

## Project concepts of programming Languages



**Name**

**Abdulaziz Abdullah AL Hamdan  
Tameem Alsulmi  
Faisal hamad Alqahtani**

**ID Number**

**436018738  
437017388  
436020958**

**Department : Computer Science  
Date : 17/3/2019**



## 1. Historical Background

### a. Developer

Yukihiro Matsumoto, born 14 April 1965, is a Japanese computer scientist and software programmer best known as the chief designer of the Ruby programming language and its reference implementation.

### a. Environment of development

Yukihiro Matsumoto started working on Ruby language in 1993 and launched its first version in 1995. He said that the reason for his invention of the ruby was his desire for a script language stronger than Perl and more of an object than a Python. He took into account the best features in other languages. The language was not popular at first. It was used only in Japan, but with the passing of time, it became famous all over the world.

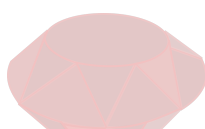


## 2. Design Process

### a. Goals of the language

Ruby is a language of careful balance. Its creator, Yukihiro Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming.

Ruby is an interpreted and object-oriented programming language which main purpose is to create simple and understandable web apps. Fast development, clarity, and syntax simplicity in Ruby are more important than the running speed of an app.



## **b.Advantages**

Truly Object Oriented from the ground up - very consistent patterns to the language

User-friendly language syntax - easy to read and easy to learn

Help the user develop nice, clean, powerful programs

Very powerful run time class extensions - make applications like Rails not only possible but very natural

## **c.Disadvantages**

The programmers did not notice any obvious flaws for Ruby, but the only drawback is that it is not used by many companies or users, especially in the Arab world, and they resort to other programming languages.



## **3.Language Overview**

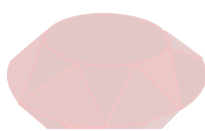
### **a. Name of variable**

#### **- Local variables:**

A local variable name starts with a lowercase letter or underscore (\_). It is only accessible or have its scope within the block of its initialization. Once the code block completes, variable has no scope.

#### **-Class variables**

A class variable name starts with @@ sign. They need to be initialized before use. A class variable belongs to the whole class and can be accessible from anywhere inside the class. A class variable is shared by all the descendents of the class.



### **-Instance variables**

An instance variable name starts with a `@` sign. It belongs to one instance of the class and can be accessed from any instance of the class within a method. They only have limited access to a particular instance of a class. They don't need to be initialize. An uninitialized instance variable will have a nil value.

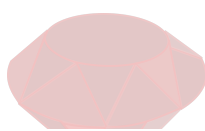
### **- Global variables**

A global variable name starts with a `$` sign. Its scope is globally, means it can be accessed from any where in a program. An uninitialized global variable will have a nil value. It is advised not to use them as they make programs cryptic and complex.

### **b.Data Types**

Data types in **Ruby** represents different types of data like text, string, numbers, etc. All data types are based on classes because it is a **pure Object-Oriented language**. There are different data types in Ruby as follows:

- Numbers
- Boolean
- Strings
- Hashes
- Arrays
- Symbols



**Numbers:** Generally a number is defined as a series of digits, using a dot as a decimal mark. Optionally the user can use the underscore as a separator. There are different kinds of numbers like integers and float. Ruby can handle both **Integers** and **floating point** numbers. According to their size, there are two types of integers, one is Bignum and second is Fixnum.

e.g. `Float("123.50") #=> 123.5` , `Integer("123.50") #=> 123`

- **Example:**



```
# Ruby program to illustrate the  
# Numbers Data Type
```

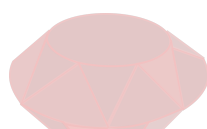


```
# float type  
distance = 0.1
```

```
# both integer and float type  
time = 9.87 / 3600  
speed = distance / time  
puts "The average speed of a sprinter is #{speed} km/h"
```

