Programming language during technology evolution time (1990-2015)

Prepared By

Dr. Sameer Bawaneh

computing and technology department
Sameer.b@outlook.com

Abstract– In my paper research I will be talking briefly about the significant development in programming language during the period of technology evolution and internet time, I will classify this period into two main milestones the first milestone will cover the period 1990s (1990-1999) and the second will cover the last 15 years (2000 – 2015), I will be more focusing on the second milestone and specially the main changes and development tolls in most popular languages this days and make comparison between the mostly used programming language in this year along with the market share for each one.

I. Introduction

While there were instances of different forms of computer programming as far back as ancient Greece (the Antikythera mechanism) it was not until the 1880s that something resembling modern programming took place. During this period, Herman Hollerith, invented a type of data recording program that could be read by machines. By utilizing a series of punched cards, known as Hollerith cards, data could be recorded by hand and read by a machine, which allowed for the invention of the tabulator and keypunch machines. More modern figures in programming include Larry Wall, inventor of Perl, Bjarne Stroustrup, C++, and James Gosling, Java. These people have made modern programming possible, and are ushering in a new technological age.

Due to the fast growth of the Internet in the mid-1990s as this revolution happening—a startling and amazing revolution that is altering everything from our traditional views to how people can and should communicate with each other, the new technologies in the Hardware and Internet and all technology fields created an great opportunity for new programming languages to be adopted and to be develop. And it changes the measurement factors and ideas for programming language.

II. First millstone (1990-1999)

In this period of time three factors effected the development of programming language and it open a new path for programming language challenges and competitions, herein the list of the major specifications for programming language for this period:

Internet: most of the programming language has been affected by the fast growth of the Internet in the mid-1990s as it was major significant factor affected the major programming languages. This revolution happening—a startling and amazing revolution that is altering everything from our

Traditional views to how people can and should communicate with each other, the Internet created an great opportunity for new languages to be adopted. In particular, the JavaScript programming language increased to popularity because of its early integration with the Netscape Navigator web browser. Also there is a different scripting languages achieved common use in developing customized applications for web servers such as PHP

Recombination and maturation of old ideas. This period began the increase of the functional languages which is a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming model, which means programming is done with expressions

Catch the technology fast growth and to follow and achieve the rapid development many of "rapid application development" (RAD) languages appeared, is both a general term used to refer to alternatives to the conventional waterfall model of software development as well as James Martin's approach to have more rapid development. In general, RAD approaches to software development put less emphasis on planning tasks and more emphasis on development. In contrast to the waterfall model, which emphasizes exact specification and planning, this causes RAD to use prototype in addition to or even sometimes in place of design specifications. RAD approaches also emphasize a flexible process that can adapt as the project evolves rather than thoroughly defining specifications and plans correctly from the start. In addition to James Martin's RAD methodology, other approaches to rapid development include Agile methods and the spiral model. Which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java.

Java was the most considerable language at the end of this period in particular as it is more radical and innovative than the RAD languages were the new scripting languages. These did not directly move down from other languages and featured

new syntaxes and more open-minded integration of features. Many consider these scripting languages to be more productive than even the RAD languages, but frequently because of choices that make small programs simpler but large programs more difficult to write and maintain

Here is list for some of programming language released in this period with few details:

TABLE I
Released programming languages between 1990-1999

Year	Name	Predecessor(s)
1990	AMOS BASIC	STOS BASIC
1990	Object Oberon	Oberon
1991	GNU E	C++
1991	Oberon-2	Object Oberon
1991	Python	ABC, ALGOL 68, Icon, Modula-3
1991	Oz	Prolog
1991	Visual Basic	QuickBASIC
1992	Borland Pascal	Turbo Pascal OOP
1992	Dylan	Common Lisp, Scheme
1993	AppleScript	HyperTalk
1993	NewtonScript	Self, Dylan
1994	Claire	Smalltalk, SETL, OPS5, Lisp, ML, C, LORE, LAURE
1994	ANS Forth	Forth
1995	Ada 95	Ada 83
1995	Borland Delphi	Borland Pascal
1995	Java	C, Simula 67, C++, Smalltalk, Ada 83, Objective-C, Mesa
1995	LiveScript	Self, C, Scheme
1995	PHP	Perl
1995	Ruby	Smalltalk, Perl
1995	JavaScript	LiveScript
1996	Curl	Lisp, C++, Tcl/Tk, TeX, HTML
1997	ECMAScript	JavaScript
1997	Tea	Java, Scheme, Tcl
1998	Standard C++	C++, Standard C, C
1998	UnrealScript	C++, Java
1999	XSLT (+ XPath)	DSSSL

