executes the program one instruction at a time, so a program written in an interpreted language must be interpreted each time it is run. Compiled programs execute faster, but interpreted programs are easier to correct or modify.

A procedural language is either compiled or interpreted, depending on the use for which it was created. FORTRAN, for example, is usually implemented with a compiler because it was created to handle large programs for scientific and mathematical applications where speed of execution is very important. On the other hand, BASIC is typically implemented with an interpreter because it was intended for use by novice programmers.

Besides FORTRAN and BASIC, early procedural languages include ALGOL and COBOL. Some of the most known languages that were later developed and use the concepts of procedural programming are Pascal, Modula, and most importantly – C, which will be used throughout this text.

C is one of the descendants of ALGOL 60. It was developed in 1972 by Ken Thompson and Dennis Ritchie, both of Bell Laboratories. Their goal was to create a language that would combine high-level structured language features with those that control low-level programming. This makes C well suited for writing operating systems, compilers, and also business applications. C compilers can basically run on all machines, and since a standard for C was

defined in 1988, most C programs are portable. Conversely, C has been defined as a programming language written by a programmer, which means that novices find it difficult to learn.

C supports structured programming and provides for several data types. For example, pointer arithmetic is an integral part of C, as is the use of functions that may be called recursively. Although input and output statements are not part of the language, they are functions found in a "library" ready to be used when needed. Some of the functions found in a standard UNIX C library include string manipulation, character functions, and memory allocation. In addition to external, automatic and static variables, C provides register variables, which shorten execution time because they use registers.

C makes it possible to work on bit data using the bit operators for AND, OR, Exclusive OR, One's complement, SHIFT LEFT, and SHIFT RIGHT, giving programmers great control over data manipulation.

When compared to other programming languages such as FORTRAN or Pascal, C has remained quite stable. Its success in the early 1980s was due in part to its close ties with UNIX and its availability on personal computers. Additionally, it satisfied the needs of both system and application programmers alike.[35]

2. Overview

Procedural programming is a programming paradigm, derived from structured programming. Structure is based on the concept of the procedure call. There are different names for procedures:

•Routines

•Subroutines
•Methods
•Functions

Procedure might be called at any point during a program's execution, including by other procedures or itself (this situation is called recursion). Procedural programming is a list or set of instructions telling a computer what to do step by step and how to perform from the first code to the second code.

Procedural programming relies on procedures, also known as routines. A procedure contains series of computational steps to be carried out (we call that a function). Procedural programming is also referred to as imperative or structured programming.
Procedural programming is intuitive in the sense that it is very similar to how you would expect a program to work. If you want a computer to do something, you should provide step-by-step instructions on how to do it. Many of the early programming languages are all procedural.

A common technique in procedural programming is to repeat a series of steps using iteration. This means a series of steps is written and then the program is told to repeat these steps a certain number of times. This makes it possible to automate repetitive tasks.[36]

3. Concepts and their representation in C
3.1. Modularity

Basic definition of modularity is as follows: “Modularity is the degree to which a system's components may be separated and recombined.”

Modular programming is important in any programming paradigm including, but by no means limited to, procedural languages. A modular approach encourages taking one step at a time, promotes team work and collaboration and, most importantly, promotes testing on small and isolated units prior to integrating them into a large system.

Many new C programmers will quite happily write a large program and have all the source code in one huge file. They press F9 and it compiles quite quickly on the latest machines, so why bother splitting it up into separate C files? Here are the reasons:

•A change to one file means that only that file needs to be recompiled
•It is easier to edit and navigate smaller self-contained files
•Allows more than one person to work on a particular project at once
•It is much simpler to reuse code that is in a self-contained module or file
•It is slightly easier to isolate compile-time and run-time bugs

So, when writing your programs, the best approach would be to try and split up source code into modules. Modules are self-contained units which can usually be reused and are easy to maintain and debug.

3.2. Variables

A variable is nothing but a name given to a storage area that programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and

lowercase letters are distinct because C is case-sensitive. Variables have to be of certain type and in C, basic variable types are:

Type	Description
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

C programming language also allows definition of various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc.

A variable definition means to tell the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, type must be a valid C data type including char, w_char, int, float, double, bool or any user-defined object, and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are:

```
int i, j, k;

char c, ch;

float f, salary;
```

```
double d;
```

The line ***int i, j, k;*** both declares and defines the variables ***i***, ***j*** and ***k***, which instructs the compiler to create variables named ***i***, ***j*** and ***k*** of type ***int***.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For a definition without an initializer, the following rules apply:

- variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0);
- the initial value of all other variables is undefined.

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that the compiler proceeds with further compilation without needing all details about the variable. A variable declaration has its meaning at the time of compilation only; compiler needs actual variable declaration at the time of linking of the program.

Declaration is useful when using multiple files and defining variable in one of the files which will be available at the time of linking of the program. Extern keyword is used to declare a variable at any place. Though a variable can be declared multiple times in a C program, it can be defined only once in a file, a function or a block of code.

Here is an example where variables have been declared at the top, but they have been defined and initialized inside the main function:

```
#include <stdio.h>


// Variable declaration:
extern int a, b;
extern int c;
extern float f;


int main ()
```

```
{

/* variable definition: */

int a, b;

int c;

float f;


/* actual initialization */

a = 10;

b = 20;


c = a + b;

printf("value of c : %d \n", c);

f = 70.0/3.0;

printf("value of f : %f \n", f);


return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
value of c : 30
value of f : 23.333334
```

Same concept applies on function declaration where a function name is provided at the time of its declaration and its actual definition can be given anywhere else. For example:

```
// function declaration
int func();

int main()
{
// function call
int i = func();
}

// function definition
int func()
```

```
{

return 0;

}
```
[37]

3.3. Special variables - pointers

Pointers are not a basic type but are very important for programming in C. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So, it becomes necessary to learn pointers to become a perfect C programmer.

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>

int main ()

{

int  var1;

char var2[10];
```

```
printf("Address of var1 variable: %x\n", &var1  );

printf("Address of var2 variable: %x\n", &var2  );


return 0;

}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6
```

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, a pointer must be declared before it can be used to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk that is used for multiplication. However, in this statement the

asterisk is being used to designate a variable as a pointer. Following are the valid pointer declarations:

```
int    *ip;    /* pointer to an integer */

double *dp;    /* pointer to a double */

float  *fp;    /* pointer to a float */

char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

When there is no exact address to be assigned, a pointer can be assigned a NULL value. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>


int main ()

{
```

```
int  *ptr = NULL;

printf("The value of ptr is : %x\n", ptr  );

return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, an if statement can be used:

```
if(ptr)     /* succeeds if p is not null */

if(!ptr)    /* succeeds if p is null */
```[38]

3.4. Variable scope

Now that the basic use of variables was reviewed, the next step is to explain variable scope, a concept very important to understand when using smaller units like procedures. A scope in programming is a region of the program where a defined variable exists but cannot be accessed beyond it. There are three places where variables can be declared in C programming language:

1. Inside a function or a block where they are called ***local*** variables;
2. Outside of all functions where they are called ***global*** variables.
3. In the definition of function parameters where they are called ***formal*** parameters.

Local variables can be used only by statements that are inside that function or block of code. They are not known to functions outside their own. Following is the example using local variables. Here, all the variables ***a***, ***b*** and ***c*** are local to ***main()*** function.

```
#include <stdio.h>


int main ()

{

/* local variable declaration */
```

```
int a, b;

int c;


/* actual initialization */

a = 10;

b = 20;

c = a + b;


printf ("value of a = %d, b = %d and c = %d\n", a, b, c);


return 0;

}
```

Global variables are defined outside of a function, usually on top of the program. They will hold their value throughout the lifetime of the program and they can be accessed inside any of the functions. A global variable is available for use throughout the entire program after its declaration. Following code is the example using global and local variables:

```
#include <stdio.h>
```

```
/* global variable declaration */

int g;


int main ()

{

/* local variable declaration */

int a, b;


/* actual initialization */

a = 10;

b = 20;

g = a + b;


printf ("value of a = %d, b = %d and g = %d\n", a, b, g);


return 0;

}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
{
/* local variable declaration */
int g = 10;

printf ("value of g = %d\n",  g);

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of g = 10
```

Function parameters, or formal parameters, are treated as local variables within that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main ()
{
/* local variable declaration in main function */
int a = 10;
int b = 20;
int c = 0;

printf ("value of a in main() = %d\n",  a);
```

```
   c = sum( a, b);
   printf ("value of c in main() = %d\n",  c);

   return 0;
   }

   /* function to add two integers */
   int sum(int a, int b)
   {
   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
   }
```

When the above code is compiled and executed, it produces the following result:

```
   value of a in main() = 10
```

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30[39]

One more example for scope of variables inside block (function):

```
int a;


void f(void)
{
float a;/* different from global `a' */
float b;


a = 0.0;/* sets `a' declared in this block to 0.0 */
/* global `a' is unaffected */
{
int a;/* new `a' variable */


a = 2;/* outer `a' is still set to 0.0 */
```

```
}

b = a;/* sets `b' to 0.0 */

}


/* global `a' is unaffected by anything done in f() */
```

3.5. Functions

All the instructions of a C program are contained in functions. Each function performs a certain task. A special function with the name ***main()*** is the first one to run when the program starts. All other functions are subroutines of the ***main()*** function (or otherwise dependent procedures, such as call-back functions), and can have any name. Every function is defined exactly once while the program can declare and call a function as many times as necessary.[40]

In C programming, all executable code resides within a function. A function is a named block of code that performs a task and then returns control to the caller. Note that other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method" but in C, these are all functions.

A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call.

As a basic example, suppose you are writing code to print out the first 5 squares

of numbers, do some intermediate processing, then print the first 5 squares again. We could write it like this:

```
#include <stdio.h>


int main(void)
{
int i;
for(i=1; i <= 5; i++)
{
printf("%d ", i*i);
}
for(i=1; i <= 5; i++)
{
printf("%d ", i*i);
}
return 0;
}
```

The same loop has to be written twice. A programmer may want to somehow put this code in a separate place and simply jump to this code whenever it needs to be used. This is what it would look like:

```
#include <stdio.h>


void Print_Squares(void)

{

int i;

for(i=1; i <=5; i++)

{

printf("%d ", i*i);

}

}


int main(void)

{

Print_Squares();

Print_Squares();
```

```
return 0;
}
```

In general, declaring a function is done in the following form:

```
type name(type1 arg1, type2 arg2, ...) /* function head*/
{
/* code = function block */
}
```

The function head specifies the name of the function, the type of its return value, and the types and names of its parameters, if any exist. The statements in the function block specify what the function does. Declaring a function means informing the compiler of its type. A declaration must indicate at least the type of the return value such as:

```
int rename();
```

The line declares ***rename*** as a function that returns a value with type ***int***. Because function names are external identifiers by default, that declaration is equivalent to the next one:

```
extern int rename();
```

This declaration does not include any information about the number and the types of the function's parameters. as a result, the compiler cannot test whether a given call to this function is correct so calling the function with arguments that are different in number or type from the parameters in its definition may result in a runtime error. To avoid this error, it is best to declare parameters as well.[41]

A function can take no arguments, or can return nothing, or both. To enable this, C keyword ***void*** is written. "Void" basically means "nothing" - so if we want to write a function that returns nothing, we would write

```
void sayHello(int number_of_times)
{
int i;
for(i=1; i <= number_of_times; i++) {
printf("Hello!\n");
}
}
```

Notice that there is no return statement in the function above. That's why we write ***void*** as the return type. Besides this, one can use the ***return*** keyword in a procedure to return to the caller before the end of the procedure, but one cannot return a value as if it were a function.

A function cannot return an array or a function. However, it can be defined so that it returns a pointer to a function or a pointer to an array. Arrays can be passed as an argument to a function, and that would be written as

```
type name[]
```

Array names are automatically converted to pointers when used as function arguments so the previous statement is equivalent to the declaration

```
type *name
```[42]

The next example shows the function that does not take arguments.

```
float calculate_number()
{
float to_return=1;
int i;
for(i=0; i < 100; i++) {
to_return += 1;
to_return = 1/to_return;
}
```

```
return to_return;

}
```

This function doesn't take any inputs, but it returns a number calculated by this function. Naturally, both ***void*** return and ***void*** in arguments can be combined together to get a valid function, as well.

C allows defining functions that can be called with a variable number of arguments. In this case, the list of parameters ends with an ellipsis after the last comma. The ellipsis represents optional arguments.

```
int printf(const char *format,...);
```

Typically, parameters of a function in C are local variables initialized with the argument values. The advantage of this approach is that any expression can be used as an argument as long as it has the appropriate type. On the other hand, the drawback is that copying large data objects to begin a function call can be expensive. Moreover, a function has no way to modify the original variables as it only knows how to access the local copy. This approach is called ***call by value***.

There is a second approach to calling a function. It can directly access any variable visible to the caller if one of its arguments is that variable’s address. These functions in C are ***call by reference*** functions and here is an example of this call:

```
int var;
```

```
scanf(“%d”, &var);
```

This call reads a string as a decimal numeral, converts it to an integer and stores the value in the location of ***var***. Even when a function needs to merely to read and not modify a variable, it can still be more efficient to pass the variable’s address rather than just its value. The reason is that passing by address avoids the need to copy the data.

Executing a function, or a function call, consists of the function’s name and the operator (). For example, the following statement calls the function ***maximum()*** to compute the maximum of the matrix ***mat*** with ***r*** rows and ***c*** columns:

```
maximum(r, c, mat);
```

Firstly, storage space for the parameters is allocated and argument values are copied to the corresponding locations. Secondly, the program jumps to the beginning of the function and its execution starts with the first variable definition or statement within the function block. If the program reaches the ***return*** statement or the closing brace } of the function block, execution of the function ends.[43]

3.6. Recursion

Here's a simple function that does an infinite loop. It prints a line and calls itself, which again prints a line and calls itself again, and this continues until the stack

overflows and the program crashes. A function calling itself is called ***recursion***, and normally there will be a condition that stops the recursion after a small, finite (defined) number of steps.

```
void infinite_recursion()
{
printf("Infinite loop!\n");
infinite_recursion();
}
```

The next example will demonstrate the usage of condition with recursive functions. Note that ++depth is used so the increment will take place before the value is passed into the function. Alternatively, value can be increment in a separate line before the recursion call. If the call of the function is ***printMe(3,0);*** the function will print the line Recursion 3 times.

```
void printMe(int j, int depth)
{
if(depth < j) {
printf("Recursion! depth = %d j = %d\n", depth, j); //j keeps its value
printMe(j, ++depth);
}
```

```
}
```

Recursion is most often used for jobs such as directory tree scans, seeking for the end of a linked list, parsing a tree structure in a database and factoring numbers (and finding primes) among other things.

3.7. Static functions

If a function should be called only from within the file in which it is declared, it is appropriate to declare it as a static function. When a function is declared static, the compiler will compile the function to an object file in a way that prevents it from being called within a code in other files. Example:

```
static int compare( int a, int b )
{
return (a+4 < b)? a : b;
}
```

4. Example with different procedural approaches

Basic approaches of procedural programming paradigm were previously

explained through language C. Now can examine a specific example and how the problem can be solved.

Since programming approaches can sound a little bit abstract, let's look at this example. The task is to calculate Fibonacci numbers. By definition, the first two numbers in a Fibonacci sequence are 0 and 1, and the following numbers are always the sum of the previous two. In mathematical terms, here is what the sequence looks like:

$$F_n = F_{n-1} + F_{n-2}$$

The result is a sequence that looks like this:

0 1 1 2 3 5 8 13 21 34 55 89 …. n

Fibonacci numbers are used for a wide range of applications, from describing patterns in nature to the structures of galaxies. So what is the Fibonacci number for n = 100? This is exactly the kind of task you want a computer program to do for you. Don't worry too much about understanding the math - the point is that we want a computer program to do the calculations.

In procedural programming, here is what the code would look like to get the 'nth' number in the Fibonacci sequence:

```
#include<stdio.h>
```

```
int main()

{

int n, first = 0, second = 1, next, c;


printf("Enter the number of terms\n");

scanf("%d",&n);


printf("First %d terms of Fibonacci series are :-\n",n);


for ( c = 0 ; c < n ; c++ )

{

if ( c <= 1 )

next = c;

else

{

next = first + second;

first = second;
```

```
second = next;
}
printf("%d\n",next);
}
return 0;
}
```

The result is the same sequence that we defined earlier:

```
0 1 1 2 3 5 8 13 21 34 55 89 …. n
```

The basic idea behind this code is to use a looping structure that iterates a series of steps. With every iteration, the next value in the sequence is processed. After one iteration, the value of the variable ***fib*** is 1, after the second iteration the value is 2, etc. There are ***n*** iterations.

Same problem can be written using recursion.

```
#include<stdio.h>

int Fibonacci(int);
```

```
main()

{

int n, i = 0, c;


scanf("%d",&n);


printf("Fibonacci series\n");


for ( c = 1 ; c <= n ; c++ )

{

printf("%d\n", Fibonacci(i));

i++;

}

return 0;

}


int Fibonacci(int n)
```

```
{
if ( n == 0 )
return 0;
else if ( n == 1 )
return 1;
else
return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Recursion method is less efficient as it involves function calls which use stack. Also, there are chances that stack overflow will happen if function is called frequently for calculating larger Fibonacci numbers. Therefore, caution is required when writing code with recursive calls.[44]

5. Review of concepts in other languages

In ALGOL, there is a difference between units called ***function procedures*** and ***proper procedures***. Functional procedures are the procedures that have resulting value while proper procedure do not return a value. While both types are just called functions in C, ALGOL and some other languages distinguish these types. In case of a proper procedure, there is no need to write ***void*** to signalize that there is no returning type. Instead, simply ***procedure*** is written without any type coming before it. In FORTRAN and BASIC, user-defined functions are referred

to as ***subroutines***.

Dividing the code so that it consists of independent units is a concept associated with a lot of programming languages. In fact, it is more of a basis of structured approach to programming. Even though it is usually linked to imperative programming, languages like Java and C# have their own implementation of smaller units called ***methods***.
Instead of being an independent paradigm with strictly procedural languages, it is an approach that can be applied to a lot of languages, no matter which paradigm they are based on. As a result, procedural programming is a style that is not associated with a specific era but is used since the beginning of high-level programming languages until today.

Bibliography

1. PRINZ, P., CRAWFORD, T. (2005) ***C in a Nutshell.*** O'Reilly Media

2. ENCYCLOPEDIA (2015) ***Procedural Languages*** [Online] Available at: http://www.encyclopedia.com/doc/1G2-3401200261.html [Accessed: 10th June 2015]

3. STUDY (2015) ***Functional Programming and Procedural Programming*** [Online] Available at: http://study.com/academy/lesson/functional-programming-and-logic-programming.html [Accessed: 10th June 2015]

4. TUTORIALSPOINT (2015) ***C-Variables*** [Online] Available at: http://www.tutorialspoint.com/cprogramming/c_variables.htm [Accessed: 10th June 2015]

5. TUTORIALSPOINT (2015) ***C-Pointers*** [Online] Available at: http://www.tutorialspoint.com/cprogramming/c_pointers.htm [Accessed: 10th June 2015]

6. TUTORIALSPOINT (2015) ***C-Scope Rules*** [Online] Available at: http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm [Accessed: 10th June 2015]

7. WIKIBOOKS (2015) ***C Programming/Procedures and functions*** [Online] Available at: http://en.wikibooks.org/wiki/C_Programming/Procedures_and_functions [Accessed: 10th June 2015]

Functional Programming

F unctional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. Programming is done with expressions so it is a declarative programming paradigm. The key of functional programming is for the output value of a function to depend only on the arguments that are input. This means that calling the same function twice with the same argument will produce the same result each time.[45]

1. History

Functional programming has roots in lambda calculus which is a formal system developed in the 1930s used to investigate computability, function definition, function application, recursion. Lambda calculus is a mathematical abstraction rather than a programming language but it forms the basis of almost all

functional programming languages today.

An early language with functional approach was Lisp developed in the late 1950s. Even though Lisp is a multi-paradigm language, it introduced features that are now found in functional languages.[46] Sometimes, IPL is cited as the first computer-based functional language. It is an assembly-style language and code can be used as data so it can be considered that IPL has higher-order functions. However, it relies on mutating list structure.

In the 1960s, ***APL*** was developed and it had influence on FP developed in 1970s. ML[47] was introduced in the same decade and is very important since it developed into popular languages today OCaml and Standard ML.

The language Scheme brought awareness of the power of functional programming to the wider programming community. In 1987, Haskell started to develop and its function was to form an open standard for functional programming research. Implementation releases have been ongoing since 1990 while this decade also introduced J, K and Q.

As functional languages matured, their practical use grew. They are being used in areas such as database processing, financial modelling, statistical analysis and bio-informatics. Functional languages simply became more popular for general programming as the execution became more efficient.[48]

Overview

A number of concepts and paradigms are specific to functional programming and generally foreign to imperative programming as well as object-oriented programming. Still, programming languages are often hybrids of several

programming paradigms so programmers using mostly imperative languages may also have used concepts associated with functional programming.[49]

Although there is still some disagreement about what functional programming is, there are few features that are generally agreed to be the part of it and they are:

- first-class functions – they accept either another function as an argument or return a function
- pure functions – they are functions without side effects (side effects are actions a function may perform that are not solely contained within the function itself)
- recursion – allows writing smaller algorithms and operating by looking only at the inputs to a function
- immutable variables – variables that cannot be changed once set
- non-strict evaluation – allows having variables that have not yet been computed (variable does not get assigned until the first time it is referenced)
- statements – evaluable pieces of code that have a return value
- pattern matching – allows better type-checking and extracting elements from an object

It is possible to use functional style of programming in languages that are not traditionally considered functional. First-class functions were slowly added to mainstream languages like PHP, C# and C++. Since Java 8, this language

supports lambda expressions but functional programming in Java can be inconvenient because of the presence of checked exceptions – it can be necessary to catch checked exception and them rethrow it. This problem does not occur in languages such as Scala.

Functional programming has been popular in academia a lot longer than it has been used for industrial purposes. However, OCaml has been used in areas such as financial analysis, driver verification, industrial robot programming and static analysis of embedded software. Haskell has been applied in areas such as aerospace systems, hardware design, and web programming, even though it was initially developed as a research language. Other languages used in industry include Scala and F# - both with support for pure imperative and functional programming – Lisp, Standard ML, and Clojure.[50]

3. Mathematical background

3.1. Functions – basic concepts

In mathematics, a function is a relation between a set of inputs and a set of permissible outputs where each input is related to exactly one output. The output of a function symbolized as ***f*** that corresponds to an input ***x*** is denoted by ***f(x)*** (read “f of x”). The input variables are sometimes referred to as the arguments of the function.

There are many ways to describe or represent a function. Some are defined by a formula or algorithm that tells how to compute the output of the given input. Others are given by picture that is called the graph of the function. Functions are sometimes also defined by a table that gives the outputs for selected inputs. The input and output of a function can be expressed as an ordered pair so that the first element is the input and the second is the output. For example, if $f(x) = x^2$

and ***x*** is 3, the ordered pair is (-3, 9).

A function is defined by its set of inputs, called the ***domain***. A set containing the set of outputs, and possibly additional elements as members, is called its ***codomain***. The set of all input-output pairs is called its ***graph.***

A function ***f*** with domain ***X*** and codomain ***Y*** is commonly denoted by

$$f : X \to Y$$

Here, the elements of ***X*** are called arguments of ***f***. For each argument ***x***, the corresponding unique ***y*** in the codomain is called the function value at ***x*** or the image of ***x*** under ***f***. It can also be said that ***f*** associates ***y*** with ***x*** or maps ***x*** to ***y***. This is written as

$$y = f(x).$$

A function is called ***injective*** (or ***one-to-one***, or an ***injection***) if $f(a) \neq f(b)$ for any two different elements ***a*** and ***b*** of the domain. It is called ***surjective*** (or ***onto***) if $f(X) = Y$. That is, it is surjective if for every element ***y*** in the codomain, there is and ***x*** in the domain such that $f(x) = y$. Finally, ***f*** is called ***bijective*** if it is both injective and surjective.

Analog with arithmetic, it is possible to define addition, subtraction, multiplication, and division of functions and the output is a number. The following rules apply:

$$\begin{aligned}(f + g)(x) &= f(x) + g(x)\\ (f - g)(x) &= f(x) - g(x)\\ (f \quad g)(x) &= f(x) \quad g(x)\end{aligned}$$

$$\left(\frac{f}{g}\right)(x) = \frac{f(x)}{g(x)}$$

Another important operation defined on functions is composition, where the output from one function is the input to another function. The composition of ***f*** with a function $g : Y \to Z$ is the function $g \circ f : X \to Z$ defined by

$$(g \circ f)(x) = g(f(x)).$$

The value of ***x*** is obtained by first applying ***f*** to ***x*** to obtain $y = f(x)$ and then applying ***g*** to ***y*** to obtain $z = g(y)$. In the notation $g \circ f$, the function on the right (***f***) acts first and the function on the left (***g***) acts second. The notation is read as "***g*** of ***f***" or "***g*** after ***f***".

The unique function over a set ***X*** that maps each element to itself is called the ***identity function*** for ***X*** and is typically denoted by ***idX***. Under composition, an identity function is "neutral": if ***f*** is any function from ***X*** to ***Y***, then $f \circ id_X = f$ and $id_Y \circ f = f$.

An inverse function for ***f***, denoted by f^{-1}, is a function in the opposite direction, from ***Y*** to ***X*** with following rules satisfied:

$$f \circ f^{-1} = id_Y$$
$$f^{-1} \circ f = id_X.$$

For example, if ***f*** converts a temperature in degrees Celsius to degrees Fahrenheit, the function converting degrees Fahrenheit to degrees Celsius would be a suitable f^{-1}. Such an inverse function exists if and only if ***f*** is bijective. In this case, ***f*** is called invertible.[51]

3.2. Lambda calculus

Lambda calculus was invented by Alonzo Church in the 1930s as a mathematical formalism for expressing computation by functions, similar to the way a Turing machine is a formalism for expressing computation by a computer. Lambda calculus can be used as a model for purely functional languages mostly in the same way as Turing machines can be used as models for imperative languages. [52]

When looking at a function mathematically, a function is a rule for determining a value from an argument. Some examples of functions are:

$$f(x) = x^2 + 3$$

$$g(x + y) = \sqrt{x^2 + y^2}.$$

In the simplest, pure form of lambda calculus, there are no domain-specific operators such as addition and exponentiation, only function definition and application. This allows writing functions such as:

$$h(x) = f(g(x))$$

h is defined solely in terms of function application and other functions that are assumed to be already defined. It is possible to add operations such as addition and exponentiation to pure lambda calculus.

The main constructs of lambda calculus are ***lambda abstraction*** and ***application***. Lambda abstraction is used to write functions: If ***M*** is some expression, then $\lambda x.M$ is the function that is a result of treating ***M*** as a function of the variable ***x***. For example, $\lambda x.x$ is a lambda abstraction that defines the identity function – the

function whose value at ***x*** is ***x***. A more familiar way to write this function would be $I(x) = x$.

This approach forces making up a name for every function while lambda calculus allows writing anonymous functions and using them inside larger expressions. In lambda notation, function application is written just by putting a function expression in front of one or more arguments; parentheses are optional. For example, identity function can be applied to the expression ***M*** by writing $(\lambda x.x)M$.

Another example of a lambda expression is $\lambda f.\lambda g.\lambda x.f(g\ x)$. Given functions ***f*** and ***g***, this function produces the composition $\lambda x.f(g\ x)$ of ***f*** and ***g***.

Pure lambda calculus can be extended by adding a variety of other constructs. Extension with extra operations is called an ***applied lambda calculus***. Relation between lambda calculus and computer science is illustrated with the following equation:

$$\begin{aligned}&\text{programming language} = \text{applied } \lambda\text{ - calculus}\\ &= \text{pure } \lambda\text{ - calculus} + \text{additional data types}\end{aligned}$$

This works even for programming languages with side effects.

If we assume there is an infinite set ***V*** of variables and ***x, y, z,...*** stand for arbitrary variables, the grammar for lambda expressions is

$$M = x \mid MM \mid \lambda x.M$$

where ***x*** may be any variable that is an element of ***V***. An expression of the form M_1M_2 is called an ***application*** and an expression of the form $\lambda x.M$ is called a

lambda abstraction.

$\lambda x.(f(gx))$ lambda abstraction
$(\lambda x.x)5$ lambda application

There are plenty of conventions in lambda calculus but only one will be used in this context – in an expression containing a λ, the scope of λ extends as far to the right as possible. This means that $\lambda x.xy$ should be read as $\lambda x.(xy)$, not $(\lambda x.x)y$.

An occurrence of a variable in an expression is either free or bound. If a variable is ***free***, it means that the variable is not declared in the expression. For example, the variable ***x*** is free in the expression ***x + 3***. This expression cannot be evaluated without putting it inside some larger expression that will associate a value with ***x***. If a variable is not free, it is ***bound***.

The symbol λ is called a ***binding operator*** because it binds a variable within a specific part of an expression. The variable ***x*** is bound in $\lambda x.M$. Here, ***x*** is just a place holder like ***x*** in integral $\int f(x)\,dx$ and the meaning of $\lambda x.M$ does not depend on ***x***. $\lambda x.x$ defines the same function as $\lambda y.y$.

In $\lambda x.M$, the expression ***M*** is called the ***scope*** of the binding λx. A variable ***x*** appearing in an expression ***M*** is bound if it appears in the scope of some λx and is free otherwise. In the expression $\lambda x.(\lambda y.xy)y$, the first occurrence of ***x*** is called a ***binding occurrence***, as this is where ***x*** becomes bound. The other occurrence of ***x*** is a bound occurrence. Reading from left to right, the first occurrence of ***y*** is a binding occurrence, the second is a bound occurrence, and the third is free as it is outside the scope of the λy in parentheses.

A useful convention is to rename bound variables so that all bound variables are different from each other and different from all of the free variables. As a result, $(\lambda y.(\lambda z.z)y)x$ will be written instead of $(\lambda x.(\lambda x.x)x)x$.

Expressions that differ only in names of bound variables are called α equivalent. To emphasize that two expressions are α equivalent, the equation is written as $\lambda x.x =_{\alpha} \lambda y.y$. The equational proof system of lambda calculus has the axiom

$$\lambda x.M = \lambda y.[y/x]M$$

where $[y/x]M$ is the result of substituting ***y*** for free occurrences of ***x*** in ***M***. The central equational axiom of lambda calculus is used to calculate the value of a function application $(\lambda x.M)N$ by substitution. Because $\lambda x.M$ is the resulting function of treating ***M*** as a function of ***x***, the value of this function can be written at ***N*** by substituting ***N*** for ***x***. Writing $[N/x]M$ for the result of substituting ***N*** for free occurrences of ***x*** in ***M***, results in $(\lambda x.M)N = [N/y]M$, which is called the axiom of β equivalence.

When there is a function ***f*** with two arguments, it can be represented by a function $\lambda x.(\lambda y.M)$ with a single argument. When this function is applied, it returns a second function that accepts a second argument and then computes the result in the same way as ***f***. For example, the function $f(g, x) = g(x)$ has two arguments but can be represented in lambda calculus by ordinary lambda abstraction. Next, f_{curry} is defined by $f_{curry} = \lambda g.\lambda x.gx$. The difference between ***f*** and f_{curry} is that ***f*** takes a pair ***(g, x)*** as an argument while the second function takes a single argument ***g***. Both functions do the same thing.

If we write $\blacksquare$ instead of = in β equations, we obtain the basic computation step called β reduction:

$$(\lambda x.M)N \to_{\beta} [N/x]M$$

In this case, we say that ***M*** β reduces to ***N***.

Intuitively, $M \to N$ means that the expression ***M*** can be evaluated to the expression ***N*** in one computation step. Generally, the process can be repeated but for many expressions, the process eventually reaches a stopping point. A stopping point, or expression that cannot be further evaluated, is called a ***normal form***.

Another important property of lambda calculus is called ***confluence***. As a result of confluence, evaluation order does not affect the final value of an expression. So, if an expression ***M*** can be reduced to a normal form, there is exactly one normal form of ***M***, independent of the order in which we choose to evaluate subexpressions.[53]

4. Concepts

4.1. Immutable variables

In object-oriented and functional programming, an immutable variable (object) is a variable whose state cannot be modified once it is created. In contrast, a mutable variable can be modified after it is created. In some cases, a variable is considered immutable even if some internally used attributes change but the variable’s state appears to be unchanging externally.

Immutable objects are often useful because they are inherently thread-safe.

Multiple threads can act on data represented by immutable objects without concern of the data being changed by other threads. Other benefits are that they are simpler to understand and offer higher security than mutable objects.[54]

The logical question about immutable variables would be: how can an application run if variables never change? To answer this question, there are couple of rules to be considered first:

- local variables do not change
- global variables can change only references[55]

In imperative programming, values held in variables whose content never changes are called ***constants***. In most object-oriented languages, objects can be referred to using references. In this case, it is important to know whether the state of an object can vary when they are shared via references. If an object is known to be immutable, it can be copied merely by making a copy of a reference to it instead of copying the entire object. Since a reference is usually much smaller than the object itself, this result in memory savings.

Sometimes, certain fields of an object can be immutable. This means that there is no way to change those parts of the object state, even though other parts of the object may be changeable. If all fields are immutable, the object is immutable. If the whole object cannot be extended by another class, the object is called ***strongly immutable***. In some languages, signalizing that a field is immutable is done with a special keyword (i.e. ***final*** in Java). On the other hand, languages like OCaml have objects that are immutable by default so mutable objects need to be explicitly marked.

In Java, a typical example of immutable object is an instance of ***String*** class.

```
String str = “A Simple String.”;
str.toLowerCase();
```

The method ***toLowerCase*** will not change the data that ***str*** contains. Instead, it will create a new object that is given the value “a simple string.”. A reference to the new string is returned, so in order to assign the new value to ***str***, the following line is necessary:

```
str = str.toLowerCase();
```

The string ***str*** now references to the object that contains “a simple string.”.

When implementing immutable primitive types and object references, the keyword ***final*** is used.[56] This prevents primitive type variables to be reassigned after being defined.

```
int i = 12; //int is a primitive type
i = 30; //this is ok

final int j = 12;
```

```
j = 30; //the code doesn't compile; j is final and cannot be reassigned
```

Reference types cannot be made immutable just by using ***final***. It only prevents reassignment.

```
final MyClass c = new MyClass(); //c is of reference type

m.field = 100; //this is ok; changing the state is possible

m = new MyClass(); //does not compile; m cannot be reassigned
```

Primitive wrapper classes Integer, Long, Short, Double, Float, Character, Byte, Boolean are also immutable.[57]

In Scala, any entity can be defined as mutable or immutable. In the declaration, ***val*** is used for immutable entities and ***var*** is used for mutable entities. Even though an immutable binding cannot be reassigned, it may refer to a mutable object so it is still possible to call mutating methods on that object. To sum it up – the binding is immutable but the underlying object may be mutable. This is similar to the previous example regarding final references in Java.

```
val immutableValue = 100;

var mutableValue = 40;
```

In pure functional programming languages, it is not possible to create mutable objects. All objects are immutable.[58]

4.2. First-class function

First-class functions are functions treated as objects themselves, meaning they can be passed as a parameter to another function, returned as a result of a function, or they can be stored in a variable. It is said that a programming language has first-class functions if it treats functions as first-class citizens. Functions are treated like ordinary variables with a function type.

A higher-order function is a function that does at least one of the following:

- takes one or more functions as an input
- outputs a function

All other functions are first-order functions. In previously described lambda calculus, all functions are higher-order.

A typical example of a higher-order function is the map function. It takes a function ***f*** and a list of elements as arguments and returns a new list with ***f*** applied to each element from the list as a result. Another common example is sorting functions which take a comparison function as a parameter so the programmer can separate sorting algorithm from the comparisons of the items being sorted.

Languages where functions are not first-class often allow writing higher-order functions through the use of features such as function pointers or delegates. Below are examples of both cases – the first one is in JavaScript (functions are first-class) and the second one is in C# (delegates are used). The called function

applies the function passed as an argument on the second argument twice.

```
//first example
function twice(f, x){
return f(f(x));
}

function f(x){
return x*3;
}

twice(f, 7); // 63

//second example
delagate int Del1(int x);
delagate int Del2(Del1 f, int x);

Del1 f1 = delegate(int x) {return x * 3;};
```

```
Del2 f2 = delegate(Del1 f, int x) {return f(f(x));};


f2(f1, 7);
```
[59]

Anonymous functions are split into two types: lambda functions and closures. Functions are made of four parts: name, parameter list, body, and return. The idea behind anonymous functions is being able to create functions that do not have a name, they have a limited scope and need to exist only for a short time. They are often passed as arguments to higher-order functions and used to construct the result of a higher-order function that needs to return a function. Not all languages support passing a function as an argument without binding it to a name.

Lambda functions are unnamed functions that contain a parameter list, a body, and a return. In many languages they can be defined with lambda expressions. Here are examples of how those expressions are built in different languages.[60]

Java – The basic form is ***parameter(s) -> body of the expression***.

```
//expression without arguments

() -> System.out.println(“Hello World”);


//expression with one argument

//parentheses are not required
```

```
ActionListener al = event -> System.out.prinln(“button clicked”);


//body of expression as a full block of code

//usual rules for methods are applied

() -> {

System.out.println(“Hello”);

System.out.println(“ World”);

};


//expression with multiple arguments

int z = (x, y) -> x + y;


//multiple arguments that with explicit type

//when not used, types are concluded by the compiler

int z = (int x, int y) -> x + y;
```
[61]

C# - In C#, lambda expressions have similar form, but instead of ->, the operator is =>. C# also has delegates, a special type of variable used to reference methods with particular parameter list and return type.

```
delegate bool greater = (a, b) => ((int)a) > ((int)b);


//generic delegate with two objects as parameters

//delegate returns a bool

Func<object, object, bool> compare = (a, b) => ((int)a) > ((int)b);


//C# has a variable type var that represents an unknown type

//var cannot be used with lambda expressions

//both examples are wrong

var add = (x, y) => x + y;

var add = (int x, int y) => x + y;
```
[62]

F# - The form of the lambda expression in F# is ***fun parameter-list -> expression***. Parameter list also has names and optionally types, like in previous examples.

```
//single parameter

fun x -> x + 1
```

```
//multiple parameters without types

fun a b c -> printfn "%A %A %A" a b c


//multiple parameters with types

fun (a: int) (b: int) (c: int) -> a + b * c
```

Lambda functions are a form of nested function meaning they allow access to variables in the scope of the containing function (non-local variables). This means closures are necessary to implement them.[63]

Closures are a lot like lambdas, except they reference variables outside the scope of the function. The body references a variable that doesn't exist in either the body or the parameter list. They are typically implemented with a special data structure that contains a pointer to the function code and a representation of the function's lexical environment at the time when the closure was created. The referencing environment binds the non-local names to the corresponding variables scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is entered later, maybe from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

It is harder to implement closures in languages that allocate all local variables on a linear stack during run-time. In this case, function's local variables are deallocated when the function returns but a closure requires that the free variables it references survive the function's execution. Therefore, those variables must be allocated so that they persist until they are no longer needed.