**Output:**

```
The average speed of a sprinter is 36.474164133738604 km/h
```



**Boolean:** Boolean data type represents only one bit of information either true or false.

- **Example:**



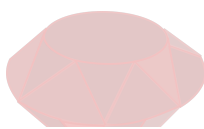
```
# Ruby program to illustrate the  
# Boolean Data Type
```



```
if true  
  puts "It is True!"  
else  
  puts "It is False!"  
end  
  
if nil  
  puts "nil is True!"  
else  
  puts "nil is False!"  
end  
  
if 0  
  puts "0 is True!"  
else  
  puts "0 is False!"  
end
```

**Output:**


```
It is True!  
nil is False!  
0 is True!
```




**Strings:** A string is a group of letters that represent a sentence or a word. Strings are defined by enclosing a text within a single (") or double (") quotes. You can use both double quotes and single quotes to create strings. Strings are objects of class String. Double-quoted strings allow substitution and backslash notation but single-quoted strings doesn't allow substitution and allow backslash notation only for \ and \'

e.g. `String(123.5) #=> "123.5"`

- **Example:**



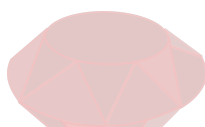
```
# Ruby program to illustrate the
# Strings Data Type
```



```
#!/usr/bin/ruby -w
puts "String Data Type";
puts 'escape using "\\"';
puts 'That\'s right';
```

**Output:**

```
String Data Type
escape using "\"
That's right
```



**Hashes:** A hash assigns its values to its key. Value to a key is assigned by `=>` sign. A key pair is separated with a comma between them and all the pairs are enclosed within curly braces. A hash in Ruby is like an object literal in JavaScript or an associative array in PHP. They're made similarly to arrays. A trailing comma is ignored.

- **Example:**



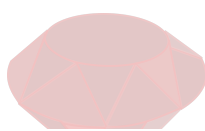
```
# Ruby program to illustrate the  
# Hashes Data Type
```



```
#!/usr/bin/ruby  
hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" :  
hsh.each do |key, value|  
  print key, " is ", value, "\n"  
end
```

**Output:**

```
red is 3840  
green is 240  
blue is 15
```





**Arrays:** An array stores data or list of data. It can contain all types of data. Data in an array are separated by comma in between them and are enclosed within square bracket. The position of elements in an array starts with 0. A trailing comma is ignored.

- **Example:**



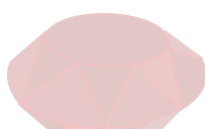
```
# Ruby program to illustrate the  
# Arrays Data Type
```



```
#!/usr/bin/ruby  
ary = [ "fred", 10, 3.14, "This is a string", "last element"  
ary.each do |i|  
  puts i  
end
```

**Output:**

```
fred  
10  
3.14  
This is a string  
last element
```



**Symbols:** Symbols are light-weight strings. A symbol is preceded by a colon (:). They are used instead of strings because they can take up much less memory. Symbols have better performance.

- **Example:**

```
# Ruby program to illustrate the
# Symbols Data Type

#!/usr/bin/ruby
domains = { :sk => "GeeksforGeeks", :no => "GFG", :hu => "Geeks" }

puts domains[:sk]
puts domains[:no]
puts domains[:hu]
```

**Output:**

```
GeeksforGeeks
GFG
Geeks
```

### a.Expressions and Assignment Statements

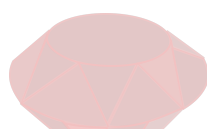
Statements are declarations that can make up a line (or multiple lines). And expressions are operations which will return values. So expressions are also a kind of statements.

Commonly seen statements are:

For loop ,switch statement ,while loop etc...

And common expressions are:

Function call like add(1, 2) , arithmetic operation like  $1 + 1 * 2$  , comparison like  $a == b$ ,  $a > c$  ,etc...