III. SECAND MILLSTONE (2000-2015)

As technology and science evolve, programmers and programming languages have had to grow and evolve along with it. Most of these changes have been influenced by demand by the consumers. As computers and technology became more popular, consumers needed and wanted easier to use technology. This tasked programmers to create new ways of writing software with new challenges, come up with languages to make it easier to create better and easier software to use and more focus in all programming language aspects which could help to do better performance, reliability, and others aspects and below is the most common trends which was the development area factors for programming languages on this particular period of time.

1) Increasing support for functional programming in mainstream languages used commercially, including pure functional programming for making code easier to reason about and easier to parallelise (at both micro- and macro- levels)
2) Constructs to support concurrent and distributed programming.
3) Mechanisms for adding security and reliability verification to the language: extended static checking, dependent typing, information flow control, static thread safety.
4) Alternative mechanisms for modularity: mixins, delegates, aspects.
5) Component-oriented software development.
6) Metaprogramming, reflection or access to the abstract syntax tree
7) Increased emphasis on distribution and mobility.
8) Integration with databases, including XML and relational databases.
9) Support for Unicode so that source code (program text) is not restricted to those characters contained in the ASCII character set; allowing, for example, use of non-Latin-based scripts or extended punctuation.
10) XML for graphical interface (XUL, XAML).
11) Open source as a developmental philosophy for languages, including the GNU compiler collection and recent languages such as Python, Ruby, and Squeak.
12) AOP or Aspect Oriented Programming allowing developers to code by places in code extended behaviors.
13) Massively parallel languages for coding 2000 processor GPU graphics processing units and supercomputer arrays including OpenCL

During this period of time most of the modern programming languages start pay more attention to the quality requirements and new measurements factors appeared to evaluate each and every part in the programming language so we founds there are more aspects that programming developers focus on to be met in terms of quality and market competitions in order for it to be more sufficient and usable. The most important quality aspects include reliability, is robustness, usability, portability, maintainability and efficiency and performance. This is guide to more focus on the algorithms, programming mistakes and logic errors. The robustness factor is how well problems are anticipated by the program that is not due to errors by the programmer. And make usability factor is simply how easy it is for users to make use of the program.

Portability is determines by the various platforms that the program can be complied and run on. This includes operating system and computer hardware platforms. Whereas the maintainability is the ease of modification and handling that can be done by future programmers should the need arise. Other programmers who use this program should be able to easily change and update the code. The efficiency and performance is referring to the amount of resources a program consumes.

Readability of Source Code

Most of the programming languages companies tried their best to concentrate more and more on the readability of the source code as it is a factor that has a direct affect on the level of quality, some of which include usability, portability and maintainability. The readability is how easily a human reader can understand the purpose, flow and operation of the code. This is specifically important due to the fact that most of the time that programmers spend is on reading, understanding and modifying existing codes. When the source code is unreadable, it can lead to duplicated codes, bugs and inefficiencies. So in this period of time more enhancements and updates for the main factors that contribute to a code's readability level include naming conventions for objects, decomposition, comments and different indentation styles.

Algorithmic Complexity

More involvement to engineering practice and academic field in the programming language developments that are both concerned a great deal with the discovery and implementation of a problem's algorithms. The algorithms that are used are classified into orders that express resource use. The orders are classified by Big O notation, which expresses memory consumption or execution time try to reach to the best level of maturity.

Methodologies

When speaking in terms of software development, most formal processes often require analysis as the first step. After analysis comes testing, implementation and failure elimination. A wide variety of approaches appeared for accomplishing each of these tasks, one of which is the Use Case analysis. Some of the more popular modeling techniques include the Unified Modeling Language, Model-Driven Architecture and Object-Oriented Analysis and Design. So now a day's most of the new philosophy of programming language to have creative and better methodologies compare to others.