Languages that support closures natively also use garbage collection.[64]

Closures can be used to associate a function with “private” variables, which persist over several function invocations. The scope of those variables is defined with a closed-over function. The first of the next two examples illustrates closed-over function written in C#. The second example illustrates regular closure written in Scala.

C#

```
public static Func<int,int> getFunction()
{
var factor = 1;
Func<int, int> increase = delegate(int x)
{
factor = factor + 1;
return x + factor;
};
return increase;
}

//reference to the created function can be assigned by calling
```

```
//the method GetFunction()
Func<int, int> increase = getFunction();
int x = increase(3); //factor becomes 2 so x becomes 5
int y = increase(5); //factor becomes 3 so y becomes 8
```

Scala

```
var factor = 3
val multiply = (i:Int) => i * factor

//using the function doesn't require creating an object
//the function is simply called
println(multiply(2));
```

4.3. Pure functions and side effects

A function or expression is said to have a side effect if it modifies some state in addition to returning a value. For example, a function may modify a global variable or a static variable, modify one of its arguments, write/read data or call other functions with side effects.

Absence of side effects is a necessary, but not sufficient, condition for referential

transparency. Referential transparency means that an expression can be replaced by its value. This requires that the expression has no side effects and is pure. An expression can be a function call. Pure functions are functions that have no side effects and always perform the same computation, resulting in the same output, given a set of inputs.[65]

Functions that perform large amounts of work are difficult to test. As code grows, functions become larger and there are more parameters. It is best to break up the function into smaller functions. These smaller functions can be pure which makes it easier to understand the code's functionality. When a function is pure, it is said that "output depends on the input".

When a set of parameters is passed into a pure function, the result is always the same. The return depends solely on the parameter list. Even if closures are passed, they bring external variables into the scope of the function, so everything the receiving function needs to operate has been passed to it locally.[66]

4.4. Currying

Currying is a transformation technique that starts with a function with multiple parameters and converts it into a sequence of functions that each accept only one parameter at a time and return the next function in the sequence. At the end of this chain of functions, all parameters are available at once, allowing the original algorithm to do its work.

This is best illustrated on an example (code is written in C#).

```
Func<int, int, int> add =
```

```
delegate(int x, int y) {
return x + y;
};
```

The function ***add*** takes two parameters and returns the result of adding the two values. When the function is called, the caller must supply both arguments ***x*** and ***y*** at once. Applying currying to this function means creating a function that accepts only one parameter, returns another function that takes second parameter, and then returns the result of the addition.

The new function is written as:

```
Func<int, Func<int, int>> curriedAdd =
delegate(int x) {
return delegate(int y) {
return x + y;
}
};
```

The type of the function has changed from ***Func<int, int, int>*** to ***Func<int, Func<int, int>>***. This reflects the new structure of the function but is also required since the code is in C#. The C# compiler refuses to use type inference for variables that store anonymous methods (whether using delegates or

lambdas). However, the explicit type is only required with the declaration of a new function.

The same can be written using lambda expressions.

```
Func<int, int, int> add = (x, y) => x + y;


Func<int, Func<int, int>> curriedAdd = x => y => x + y;
```

The process of currying follows hard rules so it is possible to automate it with a helper function that performs the process automatically. That function should take another function as a parameter and return a curried format function.

```
public static Func<T1, Func<T2, TR>> Curry<T1, T2, TR> (

this Func<T1, T2, TR> func) {

return par1 => par2 => func(par1, par2);

}
```

The input parameter is of type ***Func<T1, T2, TR>*** and the return type of the ***Curry*** function is ***Func<T1, Func<T2, TR>>***. The shape of the returned function follows the schema of the lambda expressions.

Here is how curried functions are called:

```
int result = curriedAdd(5)(3);
```

The call of the first function just takes one parameter (5), and it returns a second function which is then called with the second parameter (3). This call can also be broken down into those two steps, but it requires changing the variable type.

```
var curriedAdd5 = curriedAdd(5);
var result = curriedAdd(3);
```[67]

Here is an example in F#, a language that is functional unlike C# - object-oriented and functional hybrid – and doesn't require changing parameter types.

```
let x = 5
let y = 3



//regular function with two parameters
let add x y = x + y



//calling the function
```

```
let result  = add x y



//curried function

let add x  = //fist function

let subFunction y = //second function

x + y

subFunction //returning function



//calling the function

let fX = add x

let result  = fX y
```
[68]

4.5. Recursion

In functional programming languages, recursion is a tool often used. Many of the original functional languages didn’t have any loop constructs so recursion was used for all cases where looping was typically used in imperative languages. Some functional languages have extended syntax and allow imperative looping but recursion is still the preferred approach.[69]

The typical example of recursive function is factorial. This is how it is written in F#:

```
> let rec factorial n = if n <= 1 then 1 else n * factorial (n - 1);; val factorial
: n:int -> int
```

As it can be seem from the code, recursive function can call itself as part of its own definition. Execution of the function ***factorial 5*** can be illustrated as follows:

factorial 5

= 5 * factorial 4

= 5 * (4 * factorial 3)

= 5 * (4 * (3 * factorial 2))

= 5 * (4 * (3 * (2 * factorial 1)))

= 5 * (4 * (3 * (2 * 1)))

= 5 * (4 * (3 * 2))

= 5 * (4 * 6)

= 5 * 24

= 120

The execution of the currently executing instance of the function is suspended while a recursive call is made.

A typical error with recursion is to forget to decrement a variable at the recursive call. The function should always tend forward termination, that is, the arguments should approach the base case. That is called ***well-founded recursion***.

It is possible to define multiple recursive functions simultaneously by separating the definitions (in the following example, with ***and***). These are called ***mutually recursive functions***.

```
let rec even n = (n = 0u) || odd(n – 1u)

and odd n = (n <> 0u) && even(n – 1u)
```

Nonrecursive version of the same code would be:

```
let even n = (n % 2u) = 0u

let odd n = (n % 2u) = 1u
```

Programmers should make sure that functions are ***tail recursive***, or else the computation stack may be exhausted by large inputs. This is particularly important for functions that operate over large data structures with very large numbers of recursive calls.[70]

When a function is called, the computer must remember the place it was called

from, or the return address, so that it can return to that location with the result once the call is complete. Typically, this information is saved on the call stack, a list of return locations in order of the times that the call locations were reached. In case of tail calls, there is no need to remember the place the function is called from. Instead, tail call elimination can be performed by leaving the stack alone (except for function arguments and local variables) and the newly called function will return its result directly to the original caller.[71]

Tail recursion can only be applied to recursive functions whose final call leaves no work left to be done. If after calling itself the function has to do anything other than return, it can't be optimized. To make a function recursive, the code needs to be defined in a way that does not cause any work to be done after the recursive call is made. That is most easily done by introducing a new argument to hold whatever was supposed to be held on the stack. A function argument used only to pass itself data while performing recursion is called an accumulator.

Here is an example shown on a function that raises its argument to the power of n.

```
//without tail recursion

let rec pow x n = if n <> 0 then x * pow(x)(n – 1) else 1.0



//with tail recursion

//r acts as accumulator

let rec tailpow x n r = if n <> 0 then tailpow(r * x) (n – 1) r
```

```
else r
```

The argument ***r*** holds the result of the previous computation. Instead of performing the computation as the recursive calls return, they are performed at each step and the result is passed into ***r***. Finally, when ***n*** is zero, the result is returned.[72]

4.6. Non-strict evaluation

Evaluations are the execution of a statement, usually the execution and setting of a variable. Using strict evaluation means that the statement is immediately evaluated and assigned to the variables as soon as the variable is defined. As far as function calls go, strict evaluation means that parameters are evaluated before they are passed to functions. Conversely, non-strict evaluations do not assign the variable where it is defined. The variable isn't actually assigned until the first time it is used. This is useful when there are variables that may not be used in a specific situation. This is also known as a lazy variable and non-strict evaluation is also known as lazy evaluation.

For example, let's examine a function defined as $f(x) = a(x)/b(x)$. ***b(x)*** is evaluated first because if it equals 0, there is no point in evaluation ***a(x)*** and the entire equation fails. The lazy value is ***a(x)***.

With non-strict evaluation, immutability is maintained. The variable is assigned or evaluated only on the first reference. This means that before the variable is used, it doesn't exist. As soon as it's referenced, the variable becomes defined. [73]

Sometimes laziness can cause problems. For example, if there is a variable that a

large number of threads rely on, and it is lazy, all the threads will block until the variable has been computed. However, using lazy variables can save processing time if there is no need to evaluate a variable. Of course, it is important to know when laziness can be used.[74]

Generally speaking, strict evaluation is important when there are frequently accessed members of an object, especially if they are in a multithreaded environment. On the other hand, if there are variables that are referenced infrequently or are extremely expensive to compute, it's more useful to evaluate them only if absolutely necessary.[75]

4.7. Pattern matching

Pattern matching is usually associated with regular expressions. However, in the context of functional programming, it refers to matching objects against other objects. Using pattern matching, it is possible to extract from objects, match on members of objects, and verify that objects are of specific types – all within a statement. Pattern matching allows for fewer lines of variable assignment and more lines of understandable code. Matching on members of an objects allows writing more concise logic for when a specific segment of code should be executed.[76]

In F#, pattern matches are introduced using the ***match ... with ...*** construct and here is an example:

```
let isLikelySecretAgent url agent =

match (url, agent) with

| "http://www.control.org", 99 -> true
```

```
| “http://www.control.org”, 86 -> true

| “http://www.kaos.org”, _ -> true

| _ -> false
```

Each rule of the match is introduced with a | followed by a pattern, then ->, and a result expression. When executed, the patterns of the rules are used one by one, and the first successful pattern match determines which result expression is used.

In the example, the first rule matches if ***url*** and ***agent*** are “http://www.control.org” and 99, respectively. Likewise, the second rule matches “http://www.control.org” and 86. The third rule matches if ***url*** is “http://www.kaos.org”, regardless of agent number. The last two rules both use “wildcard” patterns represented by the underscore character; these match all inputs.

The overall conditions under which ***isLikelySecretAgent*** returns ***true*** can be determined by reading through the pattern match: agents 86 and 99 are known agents of “http://www.control.org”, all agent numbers at “http://www.kaos.org” are assumed to be agents, and no other inputs are categorized as agents.

Patterns are a powerful technique for simultaneous data analysis and decomposition. Another case of using patterns is for decomposing list values from the head downward:

```
let printFirst primes =

match primes with
```

```
| h::t -> printfn “The first prime in the list is %d” h

| [] -> printfn “No primes found in the list”
```

The first pattern-matching rule matches the input ***primes*** against the pattern ***h::t***. If ***primes*** is a nonempty list, then the match is successful, and the first ***printfn*** is executed with ***h*** bound to the head of the list and ***t*** to its tail. The second line considers the case in which the list is empty.

Pattern matching can be used to decompose structured values. The following example matches nested tuple values:

```
let highLow a b =

match (a, b) with

| (“lo”, lo), (“hi”, hi) -> (lo, hi)

| (“hi”, hi), (“lo”, lo) -> (lo, hi)

| _ -> failwith “expected both a high and low value”
```

The first rule matches if the first parts of the input pairs are the strings “lo” and “hi”, respectively. It then returns a pair made from the respective second parts of the pairs. The second rule is the mirror of this in case the values appeared in reverse order.

Individual rules of a match can be guarded by a condition that is tested if the pattern itself succeeds. Here is a simple example that computes the sign of an

integer:

```
let sign x =
match x with
| _ when x < 0 -> -1
| _ when x > 0 -> 1
| _ -> 0
```

Two patterns can be combined to represent two possible paths for matching:

```
let getValue a =
match a with
| (("lo" | "low"), v) -> v
| ("hi", v) | ("high", v) -> v
| _ -> failwith "expected both a high and low value"
```

Here, the pattern ***("lo" | "low")*** matches either string, the pattern ***("hi", v) | ("high", v)*** plays essentially the same role by matching pairs values where the left of the pair is ***"hi"*** or ***"high"*** and by binding the value ***v*** on either side of the pattern.

Forming patterns can further be done usually by building them up from other

patterns.[77]

5. Conclusion – transforming code from imperative to functional

When starting transition from imperative to functional code, it is best to break the process down into steps. Here are the transitional steps:

1. Introduce higher-order functions.
2. Convert existing methods into pure functions.
3. Convert loops over to recursive/tail-recursive methods (if possible).
4. Convert mutable variables into immutable variables.
5. Use pattern matching (if possible).

Finishing these steps can be done by moving to a more functional language like Scala or Groovy, instead of Java, for example. It would be even better to move to a fully functional language, such as Clojure. No matter which language supporting functional programming is used, the concepts and approach to programming in functional paradigm remains the same.[78]

Bibliography

1. MITCHELL, J. C. (2002) ***Concepts in Programming Languages***. Cambridge University Press

2. SEBESTA, R. W. (2012) ***Concepts of Programming Languages***. 10th Edition. Addison-Wesley

3. LOUDEN, K. C., LAMBERT, K. A. (2011) ***Programming Languages Principles and Practice.*** 3rd Edition. Cengage Learning

4. BACKFIELD, J. (2014) ***Becoming Functional.*** O'Reilly Media

5. STURM, O. (2011) ***Functional Programming in C#: Classic Programming Techniques for Modern Projects***. Wrox

6. NEWARD, T., ERICKSON, A. C., CROWELL, T. MINERICH, R. (2011) ***Professional F# 2.0.*** Wrox

7. SYME, D., GRANICZ, A., CISTERNINO, A. (2012) ***Expert F# 3.0***. 3rd Edition. Apress

8. WARBURTON, R. (2014) ***Java 8 Lambdas.*** O'Reilly Media

9. WIKIPEDIA (2015) ***Functional programming*** [Online] Available at: http://en.wikipedia.org/wiki/Functional_programming [Accessed: 31st May 2015]

10. WIKIPEDIA (2015) ***Immutable object*** [Online] Available at: http://en.wikipedia.org/wiki/Immutable_object [Accessed: 31st May 2015]

11. WIKIPEDIA (2015) ***Function (mathematics)*** [Online] Available at: http://en.wikipedia.org/wiki/Function_%28mathematics%29 [Accessed: 31st May 2015]

12. WIKIPEDIA (2015) ***Higher-order function*** [Online] Available at: http://en.wikipedia.org/wiki/Higher-order_function [Accessed: 31st

May 2015]

13. WIKIPEDIA (2015) ***Side effect (computer science)*** [Online] Available at: http://en.wikipedia.org/wiki/Side_effect_%28computer_science%29 [Accessed: 31st May 2015]

14. WIKIPEDIA (2015) ***Tail call*** [Online] Available at: http://en.wikipedia.org/wiki/Tail_call [Accessed: 31st May 2015]

15. WIKIPEDIA (2015) ***Closure (computer programming)*** [Online] Available at: http://en.wikipedia.org/wiki/Closure_%28computer_programming%29 [Accessed: 31st May 2015]

16. MSDN (2015) ***Lambda Expressions: The fun Keyword (F#)*** [Online] Available at: https://msdn.microsoft.com/en-us/library/dd233201.aspx [Accessed: 31st May 2015]

17. F# FOR FUN AND PROFIT (2012) ***Currying*** [Online] Available at: http://fsharpforfunandprofit.com/posts/currying/ [Accessed: 31st May 2015]

Event-driven programming

E vent-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. This is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input.

1. Overview

Because the program execution is determined by events, an event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure. In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected. In embedded systems the same may be achieved using hardware interrupts instead of a constantly running main loop.

Event-driven programs can be written in any programming language, although the task is easier in languages that provide high-level abstractions, such as closures. Some languages (Visual Basic, for example) are specifically designed to facilitate event-driven programming, and provide an integrated development environment (IDE) that partially automates the production of code, and provides a comprehensive selection of built-in objects and controls, each of which can respond to a range of events. Virtually all object-oriented and visual languages support event-driven programming. Visual Basic, Visual C++ and Java are examples of such languages.

A visual programming IDE such as VB.Net provides much of the code for detecting events automatically when a new application is created. The programmer can therefore concentrate on issues such as interface design, which involves adding controls such as command buttons, text boxes, and labels to standard forms (a form represents an application's workspace or window). Once the user interface is substantially complete, the programmer can add event-handling code to each control as required. Many visual programming environments will even provide code templates for event handlers, so the

programmer only needs to provide the code that defines the action the program should take when the event occurs. Each event handler is usually bound to a specific object or control on a form. Any additional subroutines, methods, or function procedures required are usually placed in a separate code module, and can be called from other parts of the program as and when needed.[79]

2. Background

Before the arrival of object-oriented programming languages, event handlers would have been implemented as subroutines within a procedural program. The flow of program execution was determined by the programmer, and controlled from within the application's main routine. The complexity of the logic involved required the implementation of a highly structured program. All of the program's code would be written by the programmer, including the code required to ensure that events and exceptions were handled, as well as the code required to manage the flow of program execution.

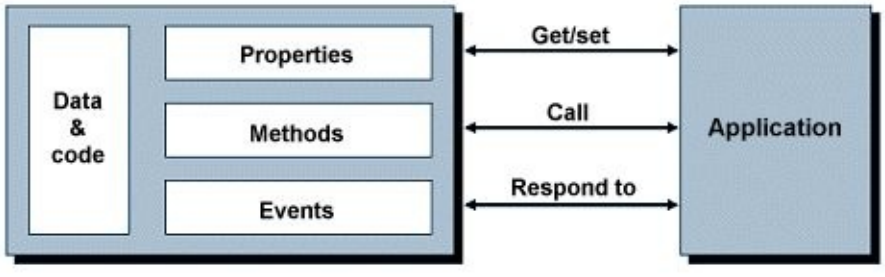
In a typical modern event-driven program, there is no discernible flow of control. The main routine is an event-loop that waits for an event to occur, and then invokes the appropriate event-handling routine. Since the code for this event loop is usually provided by the event-driven development environment or framework, and largely invisible to the programmer, the programmer's perception of the application is that of a collection of event handling routines. Programmers used to working with procedural programming languages sometimes find that the transition to an event-driven environment requires a considerable mental adjustment.

The change in emphasis from procedural to event-driven programming has been accelerated by the introduction of the Graphical User Interface (GUI) which has

been widely adopted for use in operating systems and end-user applications. It really began, however, with the introduction of object-oriented (OO) programming languages and development methodologies in the late 1970s.

By the 1990's, object-oriented technologies had largely supplanted the procedural programming languages and structured development methods that were popular during the 70s and 80s. One of the drivers behind the object oriented approach to programming that emerged during this era was the speed with which database technology developed and was adopted for commercial use. Information system designers increasingly saw the database itself, rather than the software that was used to access it, as the central component of a computerised information system. The software simply provided a user interface to the database, and a set of event handling procedures to deal with database queries and updates.

One of the fundamental ideas behind object-oriented programming is that of representing programmable entities as objects. An entity in this context could be literally anything with which the application is concerned. A program dealing with tracking the progress of customer orders for a manufacturing company, for example, might involve objects such as "customer", "order", and "order item".



An object encompasses both the data (attributes) that can be stored about an entity, the actions (methods) that can be used to access or modify the entity’s attributes, and the events that can cause the entity's methods to be invoked. The basic structure

of an object, and its relationship to the application to which it belongs, is illustrated in the diagram below.

Figure 1 – The relationship between an object and an application

The link between object-oriented programming and event-driven programming is fairly obvious. For example, objects on a Java form (usually referred to as controls) can be categorized into classes (e.g. "Button", "TextBox" etc.), and many instances of each can appear on a single form. Each class will have attributes (usually referred to as properties) that will be common to all objects of that type (e.g. "BackgroundColour", "Width", “Name” etc.), and each class will define a list of events to which an object of that type will respond. The methods (event handlers) to handle specific events are usually provided as templates to which the programmer simply has to add the code that carries out the required action.[80]

3. How It Works

Events are often actions performed by the user during the execution of a program, but can also be messages generated by the operating system or another application, or an interrupt generated by a peripheral device or system hardware. If the user clicks on a button with the mouse or hits the Enter key, it generates an event. If a file download completes, it generates an event. And if there is a hardware or software error, it generates an event. The events are dealt with by a central event handler (usually called a dispatcher or scheduler) that runs continuously in the background and waits for an even to occur. When an event does occur, the scheduler must determine the type of event and call the appropriate event handler to deal with it. The information passed to the event handler by the scheduler will vary, but will include sufficient information to

allow the event handler to take any action necessary.

Event handlers can be seen as small blocks of procedural code that deal with a very specific occurrence. They usually produce a visual response to inform or direct the user, and often change the system's state. The state of the system encompasses both the data used by the system (e.g. the value stored in a database field), and the state of the user interface itself (for example, which on-screen object currently has the focus, or the background colour of a text box). An event handler may even trigger another event to occur that will cause a second event handler to be called (note that care should be taken when writing event handlers that invoke other event handlers in order to avoid the possibility of putting the application into an infinite loop). Similarly, an event handler may cause any queued events to be discarded (for example, when the user clicks on the Quit button to terminate the program). The diagram below illustrates the relationship between events, the scheduler, and the application's event handlers.

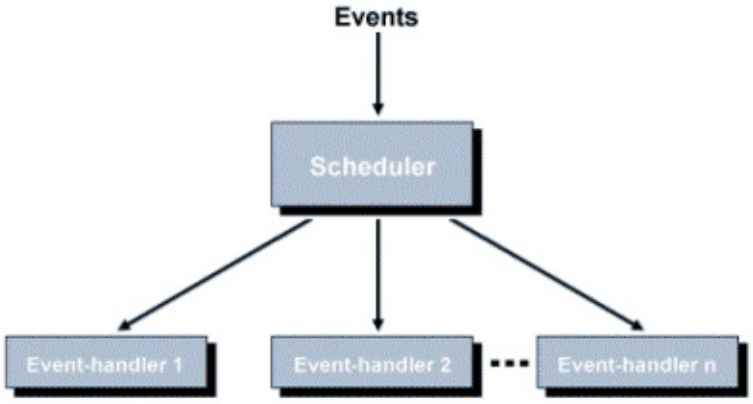


Figure 2 – Relationship between events, scheduler and event handlers

As it can be seen in the figure above, the central element of an event-driven application is a scheduler that receives a stream of events and passes each event to the relevant event handler. The scheduler will continue to remain active until it encounters an event (e.g. "End_Program") that causes it to terminate the application. Under certain circumstances, the scheduler may encounter an event for which it cannot assign

an appropriate event handler. Depending on the nature of the event, the scheduler can either ignore it or raise an ***exception*** (this is sometimes referred to as "throwing" an exception).

The pseudo-code routine below shows how a (very simple) scheduler might work. It consists of a main loop that runs continuously until a terminating condition occurs. When an event occurs, the scheduler must determine the event type, and select an appropriate event handler (or deal with the event, if no suitable event handler exists).

```
do forever: // the main scheduler loop


get event from input stream


if event type == EndProgram:

quit // break out of event loop


else if event type == event_01:

call event handler for event_01 with event parameters


else if event type == event_02:

call event handler for event_02 with event parameters
```

.

.

.

else if event type == event_nn:

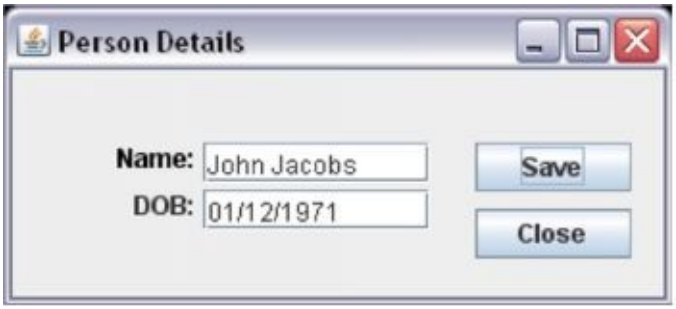
call event handler for event_nn with event parameters

else handle unrecognized event // ignore or raise exception

end loop[81]

Within an event-driven programming environment, standard events are usually identified using the ID of the object affected by the event (e.g. the name of a command button on a form), and the event ID (e.g. "left-click"). The information passed to the event handler may include additional information, such as the ***x*** and ***y*** coordinates of the mouse pointer at the time the event occurred, or the state of the ***Shift*** key (if the event in question is a key-press).

Figure 3 – A program with GUI



The example in the figure displays six graphical elements in a window. It uses two labels (“Name:” and “DOB:”), two text fields where the user will enter the Name and DOB values, and two buttons (“Save” and “Close”). A graphical user interface, compared to a command line user interface, makes the user’s interaction with a program easier. If an event handler is assigned to the button “Save”, it responds to pressing that button. It can be, for example, defined so that when the button is clicked, the user name and date of birth are saved in a database. Data is not collected until the appropriate button is clicked.

4. GUI and Event Handling

4.1. Writing Event Handlers

It should not be assumed that because most popular modern software applications have graphical user interface (GUI), event-driven programming is the right solution for every programming requirement. Some software systems have a very specific role that involves them carrying out some task to completion with little or no user intervention (a C compiler, for example). Such applications are probably better served by a procedural programming paradigm.

Having said that, most mainstream commercial software relies heavily on the availability of GUI, and most GUI software is designed to be event-driven. A visual programming language such as Visual Basic and Visual C/C++ now comes with an Integrated Development Environment (IDE) that provides an extensive array of standard controls, each with its own set of events and event handler code templates. The task of the GUI programmer is thus twofold – to create the user interface, and to write the event handler code (and any additional code modules that might be required). The IDE provides the scheduler and the event queue, and to a large extent takes care of the flow of program execution. The GUI programmer is thus free to concentrate on the application-specific

code. They will write the code required by each control or object to allow it to respond to a specific event, but do not need to know how to create the objects themselves. Java will be used here to illustrate creating GUI, event handlers etc.

The literal meaning of an event is: “An occurrence of something at a specific point in time.” The meaning of an event in a Swing application is similar. An event in Swing is an action taken by a user at a particular point in time. For example, pressing a button, pressing a key down/up on the keyboard, and moving the mouse over a component are events in a Swing application. Sometimes the occurrence of an event in Swing (or any GUI-based application) is also known as “triggering an event” or “firing an event”.

When you say that a clicked event has occurred on a button, you mean that the button has been pressed using the mouse, the spacebar, or by any other means your application allows you to press a button. Sometimes, the phrase “clicked event has been triggered” or “fired on a button” is used to say that the button has been pressed. When an important event occurs, you want to respond to the event. Taking an action in a program is nothing but executing a piece of code. Taking an action in response to the occurrence of an event is called event handling. The piece of code that is executed when an event occurs is called an event handler. Sometimes, an event handler is also called an event listener.[82]

How an event handler is written depends on the type of event and the component that generates the event. Sometimes the event handler is built into a Swing component, and sometimes you need to write the event handler yourself. For example, when you press a JButton, you need to write the event handler yourself. However, when you press a letter key on the keyboard when the focus is in a text field, the corresponding letter is typed in the text field because of the key pressed event has a default event handler that is supplied by Swing.

There are three participants in an event:

- The source of the event
- The event
- The event handler (or the event listener)

The source of an event is the component that generates the event. For example, when you press a JButton, the clicked event occurs on that JButton. In this case, the JButton is the source of the clicked event. An event represents the action that takes place on the source component. An event in Swing is represented by an object that encapsulates the details about the event such as the source of the event, when the event occurred, what kind of event occurred, etc. What is the class of the object that represents an event? It depends on the type of the event that occurs. There is a class for every type of event. For example, an object of the ActionEvent class in the java.awt.event package represents a clicked event for a JButton.

An event handler is the piece of code that is executed when an event occurs. Like an event, an event handler is also represented by an object, which encapsulates the event handling code. An object of what class represents an event handler? It depends on the type of event that the event handler is supposed to handle. An event handler is also known as an event listener because it listens for the event to occur in the source component. We will use the phrases “event handler” and “event listener”. Typically, an event listener is an object that implements a specific interface. The specific interface an event listener has to implement depends on the type of event it will listen for. For example, if you are interested in listening for a clicked event of a JButton (to rephrase, if you are

interested in handling the clicked event of a JButton), you need an object of a class that implements the ActionListener interface, which is in the java.awt.event package.

Looking at the descriptions of the three participants of an event handling, it seems you need to write a lot of code to handle an event. Not really. Event handling is easier than it seems. Here are the steps to handle an event, later followed by an example of how to handle the clicked event of a JButton. These steps can be applied to any kind of event on any Swing component:

- Identify the component for which you want to handle the event. Assume that you have named the component as sourceComponent. So your event source is sourceComponent.
- Identify the event that you want to handle for the source component. Assume that you are interested in handling Xxx event. Here Xxx is an event name that you will have to replace by an event name that exists for the source component. Recall that an event is represented by an object. The Java naming convention for event classes comes to your rescue in identifying the name of the class whose object represents Xxx event. The class whose object represents Xxx event is named XxxEvent. Usually the event classes are in the java.awt.event and javax.swing.event package.
- It is time to write an event listener for the Xxx event. Recall that an event listener is nothing but an object of a class that implements a specific interface. How do you know what specific interface you need to implement in your event listener class? Here again, the Java naming convention comes helps you out. For Xxx event, there is an XxxListener interface that you need to implement in your event

listener class. Usually the event listener interfaces are in the java.awt.event and javax.swing.event package. The XxxListener interface will have one or more methods. All methods for XxxListener take an argument of type XxxEvent because these methods are meant to handle an XxxEvent. For example, suppose you have an XxxListener interface that has a method named aMethod() as

```
public interface XxxListener {

void aMethod(XxxEvent event);

}
```

- Your event listener class will look as follows. Note that you will be creating this class.

```
public class MyXxxEventListener implements XxxListener {

public void aMethod(XxxEvent event) {

// Your event handler code goes here

}

}
```

- You are almost done. You have identified the event source, the event you are interested in, and the event listener. There is only one thing missing. You need to let the event source know that your event listener is interested in listening to its Xxx event. This is also known as registering an event listener with the event source. You register an object of your event listener class with the event source. In your case, you will create an object of the MyXxxEventListener class.

```
MyXxxEventListener myXxxListener = new MyXxxEventListener();
```

- How do you register an event listener with the event source? As we've seen before, the Java naming convention comes in handy. If a component (an event source) supports an Xxx event, it will have two methods, addXxxListener(XxxListener l) and removeXxxListener(XxxListener l). When you are interested in listening for an Xxx event of a component, you call the addXxxListener() method, passing an event listener as an argument. When you do not want to listen for Xxx event of a component anymore, you call its removeXxxListener() method. To add your myXxxListener object as the Xxx event listener for sourceComponent, you write:

```
sourceComponent.addXxxListener(myXxxListener);
```

That is all you need to do to handle an Xxx event. It may seem that you have to

perform many steps to handle an event. However, that is not the case. You can always avoid writing a new event listener class, which implements the XxxListener interface by using an anonymous inner class, which implements the XxxListener interface.

For example, you could have written the above pieces of code in two statements, like this:

```
// Create an event listener object using an anonymous inner class

XxxListener myXxxListener = new XxxListener() {
public void aMethod(XxxEvent event) {
// Your event handler code goes here
}
};

// Add the event listener to the event source component

sourceComponent.addXxxListener(myXxxListener);
```

If the listener interface is a functional interface, you can use a lambda expression to create its instance. Your XxxListener is a functional interface because it

contains only one abstract method. You can avoid creating the bulky anonymous class and rewrite the above code as follows:

```
// Add the event listener using a lambda expressions


sourceComponent.addXxxListener((XxxEvent event) -> {
// Your event handler code goes here
});
```

4.2. Handling Button Events

It is time to look at an example. Add an event listener to a JButton, and then add a JButton with text Close to a JFrame. When the JButton is pressed, the JFrame is closed and the application exits. A JButton generates an ActionEvent when it is pressed.

Once you know the name of the event, which is Action in this case, you just need to replace Xxx in the previous generic example with the word Action. You will come to know the class and method names you need to use to handle the Action event of JButton. This table compares the names of classes/interfaces/method used to handle Action event for a JButton to that of generic names we have used in the discussion.

Generic Event Xxx	Action Event for JButton	Comments
		An object of ActionEvent class in

XxxEvent	ActionEvent	java.awt.event package represents Action event for JButton
XxxListener	ActionListener	An object of a class that implements ActionListener interface represents Action event handler for a JButton
addXxxListener()	addActionListener	The addActionListener() method od a Jbutton is used to add a listener for its Action event
removeXxxListener()	removeActionListener	The removeActionListener() method of Jbutton is used to remove a listener for its Action event

The ActionListener interface is simple. It has one method called actionPerformed(). The interface declaration is as follows:

```
public interface ActionListener extends EventListener {

void actionPerformed(ActionEvent event);

}
```

All event listener interfaces inherit from the EventListener interface, which is in the java.util package. The EventListener interface is a marker interface, and it does not have any methods. It just acts as the ancestor for all event listener interfaces. When a JButton is pressed, the actionPerformed() method of all its registered Action listeners is called.

Example 1: A JFrame with a Close JButton With an Action

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JButton;

public class SimplestEventHandlingFrame extends JFrame {

JButton closeButton = new JButton("Close");
public SimplestEventHandlingFrame() {
super("Simplest Event Handling JFrame");
this.initFrame();
}

private void initFrame() {
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
// Set a FlowLayout for the content pane
this.setLayout(new FlowLayout());
```

```
// Add the Close JButton to the content pane
this.getContentPane().add(closeButton);
// Add an ActionListener to closeButton
closeButton.addActionListener(e -> System.exit(0));
}

public static void main(String[] args) {
SimplestEventHandlingFrame frame = new SimplestEventHandlingFrame();
frame.pack();
frame.setVisible(true);
}
}
```

Example 2: Let’s have one more example of adding an Action listener to JButton. This time, add two buttons to a JFrame: A Close button and another to display the number of times it is clicked. Every time the second button is clicked, its text is updated to show the number of times it has been clicked. There needs to be a variable to maintain the click count.

```
import javax.swing.JFrame;
```

```
import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import javax.swing.JButton;

import java.awt.event.ActionListener;


public class JButtonClickedCounter extends JFrame {


int counter;

JButton counterButton = new JButton("Clicked #0");

JButton closeButton = new JButton("Close");


public JButtonClickedCounter() {

super("JButton Clicked Counter");

this.initFrame();

}


private void initFrame() {

this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
// Set a FlowLayout for the content pane
this.setLayout(new FlowLayout());
// Add two JButtons to the content pane
this.getContentPane().add(counterButton);
this.getContentPane().add(closeButton);
// Add an ActionListener to the counter JButton
counterButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent event) {
// Increment the counter and set the JButton text
counter++;
counterButton.setText("Clicked #" + counter);
}
});

// Add an ActionListener to closeButton
closeButton.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent event) {
// Exit the application, when this button is pressed
```

```
System.exit(0);

}

});

}

public static void main(String[] args) {

JButtonClickedCounter frame = new JButtonClickedCounter();

frame.pack();

frame.setVisible(true);

}

}
```

Figure 4 – JFrame before and after the counter button has been clicked

4.3. Handling Mouse Events

You can handle mouse activities (clicked, entered, exited, pressed, and released) on a component. You will experiment with mouse events using a JButton. An object of the MouseEvent class represents a Mouse event on a component. Now, you can guess that to handle Mouse events, you will need to work with the MouseListener interface. Here is how the interface is declared:

```
public interface MouseListener extends EventListener {

public void mouseClicked(MouseEvent e);

public void mousePressed(MouseEvent e);

public void mouseReleased(MouseEvent e);

public void mouseEntered(MouseEvent e);

public void mouseExited(MouseEvent e);

}
```

The MouseListener interface has five methods. You cannot use a lambda expression to create mouse event handler. One of the methods of the MouseListener interface is called when a specific mouse event occurs. For example, when a mouse pointer enters a component's boundary, a mouse entered event occurs on the component, and the mouseEntered() method of the mouse listener object is called. When the mouse pointer leaves the boundary of the component, a mouse exited event occurs, and the mouseExited() method is called. The names of other methods are self-explanatory.

The MouseEvent class has many methods that provide the details about a mouse event:

- The getClickCount() – method returns the number of clicks a mouse made.
- The getX() and getY() – methods return the x and y positions of the mouse with respect to the component when the event occurs.

- The getXOnScreen() and getYOnScreen() – methods return the absolute x and y positions of the mouse at the time the event occurs.

Suppose you are interested in handling only two kinds of mouse events for a JButton: the mouse entered and mouse exited events. The text of the JButton changes to describe the event. The mouse event handler code is as follows:

```
mouseButton.addMouseListener(new MouseListener() {

@Override
public void mouseClicked(MouseEvent e) {
// Nothing to handle
}

@Override
public void mousePressed(MouseEvent e) {
// Nothing to handle
}

@Override
```

```
public void mouseReleased(MouseEvent e) {
// Nothing to handle
}

@Override
public void mouseEntered(MouseEvent e) {
mouseButton.setText("Mouse has entered!");
}

@Override
public void mouseExited(MouseEvent e) {
mouseButton.setText("Mouse has exited!");
}
});
```

In this code, an implementation for all five methods of the MouseListener interface was provided even though you were interested in handling only two kinds of mouse events. The body of three methods are left empty.

Now, we will demonstrate the mouse entered and exited event for a JButton.

When the JFrame is displayed, try moving your mouse in and out of the boundary of the JButton to change its text to indicate the appropriate mouse event.