TABLE II
Released programming languages between 2000-2015

Year	Name	Predecessor(s)
2000	Join Java	Java
2000	XL	Ada, C++, Lisp
2000	C#	C, C++, Java, Delphi, Modula-2
2001	Processing	Java, C, C++[8]
2001	Visual Basic .NET	Visual Basic
2003	Squirrel	Lua
2004	Boo	Python, C#
2004	Groovy	Java
2006	Cobra	Python, C#, Eiffel, Objective-C
2006	Windows PowerShell	C#, ksh, Perl, CL, DCL, SQL
2006	OptimJ	Java
2007	Ada 2005	Ada 95
2007	Fantom	C#, Scala, Ruby, Erlang
2007	Vala	C#
2008	Genie	Python, Boo, D, Object Pascal
2008	Pure	Q
2009	Go	C, Oberon, Limbo
2009	CoffeeScript	JavaScript, Ruby, Python, Haskell
2011	Ceylon	Java
2011	Dart	Java, JavaScript, CoffeeScript, Go
2011	Elm	Haskell, Standard ML, OCaml, F#
2011	Kotlin	Java, Scala, Groovy, C#, Gosu
2011	C++11	C++, Standard C, C
2012	Elixir	Erlang, Ruby, Clojure
2012	TypeScript	JavaScript, CoffeeScript
2012	Ada 2012	Ada 2005, ISO/IEC 8652:1995/Amd 1:2007
2013	Purescript	JavaScript, Haskell
2014	Hack	PHP
2014	Swift	Objective-C, Rust, Haskell, Ruby, Python, C#, CLU
2014	C++14	C++, Standard C, C

In this part iam going to discuss deeply the main feature for the existing programming language in the market tell the date of preparing this research, moreover, at the end of this research I will include a comparison between most popular programming languages.

A. Object-Orientation

Many languages claim to be Object-Oriented. While the exact definition of the term is highly variable depending upon who you ask, there are several qualities that most will agree an Object-Oriented language should have:

1) Encapsulation/Information Hiding
2) Inheritance
3) Polymorphism/Dynamic Binding
4) All pre-defined types are Objects
5) All operations performed by sending messages to Objects
6) All user-defined types are Objects

For the purposes of this discussion, a language is considered to be a "pure" Object-Oriented languages if it satisfies all of these qualities. A "hybrid" language may support some of these qualities, but not all. In particular, many languages support the first three qualities, but not the final three.

So how do our languages stack up?

TABLE III
Comparison between some programming languages for that most Object-Oriented qualities

criteria / Programming Language	Encapsulation / Information Hiding	Inheritance	Polymorphism / Dynamic Binding	All pre-defined types are Objects	All operations are messages to Objects	All user-defined types are Objects
Eiffel	Yes	Yes	Yes	Yes	Yes	Yes
Smalltalk	Yes	Yes	Yes	Yes	Yes	Yes
Ruby	Yes	Yes	Yes	Yes	Yes	Yes
Java	Yes	Yes	Yes	No	No	No
C#	Yes	Yes	Yes	No	No	Yes
C++	Yes	Yes	Yes	No	No	Yes
Python	No	Yes	Yes	Yes	No	Yes
Perl	Yes	Yes	Yes	No	No	No
Visual Basic	Yes	No	Yes	No	No	No

B. Static vs. Dynamic Typing

The debate between static and dynamic typing has raged in Object-Oriented circles for many years with no clear conclusion. Proponents of dynamic typing contend that it is more flexible and allows for increased productivity. Those who prefer static typing argue that it enforces safer, more reliable code, and increases efficiency of the resulting product.

It is futile to attempt to settle this debate here except to say that a statically-typed language requires a very well-defined type system in order to remain as flexible as its dynamically-typed counterparts. Without the presence of genericity (templates, to use the C++ patois) and multiple type inheritance (not necessarily the same as multiple implementation inheritance), a static type system may severely inhibit the flexibility of a language. In addition, the presence of "casts" in a language can undermine the ability of the compiler to enforce type constraints.

A dynamic type system doesn't require variables to be declared as a specific type. Any variable can contain any value or object. Smalltalk and Ruby are two pure Object-Oriented languages that use dynamic typing. In many cases this can make the software more flexible and amenable to change. However, care must be taken that variables hold the expected kind of object. Typically, if a variable contains an object of a different type than a user of the object expects, some sort of "message not understood" error is raised at run-time. Users of dynamically-typed languages claim that this type of error is infrequent in practice.

Statically-typed languages require that all variables are declared with a specific type. The compiler will then ensure that at any given time the variable contains only an object compatible with that type. (We say "compatible with that type" rather than "exactly that type" since the inheritance relationship enables subtyping, in which a class that inherits from another class is said to have an IS-A relationship with the class from which it inherits, meaning that instances of the inheriting class can be considered to be of a compatible type with instances of the inherited class.) By enforcing the type constraint on objects contained or referred to by the variable, the compiler can ensure a "message not understood" error can never occur at run-time. On the other hand, a static type system can hinder evolution of software in some circumstances.

C. Generic Classes

Generic classes, and more generally parametric type facilities, refer to the ability to parameterize a class with specific data types. A common example is a stack class that is parameterized by the type of elements it contains. This allows the stack to simultaneously be compile-time type safe and yet generic enough to handle any type of elements.

The primary benefit of parameterized types is that it allows statically typed languages to retain their compile-time type safety yet remain nearly as flexible as dynamically typed languages. Eiffel in particular uses generics extensively as a mechanism for type safe generic containers and algorithms. C++ templates are even more flexible, having many uses apart from simple generic containers, but also much more complex.

D. Inheritance

Inheritance is the ability for a class or object to be defined as an extension or specialization of another class or object. Most object-oriented languages support class-based inheritance, while others such as SELF and JavaScript support object-based inheritance. A few languages, notably Python and Ruby, support both class- and object-based inheritance, in which a class can inherit from another class and individual objects can be extended at run time with the capabilities of other objects. For the remainder of this discussion, we'll be dealing primarily with class-based inheritance since it is by far the most common model.

Although commonly thought of as simple subtyping mechanism, there are actually many different uses of inheritance. In his landmark book Object-Oriented Software Construction, Bertrand Meyer identified and classified as many as 17 different forms of inheritance. Even so, most languages provide only a few syntactic constructs for inheritance which are general enough to allow inheritance to be used in many different ways.

The most important distinction that can be made between various languages' support for inheritance is whether it supports single or multiple inheritance. Multiple inheritance is the ability for a class to inherit from more than one super (or base) class. Visual Basic has no support for inheritance of any form, although support for single inheritance is slated for the VB .NET release.

E. Feature Renaming

Feature renaming is the ability for a class or object to rename one of its features (a term we'll use to collectively refer to attributes and methods) that it inherited from a super class. There are two important ways in which this can be put to use:

Provide a feature with a more natural name for its new context

Resolve naming ambiguities when a name is inherited from multiple inheritance paths

Java and C++ both support method overloading in a similar fashion. Complexities in the mechanism to disambiguate calls to overloaded methods have lead some language designers to avoid overloading in their languages. None of the other languages under consideration support method overloading. Default argument values provide a subset of the behavior for which method overloading is used, and some languages such as Ruby and Python have chosen this route instead.

F. Operator Overloading

Operator overloading (a hotly debated topic) is the ability for a programmer to define an operator (such as +, or *) for user-defined types. This allows the operator to be used in infix, prefix, or postfix form, rather than the standard functional form.

This second point is subtle. It means that given any operator, it must be possible to invoke that operator in functional

.

G. Higher Order Functions & Lexical Closures

Higher order functions are, in the simplest sense, functions that can be treated as if they were data objects. In other words, they can be bound to variables (including the ability to be stored in collections), they can be passed to other functions as parameters, and they can be returned as the result of other functions. Due to this ability, higher order functions may be viewed as a form of deferred execution, wherein a function may be defined in one context, passed to another context, and then later invoked by the second context. This is different from standard functions in that higher order functions represent anonymous lambda functions, so that the invoking context need not know the name of the function being invoked.

Lexical closures (also known as static closures, or simply closures) take this one step further by bundling up the lexical (static) scope surrounding the function with the function itself, so that the function carries its surrounding environment around with it wherever it may be used. This means that the closure can access local variables or parameters, or attributes of the object in which it is defined, and will continue to have access to them even if it is passed to another module outside of its scope.

H. Garbage Collection

Garbage collection is a mechanism allowing a language implementation to free memory of unused objects on behalf of the programmer, thus relieving the burden on the programmer to do so. The alternative is for the programmer to explicitly free any memory that is no longer needed. There are several strategies for garbage collection that exist in various language implementations.

Reference counting is the simplest scheme and involves the language keeping track of how many references there are to a particular object in memory, and deleting that object when that reference count becomes zero. This scheme, although it is simple and deterministic, is not without its drawbacks, the most important being its inability to handle cycles. Cycles occur when two objects reference each other, and thus there reference counts will never become zero even if neither object is referenced by any other part of the program. This is the scheme that is utilized by Python and Visual Basic, although in the case of Python an extra step is taken to ensure that cycles are handled appropriately.