```
import java.awt.FlowLayout;

import javax.swing.JFrame;

import javax.swing.JButton;

import java.awt.event.MouseListener;

import java.awt.event.MouseEvent;


public class HandlingMouseEvent extends JFrame {


JButton mouseButton = new JButton("No Mouse Movement Yet!");

public HandlingMouseEvent() {

super("Handling Mouse Event");

this.initFrame();

}


private void initFrame() {
```

```
this.setDefaultCloseOperation(EXIT_ON_CLOSE);

this.setLayout(new FlowLayout());

this.getContentPane().add(mouseButton);

// Add a MouseListener to the JButton

mouseButton.addMouseListener(new MouseListener() {


@Override

public void mouseClicked(MouseEvent e) {}


@Override

public void mousePressed(MouseEvent e) {}


@Override

public void mouseReleased(MouseEvent e) {}


@Override

public void mouseEntered(MouseEvent e) {

mouseButton.setText("Mouse has entered!");
```

```
}

@Override
public void mouseExited(MouseEvent e) {
mouseButton.setText("Mouse has exited!");
}
});
}

public static void main(String[] args) {
HandlingMouseEvent frame = new HandlingMouseEvent();
frame.pack();
frame.setVisible(true);
}
}
```

It is not mandatory to always provide implementation for all event-handling methods of an event listener interface. Swing designers thought of this inconvenience and devised a way to avoid this. Swing includes a convenience class for some XxxListener interfaces. The class is named XxxAdapter. They are

called adapter classes. An XxxAdapter class is declared abstract and it implements the XxxListener interface. The XxxAdapter class provides empty implementation for all methods in the XxxListener interface. The following snippet of code shows the relationship between an XxxListener interface having two methods m1() and m2() and its corresponding XxxAdapter class.

```
public interface XxxListener {

public void m1();

public void m2();

}


public abstract class XxxAdapter implements XxxListener {


@Override

public void m1() {

// No implementation provided here

}


@Override

public void m2() {
```

```
// No implementation provided here
}
}
```

Not all event listener interfaces have corresponding adapter classes. The event listener interface, which declares more than one method, has a corresponding adapter class. For example, you have an adapter class for the MouseListener interface that is called MouseAdapter. MouseAdapter can save you a few lines of unnecessary code. If you only want to handle a few of the mouse events, you can create an anonymous inner class (or regular inner class) that inherits from the adapter class and overrides the only methods that are of interest to you. The following snippet of code rewrites the event handler used in upper snippet of code using the MouseAdapter class:

```
mouseButton.addMouseListener(new MouseAdapter() {

@Override
public void mouseEntered(MouseEvent e) {
mouseButton.setText("Mouse has entered!");
}

@Override
```

```
public void mouseExited(MouseEvent e) {

mouseButton.setText("Mouse has exited!");

}

});
```

You may notice that you did not have to worry about three other methods of the MouseListener interface because the MouseAdapter class provided empty implementation for you. There is no adapter class named ActionAdapter for the ActionListener interface. Since the ActionListener interface has only one method in it, providing an adapter class will not save any keystrokes for you. Note that using an adapter class to handle an event has no special advantage, except for saving some keystrokes. However, it does have a limitation.

If you want to create an event handler by using the main class itself, you cannot use an adapter class. Typically, your main class is inherited from the JFrame class and Java does not allow you to inherit a class from multiple classes. So, you cannot inherit your main class from the JFrame class as well as the adapter class. If you are using an adapter class to create an event handler, you must use either an anonymous inner class or a regular inner class.[83]

Time-driven Programming

T ime-driven programming is a computer programming paradigm, where the control flow of the computer program is driven by a clock and is often used in Real-time computing. A program is divided into a set of tasks (i.e., processes or threads), each of which has a periodic activation pattern. The activation patterns are stored in a dispatch table ordered by time. The Least-Common-Multiple

(LCM) of all period-times determines the length of the dispatch table. The scheduler of the program dispatches tasks by consulting the next entry in the dispatch table. After processing all entries, it continues by looping back to the beginning of the table.

The programming paradigm is mostly used for safety critical programs, since the behaviour of the program is highly deterministic. No external events are allowed to affect the control-flow of the program, the same pattern (i.e., described by the dispatch table) will be repeated time after time. However, idle time of the processor is also highly deterministic, allowing for the scheduling of other non-critical tasks through slack stealing techniques during these idle periods.

The drawback with the method is that the program becomes static (in the sense that small changes may recompile into large effects on execution structure), and unsuitable for applications requiring a large amount of flexibility. For example, the execution time of a task may change if its program code is altered. As a consequence, a new dispatch table must be regenerated for the entire task set. Such a change may require expensive retesting as is often required in safety critical systems.

So, if you want to perform a specific task once at a time, you will use timers to help you scheduled execution time.
There are two types of timers:

- Class Timer – java.util.Timer;
- Swing Timer – javax.swing.Timer;

1. Class Timer

Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

After the last live reference to a Timer object goes away and all outstanding tasks have completed execution, the timer's task execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. By default, the task execution thread does not run as a daemon thread, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread rapidly, the caller should invoke the timer's cancel method.

If the timer's task execution thread terminates unexpectedly, for example, because its stop method is invoked, any further attempt to schedule a task on the timer will result in an IllegalStateException, as if the timer's cancel method had been invoked.

This class is thread-safe: multiple threads can share a single Timer object without the need for external synchronization. This class does not offer real-time guarantees: it schedules tasks using the Object.wait(long) method.

Java 5.0 introduced the java.util.concurrent package and one of the concurrency utilities therein is the ScheduledThreadPoolExecutor which is a thread pool for repeatedly executing tasks at a given rate or delay. It is effectively a more

versatile replacement for the Timer/TimerTask combination, as it allows multiple service threads, accepts various time units, and doesn't require subclassing TimerTask (just implement Runnable). Configuring ScheduledThreadPoolExecutor with one thread makes it equivalent to Timer.

Modifier and Type	Method	Description
void	cancel()	Terminates this timer, discarding any currently scheduled tasks.
int	purge()	Removes all cancelled tasks from this timer's task queue.
void	schedule(TimerTask task, Date time)	Schedules the specified task for execution at the specified time.
void	schedule(TimerTask task, Date firstTime, long period)	Schedules the specified task for repeated fixed-delay execution, beginning at the specified time.
		Schedules the specified task

void	schedule(TimerTask task, long delay)	for execution after the specified delay.
void	schedule(TimerTask task, long delay, long period)	Schedules the specified task for repeated fixed-delay execution, beginning after the specified delay.
void	scheduleAtFixedRate(TimerTask task, Date firstTime, long period)	Schedules the specified task for repeated fixed-rate execution, beginning at the specified time.
void	scheduleAtFixedRate(TimerTask task, long delay, long period)	Schedules the specified task for repeated fixed-rate execution, beginning after the specified delay.

[84]

2. Swing Timer

A Swing timer (an instance of javax.swing.Timer) fires one or more action events after a specified delay. Do not confuse Swing timers with the general-purpose timer facility in the java.util package.

In general, it is recommended using Swing timers rather than general-purpose timers for GUI-related tasks because Swing timers all share the same, pre-existing timer thread and the GUI-related task automatically executes on the event-dispatch thread. However, you might use a general-purpose timer if you don't plan on touching the GUI from the timer, or need to perform lengthy processing.

Swing timers can be used in two ways:

- To perform a task once, after a delay - For example, the tool tip manager uses Swing timers to determine when to show a tool tip and when to hide it.
- To perform a task repeatedly - For example, you might perform animation or update a component that displays progress toward a goal.

Swing timers are very easy to use. When you create the timer, you specify an action listener to be notified when the timer "goes off". The actionPerformed method in this listener should contain the code for whatever task you need it to perform. When you create the timer, you also specify the number of milliseconds between timer firings. If you want the timer to go off only once, you can invoke setRepeats(false) on the timer. To start the timer, call its start method. To suspend it, call stop.

Note that the Swing timer's task is performed in the event dispatch thread. This means that the task can safely manipulate components, but it also means that the task should execute quickly. If the task takes a while to execute, consider using a SwingWorker instead of or in addition to the timer. See Concurrency in Swing for instructions about using the SwingWorkerclass and information on using

Swing components in multi-threaded programs.

Let's look at an example of using a timer to periodically update a component. Some applet uses a timer to update its display at regular intervals. This applet begins by creating and starting a timer:

```
timer = new Timer(speed, this);
timer.setInitialDelay(pause);
timer.start();
```

The speed and pause variables represent applet parameters, these are 100 and 1900 respectively, so that the first timer event will occur in approximately 1.9 seconds, and recur every 0.1 seconds. By specifying this as the second argument to the Timer constructor, applet specifies that it is the action listener for timer events.

After starting the timer, applet begins loading a series of images in a background thread. Meanwhile, the timer events begin to occur, causing the actionPerformed method to execute:

```
public void actionPerformed(ActionEvent e) {
//If still loading, can't animate.
if (!worker.isDone()) {
return;
```

```
}

loopslot++;

if (loopslot >= nimgs) {
loopslot = 0;
off += offset;

if (off < 0) {
off = width - maxWidth;
} else if (off + maxWidth > width) {
off = 0;
}
}
animator.repaint();

if (loopslot == nimgs - 1) {
timer.restart();
```

```
    }

    }
```

Until the images are loaded, worker.isDone returns false, so timer events are effectively ignored. The first part of the event handling code simply sets values that are employed in the animation control's paintComponent method: loopslot (the index of the next graphic in the animation) and off (the horizontal offset of the next graphic).

Eventually, loopslot will reach the end of the image array and start over. When this happens, the code at the end of actionPerformed restarts the timer. Doing this causes a short delay before the animation sequence begins again.[85]

Bibliography

1. MALIK, D. S. (2012) ***Java Programming From Problem Analysis to Program Design.*** 5th Edition. Cengage Learning

2. LIANG, Y. D. (2011) ***Introduction to Java Programming***. 8th Edition. Prentice Hall

3. TECHNOLOGYUK (2015) ***Event-driven programming.*** [Online] Available at: http://www.technologyuk.net/computing/software_development/event_dr [Accessed: 31st May 2015]

4. ORACLE (2015) ***Timer*** [Online] Available at: https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html

[Accessed: 31st May 2015]

5. ORACLE (2015) ***How to Use Swing Timers*** [Online] Available at: https://docs.oracle.com/javase/tutorial/uiswing/misc/timer.html [Accessed: 31st May 2015]

Object-oriented programming

O bject-oriented programming paradigm is based on the concept of objects, which are data structures that contain data, in the form of fields, often know as attributes; and code, in the form of procedures, often known as methods. In object-oriented programming, programs are designed by making them out of objects that interact with one another. Most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

Many of the widely used programming languages are multi-paradigm programming languages that support object-oriented programming, typically in combination with imperative, procedural programming. Some of the most significant object-oriented languages include ***C++, Java, Delphi, C#, Python, Smalltalk***.

1. History

Terms “objects” and “oriented”, in the sense of object-oriented programming, made their first appearance at MIT in the late 1950s and early 1960s. In the

environment of the artificial intelligence group, “object” could refer to identified items with properties (attributes). Another early MIT example was Sketchpad, a computer program considered to be the ancestor of modern CAD programs as well as a major breakthrough in the development of computer graphics in general, created by Ivan Sutherland. Sutherland defined notions of “object” and “instance”.[86]

The formal programming concept of objects was introduced in the 1960s in Simula 67, a programming language designed for discrete event simulation.[87] It introduced the notion of classes and instances or objects as part of an explicit programming paradigm. Simula was used for physical modelling but its ideas influenced many later languages including Smalltalk and C++.

The Smalltalk language, developed in the 1970s, introduced the term “object-oriented programming” to represent the extensive use of objects and messages as the basis of computation. Unlike Simula 67, Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically.[88]

Object-oriented programming developed as the dominant methodology in the early and mid 1990s when programming languages supporting these techniques became widely available. These languages included FoxPro, C++, and Delphi. Its dominance was enhanced by the rising popularity of graphical user interfaces (GUI), which rely heavily upon object-oriented programming techniques. The popularity of event-driven programming was also enhanced although this concept is not limited to OOP.

Object-oriented features have been added to many previously existing languages, including Ada, BASIC, Fortran, Pascal, and COBOL. More recently, a number of languages have emerged that are primarily object-oriented, but that are also

compatible with procedural methodology. Two such languages are Python and Ruby. Probably the most commercially-important recent object-oriented languages are Java, C#, and Visual Basic.NET. These languages are widely used today.

2. Overview

Object-oriented programming attempts to provide a model for programming based on objects. It integrates code and data using the concept of an “object”. An object is an abstract data type with the addition of polymorphism and inheritance. An object has both state (data) and behaviour (code).

Objects sometimes correspond to things that can be found in the real world such as “customer” and “product” in an online shopping system. This system will support behaviours like “place order” and “make payment”.

Objects are designed in class hierarchies. In the previous example illustrating a shopping system, there might be high level classes such as “electronics product”, “kitchen product”, and “book”. Further refinements of “electronic products” could be “CD Player”, “DVD Player”, etc. These classes and subclasses correspond to sets and subsets in mathematics.

Object orientation essentially merges abstract data types with structured programming and divides systems into modular objects which own their own data and are responsible for their own behaviour. This feature is known as encapsulation. With encapsulation, the data for two objects are divided so that changes to one object cannot affect the other.

The object-oriented approach encourages the programmer to place data where it

is not directly accessible by the rest of the system. Instead, the data is accessed by calling specially written functions, called methods, which are bundled with the data. These act as the intermediaries for retrieving or modifying the data they control. The construct that combines data with a set of methods for accessing and managing data is called an object.

Treating software as modular components that support inheritance makes it easy both to re-use existing components and to extend components as needed by defining new subclasses with specialized behaviours. These two goals are in the object-oriented paradigm also known as the open/closed principle. A module is open if it supports extension (it can modify behaviour, add new properties …). A module is closed if it has a well-defined stable interface that all other modules must use and that limits the interaction and potential errors that can occur in one module by changes in another.

To further explain the concepts of object-oriented programming, Java language will be used. This is a language used for both industrial and academic purposes. Besides being object-oriented, Java is often described as a web programming language because of its use in writing programs called applets that run within a web browser. Java has a clean design and it includes only features that are indispensable which makes Java easier to learn than other object-oriented programming languages.

3. Concepts

3.1. Objects and reference variables

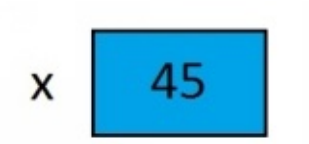
It was already mentioned that an object is a thing that can be imagined and a program written in object-oriented style consists of interacting objects. Program contains instructions to create objects. For the computer to be able to create an

object, there has to be a definition, called a class. An object is called an instance of a class, and it is an instance of exactly one class. Using the operator new to create a class object is called ***instantiating*** an object of that class. Classes will be explained in a special section.[89]
First, let's consider the following two statements:

```
int x; //line 1

String str; //line 2
```

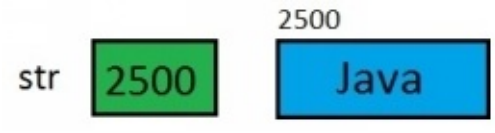
The first statement (line 1) declares ***x*** to be an int variable. It allocates memory space to store an int value and calls this memory space ***x***. The variable ***x*** can store an int value in its memory space. The following statement will store 45 in ***x*** (the result is illustrated in the figure):



x = 45;

Figure 1 – x and its data

The statement in line 2 allocates memory space for the variable ***str***. However, unlike the variable ***x***, ***str*** cannot directly store data in its memory space. The variable ***str*** stores the memory location, that is, the address of the memory space where the actual data is stored. For example, the result of the following statement is shown in the figure:



str = “Java” //line 3

Figure 2 – str and the data it points to

In this example, the statement causes the system to allocate memory space at location 2500, stores the string “Java” in this memory space, and then stores the address 2500 in the memory space of ***str***. The effect the statement in line 3 is the same as the effect of the following statement:

str = new String(“Java”);

In Java, ***new*** is an operator. It causes the system to allocate memory space of a specific type, store specific data in that memory space, and return the address of the memory space. Therefore, the statement in line 4 causes allocation of memory space large enough to store the string “Java”, stores the string in that memory space, and returns the address of the allocated memory space. After that, the assignment operator (=) stores the address of that memory space into the variable ***str***.

String is not a primitive data type. In Java terminology, the data type String is defined by the class String and variables such as ***str*** are called ***reference variables***. Reference variables are variables that store the address of a memory space. In Java, any variable declared using a class is a reference variable. Because ***str*** is a reference variable declared using the class String, it is said that ***str*** is a reference variable of the String type. The memory space 2500, where the string is stored, is called a String object. String objects are called ***instances*** of the class String.

Figure 3 – Variable str and object str

Because ***str*** is a reference variable of the String type, ***str*** can store the address of any String object, that is, it can point to or refer to any String object. This means there are two different things – the variable ***str*** and the String object that ***str*** points to. The object that ***str*** points to is called the object ***str.*** Here is an illustration to make this clearer:

To sum it up, while working with classes, a reference variable of a class type is declared, then an object is instantiated (typically with new operator) and its address is stored into the reference variable. Also, two types of variables can be declared in Java: primitive type variables and reference variables.

Primitive type variables store data directly into their own memory spaces while reference variables store the address of the object containing the data. In some languages, such as C++, reference variables are called ***pointers***.[90]

3.2. Methods – overview and predefined methods

A method in object-oriented programming is a procedure associated with an object class. An object is made up of behaviour and data where data is represented as properties of the object and behaviour as methods. Methods are also the interface an object presents to the outside world. A simple example would be a ***window*** object that has methods ***open*** and ***close***.
One of the most important capabilities that a method provides is overriding. This is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The version of a method that is executed will be determined by

the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Methods also provide the interface that other classes use to access and modify the data properties of an object. This is known as encapsulation. Encapsulation and overriding are the two primary distinguishing features between methods and procedure calls and will be further explained later in the text.[91]

A class is a collection of methods and data members. One such method in Java is ***main*** that starts executing when the program is run. All programming instructions can be packed into this method but this technique is appropriate only for short programs. For large programs, it is not practical to put the entire programming instructions into one method. The problem should be broken into manageable pieces.

Methods in Java are building blocks. They are used to divide complicated programs into smaller parts. There are both predefined methods – methods that are already written and provided by Java – and user-defined methods – methods that programmers create.

Using methods has several advantages:

- While working on one method, you can focus on just that part of the program and construct it, debug it, and perfect it.
- Different people can work on different methods simultaneously.
- If a method is needed in more than one place in a program, or in

different programs, it can be written once and used many times.

- Using methods greatly enhances the program's readability because it reduces the complexity of the method ***main***.

Methods are often called ***modules***. They are like miniature programs since they can be put together to form a larger program. This is easier to see when writing new methods. In case of predefined methods, it is not as obvious because their programming code is not available to programmers.

In Java, the concept of a method, predefined or user-defined, is similar to that of a function in algebra. Every method has a name and, depending on the values specified by the user, it does some computation. Let's take as an example the predefined power method ***pow(x, y)*** that calculates x^y. The value of ***pow(x, y)*** is x^y, so ***pow(2, 3)*** is 8.0. Because the value of this method is of type double, it is said that the method ***pow*** is of type double. Also, ***x*** and ***y*** are called ***parameters*** or ***arguments*** of the method. The method ***pow*** has two parameters.

Predefined methods in Java are organized as a collection of classes, called ***class libraries***. For example, the class ***Math*** contains mathematical methods and it is contained in the package ***java.lang.*** Another example is the class ***Character*** contained in the same package. It provides methods for manipulating characters.

In general, before using predefined methods of a class in a program, it is necessary to import the class with that method from its package. To use the method ***nextInt*** of the class ***Scanner***, for example, the class is imported from the package ***java.util***. By default, Java automatically imports classes from the package ***java.lang***.

```
import java.util.Scanner;

import java.util.*;
```

The symbol * is a wildcard and it is used to import all classes of a certain package.

A method of a class may contain the reserved word static and it is then called a static method. Otherwise, it is called a nonstatic method. An important property of static methods is that it can be called in a program using the name of the class, the dot operator, the method name, and the appropriate parameters. Methods in the ***Math*** class, that was already mentioned, are all static. This means that the general syntax for using this class' methods is:

```
Math.methodName(parameters);
```

Java 5.0 simplified the use of static methods of a class by introducing a different import statement.

```
//to use any static method of the class

import static packageName.ClassName.*;


//to use a specific static method of the class

import static packageName.ClassName.methodName;
```

These are called static import statements. If they are used, calling the method within the program is different:

```
import static java.lang.Math.*;


…

pow(2.5, 3.5);

…
```

This shows that using the class name and the dot operator is not necessary.[92]

However, there are possible problems with static imports. Suppose that there are two classes ***Class1*** and ***Class2***. Both classes contain the static method ***printAll***, and we want to use both classes in the same program. To use the right method, it should be called using the name of the class and the dot operator.

To use methods, it is important to know the following properties:

- the name of the method
- the number of parameters, if any
- the data type of each parameter
- the data type of the value computed (the value returned) by the

method, called the type of the method

3.3. Methods – user-defined methods and method overloading

User-defined methods in Java are classified into two categories:

- value-returning methods – methods that have a return data type; these methods return a value of a specific data type using the return statement;
- void methods – methods that do not have a return data type; these methods do not use a return statement to return a value.

Value-returning methods calculate and return a value. The method ***pow*** was already mentioned as a value-returning method. Some other methods of this type are ***sqrt*** (returns the square root of the parameter), ***isLowerCase*** (returns a Boolean value determining whether the characters in the string are all lower case), ***toUpperCase*** (returns a string that was passed as a parameter but transformed so all the characters are upper case). The value returned is typically unique, so it is natural to use that value in one of three ways:

- save the value for further calculation
- use the value in some calculation
- print the value

This suggests that a value-returning method is used in either an assignment statement or an output statement. That is, a value-returning method is used in an expression.

The properties of methods that were mentioned at the end of part 3.3. become part of what is called the ***heading*** of the method. In addition to those properties, it is essential for the method to contain the code required to accomplish the task. This property is called the ***body*** of the method. Together, these five properties form what is called the ***definition*** of the method.

For example, the method ***abs*** (absolute) can have the following heading:

```
public static int abs(int number)
```

The definition of this method is as follows:

```
public static int abs(int number) {
if(number < 0)
number = -number;
return number;
}
```

The variable declared in the heading (***number***) is called the ***formal parameter*** of the method. Let’s examine the next line of code:

```
int n = abs(-15);
```

This code includes the call of the method ***abs***. The value -15 is passed to this method and it is copied to the variable ***number***. This value is called an ***actual parameter*** of the method call. So, here are the two important definitions:

- Formal parameter: A variable declared in the method heading.
- Actual parameter: A variable or expression listed in a call to a method. [93]

If a formal parameter is a variable of a primitive data type, there is no connection between the formal and the actual parameter after copying the value of the actual parameter. That is, the formal parameter has its own copy of the data. Therefore, during program execution, the formal parameter manipulates the data stored in its own memory space, not the variable that was passed as the actual parameter. [94]

If a formal parameter is a reference variable, the value of the actual parameter is also copied into the corresponding formal parameter, but there is a slight difference. Reference variables do not store data directly in its own memory space but the address of the allocated memory space. Therefore, when this kind of value is passed, both actual and formal parameter refer to the same memory space, that is, the same object. If the formal parameter changes the value of the object, it also changes the value of the object of the actual parameter. Reference variables as parameters are useful in three situations:

- When you want to return more than one value from a method.
- When the value of the actual object needs to be changed.
- When passing the address would save memory space and time, relative to copying a large amount of data.[95]

The basic Java syntax of a value-returning method is:

```
modifier(s) returnType methodName(formal parameter list) {
statements
}
```

In this syntax:

- ***modifier(s)*** indicates the visibility of the method, that is, where in a program the method can be called (used). Some of the modifiers are public, private, protected, static, abstract, and final. To include more than one modifier, they must be separated with spaces. Only one modifier among public, private, and protected can be selected. Modifiers static and abstract can be chosen similarly. They will all be explained in detail later.
- ***returnType*** is the type of the value that the method returns. This type is also called the type of the value-returning method.
- ***methodName*** is a Java identifier, giving a name to the method.

- Statements enclosed between braces form the body of the method.

The Java syntax of a formal parameter list is:

datatype identifier, datatype identifier, …

The Java syntax to call a value-returning method is:

methodName(actual parameter list) //or

returnType var = methodName(actual parameter list)

The Java syntax of an actual parameter list is:

expression or variable, expression or variable, …

Thus, to call a value-returning method, you use its name with the actual parameters (if any) in parentheses. A method's formal parameter list can be empty, but the parentheses are still needed. If the formal parameter list is empty, the method heading of the method takes the following form:

modifier(s) returnType methodName()

If the formal parameter list is empty, in a method call, the actual parameter list is

also empty. In the case of an empty formal parameter list, in a method call, the empty parentheses are still needed. Thus, a call to a value-returning method is:

```
methodName()
```

In a method call, the number of actual parameters, together with their data types, must match the formal parameters in the order given. That is, actual and formal parameters have a one-to-one correspondence.

A value-returning method uses a ***return*** statement to return its value; that is, it passes a value back when the method completes its task. The ***return*** statement has the following syntax:

```
return expression;
```

where ***expression*** is a variable, constant value, or expression. The ***expression*** is evaluated and its value is returned. The data type of that value should be compatible with the return type of the method. When a ***return*** statement executes in a method, the method immediately terminates and the control goes back to the caller.

Here is an example to sum it all up. It is a method that determines and returns the larger of two numbers that are also its parameters. The data type of these parameters can be, for example, ***double*** so the method's type is also ***double***. Method can be named ***larger***.

```
public static double larger(double x, double y) {
double max;

if(x >= y) {
max = x;
}
else {
max = y;
}
return max;
}
```

Note that the method uses an additional variable ***max***, called a ***local declaration***.

There are also ways to make this code more elegant and the same method can be written as:

```
public static double larger(double x, double y) {
if(x >= y) {
return x;
```

```
}
else {
return y;
}
}
```

Because the execution of a ***return*** statement in a method terminates the method, the method definition can also be written as:

```
public static double larger(double x, double y) {
if(x >= y) {
return x;
}
//next line will only be reached if the above condition is false
return y;
}
```

The ***return*** statement can appear anywhere in the method. Because all subsequent statements are skipped once ***return*** is executed, it is a good idea to return the value as soon as it is computed.[96]

Void methods do not return a value using the ***return*** statement so the return value of these methods can be considered void. They have similar structure to that of value-returning methods – both have a heading part and a statement part. Void methods can also use ***return*** statement but without any value and it is typically used to exit the method early.

Because void methods do not return a value, they are not used in an expression. Instead, a call to a void method is a stand-alone statement. To call a void method, the method name together with the actual parameters (if any) are used in a stand-alone statement.

The definition of a void method with parameters has the following syntax in Java:

```
modifier(s) void methodName(formal parameter list) {
statements
}
```

The formal parameter list looks the same as for the value-returning method, and it can also be empty. A method call in Java has the following syntax:

```
methodName(actual parameter list);
type var = methodName(actual parameter list); //This is not possible![97]
```

The method ***larger*** that was used as an example of a value-returning method can

also be written as void:

```
public static void larger(double x, double y) {
if(x >= y) {
max = x;
}
else {
max = y;
}
}
```

Here, the value that is larger is stored in the global variable declared outside the method.

In a Java program, method definitions can appear in any order. However, when a program executes, the first statement in the method ***main*** always executes first, regardless of where in the program the method ***main*** is placed. Other methods execute only when they are called.

A method call statement transfers control to the first statement in the body of the method. In general, after the last statement of the called method executes, control is passed back to the point immediately following the method call. A value-returning method returns a value. Therefore, for value-returning methods, after executing the method, control goes back to the caller, and the value that the

method returns replaces the method call statement. The execution then continues at the point immediately following the method call.[98]

In Java, several methods can have the same name within a class. This is called ***method overloading***. Before going into rules of overloading, it is important to know that two methods are said to have different formal parameter lists if both methods have:

- A different number of formal parameters, or
- If the number of formal parameters is the same, then the data type of the formal parameters, in the listed order, must differ in at least one position.

For example:

```
public void methodOne(int x);

public void methodTwo(int x, double y);

public void methodThree(double y, int x);
```

These methods all have different formal parameter lists. However, in the following example, methods have both three formal parameters, and the data type of the corresponding parameters is the same. Therefore, these methods have the same formal parameter list.

```
public void methodFour(int x, double y, char c);

public void methodFive(int one, double u, char firstC);
```

To overload a method name, within a class, any two definitions of the method must have different formal parameter list. The signature of a method consists of the method name and its formal parameter list. Two methods have different signature if they have either different names or different formal parameter lists. If a method is overloaded, then all the methods with that same name have different signatures if they have different formal parameter lists. So, the following method headings correctly overload the method ***methodXYZ***:

```
public void methodXYZ()

public void methodXYZ(int x, double y)

public void methodXYZ(double one, int y)

public void methodXYZ(int x, double y, char c)
```

Let's consider the following method headings to overload the method ***methodABC***:

```
public void methodABC(int x, double y)

public int methodABS(int x, double y)
```

Even though these two methods differ in return type, the signature of a method

does not include the return type. Both method headings have the same name and same formal parameter list. Therefore, these methods are incorrect to overload the method ***methodABC***.

If a method's name is overloaded, then in a call to that method, the formal parameter list of the method determines which method should execute. Let's say you need to write a method that determines the larger of two items and both of those items can be integers, floating-point numbers, characters, or strings. In this case, you can write several methods:

```
int largerInt(int x, int y)

char largerChar(char first, char second)

double largerDouble(double u, double v)

String largerString(String first, String second)
```

All of these methods perform similar operations. Instead of giving different names to these methods, the same name can be used for each method – larger, for example. Then, overloading of the method ***larger*** would be as follows:

```
int larger(int x, int y)

char larger(char first, char second)

double larger(double u, double v)

String larger(String first, String second)
```

If the call is ***larger(5, 3)***, the first method executes because the actual parameters match the formal parameters of the first method. If the call is ***larger('A', '9')***, the second method executes, and so on.

Method overloading is used when it is necessary to define the same action for different types of data.[99]

3.4. Classes and instances

A class is a collection of a specific number of components. The components of a class are called the members of the class. Class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behaviour (member functions, methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (subroutine that creates objects), and as the type of objects generated.

When a constructor creates an object, that object is called an instance of the class. The creation of an instance is called instantiation. Not all classes can be instantiated, like abstract classes, while those that can be instantiated are called concrete classes. After instantiation, the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

It is important to distinct class from type. Objects have type – the interface, that is, the types of member variables, the signatures of member functions (methods), and properties these satisfy. On the other hand, a class has an implementation, a concrete data structure and collection of subroutines.

Types generally represent nouns, such as a person, place or thing, and a class represents an implementation of these. For example, an ***Orange*** type represents the properties and functionality of oranges in general, while ***ABCOrange*** class would represent ways of producing oranges. This class could then produce particular oranges: instances of the ***ABCOrange*** class would be objects of type ***Orange***. Often only a single implementation of a type is given, in which case the class name is often identical with the type name.

To design a class, the first step is to determine what data needs to be manipulated and what operations will be used for that. For example, the class ***Circle*** would implement the basic properties of a circle – radius, area, and circumference of the circle. To find area and circumference, the class has to contain basic operations to perform these calculations. The skeleton of this class would be:

```
public class Circle {

private double radius;


public double area() {

//code that determines the area of the circle

}


public double circumference() {

//code that determines the circumference of the circle
```

```
}

}
```

The general Java syntax for defining a class is:

```
modifier(s) class ClassIdentifier modifier(s) {

classMembers

}
```

where modifier(s) are used to alter the behaviour of the class and, usually, ***classMembers*** consist of named constants, variable declarations, and/or methods, but can even include other classes. That is, a member of a class can be a variable or a method or an inner class. Previous examples mostly contain modifiers ***public***, ***private***, and ***static***.

The class members can be classified into four categories – ***private***, ***public***, ***protected***, or they can have no modifier. Modifiers affect member visibility in the following manner:

- If a member of a class is ***private***, it cannot be accessed outside the class.
- If a member of a class is ***public***, it can be accessed outside the class.
- If a class member is declared without any modifiers, it can be accessed

from anywhere within the package.

The last category is also called the default visibility of class members and this type is usually avoided. Modifier ***protected*** will be explained in the section about inheritance.

Deciding which members should be private and which should be public depends on the nature of each member. The general rule is that any member that needs to be accessed from outside the class is declared public; any member that shouldn't be accessed directly by the user should be declared private. For example, the user should be able to set the value of a variable within a class through the method (set method), or print that value, so these methods should be public.

Declaring variables within a class private is the usual practice in object-oriented programming. Users should not control the direct manipulation of the data member, so the user should never be provided with direct access to the variables. Instead, variables are accessed through get and set methods. Set methods change the values of variables and are called ***setters***, as well as ***mutator*** methods. Get methods only access the value of a variable and they do not change it. Such methods are called ***getters***, or ***accessor*** methods. Let's add those to the class ***Circle***.

```
public void setRadius(double value) {

radius = value;

}
```

```
public double getRadius() {

return radius;

}
```

Methods in a class are often public because they are used for manipulating data, setting the variable values, and getting variable values. However, a method can be private and that is the case when that method is only used to support other methods of the class. Also, the user of the class does not need to access this method, only other methods do. Keeping class members private so that users cannot access them directly and manipulate data is called encapsulation.

In addition to the methods necessary to implement operations, every class can have special types of methods called constructors. A constructor has the same name as the class, and it executes automatically when an object of that class is created. Constructors are used to guarantee that the variables of the class are initialized.

There are two types of constructors: those with parameters and those without them. The constructor without parameters is called the default constructor. Constructors have the following properties:

- The name of a constructor is the same as the name of the class.
- A constructor, even though it is a method, has no return type. That is, it is neither a value-returning method nor a void method.
- A class can have more than one constructor. However, all constructors of a class have the same name, so the constructors of a class can be

overloaded.

- If a class has more than one constructor, the constructors must have different signatures.
- Constructors execute automatically when class objects are instantiated. Because they have no type, they cannot be called like other methods.
- If there are multiple constructors, the constructor that executes depends on the type values passed to the class object when the class object is instantiated.

We can add two constructors to the class ***Circle*** previously used as an example. One of the constructors will be the default constructor that sets the value of the radius to 0. The second constructor will set the past value as the value of the radius.

```
public Circle() {
radius = 0;
}

public Circle(double value) {
radius = value;
}
```

If there is no constructor included in a class, Java automatically provides the default constructor. Therefore, when object is created, variables are initialized to their default values – int to 0, etc. If there is at least one constructor, Java will not automatically provide the default constructor. Generally, if a class includes constructors, it is best to also include the default constructor.

Once a class is defined, it is possible to declare reference variables of that class type. The next two lines are examples of variable declaration:

```
Circle c1;

Circle c2;
```

These two lines declare ***c1*** and ***c2*** to be reference variables of type ***Circle***. However, these statements do not allocate memory spaces to store variable ***radius***. To store variables that a class has, it is necessary to create an object of that class. This is accomplished by using the operator ***new***.

The general Java syntax for using the operator ***new*** is:

```
new className()
```

or:

```
new className(argument1, argument2, …, argumentN)
```

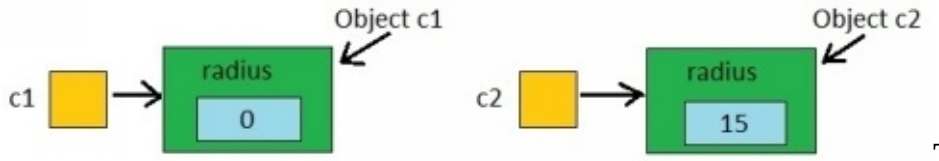
The first expression instantiates the object and initializes the instance variables of the object using the default constructor. The second expression instantiates the object and initializes the instance variables using a constructor with parameters.

Let's consider the following two lines:

```
c1 = new Circle();

c2 = new Circle(15);
```

Figure 4 – Objects c1 and c2



The first line allocates memory space for a ***Circle*** object, initializes each instance variable to the object 0 (only radius in this case), and stores the address of the object into ***c1***. The second statement allocates memory space for a ***Circle*** object, initializes the instance variable ***radius*** to 15, and stores the address of the object into ***c2***. The object to which ***c1*** points is called the object ***c1*** and the object to which ***c2*** points the object ***c2***.

Naturally, statements can be combined to declare the variable and instantiate the object into one statement. The previous instantiations can be combined like this:

```
Circle c1 = new Circle();

Circle c2 = new Circle(15);
```

So, creating an object of a class type means to:

- declare a reference variable of the class type
- instantiate the class object
- store the address of the object into the reference variable declared

Once an object of a class is created, the object can access the members of the class. The general Java syntax for an object to access a data member or a method is:

```
referenceVariableName.memberName
```

The class member that the class object can access depend on where the object is created. The following rules apply:

- If the object is created in the definition of a method of the class, then the object can access both the public and private members.
- If the object is created elsewhere (like in a user's program), then the object can access only the public members of the class.

In Java, the dot . is called the member access operator. Here is how we can access member of the Circle class:

```
c1.setRadius(10);

System.out.println("The area of the circle with the value of the radius " +
c2.getRadius() + " is " + c2.area());
```

The following statement is illegal because the variable is private:

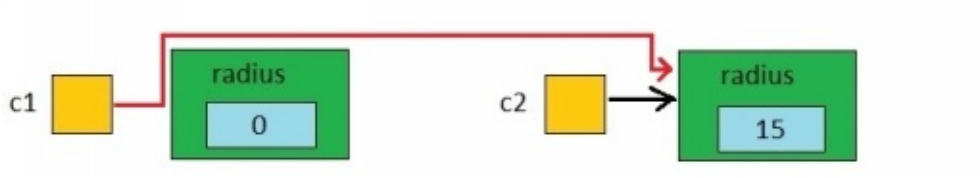
```
c1.radius();
```

Most of Java's built-in operations do not apply to classes. Arithmetic operations cannot be performed on classes, like using the operator + to add the values of two objects. Relational operators also cannot be used to compare two class objects in any meaningful way. The dot operator and assignment operator are valid but the second one should be used carefully.
Assignment operator doesn't work on reference variables the same way it does on the primitive type variables. Here is a statement with this operator:

```
c1 = c2;
```

This statements copies the value of the reference variable ***c2*** into the reference variable ***c1***. Since that value is the object address and not the object itself, after the statement executes, both ***c1*** and ***c2*** will refer to the same object.

Figure 5 – The effect of assignment operator



This is called the shallow copying of data. In shallow copying, two or more reference variables of the same type point to the same object; that is, two or more reference variables become aliases. The object originally referred to by ***c1*** now becomes inaccessible.

To copy the instance variables of the object ***c2*** into the corresponding instance variables of the object ***c1***, there needs to be a method that will copy each of the variable values from one object to another. That method would look something like this for the class ***Circle***:

```
public void copyData(Circle c) {

c.setRadius(radius);

//the circle sends data to another circle as set method parameters

}
```

This is called the deep copying of data. In deep copying, each reference variable refers to its own object, not the same object.[100]

It was mentioned before that relational operators do not give expected results when used on reference variables. If there are specific class variables that need to be compared, a new method should be written to provide that function, especially since those variables are typically private so they cannot be accessed directly. The most interesting of relational operators is == (equals).

We already know that reference variables contain only addresses. So, what happens when two reference variables are compared with the == operator?

```
if(c1 == c2) {
System.out.println(“Variables are equal”);
}
else {
System.out.println(“Variables are not equal”);
}
```

The answer is the code above will print out that variables are not equal. The reason is simple. Two values are compared the same way two int values would be compared. However, the values that are checked here are addresses. So, the operator == compares two addresses and returns ***true*** if two reference values point (refer) to the same object, and ***false*** otherwise.

Each class in Java can redefine the method ***equals***. This allows us to define when two reference variables should be considered equal. In the case of the class ***Circle***, two variables of the type ***Circle*** should be equal if they have the same value of the variable ***radius***.

```
public boolean equals(Circle c) {
return (c.getRadius() == radius);
```

```
}
```

Because ***radius*** is of type double, the equals operator will behave as expected. If we include this method in a class, reference variables can be compared like this:

```
c1.setRadius(20);
c2.setRadius(20);

if(c1.equals(c2)) {
System.out.println(“Variables are equal”);
}
else {
System.out.println(“Variables are not equal”);
}
```

Since we set both ***radius*** variables to 20, the code above will print out that ***c1*** and ***c2*** are equal, because this method compares concrete values instead of addresses.

3.5. Static class members

In the part 3.3. about predefined methods, it was explained how existing

methods are imported and used. In examples that used the class ***Math***, it was not necessary to create any objects in order to use its methods. It was enough to write the following import statement:

```
import static java.lang.Math.*;
```

After that, calling the method ***pow*** only required one line:

```
pow(5, 3);
```

In case the regular import was used instead of static import, calling ***pow*** would be done as follows:

```
Math.pow(5, 3);
```

That is, the method was called simply by using the name of the class and the dot operator.

The same approach cannot be used with the class ***Circle***. Although the methods of the class ***Math*** are public, they are also defined using the modifier static. The modifier static in the heading specifies that the method can be invoked by using the name of the class. Similarly, if a data member of a class is declared using the modifier static, it can be accessed by using the name of the class.

Let’s use a new illustrative class for this section.

```
public class Illustrate {

private int x;

private static int y;

public static int count;


//default constructor

public Illustrate() {

x = 0;

}


//constructor with parameters

public Illustrate(int a) {

x = a;

}


//method that returns the values of variables as a string

public String toString() {

return (“x = “ + x + “, y = “ + y + “, count = “ + count);
```

```
}

public static void incrementY() {
y++;
}
}
```

If there was a variable declaration for this class, that reference variable could access any public member of the class ***Illustrate***.

```
Illustrate illObject = new Illustrate();
```

Since the method ***incrementY*** is static and public, the following statement is legal:

```
Illustrate.incrementY();
```

Similarly, because the data member ***count*** is static and public, the following statement is also legal:

```
Illustrate.count++;
```

In essence, public static members of a class can be accessed either by an object, that is, by using a reference variable of the class type, or using the class name and the dot operator.

Suppose that there is a class ***MyClass*** with data members, both static and non-static. When an object of type ***MyClass*** is instantiated, only the non-static data members of the class ***MyClass*** become the data members of each object. For each static data member of the class, Java allocates memory space only once. All ***MyClass*** objects refer to the same memory space. In fact, static data members of a class exist even when no object of the class type is instantiated. Moreover, static variables are initialized to their default values.

For the class ***Illustrate***, memory space exists for the static data members ***y*** and ***count***. Now consider following statements:

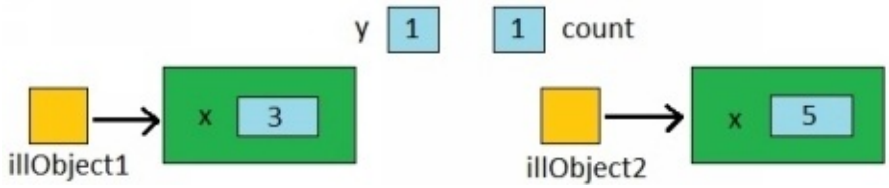
```
Illustrate illObject1 = new Illustrate(3);

Illustrate illObject2 = new Illustrate(5);
```

These statements declare two variables of type ***Illustrate*** and instantiate the two objects. Next, we can modify static members:

```
Illustrate.incrementY();

Illustrate.count++;
```



Here is a figure that illustrates objects and static variables.

Figure 6 – Illustration of static class members

As the figure shows, static data members can be considered as independent variables that do not belong to a created object but to a class itself.

Here is how it would look if we printed out the values:

```
//statement that invokes method toString
System.out.println(illObject1);

//output
x = 3, y = 1, count = 1

//statement
System.out.printl(illObject2);

//output
x = 5, y = 1, count = 1
```

```
//statements
Illustrate.count++;
System.out.println(illObject1);
System.out.println(illObject2);

//outputs
x = 3, y = 1, count = 2
x = 3, y = 1, count = 2
```

Because static members of a class do not need any object, a static method cannot use anything that depends on a calling object. In other words, in the definition of a static method, you cannot use a non-static data member or a non-static method, unless there is a locally declared object that accesses the non-static data member or the non-static method.[101]

The best example for this is the ***main*** method. It is a static method so it can call only other static methods. So how come we can normally invoke methods after we create an object? This is exactly what the previous paragraph explains. The object that is created within the ***main*** method is local for that method and, as we know by now, non-static methods can be accessed through a reference variable previously instantiated.

3.6. Keyword this

There were examples of non-static method calls previously, and one of them is:

```
c1.setRadius(10);
```

The statement uses the method ***setRadius*** to set the instance variable ***radius*** of the object ***c1*** to 10. You might be wondering how Java (the language used in this case) makes sure that the statement above sets the instance variable of the object ***c1*** and not of another ***Circle*** object. The answer is that every object has access to a reference of itself. The name of this reference is ***this***, although languages like Python, Ruby, and Swift use the keyword ***self*** and Visual Basic uses ***Me***.

Java implicitly uses the reference ***this*** to refer to both the instance variables and the methods of a class. The definition of the method ***setRadius*** is:

```
public void setRadius(int value) {
radius = value;
}
```

The statement:

```
radius = value;
```

is, in fact, equivalent to the statement:

```
this.radius = value;
```

In this statement, the reference ***this*** is used explicitly. It can be used explicitly so that the new definition of the method is as follows:

```
public void setRadius(int radius) {
this.radius = radius;
}
```

Here, the name of the formal parameter and the name of the instance variables are the same. In this definition of the method, the expression ***this.radius*** means the instance variable ***radius***, not the formal parameter ***radius***. Because the code explicitly uses the reference ***this***, the compiler can distinguish between the instance variable and the formal parameter.

In addition to explicitly referring to the instance variables and methods of an object, the reference ***this*** has another use – to implement cascaded method calls. We can use the class ***Circle*** that we've already defined and extend the definition of the class to set the radius and then return a reference to the object, using ***this.*** Let's look at the set method, since it's the only thing we are changing:

```
public Circle setRadius(int value) {
```

```
    radius = value;
    return this;
    }
```

When the method is defined like this, it is possible to do the following:

```
    //n is a variable of type double
    double radius = c1.setRadius(n).getRadius();
```

In this statement, the expression :

```
    c1.setRadius(n)
```

is executed first because the associativity of the dot operator is from left to right. This expression sets the radius to a value stored in the variable ***n*** and returns a reference to the object, which is the object ***c1***. Thus, the next expression executed is:

```
    c1.getRadius()
```

which returns the value of the radius of the object ***c1***.[102]

3.7. Inheritance

Suppose that we need to design a class ***PartTimeExmployee*** to implement and process the characteristics of a part-time employee. The main features associated with a part-time employee are the name, pay rate, and number of hours worked. Rather than designing the class ***PartTimeEmployee*** from scratch, we can first design the class ***Person*** to implement a person's name. Then, ***PartTimeEmployee*** could extend the definition of the class ***Person*** by adding additional members.

This does not mean we will make necessary changes directly to the class ***Person***, that is, edit it by adding and/or deleting members. The idea is to create a new class called ***PartTimeEmployee*** without making any physical changes to the class ***Person***, by adding only the members that are necessary to the new class. For example, the class representing a person already has fata members to store the first and last name, so those members do not need to be included in the new class as well. In fact, these data members will be inherited from the class ***Person***.

Another example could be a class representing a clock. The basic class called ***Clock*** would implement the time of day and store three data members to store the hours, minutes, and seconds. In case an application requires the time zone to be stored as well, we could extend the definition of the class ***Clock*** and create a class ***ExtClock***, to accommodate this new information. That is, the class ***ExtClock*** would be derived by adding a data member – ***timeZone*** – and the necessary method members to manipulate the time.

Java mechanisms that allow extending the definition of a class without making any physical changes to the existing class is ***inheritance***. Inheritance can be views as an "is-a" relationship. For example, every part-time employee ***is a*** person. Similarly, every extended clock ***is a*** clock.

Inheritance lets programmers create new classes from existing classes. Any new class that is created from an existing class is called a ***subclass*** or ***derived class***. Existing classes are called ***superclasses*** or ***base classes***. The inheritance relationship enables a subclass to inherit features from its superclass. Furthermore, the subclass can add new features of its own. Therefore, rather than creating completely new classes from scratch, it is better to take advantage of inheritance and reduce software complexity.

Inheritance can be viewed as a treelike, or hierarchical, structure wherein a superclass is shown with its subclasses. The following is a figure showing relationships between various shapes.

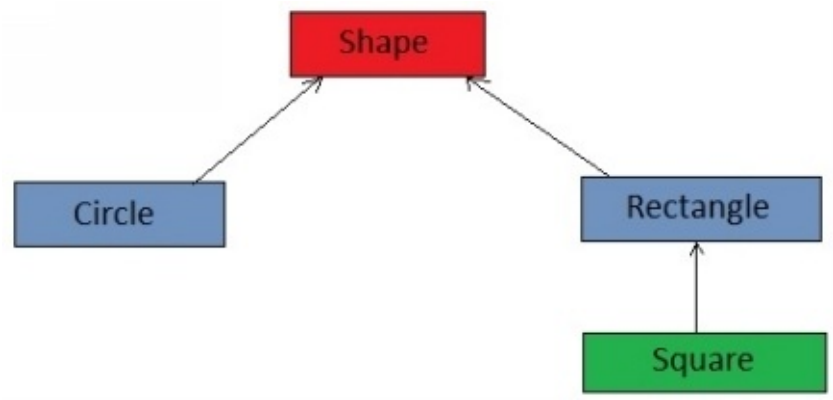


Figure 7 – Inheritance hierarchy

In this diagram, ***Shape*** is the superclass. The classes ***Circle*** and ***Rectangle*** are derived from ***Shape***, and the class ***Square*** is derived from ***Rectangle***. Every circle and every rectangle is a shape. Every square is a rectangle.

The general Java syntax to derive a class from an existing class is:

```
modifier(s) class ClassName extends ExistingClassName modifier(s) {
memberList
}
```

Here is a concrete example that specifies that the class ***Circle*** is derived from ***Shape***:

```
public class Circle extends Shape {
…
}
```

Here are the rules about superclasses and subclasses that should be kept in mind when using inheritance in Java:

- The private members of the superclass are private to the superclass; hence, the members of the subclass(es) cannot access them directly. In other words, when you write the definitions of the methods of the subclass, you cannot access the private members of the superclass directly.
- The subclass can directly access the public member of the superclass.
- The subclass can include additional data and/or method members.

- The subclass can override, that is, redefine, the public methods of the superclass. In the subclass, there can be a method with the same name, number, and types of parameters as a method in the superclass. However, this redefinition is available only to the objects of the subclass, not to the objects of the superclass. Also, objects of the subclass don't use the original method from the superclass.

- All data members of the superclass are also data members of the subclass. Even private members are inherited, they just cannot be accessed directly. Similarly, the methods of the superclass (unless overridden) are also the methods of the subclass.

Each subclass may become a superclass for a future subclass. Inheritance can be either single or multiple. In single inheritance, the subclass is derived from a single superclass; in multiple inheritance, the subclass is derived from more than one superclass. Java supports only single inheritance so in Java, a class can extend the definition of only one class. However, Java does offer options that can simulate multiple inheritance, but let's explain the basics first.

Suppose that a class ***SubClass*** is derived from a class ***SuperClass***. Further assume that both ***SubClass*** and ***SuperClass*** have some data members. It then follows that the data member of the class ***SubClass*** are its own data members, together with the data member of ***SuperClass***. Similarly, in addition to its own methods, the subclass inherits the methods of the superclass.

The subclass can give some of its methods the same signature as given by the superclass. For example, suppose that ***SuperClass*** contains a method called ***print*** that prints the values of the data members of ***SuperClass***. ***SubClass*** contains data members in addition to the data members inherited from ***SuperClass***.

Suppose that you want to include a method in the subclass that prints the data members of ***SubClass***. This method can have any name. However, it can also have the name ***print*** as it has in the superclass. This is called ***overriding***, or ***redefining***, the method of the superclass.

To override a public method of the superclass in the subclass, the corresponding method in the subclass must have the same name, the same type, and the same formal parameter list. That is, to override a method of a superclass, in the subclass the method must be defined using the same signature and the same return type as in its superclass. If the corresponding method in the superclass and the subclass has the same name but different parameter lists, then this is method ***overloading*** in the subclass, which is also allowed.

Whether a method is overridden or overloaded in the subclass, it is important to know how to specify a call to the method of the superclass that has the same name as that used by a method of the subclass. This concepts can be illustrated using an example, so here is a definition of a class:

```
public class Rectangle {

private double length;

private double width;



public Rectangle() {

length = 0;

width = 0;
```

```
}

public Rectangle(double l, double w) {
setDimension(l, w);
}

public void setDimension(double l, double w) {
if(l >= 0)
length = l;
else
length = 0;

if(w >= 0)
width = w;
else
width = 0;
}
```

```
public double getLength() {

return length;

}


public double getWidth() {

return width;

}


public double area() {

return length * width;

}


public double perimeter() {

return 2 * (length + width);

}


public String toString() {

return (“Length = “ + length + “; Width = “ + width);
```

```
}

}
```

This class represents a rectangle and has 10 members. Now let's look at the following definition of the class ***Box***, derived from the class ***Rectangle*** (the definitions of methods without body will be given below):

```
public class Box extends Rectangle {

private double height;


public Box() {

…

}


public Box(double l, double w, double h) {

…

}


public void setDimension(double l, double w, double h) {

…
```

```
}

public double getHeight() {
return height;
}

public double area() {
…
}

public double volume() {
…
}

public String toString() {
…
}
}
```

Since the class ***Box*** is derived from the class ***Rectangle***, all public members of ***Rectangle*** are public members of ***Box***. The class ***Box*** overrides the methods ***toString*** and ***area***, and overloads the method ***setDimension***.

In general, when writing the definitions of the methods of a subclass to specify a call to a public method of a superclass, you do the following:

- If the subclass overrides a public method of the superclass, then you must specify a call to that public method of the superclass by using the reserved word ***super***, followed by the dot operator, followed by the method name with an appropriate parameter list. In this case, the general syntax to call a method of the superclass is:

```
super.methodName(parameters);
```

- If the subclass does not override a public method of the superclass, you can specify a call to that public method by using the name of the method and an appropriate parameter list, like a regular method call.

Now, let's write the definition of the method ***toString*** of the class ***Box***. The class ***Box*** has three instance variables: ***length***, ***width***, and ***height***. The method ***toString*** of the class ***Box*** prints the values of these three instance variables. To write the definition of this method, keep in mind the following:

- The instance variables ***length*** and ***width*** are private members of the superclass and cannot be directly accessed in the class ***Box***. Therefore, when writing the definition of the method ***toString*** in the subclass, you cannot directly reference ***length*** and ***width***.
- The instance variables ***length*** and ***width*** are accessible in the class ***Box*** through the public methods of the superclass. Therefore, when writing the new method, you first call the method ***toString*** of the superclass to print the values of ***length*** and ***width***. After printing these values, you output the value of ***height***.

```
public String toString() {
return super.toString() + "; Height = " + height;
}
```

Here are the definitions of the remaining methods as well:

```
public void setDimension(double l, double w, double h) {
super.setDimension(l, w);

if(h >= 0)
height = h;
else
```

```
height = 0;

}
```

Note that because ***Box*** overloads the method ***setDimension***, the definition of the new method can specify a call to the superclass method without the reserved word ***super*** and the dot operator.

```
public double area() {

return 2 * (getLength() * getWidth() + getLength() * height + getWidth() *
height);

}
```

To determine the area, we cannot simply use the method of the superclass because the formula is not the same. Instead, we access private members through get methods.

```
public double volume() {

return super.area() * height;

}
```

Volume is determined by multiplying the area of the base, which is a rectangle, by its height.

Methods can have the modifier ***final***. In that case, they cannot be overridden in any subclass.

A subclass can also have its own constructors. A constructor typically serves to initialize the instance variables. But, when we instantiate a subclass object, this object inherits the instance variables even though it cannot directly access those that are private.

As a consequence, the constructors of the subclass can and should directly initialize only the instance variables of the subclass. Thus, when a subclass object is instantiated, to initialize the instance variables (private and other) – both its own and its ancestor's – the subclass object must also automatically execute one of the constructors of the superclass.

A call to a constructor of the superclass is specified in the definition of a subclass constructor by using the reserved word ***super***. The general syntax to call a constructor of a superclass in Java is:

```
super(parameters);
```

We will now go back to the definition of the class ***Box***. Two of its instance variables – ***length*** and ***width*** – are inherited. Let's write the definition of the default constructor first. Recall that if a class contains the default constructor and no values are specified during object instantiation, the default constructor executes and initializes the object. Because the class ***Rectangle*** contains the default constructor, when we write the definition of the constructor of the class ***Box***, to explicitly specify a call to the default constructor of the superclass, we

use the reserved work ***super*** with no parameters. Also, a call to the constructor of the superclass must be the first statement.

```
public Box() {
super();
height = 0;
}
```

Next, we write the definition of the constructor with parameters. Note that if you do not include the statement ***super()***, then, by default, the default constructor of the superclass will be called. To specify a call to a constructor with parameters of the superclass, we use the reserved word ***super*** with the appropriate parameters. A call to the constructor of the superclass must be the first statement here as well.

```
public Box(double l, double w, double h) {
super(l, w);
height = h;
}
```

When this constructor executes, it triggers the execution of the constructor with two parameters of type ***double*** of the class ***Rectangle***.

Note that invoking a superclass constructor's name in a subclass will result in a syntax error. Also, because a call to a constructor of the superclass always comes first, within the definition of a subclass constructor only one superclass constructor can be invoked.[103]

3.8. Protected class members and class Object

The private members of a class are private to the class and its subclasses so it cannot be directly accessed outside the class. Only methods of that class can access the private members directly. However, sometimes it may be necessary for a subclass to access a private member of a superclass. If you make a private member public, then anyone can access that member.
Recall that we mentioned the modifier ***protected*** besides ***private*** and ***public***. So, if a member of a superclass needs to be directly accessed in a subclass and yet still prevent its direct access outside the class, such as in a user program, that member is declared using the modifier ***protected***. Thus, the accessibility of a protected member of a class falls between public and private. A subclass can directly access the protected member of a superclass.

Here is an example with definitions of the classes ***BClass*** and ***DClass***:

```
public class BClass {

protected char bCh;

private double bX;


public BClass() {
```

```
bCh = '*';

bX = 0.0;

}


public BClass(char ch, double u) {

bCh = ch;

bX = u;

}


public void setData(double u) {

bX = u;

}


public void setData(char ch, double u) {

bCh = ch;

bX = u;

}
```

```
public String toString() {

return (“Superclass: bCh = “ + bCh + “, bX = “ + bX + ‘\n’);

}
```

The definition of this class contains the protected variable ***bCh*** and the private variable ***bX***. It also contains overloaded method ***setData***; one version of it is used to set both the instance variables, and the other version is used to set only the private instance variable. There is also a default constructor.

The following is the second class that extends ***BClass***. It also contains a method ***setData***, with three parameters and the method ***toString***. Definitions of empty methods will be further provided.

```
public DClass extends BClass {

private int dA;


public DClass() {

…

}


public DClass(char ch, double v, int a) {

…
```

```
}

public void setData(char ch, double v, int a) {
…
}

public String toString() {
…
}
}
```

Let’s start with the definition of the method ***setData***. Because ***bCh*** is protected, it can be directly accessed from this method. However, ***bX*** is private and cannot be accessed directly. Thus, this method ***setData*** must set ***bX*** using the method ***setData*** of its superclass. The definition can be written as follows:

```
public void setData(char ch, double v, int a) {
super.setData(v);
bCh = ch;
dA = a;
```

```
}
```

The statement ***super.setData(v)*** calls the method ***setData*** of the superclass with one parameter, to set the instance variable ***bX***. The value of ***bCh*** is set directly.

Here are the remaining definitions:

```
public DClass() {
super();
dA = 0;
}

public DClass(char ch, double v, int a) {
super(ch, v);
dA = a;
}

public String toString() {
return (super.toString() + “Subclass dA = “ + dA + ‘\n’);
}
```

It was mentioned before that every Java class has the method ***toString*** that we can redefine. If a user-defined class does not provide its own definition of this method, then the default definition of the method ***toString*** is invoked. So, the question is, where is the method ***toString*** originally defined?

This method comes from the Java class ***Object***, and it is a public member of this class. In Java, if you define a class and do not use the reserved word extends to derive it from an existing class, then the class you define is automatically considered to be derived from the class ***Object***. Therefore, the class ***Object*** directly or indirectly becomes the superclass of every class in Java.

Using the inheritance mechanism, every public member of the class ***Object*** can be overridden and/or invoked by every object of any class type. Those members include the method ***toString***, as well as ***equals*** that was already explained.[104] [105]

3.9. Polymorphism

Java allows treating an object of a subclass as an object of its superclass. In other words, a reference variable of a superclass type can point to an object of its subclass. One of previous examples will be used. There is a class ***Person*** that contains person's first and last name, and a class ***PartTimeEmployee***, that extends the class ***Person*** and adds pay rate and number of hours worked. Consider the following statements:

```
Person name, name Ref; //line 1

PartTimeEmployee employee, employeeRed; //line 2
```

```
name = new Person(“John”, “Blair”);

employee = new PartTimeEmployee(“Susan”, “Johnson”, 12.5, 45);
```

The statement in line 1 declares ***name*** and ***nameRef*** to be reference variables of type ***Person***. Similarly, the statement in line 2 declares ***employee*** and ***employeeRef*** to be reference variables of type ***PartTimeEmployee***. The statement in line 3 instantiates the object ***name*** and the statement in line 4 instantiates the object ***employee***.

Now consider the following statements:

```
nameRef = employee; //line 5

System.out.println(“nameRef: “ + nameRef); //line 6
```

The statement in line 5 makes ***nameRef*** point to the object ***employee***. After this statement executes, the object ***nameRef*** is treated as an object of the class ***PartTimeEmployee***. The statement in line 6 outputs the value of the object ***nameRef***. The output is:

```
nameRef: Susan Johnson’s wages are: $562.5
```

Even though ***nameRef*** is declared as a reference variable of type ***Person***, when the program executes, the statement in line 6 outputs the first name, the last

name, and the wages of a ***PartTimeEmployee***. This is because when the statement executes to output ***nameRef***, the method ***toString*** of the class ***PartTimeEmployee*** executes, and not that of the class ***Person***. This is called ***late binding***, ***dynamic binding***, or ***run-time binding***. The method that gets executed is determined at execution time, not at compile time.

In a class hierarchy, several methods may have the same name and the same formal parameter list. Also, a reference variable of a class can refer to either an object of its own class or an object of its subclass. Therefore, a reference variable can invoke, that is, execute, a method of its own class or of its subclass(es).

Binding means associating a method definition with its invocation, that is, determining which method definition gets executed. In early binding, a method's definition is associated with its invocation when the code is compiled. In late binding, a method's definition is associated with the method's invocation at execution time. Except for a few special cases, Java uses late binding for all methods. Furthermore, the term ***polymorphism*** means associating multiple potential meanings with the same method name. In Java, polymorphism is implemented using late binding.

The reference variable ***name*** or ***nameRef*** can point to any object of the class ***Person*** or the class ***PartTimeEmployee***. Loosely speaking, we say that these reference variables have many forms, that is, they are ***polymorphic reference*** variables. They can refer to objects of their own class or to objects of the subclasses inherited from their class. However, you cannot automatically consider a superclass object to be an object of a subclass. In other words, you cannot automatically make a reference variable of a subclass type point to an object of its superclass.

Suppose that ***supRef*** is a reference variable of a superclass type. Moreover, suppose that ***supRef*** points to an object of its subclass. You can use an appropriate cast operator on ***supRef*** and make a reference variable of the subclass point to the object. On the other hand, if ***supRef*** does not point to a subclass object and you use a cast operator on ***supRef*** to make a reference variable of the subclass point to the object, then Java will throw a ***ClassCastException*** – indicating that the class cast is not allowed.

Let's examine that through the previously used example.

```
Person name, nameRef;
PartTimeEmployee employee, employeeRef;


name = new Person("John", "Blair");
employee = new PartTimeEmployee("Susan", "Johnson", 12.5, 45);
nameRef = employee;


employeeRef = (PartTimeEmployee) name; //ILLEGAL
```

The variable ***name*** points to an object of the class ***Person*** so an exception will be thrown. The following statement is legal:

```
employeeRef = (PartTimeEmployee) nameRef;
```

Because ***nameRef*** refers to the object ***employee***, and ***employee*** is a reference variable of the ***PartTimeEmployee*** type, this statement would make ***employeeRef*** point to the object ***employee***.

To determine whether a reference variable that points to an object is of a particular class type, Java provides the operator ***instanceof***.

```
p instanceof Box
```

This expression evaluates to ***true*** if ***p*** points to an object of the class ***Box***; otherwise, it evaluates to ***false***.[106]

It is advised to use the cast operator together with ***instanceof*** operator to avoid the mentioned exception.

3.10. Abstraction and Java multiple inheritance

An abstract method is a method that has only the heading with no body. The heading of an abstract method contains the reserved word ***abstract*** and ends with a semicolon. The following are examples of abstract methods:

```
public void abstract print();

public abstract object larger(object, object);

void abstract insert(int insertItem);
```

An abstract class is a class that is declared with the reserved word ***abstract*** in its heading. Following are some facts about abstract classes:

- An abstract class can contain instance variables, constructors, and nonabstract methods.
- An abstract class can contain an abstract method.
- If a class contains an abstract method, then the class must be declared abstract.
- An object of an abstract class cannot be instantiated. It is only possible to declare a reference variable of an abstract class type.
- An object of a subclass of an abstract class can be instantiated, but only if the subclass gives the definitions of all the abstract methods of the superclass.

Here is an abstract class example:

```
public abstract class AbstractClassExample {

protected int x;


public abstract void print();
```

```
public void setX(int a) {
x = a;
}

public AbstractClassExample() {
x = 0;
}
}
```

Abstract classes are used as superclasses from which other subclasses within the same context can be derived. They serve as placeholders to store members common to all subclasses. They can also be used to force subclasses to provide certain methods by making those methods abstract. If a subclass does not implement inherited abstract method(s), it is also abstract.[107]

Whether the superclass is abstract or not, Java does not allow a class to extend more than one class. However, there are ways to simulate this behaviour – using interfaces. An interface is a type of a class that contains only abstract methods and/or named constants. Interfaces are defined using the reserved word ***interface*** in place of the reserved word ***class***.

Classes do not extend interfaces like they extend classes, but they implement them. Despite being able to only extend one class, classes can implement many interfaces. Interfaces are similar to classes and their purpose is to enable multiple

inheritance. If a class implements an interface, it must provide definitions for each of the methods of the interface; otherwise, an object of that class type cannot be instantiated. For example, a single class ***Person*** can implement multiple interfaces such as ***Driver***, ***Commuter***, and ***Biker***. So, interfaces, like inheritance, are used to share common behaviour and common code.[108]

Just as it is possible to create polymorphic references to classes in an inheritance hierarchy, it is possible to create polymorphic references using interfaces. Interface name can be used as the type of a reference variable, and the reference variable can point to any object of any class that implements that interface. However, because an interface contains only method headings and/or named constants, an object of an interface cannot be created.[109]

Java 8 introduced lambdas and as a result, it was necessary to allow interface methods with concrete implementations. These methods are called ***default methods*** and expand possibilities to implement multiple inheritance in Java. Without these methods, if an interface that has already been designed got a new method, every program containing a class that implements that interface would have to implement that same method. To avoid that, a class only has to implement ordinary methods, while it can choose to either keep the implementation of the default method, or override it.

In case there is the exact same default method defined in one interface and again in a superclass or another method, Java follows these rules:

- If a superclass provides a concrete method, default methods are simply ignored.
- If two interfaces provide a default method and a method with the same

name and parameter types (default or abstract), programmer has to resolve the conflict by overriding that method.

According to the second rule, if at least one of the two conflicted methods is default, Java compiler reports an error rather than just choosing one method. Here is an example of an interface that has a default method and a class that implements it:

```
interface Person {


int getId();

default String getName() {

return “John Doe”;

}


}


class Student implements Person {


public String getName() {
```

```
return super.getName();
}

}
```
[110]

Multiple inheritance is allowed in C++, for example. This feature can be very useful but programmers dislike it. Problems can occur when using multiple inheritance. For example, suppose that there are two separate classes ***Lawyer*** and ***Businessman,*** each with a method that processes salary – ***pay().*** If there was a new class ***LawyerBusinessman*** which inherits both ***Lawyer*** and ***Businessman*** and ***pay()*** was invoked for its object, which ***pay()*** method should be called? It could be either the one from the class ***Lawyer*** or the one from class ***Businessman*** and they have different implementations. Besides this problem, multiple inheritance is harder to visualize with class diagrams.[111]

4. Advantages and disadvantages

Like other paradigms, object-oriented programming has its advantages and disadvantages. Let's look at the advantages first. Object-oriented programming is modular, as it provides separation of duties in program development. It is also extensible, because objects can be extended to include new attributes and behaviours using inheritance. Objects can also be reused within other applications. Because of these three factors – modularity, extensibility, and reusability – object-oriented programming provides improved software-

development productivity over traditional procedure-based programming techniques.

For the same reasons, object-oriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes. Reuse also enables faster development since object-oriented programming languages come with rich libraries of objects, and code developed during projects is reusable in future projects. Another advantage of reusability is lower cost of development. More effort is put into the object-oriented analysis and design, which lowers the overall cost of development. Finally, faster software development and its lower cost allows more time and resourced to be used in the verification of the software. Object-oriented programming tends to result in higher-quality software.

When it comes to disadvantages, for starters, the thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction between objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.

Object-oriented programs typically have more lines of code than procedural programs. Besides that, programs are typically slower, as they require more instructions to be executed. There are problems that lend themselves well to functional programming style, logic programming style, or procedure-based programming style, and applying object-oriented programming in those situations does not result in efficient programs.

In conclusion, object-oriented programming offers a lot of features that may not be easy to understand at first, but are very useful once learned. And even though it cannot serve every purpose, OO programming languages that are used today

are rarely pure. Instead, they also implement concepts of functional programming, for example, offering even more programming options.[112]

Bibliography

1. MALIK, D. S. (2012) ***Java Programming From Problem Analysis to Program Design.*** 5th Edition. Cengage Learning

2. WU, C. T. (2009) ***An Introduction to Object-Oriented Programming with Java.*** 5th Edition. McGraw-Hill

3. SHARAN, K. (2014) ***Beginning Java 8 Fundamentals: Language Syntax, Arrays, Data Types, Objects, and Regular Expressions.*** Apress

4. GREGOIRE, M., SOLTER, N. A., KLEPER, S.J. (2011) ***Professional C++.*** 2nd Edition. Wrox

5. SUTHERLAND, I. E. (1963) ***Sketchpad: A Man-Machine Graphical Communication System***

6. HOLMEVIK, J. R. (1994) ***Compiling Simula: A historical study of technological genesis*** [Online] Available at: http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf [Accessed: 22nd June 2015]

7. KAY, A. (1993) ***The Early History of Smalltalk*** [Online] Available at: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

[Accessed: 22nd June 2015]

8. ORACLE (2015) ***What Is an Object?*** [Online] Available at: http://docs.oracle.com/javase/tutorial/java/concepts/object.html [Accessed: 22nd June 2015]

9. SAYLOR (2015) ***Advantages and Disadvantages of Object-Oriented Programming*** [Online] Available at: http://www.saylor.org/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf [Accessed: 1st June 2015]

Declarative Programming

D eclarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow[113]. Many languages applying this style attempt to minimize or eliminate side effects by describing what the program should accomplish in terms of the problem domain instead of describing how to go about accomplishing it. This is in contrast with imperative programming, in which algorithms are implemented in terms of explicit steps.

This paradigm often considers programs as theories of a formal logic, and computations as deductions in that logic space. Declarative style of programming may greatly simplify writing parallel programs[114].

Declarative programming is often defined as any style of programming that is not imperative. This is also a term that actually refers to a number of better-known programming paradigms. Common declarative languages include those of database query languages (SQL), logic programming, functional programming, etc.

1. History

The two main subparadigms of declarative programming are functional and logic programming.

The logic programming paradigm has its roots in automated theorem proving from which it took the notion of a deduction. What is new is that in the process of deduction some values are computed. The creation of this programming paradigm is the outcome of a long history that for most of its course ran within logic and only later inside computer science. Logic programming is based on the syntax of first-order logic, which was originally proposed in the second half of nineteenth century by Gottlob Frege and later modified to the currently used form by Giuseppe Peano and Bertland Russell.

In the 1930s, Kurt Gödel and Jacques Herbrand studied the notion of computability based on derivations. These works can be viewed as the origin of the “computation as deduction” paradigm. Additionally, Herbrand discussed in his doctoral thesis a set of rules for manipulating algebraic equations on terms that can be viewed now as a sketch of a unification algorithm. In 1965, Alan Robinson published his fundamental paper that lies at the foundation of the field of automated deduction. In this paper, he introduced the resolution principle, the notion of unification, and a unification algorithm. Using the resolution method, one can prove theorems of first-order logic, but another step was needed to see how one could compute within this framework.

This was eventually achieved by Robert Kowalski in ***Predicate logic as a programming language*** (North-Holland, 1974), in which logic programs with a restricted form of resolution were introduced. The difference between this form

of resolution and the one proposed by Robinson is that syntax is more restricted, but proving now has a side effect in the form of a satisfying substitution. This substitution can be viewed as a result of a computation and consequently certain logical formulas can be interpreted as programs. In parallel, Alain Colmerauer and his colleagues worked on a programming language for natural language processing based on automated theorem proving. This ultimately led to creation of Prolog in 1973.

Prolog can be seen as a practical realization of the idea of logic programs. It started as a programming language for applications in natural language processing, but soon after it was found that it can also be used as a general purpose programming language.

The logic programming paradigm influenced a number of developments in computer science. As early as the 1970s, it led to the creation of deductive databases that extend the relational databases by providing deduction capabilities. A further stimulation to the subject came unexpectedly from the Japanese Fifth Generation Project for intelligent computing systems, in which logic programming was chosen as its basis. More recently, this paradigm led to constraint logic programming that realizes a general approach to computing in which the programming process is limited to a generation of constraints (requirements) and a solution of them, and to inductive logic programming, a logic-based approach to machine learning.[115]

Functional programming has roots in lambda calculus which is a formal system developed in the 1930s used to investigate computability, function definition, function application, recursion. Lambda calculus is a mathematical abstraction rather than a programming language but it forms the basis of almost all functional programming languages today[116].

An early language with functional approach was Lisp developed in the late 1950s. Even though Lisp is a multi-paradigm language, it introduced features that are now found in functional languages. Another important functional language is ML that was introduced in the 1970s.[117] It developed into popular languages today OCaml and Standard ML. The language Scheme brought awareness of the power of functional programming to the wider programming community. In 1987, Haskell started to develop and its function was to form an open standard for functional programming research.

As functional languages matured, their practical use grew. They are being used in areas such as database processing, financial modelling, statistical analysis and bio-informatics. Functional languages simply became more popular for general programming as the execution became more efficient.

2. Overview of declarative paradigm

There is a number of common definitions that attempt to give declarative programming a definition other than simply contrasting it with imperative programming. For example:

- A program that describes what computation should be performed and not how to compute it
- Any programming language that lacks side effects (or more specifically, is referentially transparent)
- A language with a clear correspondence to mathematical logic

Here, the term ***side effect*** was mentioned. A function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world. For example, a function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, etc.[118]

Declarative programming contrasts with imperative and procedural programming. This is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed.

In logical programming languages, programs consist of logical statements, and the program executes by searching for proofs of the statements. In a pure functional language, like Haskell, all functions are without side effects, and state changes are only represented as functions that transform the state.[119] Other functional languages, such as Lisp, OCaml and Erlang, support a mixture of procedural and functional programming. Some logical programming languages, such as Prolog, and database query languages, such as SQL, while declarative in principle, also support a procedural style of programming.

3. Overview of logic paradigm

The logic programming paradigm differs from other programming paradigms. When stripped to the bare essentials, it can be summarized by the following three features:

- Computing takes place over the domain of all terms defined over a “universal” alphabet.

- Values are assigned to variables by means of automatically generated substitutions, called ***most general unifiers***. These values may contain variables, called ***logical variables***.
- The control is provided by a single mechanism: automatic backtracking.

Even such a brief summary shows both the strength and weakness of the logic programming paradigm. Its strength lies in an enormous simplicity and conciseness; its weakness has to do with the restrictions to one control mechanism and the use of a single data type.

This framework has to be modified and enriched to accommodate it to the customary needs of programming, for example by providing various control constructs and by introducing the data type of integers with the customary arithmetic operations. This can be done and, in fact, Prolog and constraint logic programming languages are examples of such a customization.[120]

Programs written in logic languages consist of declarations that are actually statements, or propositions, in symbolic logic. One of the essential characteristics of logic programming languages is their semantics, which is called declarative semantics. The basic concept of this semantics is that there is a simple way to determine the meaning of each statement, and it does not depend on how the statement might be used to solve a problem.

Declarative semantics is considerably simpler that the semantics of imperative languages. For example, the meaning of a given proposition in a logic programming language can be concisely determined from the statement itself.

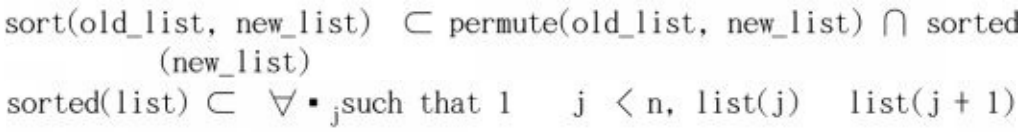
In an imperative language, the semantics of a simple assignment statement requires examination of local declarations, knowledge of the scoping rules of the language, and possibly even examination of programs in other files just to determine the types of variables in the assignment statement. Then, assuming the expression of the assignment contains variables, the execution of the program prior to the assignment statement must be traced to determine the values of those variables. The resulting action of the statement, then, depends on its run-time context.

Comparing this semantics with that of a proposition in a logic language, with no need to consider textual context or execution sequences, it is clear that declarative semantics is far simpler than the semantics of imperative languages. Thus, declarative semantics is often stated as one of the advantages of declarative languages.

Programming in a logic programming language is nonprocedural. It was already mentioned that programs in such languages do not state exactly how a result is to be computed but rather describe the form of the result. The difference is that we assume the computer system can somehow determine how the result is to be computed. What is needed to provide this capability for logic programming languages is a concise means of supplying the computer with both the relevant information and a method of inference for computing desired results. Predicate calculus supplies the basic form of communication to the computer, and resolution provides the inference technique.

A common example for illustrating the difference between procedural and nonprocedural systems is sorting. In a language like Java, sorting is done by explaining in a program all of the details of a sorting algorithm to a computers that has a Java compiler. The computer, after translating the Java program into machine code, follows the instructions and produces the sorted list.

In a nonprocedural language, it is necessary to only describe the characteristics of the sorted list: It is some permutation of the given list such that for each pair of adjacent elements, a given relationship holds between the two elements. To state this formally, suppose the list to be sorted is an array named list that has a subscript range 1…n. The concept of sorting the elements of the given list, named ***old_list***, and placing them in a separate array, named ***new_list***, can then be expressed as follows (permute is a predicate that returns true if its second parameter array is a permutation of its first parameter array):



From this description, the nonprocedural language system could produce the sorted list. That makes nonprocedural programming sound like the mere production of concise software requirements specifications. Unfortunately, it is not that simple. Logic programs that use only resolution face serious problems of execution efficiency. In the sorting example, if the list is long, the number of permutations is huge, and they must be generated and tested, one by one, until the one that is in order is found and that is a very lengthy process.[121]

4. Concepts of logic paradigm and theoretical background

The logic programming paradigm includes both theoretical and fully implemented languages but they all share the idea of interpreting computation as logical deduction. This part of text will examine these concepts and contain code examples written in Prolog, the best known fully implemented logic programming language.

4.1. Deduction as computation

A slogan that exactly captures the concepts that underpin the activity of programming is: Algorithm = Logic + Control. According to this equation, the specification of an algorithm, and therefore its formulation in programming languages, can be separated into three parts. On the one side, the logic of the solution is specified. That is, the "what" must be done is defined. On the other, those aspects related to control are specified, and therefore the "how" of finding the desired solution is clarified.

The programmer who uses a traditional imperative language must take account of both these components. Logic programming, on the other hand, was originally defined with the idea of cleanly separating these two aspects. The programmer is only required to provide a logical specification. Everything related to control is relegated to the abstract machine.

Using a computational mechanism based on a particular deduction rule, the interpreter searches through the space of possible solutions for the one specified by the "logic", defining in this way the sequence of operations necessary to reach the final results.[122]

Let's consider the following problem. It is desired to arrange three 1s, three 2s, …, three 9s in a list such that for each $i \in [1, 9]$, there are exactly ***i*** numbers between two successive occurrences numbered ***i***. Therefore, for example, 1, 2, 1, 8, 2, 4, 6, 2 could be a part of the final solution, while 1, 2, 1, 8, 2, 4, 2, 6 might not be one since there is only a single number between the last two occurrences of 2.

Resolving this issue in a declarative fashion would go as follows. First, we need

a list which we will call ***Ls***. This list will have to contain 27 elements, which we can specify using a unary predicate, that is, a relation symbol, ***list_of_27***. If we write ***list_of_27(Ls)***, we mean to say that ***Ls*** must be a list of 27 elements/ In other words, ***list_of_27*** defines a relation formed of all possible lists of 27 elements. To achieve this, we can define ***list_of_27*** like this:

```
list_of_27(Ls) :-

Ls = [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]
```

The part to the left of the : - symbol denotes what is being defined, while the part on the right indicates the definition. The list ***Ls***, in addition to being composed of 27 elements, will also have to contain a sublist in which the number 1 appears followed by any other number, by another occurrence of the number 1, then another number and finally a last occurrence of the number 1. Such a sublist can be specified as `[1, X, 1, Y, 1]` where ***X*** and ***Y*** are variables. More efficiently, we can write: `[1, _, 1, _, 1]` where _ is the anonymous variable, a variable whose name is of no interest. Assuming that we have available a binary predicate ***sublist(X, Y)*** whose meaning is that the first argument ***(X)*** is a sublist of the second ***(Y)***, so the requirement can be expressed by writing `sublist([1, _, 1, _, 1], Ls)`.

Moving on to number 2 and reasoning similarly, we obtain that the list ***Ls*** must also contain the sublist `[2, _, _, 2, _, _, 2]` and therefore the following must also be true: `sublist([2, _, _, 2, _, _, 2], Ls)`. The reasoning can be repeated as

so1(Ls):-

list_of_27(Ls),

sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ls),

sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ls),

sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ls),

sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ls),

sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ls),

sublist([4,_,_,_,_,4,_,_,_,_,4], Ls),

sublist([3,_,_,_,3,_,_,_,3], Ls),

sublist([2,_,_,2,_,_,2], Ls),

sublist([1,_,1,_,1], Ls),

list_of_27(Ls) :-

Ls = [_,_]

far as number 9, hence the ***sol*** program that we want to produce can be described as follows:

The above written says that to find a solution to our problem (***sol(Ls)***), all the properties described in the following lines must be satisfied. The commas that separate various predicates are to be interpreted as “and”, that is as conjunctions in the logical sense. What we have provided is nothing more than a specification that formally repeats the formulation of the problem and which can be interpreted in purely logical terms.

This declarative reading would not be of great use as the solution to our problem if it were not accompanied by a procedural interpretation and must therefore be translated into a conventional programming language. The important characteristic of logic paradigm is that the logical specifications we write are to all intents and purposes executable programs. The code described above is indeed a genuine Prolog program that can be evaluated by an interpreter to obtain the desired solution.

The specification above can also be read in a procedural fashion. The first line contains the declaration of a procedure called ***sol***; it has a single formal parameter ***Ls***. The body of this procedure is defined by the following 10 lines that contain the following ten procedure calls. First procedure ***list_of_27*** is

```
sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ls)
```

called with actual parameter ***Ls*** which is defined as we have seen above and which therefore instantiates its actual parameter into a list of 27 anonymous variables. In the next line, there is the call:

We assume that this call arranges matters so that the list which appears as the first parameter is a sublist of the list which is bound to the second parameter, possibly by instantiating the variable ***Ls***. The following calls are similar. Parameter passing occurs in a way similar to call by name and, in Prolog, the order in which the different procedure calls appear in the text specifies the order of evaluation, although in the case of pure logic programs, no order is specified.

Given the preceding definition of procedure ***sol***, the call ***sol(Ls)*** returns ***Ls*** instantiated with a solution to the problem. Successive calls to the same procedure will allow us to obtain the other solutions to the problem. One of the solutions is:

```
1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5,
```

With this procedural interpretation, we have a true programming language which allows us to express, in a compact and relatively simple way, programs that solve even very complex problems. For this power, the language pays the penalty of efficiency. In the preceding program, despite its apparent simplicity, the computation performed by the language's abstract machine is very complex, given that the interpreter must try the various combinations of possible sublists until it finds the one that satisfies all the conditions. In these search processes, a backtracking mechanism is used. When the computation arrives at a point at which it cannot proceed, the computation that has been performed is undone so that a decision point can be reached, if it exists, at which an alternative is chosen

that is different from the previous one. This search process can have exponential complexity.

4.2. First-order logic

Logic programs are sets of logic formulae of a particular form. The logic of interest here is first-order logic, also called predicate calculus. The terminology refers to the fact that symbols are used to express properties of elements of a fixed domain of discourse D. More expressive logics also permit predicates whose arguments are more complicated objects such as sets and functions over D (second order), sets of functions (third order), etc., in addition to elements of D.

When speaking of predicate calculus and therefore logic programs, we have to define the language. The language of first/order logic consists of three components:

1) An alphabet
2) Terms defined over this alphabet
3) Well-formed formulae over this alphabet

Let’s analyse these components in the same order.

Alphabet is usually a set of symbols. In this case, we consider the set to be partitioned into two disjoint subsets: the set of logical symbols (common to all first-order languages) and the set of non-logical symbols (which are specific to a domain of interest). For example, all first-order languages will use a logical

symbol to denote conjunction. If we are considering orderings on a set, we will probably also have the < symbol as one of the non-logical symbols.

The set of logical symbols contains the following elements:

- The logic connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\Rightarrow$ (implication) and $\Leftrightarrow$ (logical equivalence).
- The propositional constants ***true*** and ***false***.
- The quantifiers $\exists$ (exists) and $\forall$ (forall).
- Some punctuation symbols such as brackets "(" and ")" and comma ",".
- An infinite set ***V*** of variables, written ***X***, ***Y***, ***Z***,…

The non-logical symbols are defined by a signature with predicates (Σ, Π). This is a pair in which the first element is the function signature, that is a set of function symbols, each considered with its own arity (the arity denotes the number of arguments of a function or relation).

The second element of the pair, Π, is the predicate signature, a set of predicate symbols together with their arities. Functions of arity 0 are said to be constants and are denoted by the letters ***a***, ***b***, ***c***, …. Function symbols of positive arity are denoted by ***f***, ***g***, ***h***, …, while predicate symbols are denoted by ***p***, ***q***, ***r***, …. The difference between function and predicate symbols is that the former must be interpreted as functions, while the latter must be interpreted as relations.

Term is fundamental to mathematical logic and computer science and is used

implicitly in many contexts. For example, an arithmetic expression is a term obtained by applying operators to operands. Other types of constructs, too, such as strings, binary trees, lists, can be conveniently seen as terms which are obtained using appropriate constructors.

In the simplest case, a term is obtained by applying a function symbol to as many variables and constants as required by its arity. So, if ***a*** and ***b*** are constants, ***X*** and ***Y*** are both variables and ***f*** and ***g*** have arity 2, then $f(a, b)$ and $g(a, X)$ are terms. Nothing prevents the use of terms as the arguments to a function, provided that the arity is always respected. We can, for example, write $g(f(a, b), Y)$ or $g(f(a, f(X, Y)), X)$ and so on. In most general case, terms could be defined as follows.

The terms over a signature Σ (and over the se, ***V***, of variables) are defined inductively as follows:

- A variable (in ***V***) is a term.
- If f (in Σ) is a function symbol of arity ***n*** and $t_1, \cdots, t_n$ are terms, then $f(t_1, \cdots, t_n)$ is a term.

Formulae of the language allow us to express the properties of terms which, from the semantic point of view, are properties of some particular domain of interest. For example, if we have the predicate >, writing > (3, 2), we want to express the fact that the term “3” corresponds to a value greater than that associated with the term “2”. Predicates can then be used to construct complex expressions using logical symbols. The formula $>(X, Y) \quad >(Y, Z) \Rightarrow >(X, Z)$ expresses the transitivity of >; it asserts that if

$>(X, Y)$ is true and $>(Y, Z)$ is also true, it is the case that $>(X, Z)$.

To define formulae precisely, we have first atomic formulae (or atoms), constructed by the application of a predicate to the number of terms required by its arity. For example, if ***p*** has arity 2, using two terms introduced above, we can write $p(f(a, b), f(a, X))$. Using logical connectives and quantification, we can construct complex formulae from atomic ones. The following is a definition of formulae.

The formulae over the signature with terms (Σ, Π) are defined as follows:

1. If $t_1, \cdots, t_n$ are terms over the signature Σ and $p \in \Pi$ is a predicate symbol of arity ***n***, then $p(t_1, \cdots, p_n)$ is a formula.
2. ***true*** and ***false*** are formulae.
3. If ***F*** and ***G*** are formulae, then $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$ and $(F \cdot G)$ are formulae.
4. If ***F*** is a formula and ***X*** is a variable, then $\forall X.F$ and $\exists X.F$ are formulae.

4.3. Resolutions and clauses

In automatic theorem proving, and also in logic programming, particular classes of formulae, called ***clauses***, have been identified which lend themselves to more efficient manipulation, in particular using a special inference rule called ***resolution***.

A restricted version of the concept of clause, called ***definite clauses***, as well as

restricted forms of resolution are of interest when talking about logic programming. Using them, the procedure for seeking a proof is not only particularly simple but also allows the explicit calculation of the values of the variables necessary for the proof. These values can be considered as the result of computation, giving way to a model of computation based on logical deduction.

Let $H, A_1, \cdots, A_n$ be atomic formulae. A definite clause is a formula of the form:

$$H :- A_1, \cdots, A_n.$$

If n = 0, the clause is said to be a ***unit***, or a fact, and the symbol $:-$ is omitted. A logic program is a set of clauses, while a pure Prolog program is a sequence of clauses. A query is a sequence of atoms $A_1, \cdots, A_n$.

When it comes to the definition above, the symbol $:-$ is a symbol denoting (reversed) implication ( ) and is the same as often encountered in real logic languages. The commas in a clause or in a query should be interpreted as logical conjunction. The notation "$H :- A_1, \cdots, A_n.$" is therefore an abbreviation for "$H\ A_1 \cdots A_n.$" The full stop is part of the notation and is important because it tells a potential interpreter or compiler that the clause it is working on has terminated.

The part on the left of $:-$ is called the head of the clause, while that on the right is called the body. A fact is therefore a clause with an empty body. A program is a set of clauses in the case of theoretical formalism. In the case of Prolog, a program is considered as a sequence because the order of clauses is relevant. The set of clauses containing the predicate symbol ***p*** in their head is said to be the definition od ***p***. Variables occurring in the body of a clause and not in the head are said to be local variables.

4.4. Unification

The fundamental computational mechanism in logic programming is the solution of equations between terms using the unification procedure. In this procedure, substitutions are computed so that variables and terms can be bound. The composition of the different substitutions obtained in the course of computation provides the result of the calculation.

Before going into detail on the process of unification, it is important to clarify that the concept of variable is different from what we've seen before. Here, we consider the logic variable which is an unknown which can assume values from a predetermined set. In our case, this set is that of definite terms over the given alphabet. This fact, together with the use to which logic variables are put in logic programming, gives rise to three important differences between this concept and the modifiable variables in imperative languages:

1) The logic variable can be bound only once, in the sense that if a variable is bound to a term, this binding cannot be destroyed (but the term might be modified). For example, if we bind the variable ***X*** to the constant ***a*** in a logic program, the binding cannot later be replaced by another which binds ***X*** to the constant ***b***. This is possible in imperative languages using assignment.

2) The value of a logic variable can be partially defined (or undefined), to be specified later. This is because a term that is bound to a variable can contain other logic variables. For example, if the variable ***X*** is bound to the term $f(Y, Z)$, successive bindings of the variables ***Y*** and ***Z*** will also modify the value of the variable ***X***: if ***Y*** is bound to ***a*** and ***Z*** is bound to $g(W)$, the value of ***X*** will become the term $f(a, g(W))$. The process could continue by

modifying the value of ***W***. This mechanism for specifying the value of a variable by successive approximations is typical of logic languages and is somewhat different from the corresponding in imperative languages, where a value assigned to a value cannot be partially defined.

3) A third important difference concerns the bidirectional nature of bindings for logical variables. If ***X*** is bound to the term $f(Y)$ and later we are able to bind ***X*** to the term $f(a)$, the effect so produced is that of binding the variable ***Y*** to the constant ***a***. This does not contradict the first point, given that the binding of ***X*** to the term $f(Y)$ is not destroyed, but the value of $f(Y)$ is specified through the binding of ***Y***. Thus, we cannot only modify the value of a variable by modifying the term to which it is bound, but we can also modify this term by providing another binding for that variable. This second binding must be consistent with the first, that is if ***X*** is bound to the term $f(Y)$, we cannot try to bind ***X*** to a term of the form $g(Z)$.

The connection between variables and terms is made in terms of the concept of substitution, which allows the substitution of a variable by a term. A substitution, usually denoted by the Greek letters ϑ, σ, ρ, $\cdots$, can be defined as follows.

A substitution is a function from variables to terms such that the number of variables which are not mapped to themselves is finite. A substitution ϑ is denoted by the notation:

$$\vartheta = \{X_1/t_1, \cdots, X_n/t_n\}$$

where $X_1, \cdots, X_n$ are different variables $t_1, \cdots, t_n$ are terms and where we assume that t_i is different from X_i, for $i = 1, \cdots, n$.

In the preceding definition, a pair X_i/t_i is said to be a ***binding***. In the case in which all the $t_1, \cdots, t_n$ are ground terms, then ϑ is said to be a ground substitution. For the previously defined ϑ, we define the domain, codomain and variables of a substitution as follows:

$$\text{Domain}(\vartheta) = \{X_1, \cdots, X_n\},$$

$$\text{Codomain}(\vartheta) = \{Y \mid Y \text{ a variable in } t_i, \text{ for some } 1 \quad i \quad n\}.$$

A substitution can be applied to a term to modify the value of the variables present in the domain of the substitution. More precisely, if we consider an expression ***E*** (a term, a literal, a conjunction of atoms, etc), the result of the application of $\vartheta = \{X_1/t_1, \cdots, X_n/t_n\}$ to ***E***, denoted by $E\vartheta$, is obtained by simultaneously replacing every occurrence of X_i in E by the corresponding t_i, for all $1 \quad i \quad n$. So, if we apply the substitution $\vartheta = \{X/a, Y/f(W)\}$ to the term $g(X, W, Y)$, we obtain the term $g(X, W, Y)\vartheta$, that is $g(a, W, f(W))$.

The composition $\vartheta\sigma$ of two substitutions $\vartheta = \{X_1/t_1, \cdots, X_n/t_n\}$ and $\sigma = \{Y_1/s_1, \cdots, Y_m/s_m\}$ is defined as the substitution obtained by removing from the set

$$\{X_1/t_1\sigma, \cdots, X_n/t_n\sigma, Y_1/s_1, \cdots, Y_m/s_m\}$$

the pairs $X_i/t_i\sigma$ such that X_i is equal to $t_i\sigma$ and the pairs Y_1/s_i such that $Y_i \in \{X_1, \cdots, X_n\}$.

A particular type of substitution is formed from those which simply rename their variables. For example, the substitution $\{X/W, W/X\}$ does nothing more than

change the names of the variables X and W. Substitutions like this are called renamings.

A substitution ρ is a renaming if its inverse substitution ρ^{-1} exists and is such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. Note that the substitution $\{X/Y, W/Y\}$ is not renaming. It not only changes the names of the two variables, but also makes the two variables equal that were previously distinct.

It is also useful to define a preorder over substitutions, where $\vartheta \quad \sigma$ is read as: ϑ is more general than σ. Therefore we define $\vartheta \quad \sigma$ if there exists a substitution γ such that $\vartheta\gamma = \sigma$. Similarly, given two expressions, t and t', we define $t \quad t'$ if and only if there exists a ϑ such that $t\vartheta = t'$. The relation is a preorder and the equivalence induced by it is called the ***variance***.

If ϑ is a substitution that has as domain the set of variables ***V***, and ***W*** is a subset of ***V***, the restriction of ϑ to the variables in ***W*** is the substitution obtained by considering only the bindings for variables in ***W***, that is the substitution defined as:

$\{Y/t | Y \in W \text{ and } Y/t \in \vartheta\}$.

In the logic paradigm, the association of values with variables is implemented through substitutions. The application of a substitution to a term can be seen as the evaluation of the terms, and evaluation that returns another term, and therefore, a partially defined value.

The basic computation mechanism for the logic paradigm is the evaluation of equations of the form $s = t$, where s and t are terms and "=" is a predicate symbol interpreted as syntactic equality over the set of all ground terms.

If we write $X = a$ in a logic program, we mean that the variable X must be bound to the constant a. The substitution $\{X/a\}$ therefore constitutes a solution to this equation since we obtain $a = a$ which is and equation that is clearly satisfied.

Unlike in an imperative language, here we can also write $a = X$ instead of $a = X$ and the meaning does not change. Let's assume that we have a binary function symbol + which expresses the sum of two natural numbers. Now let's consider the equation $3 = 2 + 1$. It does not contain variables so it can be true or false. In logic program this equation cannot be solved because the symbol = is interpreted as syntactic equality over the set of ground terms. The constant 3 is different from the term 2+1.

Similarly, the equation $f(X) = g(Y)$ has no solutions because however the variables X and Y are instantiated, we cannot make the two different function symbols f and g equal. In general, the equation $X = t$ cannot be solved if t contains the variable X and is different from X.

On the other hand, the equation $f(X) = f(g(Y))$ is solvable. One solution is the substitution $\vartheta = \{X/g(Y)\}$ because if applied, it makes the two terms syntactically equal. We can say that the substitution ϑ ***unifies*** the two terms of the equation and it is therefore called a ***unifier***.

Given a set of equations $E = \{s_1 = t_1, \cdots, s_n = t_n\}$, where $s_1, \cdots, s_n$ and $t_1, \cdots, t_n$ are terms, the substitution ϑ is a unifier for E if the sequence $(s_1, \cdots, s_n)\vartheta$ and $(t_1, \cdots, t_n)\vartheta$ are syntactically identical. A unifier of E is said to be the most general unifier if it is more general than any other unifier of E; that is, for every other unifier σ of E, σ is equal to $\vartheta\tau$ for some substitution τ.[123]

4.5. Prolog features and applying theory

Prolog uses clauses and implements resolution via a strictly linear depth-first strategy and a unification algorithm.

Prolog uses notation almost identical to that developed for clauses, except that the implication arrow is replaced by a colon followed by a dash (or minus) “: - “. So, instead of writing

ancestor(X, Y) parent(X, Z), ancestor(Z, Y).

we write

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

Note that the variables ***X*** and ***Y*** are written in uppercase. Prolog distinguishes variables from constants and names of predicates and functions by using uppercase for variables and lowercase for constants and names. It is also possible in most Prolog systems to denote a variable by writing and underscore before the name:

ancestor(_x, _x).

In addition to the comma connective that stands for ***and***, Prolog uses the semicolon for ***or***. However, the semicolon is rarely used in programming, since it is not a standard part of clause logic.
Basic data structures are terms like

parent(X, Z)

successor(successor(0))

Prolog also includes the list as a basic data structure. A Prolog list is written using square brackets, with the list consisting of items ***x***, ***y***, and ***z*** written as

[x, y, z]

Lists may contain terms or variables. It is also possible to specify the head and tail of a list using a vertical bar. For example, `[H|T]` means that `H` is the first item in the list and `T` is the tail of the list. This type of notation can be used to extract the components of a list via pattern matching. Thus, if `[H | T] = [1, 2, 3]`, then `H = 1` and `T = [2, 3]`. It is also possible to write as many terms as one wishes before the bar. For example, `[X, Y | Z] = [1, 2, 3]` gives `X = 1, Y = 2`, and `Z = [3]`. The empty list is denoted by `[]`.

Prolog has a number of standard predicates that are always built in, such as ***not***, =, and the I/O operations ***read***, ***write***, and ***nl*** (new line). One anomaly in Prolog is that the less than or equal to operator is usually written `=<` instead of `<=`.

Prolog compilers exist, but most systems are run as interpreters. A Prolog

program consists of a set of clauses in Prolog syntax, which is usually entered from a file and stored in a dynamically maintained database of clauses. Once a set of clauses has been entered into the database, goals can be entered either from a file or from the keyboard to begin execution. Thus, once a Prolog system has begun to execute, it will provide the user with a prompt for a query, such as

```
?-_
```

Let’s say the following clauses have been entered into the database:

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

ancestor(X, X).

parent(amy, bob).
```

The following queries would cause the indicated response:

```
?- ancestor(amy, bob).

yes.


?- ancestor(bob, amy).

no.
```

?-ancestor(X, bob).

X = amy ->_

In the last query, there are two answers. Most Prolog systems will find one answer and then wait for a user prompt before printing more answers. If the user supplies a semicolon at the underscore, then Prolog continues to find more answers.

?- ancestor(X, bob).

X = amy ->;

X = bob

?-_

Prolog has built-in arithmetic operations and an arithmetic evaluator. Arithmetic terms can be written either in the usual infix notation or as terms in prefix notation: `3 + 4` and `+(3, 4)` mean the same thing. However, Prolog cannot tell when to consider an arithmetic term as a term itself, that is, strictly as data, or when to evaluate it.

?- write(3 + 5).

3 + 5

To force the evaluation of an arithmetic term, a new operation is required: the built-in predicate ***is***. Thus, to get Prolog to evaluate $3 + 5$, we need to write:

```
?- X is 3 + 5, write(X).

X = 8
```

We already saw that two arithmetic terms may not be equal as terms, even though they have the same value:

```
?- 3 + 4 = 4 + 3.

no
```

To get equality of values, we must force evaluation using ***is***, for example, by writing a predicate:

```
valequal(Term1, Term2) :- X is Term1, Y is Term2, X = Y.
```

The result is now:

```
?- valequal(3 + 4, 4 + 3).

yes
```

Here is an example of Euclid's algorithm for the greatest common divisor, first in clauses:

gcd(u, 0, u).

gcd(u, v, w) not zero(v), gcd(v, u mod v, w).

In Prolog this translates to:

gcd(U, 0, U).

gcd(U, V, W) :- not(V = 0), R is U mod V, gcd(V, R, W).

Unification is the process by which variables are instantiated, or allocated memory nd assigned values, so that patterns match during resolution. It is also the process of making two terms the same in some sense. The basic expression whose semantics is determined by unification is equality: In Prolog, the goal `s = t` attempts to unify the terms `s` and `t`. It succeeds if unification succeeds and fails otherwise. Unification in Prolog can be studied by experimenting with the effect of equality:

?- me = me.

yes

?- me = you.

no

?- me = X.

X = me

?- f(a, X) = f(Y, b).

X = b

Y = a

?- f(X) = g(X).

no

?- f(X) = f(a, b).

no

?- f(a, g(X)) = f(Y, b).

no

?- f(a, g(X)) = f(Y, g(b)).

X = b

Y = a

From previous lines, we can formulate the following unification algorithm for Prolog:

1) A constant unifies only with itself: me = me succeeds but me = you fails.

2) A variable that is not instantiated unifies with anything and becomes instantiated to that thing.

3) A structured term unifies with another term only if it has the same function name and the same number of arguments, and the arguments can be unified recursively. Thus, `f(a, X)` unifies with `f(Y, b)` by instantiating `X` to `b` and `Y` to `a`.

A variation on point 2 is when two variables that are not instantiated are unified:

?- X = Y.

X = _23

Y = _23

The number printed on the right side (23, in this case) will differ from system to system and indicates an internal memory location set aside for that variable. Thus, unification causes uninstantiated variables to share memory, that is, to become aliases of each other.[124]

Prolog applies resolution in a strictly linear fashion, replacing goals left to right and considering clauses in the database in top-to-bottom order. Subgoals are also considered immediately once they are set up. This search strategy results in a depth-first search on a tree of possible choices.

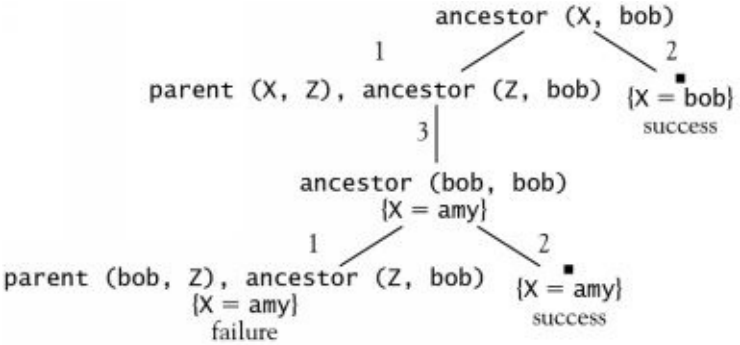
Let’s consider the clauses used before:

(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

(2) ancestor(X, X).

(3) parent(amy, bob).

Given the goal “ancestor(X, bob)”, Prolog’s search strategy is left to right and depth first on the tree of subgoals shown in the following figure. Edges in the figure are labelled by the number of the clause above used by Prolog for resolution, and instantiations of variables are written in curly brackets.



Leaf nodes in this tree occur either when no match is found for the leftmost clause or when all clauses have been eliminated, thus indicating success. Whenever failure occurs, or the user indicates a continued search with a semicolon, Prolog backtracks up the tree to find further path to a leaf, releasing instantiations of variables as it does so.

Depth-first search with backtracking can be used to perform loops and repetitive searches. We must force backtracking even when a solution is found and that is done with the built-in predicate ***fail***. As an example, we can get Prolog to print all solutions to a goal such as ***append***, with no need to give semicolons to the system. Define the predicate:

```
printpieces(L) :- append(X, Y, L), write(X), write(Y), nl, fail.
```

With this, we get the following behaviour:

```
?- printpieces([1, 2]).

[][1, 2]

[1][2]

[1, 2][]

no
```

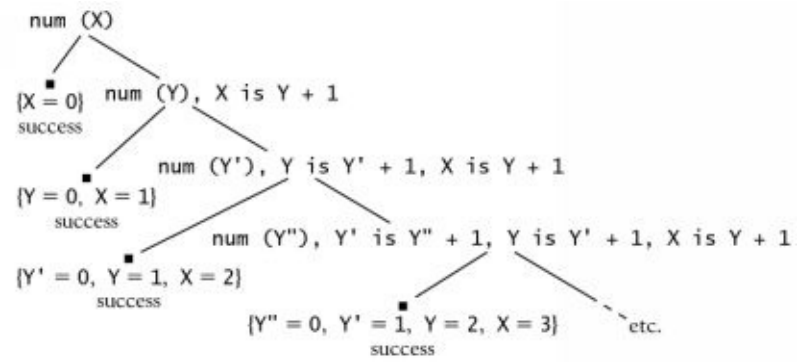
Backtracking on failure forces Prolog to write all solutions at once.

We can also use this technique to get repetitive computations. For example, the following clauses generate all integers 0 as solutions to the goal ***num(X)***:

(1) num(0).

(2) num(X) :- num(Y), X is Y + 1.

The search tree is displayed in the figure below. This tree has an infinite branch to the right, while different uses of ***Y*** from clause (2) are indicated by adding quotes to the variable name.



There is a way to stop the search from continuing through the whole tree so we can prevent infinite loops. There is an operator for this, the ***cut***, usually written as an exclamation point. The cut freezes a choice when it is encountered. If a cut is reached on backtracking, the search of the subtrees of the parent node of the node containing the cut stops, and the search continues with the grandparent node. In effect, the cut prunes the search tree of all other siblings to the right of the node containing the cut.

Let's rewrite the used example using the cut:

(1) ancestor(X, Y) :- parent(X, Z), !, ancestor(Z, Y).

(2) ancestor(X, X).

(3) parent(amy, bob).

The only solution that will be found is ***x = amy***, since the branch containing ***X = bob*** will be pruned from the search.

The cut can also be used to imitate ***if-else*** constructs in imperative and functional languages. To write a clause such as:

D = if ***A*** then ***B*** else ***C***

we write the following in Prolog:

D :- A, !, B.

D :- C.

Almost the same could have been achieved without the cut:

D :- A, B.

D :- not(A), C.

but this is different since ***A*** is executed twice.[125]

5. Other known examples of declarative languages

5.1. SQL

SQL (Structured Query Language) is the standard language used to communicate with a relational database. It can be used to retrieve data from a database using a query but it can also be used to create and destroy databases, as well as modify their structure. In addition, it is possible to add, modify, and delete data with SQL. Even with all that capability, SQL is still considered only a data sublanguage, meaning that it does not have all the features of general-purpose programming languages such as C# or Java.

SQL is specifically designed for dealing with relational databases, and thus does not include a number of features that are needed for creating useful application programs. As a result, to create a complete application that handles queries as well as provides access to a database, you must write the code in one of the general-purpose languages and embed SQL statements within the program whenever it communicates with the database.[126]

The language is subdivided into several language elements, including:

- Clauses
- Expressions
- Predicates
- Queries
- Statements

- Insignificant whitespace

Although SQL is a declarative language to a great extent, it also includes procedural elements.

5.2. Markup languages

Many markup languages such as HTML, MXML, XAML are often declarative. More on markup languages in general can be found on this website: http://en.wikipedia.org/wiki/Markup_language.

XAML (Extensible Application Markup Language) is a declarative XML-based language that is used for initializing structured values and objects. It is used extensively in .NET Framework 3.0 and .NET Framework 4.0 technologies, particularly WPF (Windows Presentation Foundation), Silverlight, Windows Store Apps… In WPF, XAML forms a user interface markup language to define UI elements, data binding, eventing, and other features. Anything that is created or implemented in XAML can be expressed using a more traditional .NET language such as C# or Visual Basic .NET.

HTML is the standard markup language used to create web pages. It is written in the form of HTML elements consisting of tags enclosed in angle brackets and those tags commonly come in pairs like ***<h1>*** and ***</h1>***, although some tags are unpaired. The first tag in a pair is the start tag, and the second tag is the end tag. Web browser read HTML files and compose them into visible or audible web pages without displaying the tags. Instead, tags are used to interpret the content of the page. HTML describes the structure of a website semantically along with cues for presentation, making it a markup language rather than a programming

language.

Bibliography

1. MITCHELL, J. C. (2002) ***Concepts in Programming Languages.*** Cambridge University Press

2. LOUDEN, K. C., LAMBERT, K. A. (2011) ***Programming Languages Principles and Practice.*** 3rd Edition. Cengage Learning

3. SEBESTA, R. W. (2012) ***Concepts of Programming Languages.*** 10th Edition. Addison-Wesley

4. GABBRIELLI, M., MARTINI, S. (2010) ***Programming Languages: Principles and Paradigms.*** Springer

5. TAYLOR, A. G. (2011) ***SQL All-in-One For Dummies***. 2nd Edition. For Dummies

6. LLOYD, J. (1994) ***Practical advantages of declarative programming***

7. UNSW SCHOOL OF COMPUTER SCIENCE AND ENGINEERING (2015) ***DAMP 2009: Workshop on Declarative Aspects of Multicore Programming*** [Online] Available at: http://www.cse.unsw.edu.au/~pls/damp09/ [Accessed: 21st June 2015]

8. HUDAK, P. (1989) ***Conception, Evolution, and Application of Functional Programming Languages*** [Online] Available at: http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf [Accessed: 21st June, 2015]

9. BURSTALL, R. M., DARLINGTON, J. (1977) ***A Transformation System for Developing Recursive Programs*** [Online] Available at: http://www.diku.dk/OLD/undervisning/2003e/235/Burstall-1977-TransSystem.pdf [Accessed: 21st June 2015]

10. JONES, S. P., WADLER, P. (1993) ***Imperative Functional Programming***

Automata-based Programming

1. Overview

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other formal automation. FSM-based programming is similar but it doesn't cover all possible variants as FSM stands for finite state machine and automate-based programming doesn't necessarily employ FSMs.

The following properties are key indicators for automata-based programming:

- The time period of the program's execution is clearly separated down to the steps of the automation. Each of the steps is effectively an execution of a code section, which has a single entry point. Such a section can be a function or other routine, or just a cycle body. Although not necessary, a section can be divided down to subsection to be executed depending on different states.
- Any communication between the steps is only possible via the

> explicitly noted set of variables named the state. Between any two steps, the program cannot have implicit components of its state, such as local variables, return addresses, etc. So, the state of the whole program, taken at any two moments of entering the step of the automation, can only differ in the values of the variables being considered as the state of the automation.

The whole execution of the automata-based code is a cycle of the automation's steps and the programmer's style of thinking about the program in this technique is very similar to that used to solve math-related tasks using Turing machine.

To describe automata we must begin with the definition of automata, their presentation and mathematical basis in order to know how automata is defined.

2. Theoretical background

2.1. Finite state systems

The finite automation is a mathematical model of a system, with discrete inputs and outputs. The system can be in any one of a finite number of internal configurations or states. The state of the system summarizes the information concerning past inputs that is needed to determine the behaviour of the system on subsequent inputs.