I. Uniform Access

The Uniform Access Principle, as published in Bertrand Meyer's Object-Oriented Software Construction, states that "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." It is described further with "Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation. It follows from the Continuity criterion; you may also view it as a special case of Information Hiding."

J. Class Variables/Methods

Class variables and methods are owned by a class, and not any particular instance of a class. This means that for however many instances of a class exist at any given point in time, only one copy of each class variable/method exists and is shared by every instance of the class.

K. Reflection

Reflection is the ability for a program to determine various pieces of information about an object at run-time. This includes the ability to determine the type of the object, its inheritance structure, and the methods it contains, including the number and types of parameters and return types. It might also include the ability for determining the names and types of attributes of the object.

Most object-oriented languages support some form of reflection. Smalltalk, Ruby, and Python in particular have very powerful and flexible reflection mechanisms. Java also supports reflection, but not in as flexible and dynamic fashion as the others. C++ does not support reflection as we've defined it here, but it does supported a limited form of run-time type information that allows a program to determine the type of an object at run-time. Eiffel also has support for a limited form of reflection, although it is much improved in the most recent versions of Eiffel, including the ability to determine the features contained in an object.

L. Access Control

Access control refers to the ability for a modules implementation to remain hidden behind its public interface. Access control is closely related to the encapsulation/information hiding principle of object-oriented languages. Most object-oriented languages provide at least two levels of access control: public and protected. Protected features are not available outside of the class in which they are contained, except for subclasses. This is the scheme supported by Smalltalk, in which all methods are public and all attributes are protected.

M. Design by Contract

Design by Contract is another idea set forth by Bertrand Meyer and discussed at length in Object Oriented Software Construction as well as the Eiffel Home Page. In short, Design by Contract (DBC) is the ability to incorporate important aspects of a specification into the software that is implementing it. The most important features of DBC are:

Pre-conditions, which are conditions that must be true before a method is invoked

Post-conditions, which are conditions guaranteed to be true after the invocation of a method

Invariants, which are conditions guaranteed to be true at any stable point during the lifetime of an object

There is much more to DBC than these simple facilities, including the manner in which pre-conditions, post-conditions, and invariants are inherited in compliance with the Liskov Substitution Principle. However, at least these facilities must be present to support the central notions of DBC.

N. *Multithreading*

Multithreading is the ability for a single process to process two or more tasks concurrently. (We say concurrently rather than simultaneously because, in the absence of multiple processors, the tasks cannot run simultaneously but rather are interleaved in very small time slices and thus exhibit the appearance and semantics of concurrent execution.) The use of multithreading is becoming increasingly more common as operating system support for threads has become near ubiquitous.

O. Regular Expressions

Regular expressions are pattern matching constructs capable of recognizing the class of languages known as regular languages. They are frequently used for text processing systems as well as for general applications that must use pattern recognition for other purposes. Libraries with regular expression support exist for nearly every language, but ever since the advent of Perl it has become increasingly important for a language to support regular expressions natively. This allows tighter integration with the rest of the language and allows more convenient syntax for use of regular expressions.

P. Pointer Arithmetic

Pointer arithmetic is the ability for a language to directly manipulate memory addresses and their contents. While, due to the inherent un-safety of direct memory manipulation, this ability is not often considered appropriate for high-level languages, it is essential for low-level systems applications. Thus, while object-oriented languages strive to remain at a fairly high level of abstraction, to be suitable for systems programming a language must provide such features or relegate such low-level tasks to a language with which it can interact. Most object-oriented languages have foregone support of

pointer arithmetic in favor of providing integration with C.

Q. Language Integration

For various reasons, including integration with existing systems, the need to interact with low level modules, or for sheer speed, it is important for a high level language (particularly interpreted languages) to be able to integrate seamlessly with other languages. Nearly every language to come along since C was first introduced provides such integration with C. This allows high level languages to remain free of the low level constructs that make C great for systems programming, but add much complexity.

R. Built-In Security

Built-in security refers to a language implementation's ability to determine whether or not a piece of code comes from a "trusted" source (such as the user's hard disk), limiting the permissions of the code if it does not. For example, Java applets are considered untrusted, and thus they are limited in the actions they can perform when executed from a user's browser. They may not, for example, read or write from or to the user's hard disk, and they may not open a network connection to anywhere but the originating host.
Several languages, including Java, Ruby, and Perl, provide this ability "out of the box". Most languages defer this protection to the user's operating environment.