The control mechanism of an elevator is a good example of a finite state system. That mechanism does not remember all previous requests for service but only the current floor, the direction of motion, and the collection of not yet satisfied requests for service.

In computer science there are many examples of finite state systems and the theory of finite automata is a useful design tool for these systems. A primary example is a switching circuit, such as the control unit of a computer. A switching circuit is composed of a finite number of gates, each of which can be in one of two conditions, usually denoted as 0 and 1. The state of a switching network with ***n*** gates is thus any one of the 2^n assignments of 0 or 1 to the various gates.[127]

A finite automata (FA) consists of a finite set of states and a set of transitions from state to state that occur on input symbols chosen from an alphabet Σ. For each input symbol there is exactly one transition out of each state (possibly back to the state itself). One state, usually denoted q_0, is the initial state, in which automation starts. Some states are designated as final or accepting states.

A direct graph, called transition diagram, is associated with an FA as follows. The vertices of the graph correspond to the states of the FA. If there is a transition from state q to state p on input a, there is an arc labelled a from state q to state p in the transition diagram. The FA accepts a string x if the sequence of transitions corresponding to the symbols of x leads from the start state to an accepting state.

We formally denote a finite automaton by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, q_0 in Q is the initial state, $F \subseteq Q$ is the set of final states, and δ is the transition function mapping $Q \times \Sigma$ to Q. That is, $\delta(q, a)$ is a state for each state q and input symbol a.[128]

2.2. Regular expressions

The languages accepted by finite automata are easily described by simple

expressions called regular expressions.

Let Σ be a finite set of symbols and let L, L_1, and L_2 be sets of strings from Σ^*. The concatenation of L_1 and L_2, denoted L_1L_2, is the set $\{xy \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$. That is, the strings in L_1L_2 are formed by choosing a string L_1 and following it by a string in L_2, in all possible combinations. Define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \quad 1$. The Kleene closure of L, denoted L^*, is the set

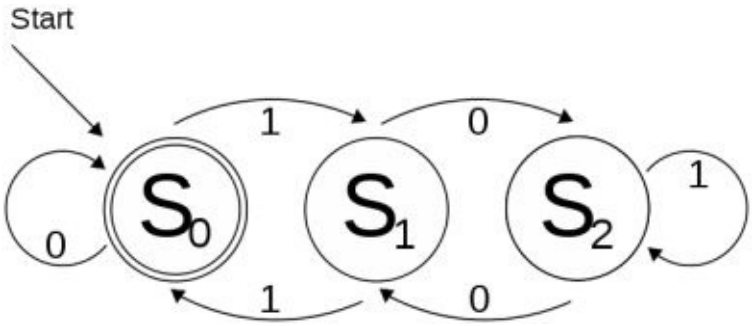
$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1) $\varnothing$ is a regular expression and denotes the empty set.
2) ϵ is a regular expression and denotes the set $\{\epsilon\}$.
3) For each a in Σ, a is a regular expression and denotes the set $\{a\}$.
4) If r and s are regular expressions denoting the languages R and S, respectively, then $(r + s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

If L is accepted by a FA, then L is denoted by a regular expression.[129]

2.3. Example

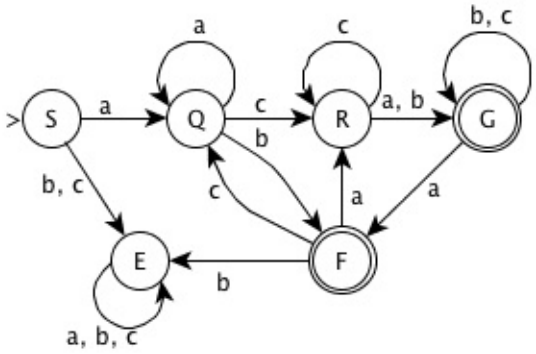


Here is a figure that illustrates a deterministic finite automation using a state diagram:

In the automaton, there are three states: ***S0***, ***S1***, and ***S2*** (graphically denoted by circles). The automaton takes a finite sequence of 0s and 1s as input. For each state, there is a transition arrow leading out to a next state for both 0 and 1.

Upon reading a symbol, a DFA jumps deterministically from a state to another by following the transition arrow. For example, if the automaton is currently in state ***S0*** and current input symbol is 1, then it deterministically jumps to state ***S1***.

The start state is denoted graphically by an arrow coming in from nowhere while final states or accept states are denoted graphically by a double circle.



3. Programming paradigm

3.1. Finite state machines

A finite-state machine (FSM) or finite-state automaton, or simply a state

machine, is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time. The state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition. This is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

The behavior of state machines can be observed in many devices in modern society which perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines which dispense products when the proper combination of coins is deposited, elevators which drop riders off at upper floors before going down, traffic lights which change sequence when cars are waiting, and combination locks which require the input of combination numbers in the proper order.

Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, language parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines have been used to describe neurological systems. In linguistics, they are used to describe simple parts of the grammars of natural languages.

Considered as an abstract model of computation, the finite state machine is weak; it has less computational power than some other models of computation such as the Turing machine. That is, there are tasks which no FSM can do, but some Turing machines can. This is because the FSM has limited memory. The memory is limited by the number of states. Sometimes a potentially-infinite set of possible states is introduced, and such a set can have a complicated structure,

not just an enumeration.[130]

3.2. Code example

Let’s consider a program written in, for example, C that reads a text from standard input stream, line by line, and prints the first word of each line. First, we need to read and skip the leading spaces, if any, then read characters of the first word and print them until the word ends. The remaining characters until the end-of-line character are read and then skipped. Here is that example in C:

```
#include <stdio.h>

int main(void)

{

int c;

do {

c = getchar();

while(c == ' ')

c = getchar();

while(c != EOF && c != ' ' && c != '\n') {

putchar(c);

c = getchar();

}
```

```
putchar('\n');

while(c != EOF && c != '\n')

c = getchar();

} while(c != EOF);

return 0;

}
```

The same problem can be solved by thinking in terms of finite state machines. Parsing has three stages: skipping the leading spaces, printing the word and skipping the trailing character. These states can be called ***before, inside***, and ***after***, and here is that version of the program:

```
#include <stdio.h>

int main(void)

{

enum states {

before, inside, after

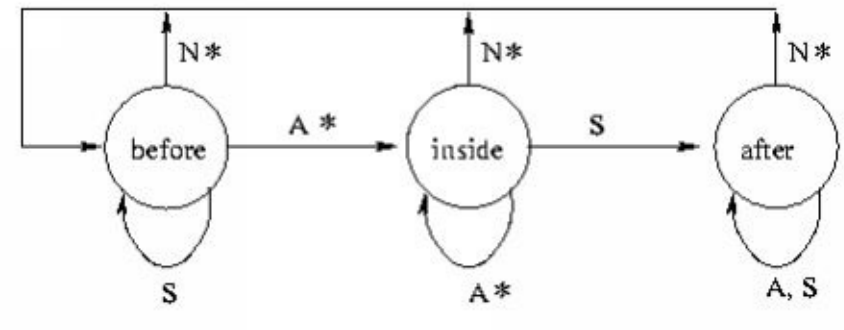
} state;

int c;

state = before;
```

```
while((c = getchar()) != EOF) {
switch(state) {
case before:
if(c == '\n') {
putchar('\n');
} else
if(c != ' ') {
putchar(c);
state = inside;
}
break;
case inside:
switch(c) {
case ' ':  state = after; break;
case '\n':
putchar('\n');
state = before;
break;
```

```
default:   putchar(c);
}
break;
case after:
if(c == '\n') {
putchar('\n');
state = before;
}
}
}
return 0;
}
```



The second code is longer but it has only one reading instruction, that is, call to the ***getchar()*** function. Besides that, there's only one loop instead of the previous four. The

body of the while loop is the automaton step, and the loop itself is the cycle of the automaton's work.

The program implements the work of a finite state machine shown on the picture above. The ***N*** denotes the end of line character, the ***S*** denotes spaces, and the ***A*** stands for all the other characters. Some state switches are accompanied with printing the character (arrows marked with asterisks).

It is not necessary to divide the code down to separate handlers for each unique state. Furthermore, the very notion of the state can be composed of several variables' values in some cases. This makes it impossible to handle each possible state explicitly. It is possible to reduce the length of the previous code by noticing that the actions taken in response to the end of line character are the same for all the possible states.

```
#include <stdio.h>

int main(void)

{

enum states {

before, inside, after

} state;

int c;

state = before;

while((c = getchar()) != EOF) {
```

```
if(c == '\n') {
putchar('\n');
state = before;
} else
switch(state) {
case before:
if(c != ' ') {
putchar(c);
state = inside;
}
break;
case inside:
if(c == ' ') {
state = after;
} else {
putchar(c);
}
break;
```

```
    case after:
    break;
    }
    }
    return 0;
    }
```

3.3. Transition table

A finite automaton can be defined by an explicit state transition table. Generally speaking, an automata-based program code can naturally reflect this approach. In the program below there's an array named ***the_table***, which defines the table. The rows of the table stand for three states, while columns reflect the input characters (first for spaces, second for the end of line character, and the last is for all the other characters).

For every possible combination, the table contains the new state number and the flag, which determines whether the automaton must print the symbol. In a real life task, this could be more complicated; e.g., the table could contain pointers to functions to be called on every possible combination of conditions.

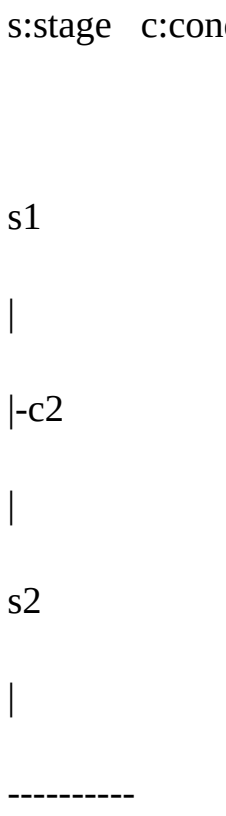
```
    #include <stdio.h>
    enum states { before = 0, inside = 1, after = 2 };
```

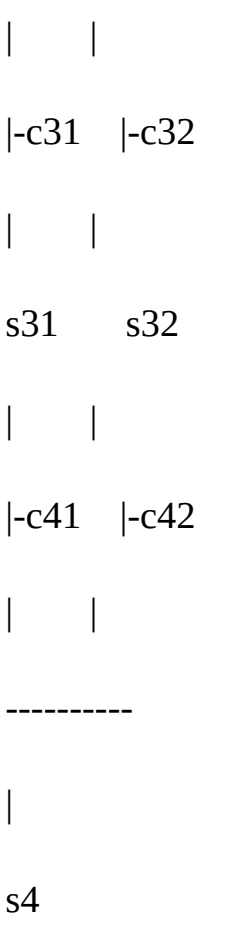
```
struct branch {
unsigned char new_state:2;
unsigned char should_putchar:1;
};
struct branch the_table[3][3] = {
/* ' '        '\n'       others */
/* before */ { {before,0}, {before,1}, {inside,1} },
/* inside */ { {after, 0}, {before,1}, {inside,1} },
/* after  */ { {after, 0}, {before,1}, {after, 0} }
};
void step(enum states *state, int c)
{
int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
struct branch *b = & the_table[*state][idx2];
*state = (enum states)(b->new_state);
if(b->should_putchar) putchar(c);
}
```

3.4. Automation and aspects of other paradigms

In the field of automation, stepping from step to step depends on input data coming from the machine itself. This is represented in the program by reading characters from a text. In reality, those data inform about position, speed, temperature, etc. of critical elements of a machine.

Like in GUI programming, changes in the machine state can thus be considered as events causing the passage from a state to another, until the final one is reached. The combination of possible states can generate a wide variety of events, thus defining a more complex production cycle. As a consequence, cycles are usually far to be simple linear sequences. There are commonly parallel branches running together and alternatives selected according to different events, schematically represented below:





If the implementation language supports object-oriented programming, a simple refactoring is to encapsulate the automaton into an object, thus hiding its implementation details. For example, an object-oriented version in C++ of the same program is below.

```
#include <stdio.h>
class StateMachine {
enum states { before = 0, inside = 1, after = 2 } state;
struct branch {
```

```
enum states new_state:2;
int should_putchar:1;
};
static struct branch the_table[3][3];
public:
StateMachine() : state(before) {}
void FeedChar(int c) {
int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
struct branch *b = & the_table[state][idx2];
state = b->new_state;
if(b->should_putchar) putchar(c);
}
};
struct StateMachine::branch StateMachine::the_table[3][3] = {
/* ' '          '\n'        others */
/* before */ { {before,0}, {before,1}, {inside,1} },
/* inside */ { {after, 0}, {before,1}, {inside,1} },
/* after  */ { {after, 0}, {before,1}, {after, 0} }
```

```
};

int main(void)

{

int c;

StateMachine machine;

while((c = getchar()) != EOF)

machine.FeedChar(c);

return 0;

}
```

3.5. Use of automata

Automata-based programming is widely used in lexical and syntactic analyses. Besides that, thinking in terms of automata – breaking the execution process down to automaton steps and passing information from step to step through the explicit state – is necessary for event-driven programming as the only alternative to using parallel processes or threads.

The notions of states and state machines are often used in the field of formal specification. For instance, UML based software architecture development uses state diagrams to specify the behaviour of the program. Also, various communication protocols are often specified using the explicit notion of state.

Thinking in terms of automata (steps and states) can also be used to describe semantics of some programming languages. For example, the execution of a program written in the Refal language is described as a sequence of steps of a so-called abstract Refal machine. The state of the machine is a view (an arbitrary Refal expression without variables).

Refal (Recursive functions algorithmic language) is a functional programming language oriented toward symbol manipulation, including string processing, translation, and artificial intelligence. It is one of the oldest members of this family, first conceived in 1966 as a theoretical tool with the first implementation appearing in 1968. Refal combines mathematical simplicity with practicality for writing large and sophisticated programs.

Continuations in the Scheme language require thinking in terms of steps and states, although Scheme itself is in no way automata-related (it is recursive). To make it possible the call/cc feature to work, implementation needs to be able to catch a whole state of the executing program, which is only possible when there's no implicit part in the state. Such a caught state is the very thing called continuation, and it can be considered as the state of a relatively complicated automaton. The step of the automaton is deducing the next continuation from the previous one, and the execution process is the cycle of such steps.[131][132]

Bibliography

1. HOPCROFT, J.E., ULLMAN, J.D. (1979) ***Introduction to automata theory, languages, and computation.*** Addison-Wesley

2. BELZER, J., HOLZMAN, A. G., KENT, A (1975) ***Encyclopedia of***

***Computer Science and Technology, Vol. 25*.** CRC Press

3. AHO, A. V., ULLMAN, J. D. (1973) ***The theory of parsing, translation and compiling 1***. Prentice-Hall

4. IETF TOOLS (2015) ***Transmission Control Protocol.*** [Online] Available at: https://tools.ietf.org/html/rfc793 [Accessed: 21st June 2015]

Coding languages vs. markup languages

P rogramming languages can be classified by many different standards – a language can be associated with one or many programming paradigms because of its elements, for example. Another example is classifying languages based on the field of programming they are used in, like network programming, database programming, website development, etc.

This document will identify two categories of languages based on their purpose in Model-View-Controller (MVC) software design pattern. This design pattern will be explained first which will be followed by implementation examples illustrating distinctions between different languages.

1. Model-View-Controller design pattern

First of all, it is important to note that MVC is one of many design patterns and to understand why design patterns are needed. The answer lies in the definition of good software which states that good software is, among other things, easy to change.

Software changes over time for many reasons:

- A new feature needs to be added.
- A bug needs to be fixed.
- Software needs to be optimized.
- The software design needs to be improved.

Bad software is difficult to change and the indicators of bad software include:

- Rigidity – Software requires a cascade of changes when a change is made in one place.
- Fragility – Software breaks in multiple places when a change is made.
- Needless complexity – Software is overdesigned to handle any possible change.
- Needless repetition – Software contains duplicate code.
- Opacity – Software is difficult to understand.

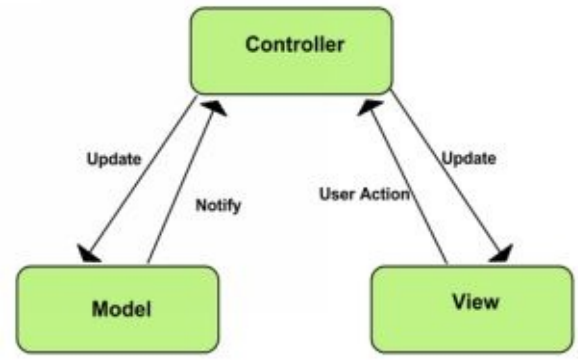
Software can be designed from the beginning to survive changes. The best strategy is to make the components of the application loosely coupled. In a loosely coupled application, you can make a change to one component of an application without making changes to other parts. There are several principles

that enable reducing dependencies between different parts of an application.

Software design patterns represent strategies for applying software design principles. In other words, a software design principle is a good idea while a software design pattern is the tool used to implement that idea. MVC is one of those patterns.[133]

As the name itself suggests, MVC consists of three components that communicate – Model, View, and Controller. Model is where the application's data objects are stored. It represents knowledge as a structure of objects. The model doesn't know anything about views and controllers but it can contain logic to update controller if its data changes.

View is a visual representation of its model so it is attached to it in order to get the necessary data for the presentation. View can also update the model by sending appropriate messages. Users interact with an application through its View.



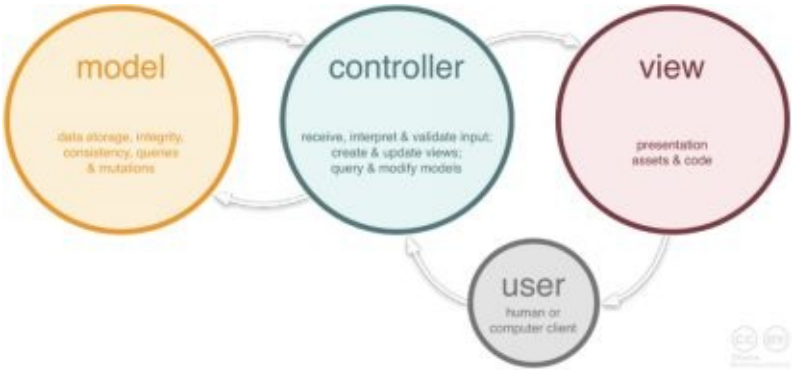
It is important to note that both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independently of the visual presentation. The separation of view and controller is secondary in many rich-client applications, and, in fact, many user interface frameworks implement the roles as one object. In Web applications, on

the other hand, the separation between view (the browser) and controller (the server-side components handling the HTTP request) is very well defined.

Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic. Unfortunately, the popularity of the pattern has resulted in a number of faulty descriptions. In particular, the term "controller" has been used to mean different things in different contexts. Fortunately, the advent of Web applications has helped resolve some of the ambiguity because the separation between the view and the controller is so apparent.

Because they can be developed separately, there are less problems once an application reaches enterprise level since the source code is less messy and is easier to debug. Having three layers of the application separated means easier control, development, debugging and feature adding to any of the layers.

Another good feature of the MVC framework is that it hides the data access layer from the users, that is, this layer or the data itself is never actually accessed directly by the user through the interface. This way, the user has to perform actions and developers can create groups or roles of users that are allowed to access the data, such as Admins, Guests etc.



The image above clarifies that the user is unable to connect to the data storage directly. Instead, the user has to interact with the top-level layers to access the data. For example, using the view, the user interacts using buttons which then call the controller in order to perform an action on the data (if required). This makes the data source secure from potential users.

Layer independency also allows programmers to find a problem if one occurs since source codes of these layers are in separate packages and do not interfere. Common examples of problems throughout the layers would be:

- Problem is in the data access layer. For example, null entries are allowed in the database, so users are able to leave the data inconsistent and cause a problem of redundancy. This means the database has to be redesigned so it does not accept null values.

- The view is not showing the data correctly. Such are scenarios where binding of the data is not accurate, or the format is incorrect (i.e. the format of the date and time values). Instead of editing back-end or data layer code, it is enough to simply edit the view section of the

application.

- The logic is not correctly defined. Sometimes the data is correct, the view is correctly bound to the resources, but the logic of the application is incorrect. This type of problem is hard to figure out because the developer has to find the part in the code that is causing problems by debugging it using the IDE and putting a few breakpoints at the locations.

2. Coding languages

Coding (programming) languages are used by programmers to write instructions that computer understands in order to execute operations that users tell it to. A coding language is a formal constructed language designed to communicate instructions to a machine, particularly, a computer. Coding languages can be used to create programs to control the behavior of a machine or to implement algorithms.

The most basic (called low-level) computer language is the machine language that uses binary ('1' and '0') code which a computer can execute very fast without using an interpreter program. However, it is tedious and complex. High-level languages (such as Basic, C, Java) are much simpler (more 'English-like') to use but need to use another program – a compiler or an interpreter – to convert the high-level code into the machine code, and are therefore slower.

There are many programming languages and new ones are continuously developed. Description of a programming language is usually split into two components – syntax (form) and semantics (meaning).

In computer science, the syntax of a computer language is a set of rules that defines combinations of symbols considered to be a correctly structured document or fragment in that language. This applies both to programming languages, where the document represents source code, and markup languages, where the document represents data.

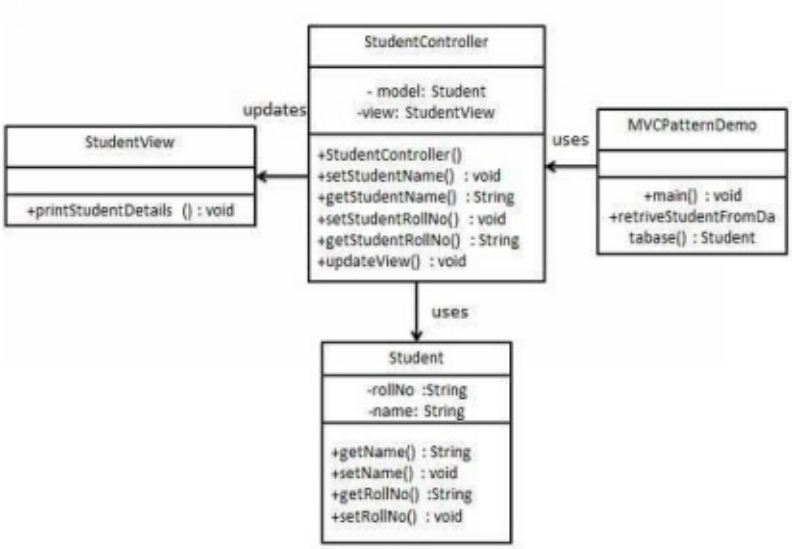
The syntax of a language defines its surface form. Text-based computer languages are based on sequences of characters, while visual programming languages are based on the spatial layout and connections between symbols (which may be textual or graphical). Documents that are syntactically invalid are said to have a syntax error.

Syntax – the form – is contrasted with semantics – the meaning. In processing computer languages, semantic processing generally comes after syntactic processing, but in some cases, semantic processing is necessary for complete syntactic analysis, and these are done together or concurrently. In a compiler, the syntactic analysis comprises the frontend, while semantic analysis comprises the backend (and middle end, if this phase is distinguished).

In programming language theory, semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages. It evaluates the meaning of syntactically legal strings defined by a specific programming language, showing the computation involved. In such a case where the syntactically illegal strings are evaluated, the result would be non-computation. Semantics describe steps a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program is executed on a certain platform, hence creating a model of computation.

In order to illustrate using coding languages to implement MVC architecture, we are going to use Java coding language in the following examples. Java is an object-oriented language similar to C++, but simplified to eliminate language features that cause common programming errors. Java source code files (files with a .java extension) are compiled into a format called bytecode (files with a .class extension), which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines (VMs), exist for most operating systems. As of 2015, Java is one of the most popular programming languages in use particularly for client-server web applications, with a reported 9 million developers.

We are going to create a ***Student*** object acting as a model. ***StudentView*** will be a view class which can print student details on console and ***StudentController*** is the controller class responsible for storing data in the ***Student*** object and updating ***StudentView*** accordingly. ***MVCPatternDemo***, our demo class, will use ***StudentController*** to demonstrate the use of MVC pattern.



2.1. Model

This model will provide the data to our application. Primarily, model is a class that defines the structure of the application’s data, like it is defined in a database.

However, classes like that can also have methods implementing operations that can be done on that data. In the model, the structure has to be defined while operations are not mandatory or even necessary in case of simple applications.

The model contains, provides and updates data. Even the controller makes a call to the model requesting it to update itself, which means that the controller sends the data to the model which then updates the data on the disk.

MVC relies on getting the needed data from a database through the model so the model also needs to be able to connect with a certain database. There are a lot of different databases so there is no one particular implementation of that connection. Since this is irrelevant to the topic at hand, we will assume that all communication is done through the variable ***database*** that is defined in the model. Operations executed will be clear from the method name.

```
public class Student {


private String id;

private String name;


public String getId() {

return id;

}
```

```
public void setId(String id) {

this.id = id;

}


public String getName() {

return name;

}


public void setName(String name) {

this.name = name;

}


public void getForId(String id) {

this.id = id;

name = database.getNameForId(id);

}


}
```

The ***Student*** class contains only the basic get and set methods because of encapsulation, an important concept of object-oriented programming. This means fields containing data are not accessed directly but through methods instead. Therefore, it is important to be able to implement operations as well as define the data structure in the Model classes.

2.2. View

As previously mentioned, the view's role is to display information the controller sends as well as enable users to communicate with the controller by handling different input methods. For now, we will stick to a simple console application that only prints out data about a student with a certain ID. Thus, the view class would be as follows:

```
public class StudentView {

private StudentController controller;

public StudentView() { }

public StudentView(StudentController controller) {

this.controller = controller;
```

```
}

public void setController(StudentController controller) {

this.controller = controller;

}

private void inputId() {

System.out.println(“Input an id: ”);

//in is previously defined as an instance of an input class

//there are many different classes used for input

//so we will not go into detail here as it is irrelevant

controller.getStudentForId(in.readLine());

}

public void printStudentDetails(String studentName, String studentId){
```

```
    System.out.println("Student: ");
    System.out.println("Name: " + studentName);
    System.out.println("ID : " + studentId);

    }

    }
```

2.3. Controller

The controller is a vital section of an application. It contains the back-end code and the logic for connecting the application's views (indirectly the user; because the user is going to use the view to interact with the application) and the models (the data source of our applications). This class controls all of the logic of an application and it needs to have all methods necessary to handle events and requests sent by the remaining two layers. For example, whenever the view needs to do something because an event was triggered or input needs to be processed, the controller must have a defined method that will respond. Also, there should be methods called if data needs to be loaded from the model into the view and vice-versa, as well as to get the model to save data on the disk.

```
    public class StudentController {
```

```
private Student model;

private StudentView view;


public StudentController(Student model, StudentView view){

this.model = model;

this.view = view;

}


public void setStudentName(String name){

model.setName(name);

}


public String getStudentName(){

return model.getName();

}


public void setStudentId(String id){

model.setId(id);
```

```
}

public String getStudentId(){
return model.getId();
}

public void getStudentForId(String id) {
model.getForId(id);
updateView();
}

public void updateView(){
view.printStudentDetails(model.getName(), model.getId());
}

}
```

2.4. Demo

Model, view, and controller are separate layers each with their own functions and none of them is a primary layer that controls the flow of execution. Instead, an instance of each class representing a layer is created in a class that represents the main thread of an application.

In this example, the class demonstrating the connection and communication of classes previously described is ***MVCPatternDemo***. Here, a value of an id is input from the console and as a result, the output is the name and the id of the student with the chosen id.

When users are given freedom with value input, there are chances that the value will be of improper format or it will not exist in the database. To make sure the application doesn't crash if such a mistake occurs, the following class as well as previous ones should be expanded with try/catch blocks. The demo class can also be edited so that it asks for a user input several times instead of just once.

```
public class MVCPatternDemo {



public static void main(String[] args) {



private Student model = new Student();

private StudentView view = new StudentView();

private StudentController controller = new StudentController(model, view);

view.setController(controller);
```

```
    view.inputId();


    }


    }
```

Let’s say the student with the id 25 is Robert Jones. After starting the program output will look like this:

```
    Input an id:

    25

    Student:

    Name: Robert Jones

    ID: 25
```

3. Markup languages

A markup language is a set of tags and/or a set of rules for creating tags that can be embedded in digital text to provide additional information about the text in order to facilitate automated processing of it, including editing and formatting

for display or printing. Markup languages are fundamental to displaying documents in web browsers, and they are also employed by every word processing program and by nearly every other program that displays text. However, such languages and their tags are typically hidden from the user.

By far the most familiar markup language to most people is HTML (hypertext markup language), which is used to allow documents to be displayed in web browsers. A newer and much more flexible (but also more difficult to learn and use) approach is to use languages based on XML (extensible markup language), which is a standard for creating languages that describe the content of documents rather than how they should be displayed.

A tag is a special string (i.e., sequence of characters). In HTML, XML and related languages, every tag begins with a leftward pointing angular bracket, contains one or more alphanumeric characters, and ends with a rightward pointing angular bracket. These brackets indicate to the browser or other program that renders (i.e., converts to its final form to be viewed by users) that they, along with the enclosed characters, are instructions for the computer rather than ordinary text and that they are not to be visible in the rendered document.

Most tags are designed to be used in pairs (consisting of a start tag and an end tag) and to enclose text within the pair. An example is the pair of HTML tags that is used to indicate bold text:

```
<b>and</b>
```

Thus, for example, the phrase ‘bold text’ is tagged in the source code for this page as

bold text

Other examples of commonly used tag pairs are and to indicate the start and end of a paragraph, and and to indicate that the enclosed text should be rendered in italics.

Every HTML document begins with the tag <html> and ends with the tag </html>. It can be seen that the closing tag in every pair differs from its opening counterpart by the inclusion of a forward slash before the character(s) enclosed within the tag. Some tags are designed to be used individually because they do not enclose any content. An example is
 which stands for break and is used to indicate the start of a new line of text. Another HTML example is <hr/> which stands for horizontal rule and is used to create a horizontal line.

In XML-based markup languages and in modern versions of HTML it is required that even tags that are used singularly be closed, and this is accomplished by the space and forward slash after all other characters within the brackets.

The tag pair <h2> and </h2> is used in HTML to tell the browser that the enclosed text is a headline and should be rendered in the second to largest type size. In contrast to this rigid information about how to display the headline, XML would employ a descriptive tag pair such as <section head> and </section head> so that the details of the type size, font and style can be easily modified by another program according to the particular application and desires of the author or user.

A major advantage of using markup languages that can describe content is that it becomes practical to automatically manipulate the content. For example, a tag pair such as <price> and </price> could be created so that every instance of a price in a document could easily be reformulated to be written in some special typeface and/or in plain, bold or italic style, could be converted to another currency (e.g., from dollars to euro), could be increased or decreased by a certain percentage or could have sales tax added in (for all prices or only prices over a specified minimum), etc.[134]

We will make example for student model using markup language, HTML in this case. As we said, the advantage of markup languages is that when defining a data set, it is enough to write the tags that are identical in appearance, the only difference is their contents. We had an example class ***Student*** typed in Java and it contained two fields – ID and name – as well as a couple of methods to access and modify these fields. Using HTML, we will create a static page that displays student data stored in a table.

The following code is a simple display table consisting of columns ***ID*** and ***Name***. For example, there are two students, Robert and John and we will only use these as constants instead of including code for data retrieval from a database. If this was a functional web site, there would be a menu, search option, etc. besides a table and its data. So, this would be the simplest way to implement view of the MVC.

```
<!DOCTYPE html>

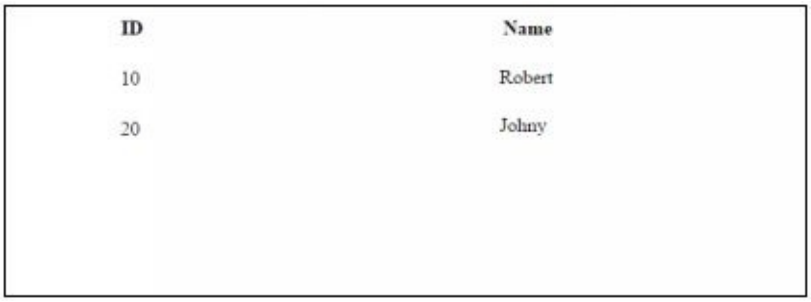
<html>

<head>
```

```
<table style="width:100%">
<tr>
<th>ID</th>
<th>Name</th>
</tr>
<tr>
<td>10</td>
<td>Robert</td>

</tr>
<tr>
<td>20</td>
<td>Johny</td>
</tr>
</table>
</body>
</html>
```

The result page is:



ID	Name
10	Robert
20	Johny

As mentioned, the table provides static information, that is, it consists of predefined fields that do not change. If we want to fully implement the MVC model, we use JavaScript for the implementation of the model and controller components. JavaScript is most commonly used as a client side scripting language. This means that JavaScript code is written into an HTML page. When a user requests an HTML page with JavaScript in it, the script is sent to the browser and it's up to the browser to do something with it.

The main difference is that Java can stand on its own while JavaScript must (primarily) be placed inside an HTML document to function. Java is a much larger and more complicated language that creates "standalone" applications.

JavaScript is a text fed into a browser that can interpret it and then it is enacted by the browser – although today's web apps are starting to blur the line between traditional desktop applications and those created using traditional web technologies: JavaScript, HTML and CSS.

Another major difference is how the language is presented to the end user. Java must be compiled into what is known as a "machine language" before it can be run on the Web. Basically what happens is after the programmer writes the Java program and checks it for errors, he or she hands the text over to another computer program that changes the text code into a smaller language. That smaller language is formatted so that it is seen by the computer as a set program with definite beginning and ending points. Nothing can be added to it and nothing can be subtracted without destroying the program.

JavaScript is text-based. You write it to an HTML document and it is run through a browser. You can alter it after it runs and run it again and again. Once the Java is compiled it cannot be changed. That is, every alteration requires the code to be recompiled.[135]

So, JavaScript enables interactions, or in this case, communication between MVC layers. For a better design of the view itself, CSS is used. Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language. Along with HTML and JavaScript, CSS is a cornerstone technology used by most websites to create visually engaging webpages, user interfaces for web applications, and user interfaces for many mobile applications.

CSS is designed primarily to enable the separation of document content from document presentation, including aspects such as the layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and

control in the specification of presentation characteristics, enable multiple HTML pages to share formatting by specifying the relevant CSS in a separate .css file, and reduce complexity and repetition in the structural content, such as semantically insignificant tables that were widely used to format pages before consistent CSS rendering was available in all major browsers.

CSS makes it possible to separate presentation instructions from the HTML content in a separate file or style section of the HTML file. For each matching HTML element, it provides a list of formatting instructions. For example, a CSS rule might specify that "all heading 1 elements should be bold", leaving pure semantic HTML markup that asserts "this text is a level 1 heading" without formatting code such as a <bold> tag indicating how such text should be displayed. This separation of formatting and content makes it possible to present the same markup page in different styles for different rendering methods, such as on-screen, in print, by voice (when read out by a speech-based browser or screen reader). It can also be used to display the web page differently depending on the screen size or device on which it is being viewed.

Although the author of a web page typically links to a CSS file within the markup file, readers can specify a different style sheet, such as a CSS file stored on their own computer, to override the one the author has specified. If the author or the reader did not link the document to a style sheet, the default style of the browser will be applied. Another advantage of CSS is that aesthetic changes to the graphic design of a document (or hundreds of documents) can be applied quickly and easily, by editing a few lines in one file, rather than by a laborious (and thus expensive) process of crawling over every document line by line, changing markup.

The CSS specification describes a priority scheme to determine which style rules apply if more than one rule matches against a particular element. In this so-

called cascade, priorities are calculated and assigned to rules, so that the results are predictable. CSS has a simple syntax and uses a number of English keywords to specify the names of various style properties. A style sheet consists of a list of rules. Each rule or rule-set consists of one or more selectors, and a declaration block.

Before CSS, nearly all of the presentational attributes of HTML documents were contained within the HTML markup; all font colours, background styles, element alignments, borders and sizes had to be explicitly described, often repeatedly, within the HTML. CSS allows authors to move much of that information to another file, the style sheet, resulting in considerably simpler HTML.

For example, headings (h1 elements), sub-headings (h2), sub-sub-headings (h3), etc., are defined structurally using HTML. In print and on the screen, choice of font, size, colour and emphasis for these elements is presentational.

Before CSS, document authors who wanted to assign such typographic characteristics to, say, all h2 headings had to repeat HTML presentational markup for each occurrence of that heading type. This made documents more complex, larger, and more error-prone and difficult to maintain. CSS allows the separation of presentation from structure. CSS can define colour, font, text alignment, size, borders, spacing, layout and many other typographic characteristics, and can do so independently for on-screen and printed views. CSS also defines non-visual styles such as the speed and emphasis with which text is read out by aural text readers.

For example, under pre-CSS HTML, a heading element defined with red text would be written as:

```
<h1><font color="red">Chapter 1</font></h1>
```

Using CSS, the same element can be coded using style properties instead of HTML presentational attributes:

```
<h1 style="color:red">Chapter 1</h1>
```

Using web pages has become a part of our everyday lives because communication via the Internet is nowadays dominant. Consequently, certain technologies that enable the production of sites far easier were developed, which also means easier modelling of MVC. These technologies (tools) are quite powerful and versatile. Some of them are Angular and Bootstrap.

Bootstrap is an open-source JavaScript framework developed by the team at Twitter. It is a combination of HTML, CSS, and JavaScript code designed to help build user interface components. Bootstrap was also programmed to support both HTML5 and CSS3.

It is a free and open-source collection of tools for creating websites and web applications. It contains HTML- and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It aims to ease the development of dynamic websites and web applications. Bootstrap is a front end framework, that is, an interface for the user, unlike the server-side code which resides on the "back end" or server.