S. Capers Jones Language Level

The Capers Jones Language Level is a study that attempts to identify the number of source lines of code is necessary in a given language to implement a single function point. The higher the language level, the fewer lines of code it takes to implement a function point, and thus presumably is an indicator of the productivity levels achievable using the language.
The study is considered flawed by many since not every language was examined in detail. Some languages were assumed to be approximately equal to another language, and so the study at best represents an approximation. However, the study is thorough enough to determine ballpark estimates on the general productivity levels of the languages.

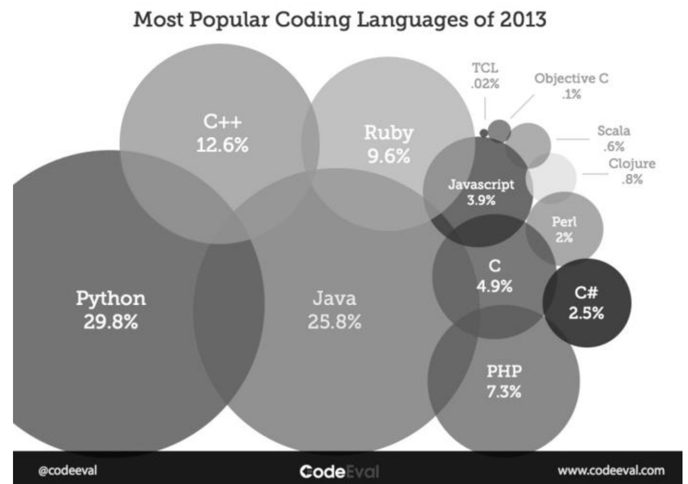
TABLE VI
Comparison between main programming languages and available feature

Programming Language / criteria	C#	C++	Python	Perl	Visual Basic
Object-Orientation	Hybrid	Hybrid / Multi-Paradigm	Hybrid	Add-On / Hybrid	Partial Support
Static / Dynamic Typing	Static	Static	Dynamic	Dynamic	Static
Generic Classes	No	Yes	N/A	N/A	No
Inheritance	Single class, multiple interfaces	Multiple	Multiple	Multiple	None
Feature Renaming	No	No	No	No	No
Method Overloading	Yes	Yes	No	No	No
Operator Overloading	Yes	Yes	Yes	Yes	No
Higher Order Functions	No	No	Lambda Expressions	Yes	No
Lexical Closures	No	No	Yes	Yes	No
Garbage Collection	Mark and Sweep or Generational	None	Reference Counting	Reference Counting	Reference Counting
Uniform Access	No	No	No	No	Yes
Class Variables / Methods	Yes	Yes	No	No	No
Reflection	Yes	No	Yes	Yes	No
Access Control	public, protected, private, internal, protected internal	public, protected, private, "friends"	Name Mangling	None	public, private
Design by Contract	No	No	No	No	No
Multithreading	Yes	Libraries	Yes	No	No
Regular Expressions	Standard Library	No	Standard Library	Built-in	No
Pointer Arithmetic	Yes	Yes	No	No	No
Language Integration	All .NET Languages	C, Assembler	C, C++, Java	C, C++	C (via DCOM)
Built-In Security	Yes	No	No	Yes	No
Capers Jones Language Level*	N/A	6	N/A	15	11