AngularJS (commonly referred to as "Angular") is an open-source web

application framework maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. The AngularJS library works by first reading the HTML page, which has embedded into it additional custom tag attributes. Angular interprets those attributes as directives to bind input or output parts of the page to a model that is represented by standard JavaScript variables.

AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's data binding and dependency injection eliminate much of the code you would otherwise have to write. And it all happens within the browser, making it an ideal partner with any server technology.

Angular is what HTML would have been, had it been designed for applications. HTML is a great declarative language for static documents. It does not contain much in the way of creating applications, and as a result building web applications is an exercise in what do I have to do to trick the browser into doing what I want?

The impedance mismatch between dynamic applications and static documents is often solved with:

- a library - a collection of functions which are useful when writing web apps. Your code is in charge and it calls into the library when it sees fit. E.g., jQuery.
- frameworks - a particular implementation of a web application, where your code fills in the details. The framework is in charge and it calls

into your code when it needs something app specific. E.g., durandal, ember, etc.

Angular takes another approach. It attempts to minimize the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs.[136]

4. Conclusion – language differences and usage in MVC implementation

To conclude, MVC has many advantages which include easier debugging, code changing, feature adding, etc. Depending on the type of an application, MVC can be implemented using coding or markup languages. Coding languages, such as Java, C#, Scala, are used for standalone desktop or mobile applications while markup languages are often used for web development and web applications. Besides the domain in which these languages are used, they differ in their syntax, as can be seen from previous chapters.

It is possible to implement an entire application with an MVC architecture using only coding or markup languages because coding languages often offer classes for developing graphical interfaces and handling events while markup languages of different domains are compatible. A typical example is HTML and CSS – used for appearance – and JavaScript – used for event handling.

Because of numerous different technologies available, it is possible to use both markup and coding languages within the same project. Since we used Java in examples, one technology that would allow Java and HTML to be used together is JSF – JavaServer Faces.

Being able to use different types of languages together allows us to use the best option for every part of an application. MVC implementation includes deciding on the way the model, view and controller are implemented since they are separate layers. Model needs to represent data and have a defined data structure which can be done with classes, for example, a concept of coding languages. Controller needs to be able to execute operations and receive and handle, as well as send requests from and to the model and view. With coding languages, this would mean having an instance of remaining two layers and method implementation which is also easily implemented. As far as view goes, it may be better to use markup languages because of how domain specific they are, but as mentioned, it all depends on the application type.

When implementing MVC, it is best to understand advantages of both coding and markup languages in order to choose the best option for implementing the model, view, and controller as well as the additional technology used to enable layer communication.

Bibliography

1. WALTHER, S. (2009) ***ASP.NET MVC Framework Unleashed***. Sams Publishing

2. HTML GOODIES (2015) ***Java vs. JavaScript*** [Online] Available at: http://www.htmlgoodies.com/beyond/javascript/article.php/3470971/Java vs-JavaScript.htm [Accessed: 11th October 2015]

3. LINFO (2006) ***Markup Language Definition*** [Online] Available at: http://www.linfo.org/markup_language.html [Accessed: 11th October 2015]

4. ANGULARJS DOCUMENTATION (2015) ***What is Angular?*** [Online] Available at: https://docs.angularjs.org/guide/introduction [Accessed: 11th October 2015]

Comparison of Different Paradigms

1. Overview and most common paradigms

There are many different programming paradigms but they overlap to some degree. In the following table are paradigms that are considered the main programming paradigms along with their main features and differences.

Paradigm	Description	Characteristics
Imperative	Computation as statements that directly change a program state	Direct assignments, common data structures, global variables
Structured	A style of imperative programming with more logical program structure	Structograms, indentation, no use of goto statements
Procedural	Derived from structured programming, based on the concept of modular programming or the procedure call	Local variables, sequence, selection, iteration, and modularization
Functional	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Lambda calculus, compositionality, formula, recursion, referential transparency, no side effects
Event-driven + Time-driven	Program flow is determined mainly by events, such as mouse clicks or interrupts including timer	Main loop, event handlers, asynchronous processes

Object-oriented	Treats datafields as objects manipulated through pre-defined methods only	Objects, methods, message passing, information hiding, data abstraction, encapsulation, polymorphism, inheritance
Declarative	Defines computation logic without defining its detailed control flow	4GLs, spreadsheets, report program generators
Automata-based programming	Treats programs as a model of a finite state machine or any other formal automata	State enumeration, control variable, state changes, isomorphism, state transition table

The most common of these paradigms are imperative, logic, functional, and object-oriented. There are two reasons why imperative languages are popular: the imperative paradigm most closely resembles the actual machine the actual machine itself so the programmer is closer to the machine and because of that closeness, this paradigm was the only one efficient enough for widespread use until recently. Another advantage of imperative programming is efficiency.

One of the advantages of logic oriented programming is that the programming steps themselves are kept to a minimum because the system solves the problem. Another advantage is that proving the validity of a program is simple.

Functional programming offers a high level of abstraction, especially when functions are used, and since it supresses many of the details of programming, it removes the possibility of committing many errors. Function oriented languages are good for programming massively parallel computers because programs can be evaluated in many different orders. Programs written in these languages also possess referential transparency, therefore, they are more inclined to mathematical proof and analysis.

When it comes to object-oriented programming, real-world objects are each

viewed as separate entities having their own state. Because objects are organized into classes which can also be created from other classes (inheritance), object-oriented paradigm provides key benefits of reusable code and code extensibility. Also, the mechanism of modelling a program as a collection of objects of various classes provides a high degree of modularity. The state of an object is manipulated and accessed only by that object's methods which results in separating a class' interface and class' implementation which is another benefit. [137]

2. Obsolete concepts

Structured programming is based on not using the ***goto*** statement but constructs of selection and repetition instead. However, few languages are purely structured while others use different deviations. Early exit from a function or loop is the most common deviation accomplished by using ***return***, ***break*** or ***continue*** statement.

Exception handling is another deviation as it changes the normal flow of program execution when an exception – anomalous or exceptional condition requiring special processing – occurs. Generally, an exception is handled by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. Depending on the type of exception, the handler may later resume the execution at the original location using the saved information.

Even though ***goto*** statement is not used, these deviations are common and present in many languages. Whether one of these deviations is used or any other,

the result is the same – there are multiple exit points instead of the single exit point required by structured programming. Therefore, the main idea of structured programming, which is avoiding “spaghetti code” is mostly abandoned.

Procedural paradigm, derived from structured programming, might also be considered obsolete. The idea of dividing the code into subroutines, methods, or functions is very important. However, these concepts are included in “bigger” paradigms, like object-oriented, so popular languages today are not solely based on procedural paradigm.

3. Abstractions in programming languages – common concepts

Abstraction plays the essential role in making programs easier for people to read. No matter which paradigm a language is based on, programming languages, in general, provide some common abstractions.

Programming language abstractions fall into two general categories: ***data abstraction*** and ***control abstraction***. Data abstractions simplify for human users the behaviour and attributes of data, such as numbers, character strings, and search trees. Control abstractions simplify properties of the transfer of control, that is, the modification of the execution path of a program based on the situation at hand. Examples of control abstractions are loops, conditional statements, and procedure calls.

Abstractions are also categorized in terms of levels, which can be viewed as measures of the amount of information contained (and hidden) in the abstraction. ***Basic abstractions*** collect the most localized machine information. ***Structured abstractions*** collect intermediate information about the structure of a program. ***Unit abstractions*** collect large-scale information in a program.

3.1. Data: Basic abstractions

Basic data abstractions in programming languages hide the internal representation of common data values in a computer. For example, integer data values are often stored in a computer using a two's complement representation. On some machines, the integer value -64 is an abstraction of the 16-bit twos complement value 1111111111000000. These values are also called "primitive" or "atomic", because the programmer cannot normally access the component parts or bits of their internal representation.

Another basic data abstraction is the use of symbolic names to hide locations in computer memory that contain data values. Such named locations are called ***variables***. The kind of data value is also given a name and is called a ***data type***. Data types of basic data values are usually given names that are variations of their corresponding mathematical values, such as ***int***, ***double***, and ***float***. Variables are given names and data types using a ***declaration***, such as the Pascal:

```
var x : integer;
```

Finally, standard operations, such as addition and multiplication, on basic data types are also provided.

3.2. Data: Structured abstractions

The data structure is the principal ethod for collecting related data values into a single unit. For example, an employee record may consist of a name, address,

phone number, and salary, each of which may be a different data type, but together represent the employee's information as a whole.

Another example is that of a group of items, all of which have the same data type and which need to be kept together for purposes of sorting or searching. A typical data structure provided by programming language is the ***array***, which collects data into a sequence of individually indexed items. Variables can name a data structure in a declaration, as in the C:

```
int a[10];
```

which establishes the variable ***a*** as the name of an array of ten integer values.

Like primitive data values, a data structure is an abstraction that hides a group of component parts, allowing the programmer to view them as one thing. Unlike primitive data values, however, data structures provide the programmer with the means of constructing them from their component parts (which can include other data structures as well as primitive values) and also the means of accessing and modifying these components.

3.3. Data: Unit abstractions

In a large program, it is useful and even necessary to group related data and operations on these data together, either in separate files or in separate language structures within a file. Typically, such abstractions include access conventions

and restrictions that support information hiding.

These mechanisms vary widely from language to language, but they allow the programmer to define new data types (data and operations) that hide information in much the same manner as the basic data types of the language. Thus, the unit abstraction is often associated with the concept of an abstract data type, broadly defined as a set of data values and the operations on those values. Its main characteristic is the separation of an interface (the set of operations available to the user) from its implementation (the internal representation of data values and operations).

Examples of large-scale unit abstractions include the module of ML, Haskell, and Python and the package of Lisp, Ada, and Java. Another, smaller-scale example of a smaller-scale example of a unit abstraction is the class mechanism of object-oriented languages.

An additional property of a unit data abstraction that has become increasingly important is its reusability – the ability to reuse the data abstraction in different programs, thus saving the cost of writing abstractions from scratch for each program. Typically, such data abstractions represent components and are entered into a library of available components. As such, unit data abstractions become the basis for language library mechanisms.

The combination of units is enhanced by providing standard conventions for their interfaces. Many interface standards have been developed, either independently of the programming language, or sometimes tied to a specific language. Most of these apply to the class structure of object-oriented languages, since classes have proven to be more flexible for reuse than most other language structures.

When programmers are given a new software resource to use, they typically study its application programming interface (API). An API gives the programmer only enough information about the resource's classes, methods, functions, and performance characteristics to be able to use that resource effectively.

3.4. Control: Basic abstractions

Typical basic control abstractions are those statements in a language that combine a few machine instructions into an abstract statement that is easier to understand than the machine instructions. For example, the algebraic notation of the arithmetic and assignment expressions:

```
SUM = FIRST + SECOND
```

This code fetches the values of the variables ***FIRST*** and ***SECOND***, adds these values, and stores the result in the location named by ***SUM***.

The term ***syntactic sugar*** is used to refer to any mechanism that allows the programmer to replace a complex notation with a simpler, shorthand notation. For example, the extended assignment operation ***x += 10*** is shorthand for the equivalent but slightly more complex expression ***x = x + 10***, in C, Java, and Python.

3.5. Control: Structured abstractions

Structured control abstractions divide a program into groups of instructions that are nested within tests that govern their execution. They, thus, help the

programmer to express the logic of the primary control structures of sequencing, selection, and iteration (loops). At the machine level, the processor executes a sequence of instructions simply by advancing a program counter through the instructions' memory addresses.

Selection and iteration are accomplished by the use of ***branch instructions*** to memory locations other that the next one. To illustrate these ideas, the following code shows an LC-3 assembly language code segment that computes the sum of the absolute values of 10 integers in an array named ***LIST***.

```
LEA  R1, LIST; Load the base address of the array (the first cell)

AND  R2, R2, #0; Set the sum to 0

AND  R3, R3, #0    ; Set the counter to 10 (to count down)

ADD  R3, R3, #10

WHILE LDR  R4, R1, #0; Top of the loop: load the datum from the

; current array cell

BRZP INC; If it's >= 0, skip next two steps

NOT  R4, R4; It was < 0, so negate it using twos complement

; operations

ADD  R4, R4, #1

INC   ADD  R2, R2, R4; Increment the sum

ADD  R1, R1, #1; Increment the address to move to the next array
```

```
                                ; cell
    ADD  R3, R3, #-1   ; Decrement the counter
    BRP  WHILE; Goto the top of the loop if the counter > 0
    ST   R2, SUM       ; Store the sum in memory
```

This assembly language code can be compared with the use of the structured ***if*** and ***for*** statements in the functionally equivalent C++ or Java code:

```
int sum = 0;
for (int i = 0; i < 10; i++){
int data = list[i];
if (data < 0)
data = -data;
sum += data;
}
```

Another structured form of iteration is provided by an ***iterator***. Typically found in object-oriented languages, an iterator is an object that is associated with a collection, such as an array, a list, a set, or a tree. The programmer opens an iterator on a collection and then visits all of its elements by running the iterator's methods in the context of a loop. For example, the following Java code segment uses an iterator to print the contents of a list, called ***exampleList***, of strings:

```
Iterator<String> iter = exampleList.iterator();
while (iter.hasNext())
System.out.println(iter.next());
```

The iterator-based traversal of a collection is such a common loop pattern that some languages, such as Java, provide syntactic sugar for it, in the form of an enhanced ***for*** loop:

```
for (String s : exampleList)
System.out.println(s);
```

Another powerful mechanism for structuring control is the ***procedure***, sometimes also called a ***subprogram*** or ***subroutine***. This allows a programmer to consider a sequence of actions as a single action that can be called or invoked from many other points in a program.

Procedural abstraction involves two things. First, a procedure must be defined by giving it a name and associating with it the actions that are to be performed. This is called ***procedure declaration***, and it is similar to variable and type declaration. Second, the procedure must actually be called at the point where the actions are to be performed. This is sometimes also referred to as procedure ***invocation*** or procedure ***activation***.

An abstraction mechanism closely related to procedures is the ***function***, which

can be viewed simply as a procedure that returns a value or result to its caller. The importance of functions is much greater than the correspondence to procedures implies, since functions can be written in such way that they correspond more closely to the mathematical abstraction of a function. Thus, unlike procedures, functions can be understood independently of the von Neumann concept of a computer or runtime environment.

Moreover, functions can be combined into higher-level abstractions known as ***higher-order functions***. Such functions are capable of accepting other functions as arguments and returning functions as values. An example of a higher-order function is a ***map***. This function expects another function and a collection, usually a list, as arguments. The map builds and returns a list of the results of applying the argument function to each element in the argument list.

3.6. Control: Unit abstractions

Control can also be abstracted to include a collection of procedures that provide logically related services to other parts of a program and that form a unit, or stand-alone, part of the program. For example, a data management program may require the computation of statistical indices for stored data, such as mean, median, and standard deviation. The procedures that provide these operations can be collected into a program unit that can be translated separately and used by other parts of the program through a carefully controlled interface. This allows the program to be understood as a whole without needing to know the details of the services provided by the unit.

One kind of control abstraction that is difficult to fit into any one abstraction level is that of parallel programming mechanisms. Many modern computers have several processors or processing elements and are capable of processing

different pieces of data simultaneously. A number of programming languages include mechanisms that allow for the parallel execution of parts of programs, as well as providing for synchronization and communication among such program parts.

Java has mechanisms for declaring ***threads*** and ***processes***. Ada provides the ***task*** mechanism for parallel execution. Ada's tasks are essentially a unit abstraction, whereas Java's threads and processes are classes and so are structured abstractions.[138]

4. Differences

Most of main programming languages are imperative or declarative, or use concepts of those paradigms. Imperative paradigm includes procedural programming, while object-oriented programming languages are usually multi-paradigm and typically come in combination with imperative paradigm. Declarative paradigm, on the other hand, includes logical and functional programming.

The main difference between these two paradigms is in the way of communicating with the machine. Imperative languages tell the machine ***how*** to do something and ***what*** we want to happen will be the result. With declarative languages, we tell the machine ***what*** we would like to happen while the computer figures out ***how*** to do it.

As an example, we can take doubling all the numbers in an array. In imperative style, the code would be as follows:

```
var numbers = [1,2,3,4,5]

var doubled = []


for(var i = 0; i < numbers.length; i++) {

var newNumber = numbers[i] * 2

doubled.push(newNumber)

}

console.log(doubled) //=> [2,4,6,8,10]
```

This code explicitly iterates over the length of the array, pulls each element out of the array, doubles it, and adds the doubled value to the new array, mutating the array ***doubled*** at each step.

A declarative approach might use the function ***map***:

```
var numbers = [1,2,3,4,5]


var doubled = numbers.map(function(n) {

return n * 2

})
```

```
console.log(doubled) //=> [2,4,6,8,10]
```

This function creates a new array from an existing array, where each element in the new array is created by passing the elements of the original array into the function passed to ***map***, in this case ***function(n) { return n*2 }***.

The function ***map*** abstracts away the process of explicitly iterating over the array, and lets us focus on what we want to happen. The function passed to ***map*** is pure, meaning it does not have side effects, it just takes in a number and returns the number doubled.

Another example is adding all the items in a list. Imperatively, it would be done as follows:

```
var numbers = [1,2,3,4,5]

var total = 0


for(var i = 0; i < numbers.length; i++) {

total += numbers[i]

}

console.log(total) //=> 15
```

To solve this declaratively, we can use the function ***reduce***:

```
var numbers = [1,2,3,4,5]


var total = numbers.reduce(function(sum, n) {
return sum + n
});
console.log(total) //=> 15
```

This function reduces a list down into a single value using the given function. It takes the function and applies it to all items in the array. On each invocation, the first argument ***sum*** is the result of calling the function on the previous element, and the second, ***n***, is the current element.[139]

Functional languages (which are declarative in nature) can have a very simple syntactic structure. The syntax of the imperative languages is much more complex which makes them more difficult to learn and to use. The semantics of functional languages is also simpler than that of the imperative languages.[140]

An appealing aspect of pure functional languages and of programs written in the pure functional subset of larger languages is that programs can be executed concurrently. We can see how parallelism arises in pure functional languages by using the example of a function call $f(e_1, \cdots, e_n)$, where function arguments $e_1, \cdots, e_n$ are expressions that may need to be evaluated.

In functional programming, we can evaluate $f(e_1, \cdots, e_n)$ by evaluating $e_1, \cdots, e_n$ in

parallel because values of these expressions are independent. When it comes to imperative programming, for an expression such as $f(g(x),\ h(x))$, the function g might change the value of x. Hence, the arguments of functions in imperative languages must be evaluated in a fixed, sequential order. This ordering restricts the use of concurrency.

Backus used the term ***von Neumann bottleneck*** for the fact that in executing an imperative program, computation must proceed one step at a time. Because each step in a program may depend on the previous one, we have to pass values one at a time from memory to the CPU and back. This sequential channel between the CPU and memory is what he called the von Neumann bottleneck.

Although functional programs provide the opportunity for parallelism, and parallelism is often an effective way of increasing the speed of computation, effectively taking advantage of inherent parallelism is difficult. One problem that is fairly easy to understand is that functional programs sometimes provide too much parallelism. If all possible computations are performed in parallel, many more computation steps will be executed than necessary.[141]

Some claim that the use of functional programming results in an order-of-magnitude increase in productivity, largely due to functional programs being claimed to be only 10 percent as large as their imperative counterparts. These factors allow proponents of functional programming to claim productivity advantages over imperative programming of 4 to 10 times.

However, program size alone is not necessarily a good measure of productivity. Certainly not all lines of source code have equal complexity, nor do they take the same amount of time to produce. In fact, because of the necessity of dealing with variables, imperative programs have many trivially simple lines for initializing and making small changes to variables.

Execution efficiency is another basis for comparison. When functional programs are interpreted, they are of course much slower than their compiled imperative counterparts. However, there are now compilers for most functional languages, so that execution speed disparities between functional languages and compiled imperative languages are no longer so great.

One might say that because functional programs are significantly smaller, they should execute much faster than the imperative programs. However, this is often not the case, because of a collection of language characteristics of the functional languages, such as lazy evaluation, that have a negative impact on execution efficiency.

Considering the relative efficiency of functional and imperative programs, it is reasonable to estimate that an average functional program will execute in about twice the time of its imperative counterpart. This may sound like a significant difference, one that would often lead one to dismiss the functional languages for a given application. But, this difference is important only in situations where execution speed is of the utmost importance.

There are many situations where a factor of two in execution speed is not considered important. For example, consider that many programs written in imperative languages, such as the Web software written in JavaScript and PHP, are interpreted and therefore are much slower than equivalent compiled versions. For these applications, execution speed is not the first priority.

Another source of the difference in execution efficiency between functional and imperative programs is the fact that imperative languages were designed to run efficiently on von Neumann architecture computers, while the design of functional languages is based on mathematical functions. This gives the

imperative languages a large advantage.

Functional languages have not attained greater popularity because the vast majority of programmers learn programming using imperative languages, which makes functional programs appear to them to be strange and difficult to understand. For many who are comfortable with imperative programming, the switch to functional programming is an unattractive and potentially difficult move. On the other hand, those who begin with a functional language never notice anything strange about functional programs.[142]

Paradigms like event-driven and automata based differ greatly from these two main paradigms. Code execution of programs based on these concepts is triggered by events or state changes. Since the part of the code associated with certain change executes when that change happens, there is no typical flow of program execution that we are familiar with from other paradigms.

Bibliography

1. LOUDEN, K. C., LAMBERT, K. A. (2011) ***Programming Languages Principles and Practice.*** 3rd Edition. Cengage Learning

2. MITCHELL, J. C. (2002) ***Concepts in Programming Languages.*** Cambridge University Press

3. SEBESTA, R. W. (2012) ***Concepts of Programming Languages.*** 10th Edition. Addison-Wesley

4. LATENTFLIP (2013) ***Imperative vs Declarative*** [Online] Available at: http://latentflip.com/imperative-vs-declarative/ [Accessed: 30th May 2015]

5. UCF DEPARTMENT OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE (2015) ***Major Programming Paradigms*** [Online] Available at: http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html [Accessed: 30th May 2015]

Experience Oriented Programming

1. Why base programming on interactions

Today, a lot of applications used on computers and smart phones can communicate with each other. Some are solely used for communication, such as messengers, while others have options to share certain information with others, like location, photos, videos, etc.

Interactions could be classified by two standards. One is based on how many software units one software communicates with, while the other can be described by how much interaction/communication is actually going on.

When it comes to how many software units communicate, software can interact with exactly one unit or many, which can be either a known number or not. An example of communicating with exactly one unit are Facebook Messenger, Skype, Viber, What's Up, etc. when they are used to exchange messages with one person.

Communicating with many people, on the other hand, has two aspects, as already mentioned. When sharing information with a Facebook Group you are a member of, or with members of a Group Chat, Skype Conference Call, and so on, you are aware of how many people you are communicating with. Some web

services can be used in a way that the result is open to the public. YouTube (commenting, posting videos), Trip Advisor (writing a review, grading) and blogging are some of them. These services offer registration but the content on these websites are public even to users without an account. Therefore, the number of units that access the information is not known.

Now, let's examine the second standard. Some games, for example, are singleplayer but offer users to share their scores, records, or accomplishments. This means that they are not initially based on communication and data exchange, but can occasionally interact with other units. Contrary to that, multiplayer games have to exchange information in order to work properly. Playing itself requires communication, but on top of that, there is usually a chat room that is additionally based on sending and receiving messages.

All these web services have changed the way people communicate. Communication is not only modernized, it is also expanded. With software being the mediator in communication, this changed the way the software is developed. As a result, there is need for combining computation and social elements.

2. Language support for Social-oriented programming

Social-oriented programming (SOP), also known as Interaction-oriented programming, can be considered as an extension or improvement of Object-oriented programming. The reason for this is that implementing software using SOP includes creating classes and using other known concepts of OOP, only the focus shifts.

In OOP, programmers create classes that represent models of real-life objects.

These classes contain fields and methods that modify those fields. With this approach, the main program is focused on creating instances of classes, called objects, storing data in their fields and manipulating them through their methods.

Unlike OOP, SOP has its focus on interactions between objects. In this case, there is semantic difference between classes although their syntax is the same. What makes them different from each other is the concepts they use and situations they are used in.

When a class needs to represent a real-life object, it contains fields and, perhaps, getters and setters to maintain encapsulation. Here, we will call these classes entity classes, or just entities. Unlike this, a class can represent interactions between entities, therefore it will be referred to as interaction class.

Interaction classes are the focus in SOP and they contain all methods related to software communication. Because of these classes, objects of entity classes are not instantiated until there is an actual interaction. This means that entity objects are not stand-alone. Instead, interaction classes instantiate them only when there is need for object communication. When they are no longer needed, they can be disposed.

This is made possible by adding fields of entity type to interaction classes. Beside entity fields, interactions ideally contain hashtables. The purpose of hashtables is storing interaction data. Interaction data is composed of objects participating in communication and methods called during that communication. It is important to store this in interaction classes because entities should not be stand-alone objects so accessing their fields (hashtable in this case) would violate this principle.

Because interaction classes are written using the same syntax as OOP classes in

general, SOP is supported by OOP languages such as Java, C#, and JavaScript. The important thing to keep in mind when programming in this manner is separating entity classes from interaction classes, as well as writing methods that represent actions to be performed when an interaction occurs. Also, objects of entity classes should not be instantiated before they interact with another entity.

3. Implementation examples

SOP is relatively new so examples of different software and web services that have already been mentioned are not actually implemented using this approach. Still, they are suitable when explaining this paradigm. In the following code examples, Java will be used to demonstrate how software based on communication should actually be implemented.

For starters, let’s examine some of the options Facebook has to offer its users. First, user contains some basic information about a person like their name, hometown, gender, date of birth, relationship status, etc. Each user also has friends, each of them being a user themselves, and messages that are exchanged between a user and one of their friends. So, entity class representing the Facebook user could be similar to the following:

```
public class User {

private String firstName;

private String familyName;

private Date dateOfBirth;
```

```
private List<User> friends;

//Post is another entity class

private List<Post> posts;

…

//remaining fields and getters and setters
}
```

Post basically consists of textual content but it can also contain video or photo. The way videos and photos are represented will not be addressed here.

```
public class Post {

...

private String content;

...

}
```

If we were to choose C# for implementation, instead of using private fields along with getters and setters, it would be better to use properties. In this case, a field would be represented as:

```
public FirstName {
```

```
    get;
    set;
    }
```

or:

```
    private String firstName;
    public FName {
    get {return firstName;}
    set {firstName = value;}
    }
```

User interactions include posting a status, posting to a friend’s wall, sending a private message, sending a friend request, liking a status, or another sort of shared content (photo, video), among others. The user that requests interaction can be considered a sender, while the other users are on the receiving end. Interaction can have one or many receivers.

```
    public class UserInteraction {
    private User sender;
    private List<User> receivers;
```

```
public void addFriend(Integer requestFromId, Integer requestToId)   {

sender = new User(requestFromId);

receivers = new LinkedList<User>().add(new User(requestToId));

receivers[0].getFriends().add(sender);

...

//new friend is added to the list of both sender and receiver

//sender and receiver data are refreshed in the database

//friend lists are merged with ones in the database

...

sender = null;

receivers = null;

}


public void likeAPost(Integer userThatLikesId, Integer userThatPostedId,
Integer postId) {

sender = new User(userThatLikesId);

Post post = new Post(userThatPostedId, postId);
```

```
post.getLikes().add(sender);

...

//post is merged in database

...

sender = null;

}

}
```

These examples illustrate how SOP can be more efficient than OOP. Whenever an interaction is over, all instantiated objects are set to null, which destroys their reference. Sometimes, it is necessary to create local variables. Even with them involved, objects are not stand-alone since their reference is destroyed after the method finishes.

Including a hashtable in interaction class would mean adding a field of type HashedMap in case of Java. HashMap consists of entries that have a key and a set of mapped values. Key represents a hashed value of two IDs of objects that interact with each other. Hashed value is calculated through the redefined ***hash*** method that is also in the interaction class.

Mapped value can be a single value or consisted of several values stored in a list. HashMap needs to be an interface for interactions. Whenever we need information about an interaction, we can access it through the hashed map. This requires the map to always be instantiated so it cannot contain references to entities as values. That would violate the basic rule of SOP. Instead, an ID of the

entity object exchanged during interaction is stored in the map. For example, if we were to store chat history between ***personA*** and ***personB***, HashMap would be defined as follows:

```
private Map<Integer,List<Integer>> map = new
HashMap<Integer,List<Integer>>();
```

In the previous code snippets, interaction method created and added to the database new users and posts – User and Post being entity classes. With everything exchanged being an entity instance, we are making sure that any kind data can be stored in the map when we set the type of map values to Integer – every entity has an ID.

A new entry is added to the map when an interaction happens, before we set entity fields back to null. In this case, the key would be the result of hash function with parameters ***personA.getId()*** and ***personB.getId()***.

Twitter functions a bit differently than Facebook but its method implementation is basically the same. Instead of friends, a user has followers. There are no posts, but users write tweets that can be retweeted by other users. LinkedIn also has users that can either be employers or potential employees. Their interaction is based on posting job offers, applying for jobs, employer-employee messaging, filling out forms, etc.

Whatever the software is based on, the implementation principle is the same – objects are instantiated when an interaction starts by acquiring their information from a database. It is then simple to access an entity only using its database ID.

It is possible to access a user through the ID because the first interaction between two systems is user registration. When registering, an object is created just like with any other interaction, but the method for creating a new user/member does not accept an ID as a parameter. Persisting the new user into the database defines the ID the user will have as long as his/her account is active. So, registration method would be similar to the following:

```
public void register() {
User u = new User();
//setting user fields
u.setFirstName(firstName);
...
u.setDateOfBirth(dateOfBirth);
...
//persisting the user into the database
u = null;
}
```

Since the reference of the object is gone after registration, there needs to be a way to remember the user's ID so it can be passed on to different methods made for interactions. When the software is started, it requires the user to log in. This way, the software knows the user's username, e-mail, or whatever else is used to

uniquely identify a person. Through this property, all of the user's information can be now found in the database. Therefore, we can store the user's ID in a variable that has a value as long as the software is running, and use it when it needs to be passed to a method.

Some web services do not require registration in order for the user to interact with others. For example, online newspapers have a comment section on every article that can be used by anyone. This is the case of unknown number of receivers because these articles are available to everyone and worldwide. The sender is unknown in this case, so it cannot be found in the database. Instead, the sender's side is instantiated as a message/comment send by an anonymous user.

```
public class Comment {

private String content;

...

}
```

There are also no known receivers, so the software on the receiving side instantiates an object of type Article, for example, which is acquired from the database, and adds the comment to the list of article's comments. To summarize, senders and receivers are not necessarily people.

4. Advantages of SOP

Even though Social-oriented paradigm is basically expanded object-oriented

paradigm, developing a tool that would base programming solely on SOP would have many advantages. For starters, making sure objects are not stand-alone results in programmer not having to worry about their references – whether the object is instantiated or not (which can also cause an exception if not careful), which object the reference points to, how to manipulate references, etc.

A tool like that would base programming on actions, so programmers would be focused on ***how*** instead of ***what***. Importance would then lie in implementing actions well, including steps needed to accomplish a task.
This approach can be incorporated in development of various types of software. This includes fields like social networking, gaming, blogging, freelancing, online shopping, and so on. This is the reason why it would be simple to start using the social-oriented approach in problem-solving.

The type of software SOP can be used to implement already exists. It can improve the way the known software works, but it can also easily solve tasks that are typical for new projects today, because a lot of them support a form of sharing. With SOP being based on known concepts, it is easier for programmers to switch to this approach without having to additionally learn new syntax or an entire language.

What Does the Future Hold

1. Factors in the development of languages

High-level languages have always been designed with the aim of assisting the task of programming computers. However, from the 1950s to the present, the importance and the cost of the various components involved in the

implementation of programs has changed and therefore priorities when designing a language have completely changed.

In the 1950s, the hardware was certainly the most expensive and important resource. The first high-level languages were therefore designed with the aim of obtaining efficient programs which would use the potentiality of the hardware to the maximum. This attitude in the first languages is reflected by the presence of many constructs that were inspired directly by the structure of the physical machine. For example, the “three-way jump” present in FORTRAN directly derives from the corresponding instruction on the IBM 704.

The fact then that programming was very difficult and required very long times was considered as a problem of secondary importance, which could be solved by means of large amounts of human resources which were certainly less expensive than the hardware.

Today, the situation is the direct opposite. Hardware is relatively cheap and efficient and the preponderant costs of information-system development are linked to the tasks performed by computing specialists. Furthermore, given the increasingly critical application of computer systems, there are considerations of correctness and security that were minimal if not wholly absent 50 years ago. Modern languages are therefore designed taking into account first the improvement of various software project activities, while the preoccupation with efficient use of the physical machine has dropped to second place, except in some particular cases.

The development of programming languages has followed a long, continuous process governed by a number of factors. Here are some of the most important:

- ***Hardware*** – The type and performance of available hardware devices clearly influenced the languages that used it.

- ***Applications –*** Applications of computers, initially solely of a numeric type, rapidly extended to many different fields, including some that required the processing of non-numeric information. New application fields can require languages with specific properties. For example, in Artificial Intelligence and knowledge processing languages are needed that allow the manipulation of symbolic formalisms rather than the solution of mathematical problems. As another example, computer games are usually implemented using particular programming languages.

- ***New methodologies –*** The development of new programming methodologies, particularly programming in the large, has influenced the development of new languages. A significant example in this sense is object-oriented programming.

- ***Implementation*** – The implementation of language constructs is significant for the development of successive languages because it allows to understand the validity of a construct to be taken into consideration as well as its practical use.

- ***Theory*** – Finally, the role of theoretical studies should not be forgotten. They play an important role in selecting some typologies of structure and in particular in identifying new technical tools to improve the programming activity.[143]

2. Future of programming languages

In the 1960s, some computer scientists dreamed of a single universal programming language that would meet the needs of all computer users. Attempts to design and implement such a language, however, resulted in frustration and failure. In the late 1970s and early 1980s, a different dream emerged – a dream that programming languages themselves would become obsolete, that new specification languages would be developed that would allow computer users to just say what they wanted to a system that would then find out how to implement the requirements.

That is what logic programming languages attempt to do. However, even though these languages can be used for quick prototyping, programmers still need to specify algorithms step by step when efficiency is needed. Little progress has been made in designing systems that can on their own construct algorithms to accomplish a set of given requirements.

Programming has, thus, not become obsolete. In a sense it has become even more important, since it now can occur at so many different levels, from assembly language to specification language. And with the development of faster, cheaper, and easier-to-use computers, there is a tremendous demand for more and better programs to solve a variety of problems.

It is difficult to predict the future of programming language design but two of the most interesting perspectives on the evolution of programming languages in the last 20 years come from a pair of second-generation Lisp programmers, Richard Gabriel and Paul Graham.

Gabriel is puzzled by the fact that very high-level, mathematically elegant languages such as Lisp have not caught on in industry, whereas less elegant and

even semantically unsafe languages such as C and C++ have become the standard.

He likens the spread of C in the programming community to that of a virus. The simple footprint of the C compiler and runtime environment and its connection to the UNIX operating system has allowed it to spread rapidly to many hardware platforms. Its conventional syntax and lack of mathematical elegance have appealed to a very wide range of programmers, many of whom may not necessarily have much mathematical sophistication. For these reasons, Gabriel concludes that C will be the ultimate survivor among programming languages well into the future.

Graham, however, sees a different trend developing. He believes that major recent languages, such as Java, Python, and Ruby, have added features that move them further away from C and closer to Lisp. However, like C in an earlier era, each of these languages has quickly migrated into new technology areas, such as Web-based client/server applications and mobile devices.

Like most writers on programming languages, Graham classifies them on a continuum, from fairly low level (C) to fairly high level (Java, Python, Ruby). But he then asks two interesting questions: If there is a range of language levels, which languages are at the highest level? And if there is a language at the highest level, and it still exists, why wouldn't people prefer to write their programs in it?

Graham claims that Lisp, after 50 years, always has been and still is the highest-level language. He then argues, in a similar manner to Gabriel, that Lisp's virtues have been recognized only by the best programmers and by the designers of the aforementioned recent languages. However, Graham believes that the future of Lisp may lie in the rapid development of server-side applications.[144]

3. Generic programming

During the last two decades, there has been an ever-growing interest in generic programming. There are several definitions of generic programming in use and it would be simplest to say that generic programming is equated to a set of language mechanism for implementing type-safe polymorphic containers. A good example is ***List<T>*** in Java.

The notion that motivated the design of Standard Template Library advocates a broader definition: a programming paradigm for designing and developing reusable and efficient collections of algorithms.[145] Another definition says generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are instantiated when needed for specific types provided as parameters.

This approach permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication, and it was pioneered by ML in 1973. These software entities are known as ***generics*** in Ada, Delphi, Eiffel, Java, C#, F#, Swift, and Visual Basic .NET; ***parametric polymorphism*** in ML, Scala and Haskell; ***templates*** in C++ and D; and ***parametrized types*** in the book ***Design Patterns***.

The notion of generic programming was popularized in the ‘60s with the LISP programming languages and its descendants because it provides direct support for higher-order functions. Since then, programming techniques and linguistic support for defining algorithms that are capable of operating over a wide range of data structures have been subjects of a large body of work.

Language features for writing some classes of polymorphic functions and data structures have received more attention than sound programming techniques at the foundation of generic libraries. In fact, generic programming is often equated with language features. It is common to see definitions of this type of programming that are to some degree crafted to mean what the specific programming languages under consideration support.

There have been different approaches to generic programming in both the functional and imperative programming communities. Two main schools of thought are identified:

- The gradual lifting of concrete algorithms, a discipline first described by David Musser, Alexander Stepanov, Deepak Kapur, and collaborators.
- A calculational approach to programming, with foundations laid by Richard Bird and Lambert Meertens.

The first school defines the discipline of generic programming as follows: start with a practical, useful, algorithm and repeatedly abstract over details. At Any stage of abstraction, the generic version of the algorithm should, when instantiated, match the original algorithm both in semantics and efficiency. The gradual lifting stops when these conditions cease to hold.[146]

In other words, Musser and Stepanov defined the generic programming paradigm as an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, analogously to the abstraction of algebraic theories in abstract algebra.

The best example of this programming approach is the Standard Template Library, which developed a theory of iterators that is used to decouple sequence data structures and the algorithms operating on them. For example, given N sequence data structures (singly linked list, vector, etc.), and M algorithms to operate on them (fund, sort, etc.), a direct approach would implement each algorithm specifically for each data structure resulting in N x M combinations to implement. However, in the generic programming approach, each data structure returns a model of an iterator concept that is a simple value type that can be dereferenced to retrieve the current value, or changed to point to another value in a sequence. Then, each algorithm is written generically with arguments of such iterators. Thus, only N + M data structure-algorithm combinations need to be implemented.

It is fundamental to require abstract specification that is independent of the actual data representation for two reasons:

- It is at the basis of substitution of one datatype interface for another when they are similar.
- It allows classification of similar interfaces based on their efficiency.

The second school of thought in generic programming has its roots in the initial algebra approach to datatypes and a calculational approach to program construction. In this setting, category theory is an essential tool.

In this approach, also referred to as datatype generic programming, structures of datatypes are parameters of generic programs. Datatype generic programming has had a strong focus on regular datatypes essentially described by algebras

generated by the functors ***sum***, ***product***, and ***unit***. Algorithms written for those functors can then operate on any inductive datatype, and are thus inherently very generic.

The first style of generic programming that was mentioned emphasizes concept analysis, the process of finding and establishing the important classes of concepts that enable many useful algorithms to work. Then, programmers explicitly define correspondence from their datatypes to those classes of concepts.

The two definitions of generic programming are fundamentally very close to each other, but the emphasis in each view is on different aspects. One is focused on a particular structural algebra for datatypes and the algorithms defined in terms of that algebra, whereas the other is focused on finding and classifying classes of algebras based on some notions of efficiency.

Although both methodologies have an underlying theoretical language-independent model, C++ has become the dominating platform for the Musser-Stepanov style, whereas Haskell and its variants are the almost exclusive tool for datatype generic programming.[147]

When it comes to programming language support for genericity, it comes down to two main implementations – in object-oriented languages and in functional languages.

When working with statistically typed languages in order to create container classes, it is inconvenient to have to write specific implementations for each datatype contained, especially if the code for each one of them is virtually identical. As an example, we will examine C++ code. With C++, the duplication of code is avoided by defining a class template:

```
template<typename T>
class List
{
/* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

In this code, ***T*** is a placeholder for whatever type is specified when the list is created. These templates or “containers-of-type-T” allow a class to be reused with different datatypes as long as certain contracts such as subtypes and signatures are kept.

Templates can also be used for type-independent functions as bellow:

```
template<typename T>
void Swap(T & a, T & b) //"&" passes parameters by reference
{
T temp = b;
```

```
b = a;

a = temp;

}


string hello = "world!", world = "Hello, ";

Swap( world, hello );

cout << hello << world << endl; //Output is "Hello, world!"
```

An important feature of templates in C++ is ***template specialization***. This allows alternative implementations to be provided based on certain characteristics of the parameterized type that is being instantiated. This feature has two purposes: to allow certain forms of optimization, and to reduce code bloat.

First, let's take the function ***sort*** for example. One of the primary activities that this function does is swapping or exchanging the values in two of the container's positions. If the values are large, it is often quicker to first build a separate list of pointers to the objects and sort those pointers before building the final sorted sequence. On the other hand, values could be quite small so it would be faster to just swap them in-place as needed. Also, parametrized type could already be of some pointer-type, which means there is no need to build a separate pointer array. Specialization allows the programmer to write different implementations and specify the characteristics that the parametrized types must have for each implementation to be used.

Class templates are slightly different as they can be partially specialized. That

means that an alternate version of the class template code can be provided when some of the template parameters are known, while leaving other template parameters generic. This can be used to create a default implementation that assumes copying a parameterizing type is expensive, thus increasing overall efficiency. Class templates can also be fully specialized so that there is an alternate implementation when all of the parameterizing types are known.

When speaking of generic programming with functional languages, we can take Haskell as an example. Haskell supports generic programming through the type class mechanism. Six of the predefined Haskell type classes have the special property of supporting derived instances. This means that when defining a new type, a programmer can state that this type is to be an instance of one of these special type classes. This does not require implementation of the class methods as it is usually necessary. Instead, all necessary methods will be derived based on the structure of the type.

The following code snippet presents a declaration of a type of binary trees and it states that it is to be an instance of the classes ***Eq*** and ***Show***:

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)

deriving (Eq, Show)
```

This results in an equality function and a string representation function being automatically defined for any type of the form ***BinTree T*** provided that ***T*** itself supports those operations. The support for derived instances of ***Eq*** and ***Show*** makes their methods == and ***show*** generic and these functions can be applied to values of various types. Although they behave differently for every argument type, little work is needed to add support for a new type.

4. Metaprogramming

Metaprogramming is the writing of computer programs with the ability to treat programs as their data. As a result, a program could be designed to read, generate, analyse or transform other programs, and even modify itself while running. By literally dissecting the word ***meta-program***, we get the meaning “a program about a program”. In other words, a meta-program is a program that manipulates code. One example of a meta-program is the C++ compiler because it manipulates C++ code to produce assembly language or machine code.[148]

Meta-programs analyze, generate, and transform object programs. In this process, object programs are structured data. It is common practice to use abstract syntax trees instead of the textual representation of programs. The language in which the meta-program is written is called the meta-language. Abstract syntax trees are represented using the data structuring facilities of the meta-language: records (structs) in imperative languages like C, objects in object-oriented languages like C++ and Java, algebraic data types in functional languages like ML and Haskell, and terms in term rewriting systems such as Stratego.

This kind of representations allow the full capabilities of the meta-language to be applied in the implementation of meta-programs. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching, it is easy to compose and decompose abstract syntax trees. For meta-programs such as compilers, programming with abstract syntax is adequate since only small fragments are manipulated at a time.

Object programs are often reduced to a core language that only contains the

essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages. However, there are many applications of meta-programming in which the use of abstract syntax is not satisfactory. The reason for this is that the conceptual distance between the concrete programs that we understand and the data structure access operations used for composition and decomposition of abstract syntax trees is too large.

This can easily be seen in the case of record manipulation in C, where the construction and deconstruction of patterns of more than a couple of constructors becomes unreadable. But even in languages that support pattern matching on algebraic data types, the construction of large code fragments in a program generator can be unpleasant. Here is a tiny program pattern that is, although small, easier to read in the concrete variant (first code) than the abstract variant (second code).

```
let ds
in let var x ta := (es1)
in es2 end
end
```

```
Let(ds,
[Let([VarDec(x,ta,Seq(es1))],
es2)])
```

Besides understandability and complexity problems, there are other reasons why using abstract syntax may be undesirable. Desugaring to a core language is not always possible. For example, in the renovation of legacy code the goal is to repair the bugs in a program, but leave it intact. This entails that a much larger abstract syntax needs to be dealt with.

Another example that shows using concrete syntax is preferable is the definition of transformation or generation rules by programmers rather than by compiler writers (meta-programmers). Other application areas that require concrete syntax are application generation and structured document processing (i.e. XML).

Hence, it is desirable to have a meta-programming language that lets us write object-program fragments in the concrete syntax of the object language. In general, we would like to write a meta-program in meta-language ***M*** that manipulates a program in object language ***O***, where ***M*** and ***O*** could be the same.

This requires the extension of the syntax of ***M*** with that of ***O*** such that ***O*** expressions are interpreted as data construction and analysis patterns. This problem is traditionally approached by extending ***M*** with a quotation operator that lets the meta-programmer indicate object language fragments. Antiquotation allows the use of meta-programming language constructs in these object language fragments to splice meta-computed object code into a fragment. If ***M*** equals ***O*** then the syntax extension is easy by just adding quote and antiquote operators to ***M***.

Building a meta-language that supports a different object language is a difficult task with traditional parser technology. Building a meta-language that allows the user to define the object language to use and determine the syntax for quotation and antiquotation operators is even harder.[149]

When it comes to the benefits of metaprogramming, we can examine two different approaches that are simpler when addressing the same problems. With metaprogramming, we could do the computation at runtime instead of compile time. The main reason to rely on a meta-program is that by doing as much work as possible before the resulting program starts, we get faster programs. However, compiling grammar, parsing and optimization during runtime could actually degrade a program's overall performance.

The second alternative to metaprogramming is user analysis. Instead of doing computation at runtime or compile time, we could just do it by hand. If the alternative is writing a meta-program that will only be used once, one could argue that user analysis is more convenient. It is certainly easier to translate one binary number than to write a correct meta-program to do so. It only takes a few such instances to tip the balance of convenience in the opposite direction, though. Regardless of how many times it's used, a meta-program enables its users to write more expressive code, because they can specify the result in a form that corresponds to their mental model. Finally, because the logic of a meta-program only needs to be written once, the resulting program is more likely to be correct and maintainable.

After the why, it is also important to address when metaprogramming is appropriate. Here are three situations when a meta-programmed solution may be appropriate:

- You want the code to be expressed in terms of the abstractions of the problem domain.
- You would otherwise have to write a great deal of boilerplate

implementation code.

- You need to choose component implementations based on the properties of their type parameters.[150]

5. Language-oriented programming

Designing and developing large-scale software systems comes with problems and they are usually caused by the following properties of large software systems:

- ***Complexity*** – This is an essential property of all large pieces of software. This leads to several problems. There are often communication difficulties among a large team of developers and these can lead to product flaws, cost overruns and schedule delays. It may also be difficult or impossible to visualise all the states of the system which makes it impossible to understand the system completely. Maintaining conceptual integrity becomes difficult because it is difficult to get an overview of the system.
- ***Conformity*** – Many systems are constrained by the need to conform to complex human institutions and systems.
- ***Change*** – Any successful system will be subject to change as it is used because it is modified to enhance its capabilities, or even to be applied beyond the original domain. Also, it should survive beyond the normal life of the machine it runs on and be able to be ported to other

machines and environments.

- ***Invisibility*** – With complex mechanical or electronic machines or large buildings the designers and constructors have blueprints and floorplans which provide an accurate overview and geometric representation of the structure. For complex software systems there is no such representation. There are several distinct but interacting graphs of links between parts of the system to be considered (control flow, data flow, time sequence, etc.). Even an accurate model or abstraction of the system may become unreliable as the system is enhanced and modified over a period of time.

In the history of computer science, the greatest single gain in software productivity has been achieved through the development of high-level languages with suitable compilers and interpreters. The use of high-level language often allows a program to be implemented with an order of magnitude fewer lines of code than if everything was written in assembler. In addition, these lines of code will typically be easier to read, analyse, understand, and modify.

In addition to these benefits, another benefit is that high-level languages encapsulate a great deal of programming knowledge in an easily usable form. For example, the programmer can let the compiler deal with subroutine call and return linking, procedure arguments, simple optimisations, etc.

Besides reduction in total code size, the aim of the language-oriented programming is for the domain-oriented language to form a repository of domain knowledge in a form which is readily useable by programmers working in that domain. Common objects in the domain will also appear in the language, common operations in the domain will be readily available as language

constructs, even though the implementation of these operations may be large and complicated.

The first stage in a language-oriented development is a language design, providing a formal syntax and semantics for this language. A language consists of a set of primitive operations together with language constructs and specialised abstract data types.

Having completed the language design, the development of the system breaks down into two largely independent stages:

- Implement the software system in the new language.
- Implement the language in some existing computer language, i.e. write a compiler or interpreter or translator for the language.

Either or both of these stages may benefit from a recursive application of the method. Such development will result in a series of language layers with lower-level languages at the bottom and very high-level languages at the top. Each level is implemented in terms of the next lower level.

Language-oriented programming is beneficial because the method provides a complete separation of concern between design issues, and implementation issues. A system implemented using the language-oriented method, as a series of language levels, ends up much smaller than a bottom up or top down implementation of the same system. This is due to the fact that with a problem-specific very high level language, a few lines of code are sufficient to implement highly complex functions. The implementation of the language is also kept small since only features relevant to the particular problem domain need to be

implemented.

The small size of the final system means that the total amount of development work required is reduced, without increasing the complexity of the system, and for the same or higher functionality. This leads to improved maintainability, fewer bugs, and improved adaptability. The very high-level language means that a small amount of code in this language can achieve a great deal of work.

The most important factor affecting maintainability is the size of the software system so more lines of code will generally require more maintenance effort. The small total size of a system produced by the language-oriented approach implies that it will therefore be highly maintainable.

With traditional programming methods, many design decisions are spread out through the code. It becomes difficult for maintainers to determine all the impacts of a particular design decision, or conversely, to determine which design decisions led to a particular piece of code being written in a specific way. With language-oriented development, the effects of a design decision will usually be localised to one part of the system.

Porting to a new operating system or programming language becomes greatly simplified. Only the middle-language needs to be re-implemented on the new machine, the implementation of the system can then be copied across without change. Also, there is a great potential for re-use of the middle-level languages for similar development projects.[151]

6. Agent-oriented programming

Agent-oriented programming (AOP) is a paradigm where the construction of the

software is centred on the concept of software agents. AOP has externally specified agents with interfaces and messaging capabilities at its core, unlike object-oriented which has objects at its core. The concept of agent-oriented programming was first used by Yoav Shoham, a computer scientist at Stanford University, within his Artificial Intelligence studies in 1990.

The state of an agent consists of components called beliefs, choices, capabilities, commitments. For this reason, the state of an agent is called its ***mental state***. The mental state of agents is captured formally in an extension of standard epistemic logics: beside temporalizing the knowledge and belief operators, AOP introduces operators for commitment, choice and capability. Agents are controlled by agent programs, which include primitives for communicating with other agents.
According to Shoham, the term ‘agent’ is used in diverse ways but the question of what an agent is becomes the question of what can be described in terms of knowledge, belief, commitment, etc. Although it is not always advantageous to do so, anything can be so described.

As an example, he treats a light switch as an agent with the capability of transmitting current at will, who invariably transmits current when it believes we want it transmitted and not otherwise. That makes flicking a switch our way of communicating our desires.

This mechanism is easy enough to understand to have a simpler, mechanistic description of the behaviour. In contrast, we do not have equally good knowledge of the operation of complex systems like robots or people. In this cases, it would often be most convenient to employ mental terminology.

Adopting this sense of agenthood, AOP specializes the framework by allowing the modules, called agents, to possess knowledge and beliefs about themselves and about one another, to have certain capabilities and make choices. A

computation consists of these agents informing, requesting, offering, accepting, rejecting, competing with and assisting one another. AOP can be viewed as a specialization of the object-oriented programming which proposes viewing a computational system as made up of modules that are able to communicate with one another and that have individual ways of handling incoming messages.

An alternative view of AOP is as an application of a formal language. The formal language being applied is a generalization of epistemic logics, which have been used extensively in artificial intelligence (AI) and distributed computation. These logics, which were imported directly from analytic philosophy first to AI and then to other areas of computer science, describe the behaviour of machines in terms of notions such as knowledge and belief.

In computer science, these notions are actually given precise computational meanings, and are used not only to prove properties of distributed system, but to program them as well. A typical rule in such systems that are knowledge based is: if processor A does not know that processor B has received its message, then processor A will not send the next message.

AOP arguments these logics with formal notions of choices, capabilities, commitment, and possibly others. A typical rule in the resulting system would be: if agent A knows that agent B has chosen to do something harmful to agent A, then A will request that B change its choice.

Shoham also considers AOP as a rigorous implementation of a fragment of direct-speech-act theory. Speech-act theory categorizes speech, distinguishing between informing, requesting, offering and so on. Speech-act theory has been applied in AI, in natural language research as well as in plan recognition.

Agent-oriented programming paradigm is the result of making engineering sense

out of abstract concepts such as knowledge, beliefs and commitments. The AOP framework includes three primary components:

- A restricted formal language with clear syntax and semantics for describing mental state.
- An interpreted programming language in which to program agents, with primitive commands such as ***request*** and ***inform***. The semantics of the programming language will derive from the semantics of mental state.
- A compiler from the agent-level language to an abstract model of processes.

6.1. Time

The basis for a language is a simple temporal language. Shoham particularly adopts explicit-time point-based logic. Time is a linear order and integers are assumed as a model for time. With each time point, a time line associates a set of propositions, those that are true at that time.
Let’s assume the following:

- ***TC*** is a set of time point constants,
- ***P*** is a set of predicate symbols, each with a fixed arity ≥ 0,
- ***F*** is a set of function symbols, each with a fixed arity ≥ 0,
- ***V*** is a set of variables.

The set of terms is defined as follows: a variable is a term, and if f is an n-ary function symbol and $trm_1, \cdots, trm_n$ are terms then $f(trm_1,\cdots,trm_n)$ is also a term. The set of well-formed formulas is then defined: if $t_1, t_2 \in TC$ then $t_1 < t_2$ is a well-formed formula (wff). If t is a time point symbol, r is an n-ary predicate symbol, and $trm_1,\cdots,trm_n$ are terms, then $f(trm_1,\cdots,trm_n)^t$ is a wff. If φ and ψ are wffs and v is a variable, then $\varphi \wedge \psi$, $\neg\varphi$ and $\forall v\,\varphi$ are all wffs.

The models for sentences are conventional: time point constants are interpreted as integers, variables and 0-ary function symbols as objects, other function symbols as functions from time and objects to objects, and predicate symbols as functions from time to tuples of objects.

6.2. Belief

Language is now augmented with a modal operator B, denoting belief. The most common logic of belief consists of the axioms of propositional calculus and inference rules, as well as following axioms:

$$B\varphi \wedge B(\varphi_1 \supset \varphi_2) \supset B\varphi_2$$
$$B\varphi \equiv BB\varphi$$
$$\neg B\varphi \equiv B\neg B\varphi$$
$$\neg B(\varphi \wedge \neg\varphi)$$

The logic gets a bit more complicated when two more arguments are added to the operator: the agent doing the believing and the time of belief.
Given the temporal logic mentioned before, we assume another set of constants – ***AG***, the agent constants. One wff-formation rule is now added: if t is a time-point constant, a is an agent constant, and φ is a sentence in the language, then

$B^t(a, \varphi)$ is also a sentence in the language. For example, $B^3(a,B^{10}(b, like(a,b)^7))$ will mean that at time 3, agent a believes that at time 10 agent b will believe that at time 7 a liked b.
Operator B has following properties:

$$B^t(a, \varphi_1) \wedge B^t(a, \varphi_1 \supset \varphi_2) \supset B^t(a, \varphi_2)$$
$$B^t(a, \varphi) \supset B^t(a, B^t(a, \varphi))$$
$$\neg B^t(a, \varphi) \supset B^t(a, \neg B^t(a, \varphi))$$
$$\neg B^t(a, \varphi \wedge \neg \varphi)$$

A B-structure is a tuple (L, m) where L is a set of time lines, and m is a function that specifies for every time point and agent an accessibility relation $R_{Bt, a}$ on L. Each such accessibility relation is transitive and Euclidian, and B is defined to be the necessity operator for this modality.

6.3. Commitment

The temporal logic and the logic of belief are standard, but when it comes to commitment, a new modal operator is introduced – CMT. Unlike B, CMT is a ternary operator: $CMT(a,b, \varphi)$ will mean that agent a is committed to agent b about φ.

What Shoham wants us to notice is that the agent is committed about the truth of a sentence, not about his taking action. For example, rather than say that the robot is committed to taking the action ‘raise arm’ at time t, we will say that the robot is committed to the proposition ‘the robot raises its arm at time t’. Since actions are a natural concept, in the actual programming language they will be introduced as syntactic sugar.

The syntax of the language is further augmented as follows: if a and b are agent

terms, t is a temporal term and φ is a sentence, then $CMT^t(a,b,\varphi)$ is also a sentence. Properties of the CMT operator are following:

$$CMT^t(a,b,\varphi_1) \wedge CMT^t(a,b,\varphi_1 \supset \varphi_2) \supset CMT^t(a,b,\varphi_2)$$
$$CMT^t(a,b,\varphi) \supset CMT^t(a,b,CMT^t(a,b,\varphi))$$
$$\neg CMT^t(a,b,\varphi \wedge \neg\varphi)$$

A B-CMT structure is the result of adding a second function to a B-structure, $R_{CMTt,a,b}$. This function specifies for each time point t and each ordered pair of agents a,b a second accessibility relation on time lines. Each such accessibility relation is transitive and serial, and CMT is defined to be the necessity operator of this modality.

Beliefs and commitment must not only be internally consistent, they must also be mutually consistent. We assume that an agent is completely aware of his commitments:

$$CMT^t(a,b,\varphi) \equiv B^t(a,CMT^t(a,b,\varphi))$$
$$\neg CMT^t(a,b,\varphi) \equiv B^t(a,\neg CMT^t(a,b,\varphi))$$

An agent is not necessarily aware of which commitments are made to him, only of commitment he has toward others.

We also assume that agents only commit in good faith:

$$CMT^t(a,b,\varphi) \supset B^t(a,\varphi)$$

Commitments must be consistent:

$$CMT^t(a,b,\varphi) \supset \neg CMT^t(a,c,\neg\varphi)$$

The freedom to choose is central to the notion of agenthood. The definition of commitment provides an alternative: choice is defined to be simply commitment to oneself:

$$CH^t(a, \varphi) =_{def} CMT^t(a, a, \varphi)$$

6.4. Capability

Capability is also bound to the notion of agenthood. We may choose to do something, but we will not do it if we are not capable of it. Unlike previous notions, capability is not a purely internal property of the agent. Philosophically, capability is even completely dissociated from mental state. But in this case, the notion will be viewed as a relation between the agent's mental state and the world.

The intuition behind the following definition is that 'to be able to X' means to have power to make X true by merely choosing that it be so:

$$CAN^t(a, \varphi) =_{def} CH^t(a, \varphi) \supset \varphi$$

The definition has some intuitive properties, such as:

$$CMT^t(a, b, \varphi) \supset B^t(a, CAN^t(a, \varphi))$$

6.5. Persisting mental state

Mental states can change over time, for example, belief. Axioms shown so far allow models in which at one time an agent believes no propositional sentences

but tautologies, at the next time he has a belief about every sentence, and at the time following that he is again agnostic. We have the intuition that beliefs tend to be more stable than that.

We will assume that agents have perfect memory of and faith in their beliefs, and only let go of a belief if they learn a contradictory fact. Therefore, beliefs persist by default. Furthermore, we assume that the absence of belief also persists by default, in a slightly different sense. If an agent does not believe a fact at a certain time, then the only reason he will come to believe it is if he learns it. Shoham used the following sentences to formally capture these two kinds of default persistence:

$$B^t(a,\varphi) \wedge \neg LEARN^t(a,\neg\varphi) \supset B^{t+1}(a,\varphi)$$
$$\neg B^t(a,\varphi) \wedge \neg LEARN^t(a,\varphi) \supset \neg B^{t+1}(a,\varphi)$$

Commitments should too persist. Although commitments persist by default, there are conditions under which commitments are revoked. These conditions include explicit release of the committer by the committee, or alternatively a realization on the part of the committer of the impossibility of the commitment. The basic sentence used is:

$$CMT^t(a,b,\varphi) \wedge \neg REVOKE^t(a,b,\varphi) \supset CMT^{t+1}(a,b,\varphi)$$

Since choice is defined in terms of commitment, it inherits the default persistence. However, while an agent cannot revoke commitments he made to others, he can cancel commitments that were made to him, including commitments made to himself, namely choices. Therefore, an agent can freely modify his choices.

Capabilities could be considered fixed: what an agent can do at one time, it can

do at any other time.

6.6. Programming of agents

Each agent is controlled by his own, private program. Agent programs are similar to standard programs, containing primitive operations, control structures and input-output instructions. What is unique about these programs is that the control structures refer to the mental-state constructs and the IO commands include methods for communicating with other agents.

In Shoham (1990), a hypothetical agent language was described, called AGENT0. What AGENT0 has in common with conventional programming languages is that the syntax defines the primitive operations available, and the programmer specifies which of those are to be carried out and in which order. However, in regular languages, there is a simple mapping between the structure of the program and the order of execution, usually a linear command sequence translates to the same execution order. In contrast, with AGENT0 there is complete decoupling between the time a command is issued and the time at which it is executed. An agent is continually engaged in two types of activity: making commitments about the future and honouring previous commitments whose execution time has come.

Fact statements are fundamental to AGENT0 since they are used to specify the content of actions as well as conditions for their execution. Fact statements are sentences in the temporal-modal logic in which the mental states of agents are defined. They can be simple facts about the world – $[t, employee(smith, acme)]$ – or

they may refer to the mental state of agents –$[t^{'}, \neg B(a,[t,employee(smith,acme)])]$.

Agents commit to action and there are two orthogonal distinctions between types of action: action may be ***private*** or ***communicative***, and they may be ***conditional*** or ***unconditional***. First, let's look at the unconditional action statements.

The syntax of private actions is

```
DO(t,p - action)
```

where t is a time point and $p - action$ is a name of a private action. Private actions are analogous to the implementation of specific methods in OOP. These actions may be invisible to other agents, but need not be so.

Private actions may or may not involve IO, but communicative actions always involve IO. Communicative actions are uniform and common to all agents, unlike private actions. AGENT0 has three types of communicative action: informing, requesting, and cancelling a request.

The syntax of informing is

```
INFORM(t,a,fact)
```

where t is a time point at which the informing is to take place, a is an agent name and $fact$ is a fact statement that itself contains other temporal information.

The syntax of requesting is

```
REQUEST(t,a,action)
```

where t is a time point, a is an agent name and action is an action statement, defined recursively. If we take the code below as an example, we can distinguish between the time at which the requesting is to be done – 1 – and the time of the requested action – 10.

```
REQUEST(1,a,DO(10,update-database))
```

Requests can also be further embedded, as in the code below

```
REQUEST(1,a,REQUEST(5,b,INFORM(10,c,fact)))
```

The syntax of cancelling a request is

```
UNREQUEST(t,a,action)
```

where t is a time point, a is an agent name, and action is an action statement.

The last unconditional action in AGENT0 has the following syntax

```
REFRAIN action
```

This is actually a nonaction where action is an action statement which does not itself contain a REFRAIN. The role of refraining is preventing commitment to other actions.

Next are conditional action statements. In AGENT0, we can distinguish commitments for conditional actions, which include conditions to be tested right

before acting, from conditions for making the commitment to act in the first place.

Conditional actions rely on one form of condition, called a ***mental condition***, while conditions for making a commitment include both mental conditions and ***message conditions***. Mental conditions refer to the mental state of the agent and the intuition behind them is that when the time comes to execute the action, the mental state at that time will be examined to see whether the mental condition is satisfied.

A mental condition is a logical combination of ***mental patterns***. A mental pattern is one of two pairs

```
(B,fact) or ((CMT,a),action)
```

where `fact` is a fact statement, `a` is an agent name, and `action` is an action statement. An example would be

(B,employee(smith,acme))

The syntax of a conditional action is now

```
IF mntlcond THEN actior
```

where `mntlcond` is a mental condition and `action` is an action statement. For example, a conditional action is

IF(B,employee(smith,acme))THEN INFORM(t,a,employee(smith,acme))

This can be read as: if at time t you believe that smith is an employee of acme, then at that time inform agent a of that fact.

In AGENT0, procedures are invoked in a pattern-directed fashion. To be specific, commitment rules are activated on certain patterns in the incoming messages and current mental state. Variables play a crucial role in these patterns.

A variable is denoted by the prefix '?' and it can substitute agent name, fact statement or action statement. The following is thus a legitimate conditional action:

IF NOT ((CMT,?x),REFRAIN action) THEN action

Variables in action statements are interpreted as existentially quantified, in the tradition of logic programming. The scope of the quantifier is upwards until the scope of the first NOT, or it is the entire statement if the variable does not lie in the scope of a NOT. As a result, the statement above can informally be read as: if you are not currently committed to anyone to refrain from action, then take that action.

The universally quantified variables are denoted by the prefix '?!'. The scope of these variables is always the entire formula. For example, the conditional action

IF (B(emp(?!x,acme)) THEN INFORM(a,emp(?!x,acme))

results in informing a of all the individuals who the agent believes to be acme

employees.

Besides action statements, there is another crucial layer of abstraction in AGENT0. Most of the action statements are unknown at programming time and are, instead, later communicated by other agents. The program itself merely contains conditions under which the agent will be committed to actions.

The conditions under which a commitment is made include both mental conditions and message conditions, which refer to the current incoming messages. A message condition is a logical combination of ***message patterns*** – a triple

(From,Type,Content)

where From is the sender's name, Type is INFORM or REQUEST, and Content is a fact statement or an action statement, depending on the type.

An example of a message pattern is

(a,INFORM,fact)

meaning that one of the new messages is from a, informing the agent of fact. A more complex message condition is

(a,REQUEST,DO(t,walk)) AND NOT (?x,REQUEST,DO(t,chew-gum))

meaning that there is a new message from a requesting the agent to walk, and

there is no new request from anyone that the agent chew - gum.

The syntax of a commitment statement is then

```
IF MSGCOND:msgcond AND MNTLCOND:mntlcond
THEN COMMIT TO agent ABOUT actio
```

where msgcond and mntlcond are respectively message and mental conditions, agent is an agent name and action is an action statement.

A more concise syntax for commitment statements can be used:

```
COMMIT(msgcond,mntlcond,agent,action)
```

An example of a commitment statement is as follows

COMMIT((?a,REQUEST,?action),

(B,myfriend(?a)),

?a,

?action)

To conclude, a program is simply a sequence of commitment statements, preceded by a definition of the agent's capabilities and initial beliefs.[152]

Bibliography

1. LOUDEN, K. C., LAMBERT, K. A. (2011) ***Programming Languages Principles and Practice.*** 3rd Edition. Cengage Learning

2. GABBRIELLI, M., MARTINI, S. (2010) ***Programming Languages: Principles and Paradigms.*** Springer

3. ABRAHAMS, D., GURTOVOY, A. (2004) ***C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.*** Addison-Wesley

4. DOS REIS, G., JÄRVI, J. (2005) ***What is Generic Programming?*** [Online] Available at: http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf [Accessed: May 25th 2015]

5. VISSER, E. (2002) ***Meta-Programming with Concrete Object Syntax*** [Online] Available at: http://dspace.library.uu.nl/bitstream/handle/1874/23952/visser_02_metap [Accessed: May 28th 2015]

6. WARD, M. P. (1994) ***Language Oriented Programming*** [Online] Available at: http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf

7. SHOHAM, Y. (1990) ***Agent-Oriented Programming*** [Online] Available at: http://cife.stanford.edu/sites/default/files/TR042.pdf

[1] Louden and Lambert, 2011, p. 3-8

[2] Sebesta, 2012, p. 21

[3]

[4] Eastern Mediterranean University, 2015

[5] Eastern Mediterranean University, 2015

[6] Niklaus Wirth is a Swiss computer scientist born February 15, 1934 in Winterthur, Switzerland

[7] Eastern Mediterranean University, 2015

[8] Hofstedt, 2011, p. 21

[9] Eastern Mediterranean University, 2015

[10] Eastern Mediterranean University, 2015

[11] Dowek, 2009, p. 1

[12] Open Book Project, 2015

[13] Dowek, 2009, p. 3

[14] Malik, 2012, p. 55

[15] Dowek, 2009, p. 3-5

[16] Dowek, 2009, p. 6

[17] Malik, 2012, p. 290

[18] Oracle, 2015

[19] Dowek, 2009, p. 7

[20] Eastern Mediterranean University, 2015

[21] Tutorialspoint, 2015

[22] Paper: Böhm, C., Jacopini, G. (1966) Flow diagrams, Turing machines and languages with only two formation rules***. Communications of the ACM***. 9 (5). p. 366–371.

[23] Scribd, 2015, p. 4

[24] Letter: Dijkstra, E. W. (1968) Letters to the editor: Go to statement considered harmful. ***Communications of the ACM.*** 11 (3). p. 147–148.

[25] Scribd, 2015, p. 5

[26] Scribd, 2015, p. 6-8

[27] Gabbrielli and Martini, 2010, p. 151
[28] Stern and Stern, 1991, p. 13
[29] Scribd, 2015, p. 9
[30] Fill, 2009
[31] Gabbrielli and Martini, 2010, p. 72
[32] Stern and Stern, 1991, p. 133-138
[33] Gabbrielli and Martini, 2010, p. 136-138
[34] Gabbrielli and Martini, 2010, p. 140-150
[35] Encyclopedia, 2015
[36] Study, 2015
[37] Tutorialspoint, 2015
[38] Tutorialspoint, 2015
[39] Tutorialspoint, 2015
[40] Prinz and Crawford, 2005, p. 96
[41] Prinz and Crawford, 2005, p. 103
[42] Prinz and Crawford, 2005, p. 99
[43] Prinz and Crawford, 2005, p. 103-105
[44] Study, 2015
[45] Wikipedia, 2015
[46] McCarthy, John (June 1978). "History of Lisp". ***In ACM SIGPLAN History of Programming Languages Conference***: 217–223. doi:10.1145/800025.808387
[47] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. Journal of the Association for Computing Machinery 24(1):44–67 (1977)
[48] Wikipedia, 2015
[49] Dick Pountain. "Functional Programming Comes of Age". ***BYTE.com (August 1994)***. Retrieved August 31, 2006.
[50] Wikipedia, 2015
[51] Wikipedia, 2015

[52] Louden and Lambert, 2011, p. 90

[53] Mitchell, 2002, p. 57-66

[54] Wikipedia, 2015;Goetz et al. ***Java Concurrency in Practice***. Addison Wesley Professional, 2006, Section 3.4. Immutability

[55] Backfield, 2014, p. 43

[56] "How to create Immutable Class and Object in Java - Tutorial Example". Javarevisited.blogspot.co.uk. 2013-03-04. Retrieved 2014-04-14.

[57] "Immutable objects". javapractices.com. Retrieved November 15, 2012.

[58] Wikipedia, 2015

[59] Wikipedia, 2015 – nje našla som toto

[60] Backfield, 2014, p. 16

[61] Warburton, 2014, p. 7

[62] Sturm, 2011, p. 24

[63] MSDN, 2015

[64] Wikipedia, 2015

[65] Wikipedia, 2015

[66] Backfield, 2014, p. 25

[67] Sturm, 2011, p. 78-80

[68] F# for fun and profit, 2012

[69] Sturm, 2011, p. 117

[70] Syme, Granicz and Cisternino, 2012, p. 28-30

[71] Wikipedia, 2015

[72] Neward, Erickson, Crowell, and Minerich, 2011, p. 219

[73] Backfield, 2014, p. 67

[74] Backfield, 2014, p. 73

[75] Backfield, 2014, p. 77

[76] Backfield, 2014, p. 93

[77] Syme, Granicz, and Cisternino, 2012, p. 34-37

[78] Backfield, 2014, p. 113

[79] TechnologyUK, 2015

[80] TechnologyUK, 2015

[81] TechnologyUK, 2015

[82] Malik, 2012, p. 760

[83] Liang, 2011, p. 534-554

[84] Oracle, 2015

[85] Oracle, 2015

[86] Sutherland, 1963

[87] Holmevik, 1994

[88] Kay, 1993

[89] Wu, 2009, p. 17

[90] Malik, 2012, p. 114-118

[91] Oracle, 2015

[92] Malik, 2012, p. 384-389

[93] Malik, 2012, p. 391-393

[94] Malik, 2012, p. 411

[95] Malik, 2012, p. 414

[96] Malik, 2012, p. 393-397

[97] Malik, 2012, p. 407

[98] Malik, 2012, p. 404

[99] Malik, 2012, p. 427-429

[100] Malik, 2012, p. 467-478

[101] Malik, 2012, p. 501-507

[102] Malik, 2012, p. 512-517

[103] Malik, 2012, p. 640-649

[104] Malik, 2012, p. 657-662

[105] The Java class ***Object*** can be viewed in detail on the following website: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

[106] Malik, 2012, p. 664-670

[107] Malik, 2012, p. 674

[108] Wu, 2009, p. 758

[109] Malik, 2012, p. 683

[110] Sharan, 2014, p. 667, 685

[111] Gregoire, Solter, and Kleper, 2012, p. 91

[112] Saylor, 2015

[113] Lloyd (1994)

[114] UNSW School of Computer Science and Engineering (2015)

[115] Mitchell, 2002, p. 475

[116] Hudak, 1989

[117] Burstall and Darlington, 1977

[118] More on the topic can be found on the following link:
http://en.wikipedia.org/wiki/Side_effect_%28computer_science%29

[119] Jones and Wadler, 1993

[120] Mitchell, 2002, p. 476

[121] Sebesta, 2012, p. 734

[122] Gabbrielli and Martini, 2010, p. 369

[123] Gabbrielli and Martini, 2010, p. 371-383

[124] Louden and Lambert, 2011, p. 116-119

[125] Louden and Lambert, 2011, p. 121-126

[126] Taylor, 2011, p. 52

[127] Hopcroft and Ullman, 1979, p. 13

[128] Hopcroft and Ullman, 1979, p. 16

[129] Hopcroft and Ullman, 1979, p. 28

[130] Belzer, Holzman and Kent, 1975

[131] Aho and Ullman, 1973

[132] IETF Tools, 2015

[133] Walther, 2009, p. 9-11

[134] LINFO, 2006

[135] HTML Goodies, 2015

[136] AngularJS Documentation, 2015

[137] UCF Department Of Electrical Engineering & Computer Science, 2015

[138] Louden and Lambert, 2012, p. 8-14
[139] Latentflip, 2013
[140] Sebesta, 2012, p. 717
[141] Mitchell, 2002, p. 80
[142] Sebesta, 2012, p. 717-719
[143] Gabbrielli and Martini, 2010, p. 415
[144] Louden and Lambert, 2011, p. 19
[145] Dos Reis and Järvi, 2005, p. 1
[146] Dos Reis and Järvi, 2005, p. 1
[147] Dos Reis and Järvi, 2005, p. 2
[148] Abrahams and Gurtovoy, 2004, p. 19
[149] Visser, 2002, p. 1
[150] Abrahams and Gurtovoy, 2004, p. 23
[151] Ward, 1994, p.2
[152] Shoham, 1990, p. 4-32