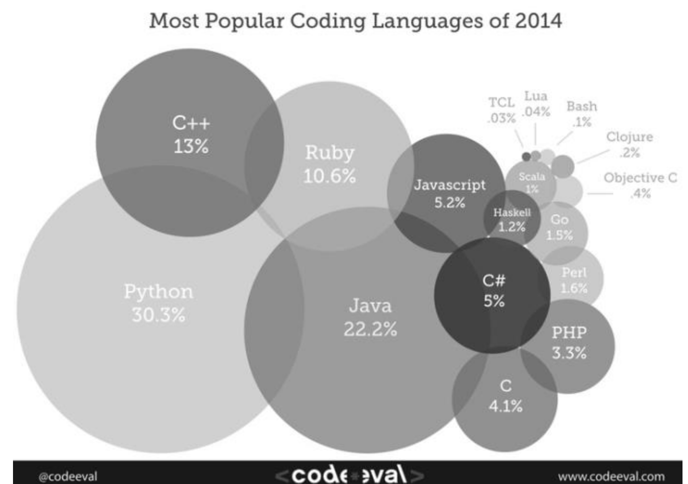
Criteria	Eiffel	Smalltalk	Ruby	Java
Object-Orientation	Pure	Pure	Pure	Hybrid
Static / Dynamic Typing	Static	Dynamic	Dynamic	Static
Generic Classes	Yes	N/A	N/A	No
Inheritance	Multiple	Single	Single class, multiple "mixins"	Single class, multiple interfaces
Feature Renaming	Yes	No	Yes	No
Method Overloading	No	No	No	Yes
Operator Overloading	Yes	Yes	Yes	No
Higher Order Functions	Agents (with version 5)	Blocks	Blocks	No
Lexical Closures	Yes	Yes	Yes	No
Garbage Collection	Mark and Sweep or Generational	Mark and Sweep or Generational	Mark and Sweep	Mark and Sweep or Generational
Uniform Access	Yes	N/A	Yes	No
Class Variables / Methods	No	Yes	Yes	Yes
Reflection	Yes	Yes	Yes	Yes
Access Control	Selective Export	Protected Data, Public Methods	public, protected, private	public, protected, "package", private
Design by Contract	Yes	No	Add-on	No
Multithreading	Implementation- Dependent	Implementation- Dependent	Yes	Yes
Regular Expressions	No	No	Built-in	Standard Library
Pointer Arithmetic	No	No	No	No
Language Integration	C, C++, Java	C	C, C++, Java	C, some C++
Built-In Security	No	No	Yes	Yes
Capers Jones Language Level*	15	15	N/A	6

N/A indicates that a topic or feature is not applicable to the language

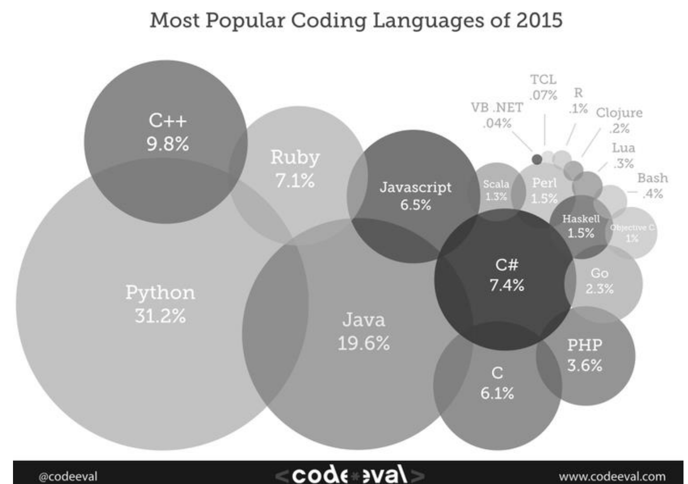
Market share of programming language for the last 3 years:



Pic I: show the most popular programming language of 2013 According to Code Eval



Pic II: show the most popular programming language of 2014 According to Code Eval



Pic III: show the most popular programming language of 2015 According to Code Eval

VI. CONCLUSION:

As of now and as per the latest research and market share, we can easily notice that Python and Java have more than 50% of the market share and they are the most popular programming languages at this time, C++ and Ruby in the second level, and they have more than 17% of the market share.
Python popularity has been increased dramatically during the last few years, while as C++ and Java decreased, from my point of view I believe that Python will be leading the programming languages for the coming years and it will be more popular and it might have the half of the market share within the coming 5 years.

At the end programming languages have been under development for many years and will remain so for many years to come. They got their start with a list of steps to wire a computer to perform a task. These steps eventually found their way into software and began to acquire newer and better features. The first major languages were characterized by the simple fact that they were intended for one purpose and one purpose only, while the languages of today are differentiated by the way they are programmed in, as they can be used for almost any purpose. And perhaps the languages of tomorrow will be more natural.

REFERENCES

[1] http://blaag.haard.se/Why-Python-is-important-for-you/
[2] http://www.tiobe.com/
[3] http://pypl.github.io/PYPL.html
[4] https://en.wikipedia.org/wiki/Python_(programming_language)
[5]http://archive.oreilly.com/pub/a/oreilly/frank/rossum_1099.html
[6]http://www.c-sharpcorner.com/Blogs/12972/importance-of-python-programming.aspx
[7]http://python-history.blogspot.com.cy/2009/01/personal-history-part-1-cwi.html
[8] http://www.theadvisors.com/langcomparison.htm
[9] http://www.jvoegele.com/software/langcomp.html