## **a.Control Statements**

Ruby can control the execution of code using Conditional branches. A conditional Branch takes the result of a test expression and executes a block of code depending whether the test expression is true or false.

e.g.

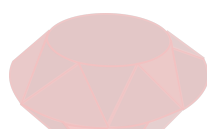
If , elsif , else , end , Case statement , Truthy and Falsy , unless , while, until

## **a.Subprograms**

Each subprogram has a single entry point. The calling program is suspended during execution of the called subprogram. Control always returns to the caller when the called subprogram's execution terminates. In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of Object. They can be called without an object, like a function.

### **i- Abstract Data Types**

- Encapsulation construct is the class
- Local variables have “normal” names
- Instance variable names begin with “at” signs (@)
- Class variable names begin with two “at” signs (@@)
- Instance methods have the syntax of Ruby functions (def ... end)
- Constructors are named initialize (only one per class)—implicitly called when new is called
  - If more constructors are needed, they must have different names and they must explicitly call new
- Class members can be marked private or public, with public being the default
- Classes are dynamic



An Example in Ruby :

```
class StackClass {  
    def initialize  
        @stackRef = Array.new  
        @maxLen = 100  
        @topIndex = -1  
    end  
  
    def push(number) ... end  
    def pop ... end  
    def top ... end  
    def empty ... end  
end
```

## **Support for Object\_Oriented Programming**

General Characteristics

Everything is an object

All computation is through message passing

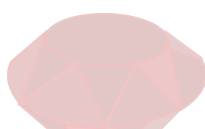
Class definitions are executable, allowing secondary definitions to add members to existing definitions

Method definitions are also executable

All variables are type-less references to objects

Access control is different for data and methods

It is private for all data and cannot be changed



Methods can be either public, private, or protected

Method access is checked at runtime

Getters and setters can be defined by shortcuts

Support for OOP in Ruby (continued)

Inheritance

Access control to inherited methods can be different than in the parent class

Subclasses are not necessarily subtypes

Mixins can be created with modules, providing a kind of multiple inheritance

Dynamic Binding

All variables are typeless and polymorphic

Evaluation

Does not support abstract classes

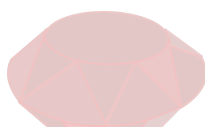
Does not fully support multiple inheritance

Access controls are weaker than those of other languages that support OOP

Exception Handling and Event Handling

Many third level languages, such as C++, Java, and C#, have try-catch blocks to allow exception handling. Much like these, Ruby has rescue blocks:

```
begin
...
rescue (Exception)
...
end
```



This rescue block attempts to execute the code after the begin, much like inside the try{} block in C#, and if an exception is thrown, the code inside the rescue command block corresponding to the exception thrown: just like the catch{} block. To add more rescue blocks for more exceptions, the programmer can just add them before the end. After all exceptions are thrown additional work or cleanup may be required, in C# there is the finally{} block; in Ruby there is the ensure block, that will execute regardless of the outcome of the rescue block. The programmer may also want to only run a section of code if the main code successfully runs (i.e. process a file after making sure that the file exists), which is where the else command comes in handy. If the code before the rescue block executes before without an exception being raised, the code inside the else block is executed.

Ruby also has a keyword “retry” that tells the program to retry the code before the rescue block, which is useful if the exception or error may be user based or just a spontaneous, unique event. There is also the capability to re-raise the exception, while seeming counter intuitive; this can be very helpful if the programmer wants to send the exception to a method. Ruby also allows for programmers to create custom exceptions thanks to the OOP.

**Give a coding and result for any suitable problem written in the language.**

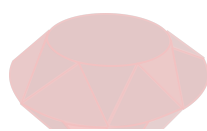
Ruby is case sensitive

```
puts("Hello world!")  
puts("Hello world!")
```

```
PUTS("Hello world!")
```

OUTPUT :

```
Hello world!  
hello world!  
undefined method 'PUTS' for main: Object
```





#### 4. Evaluation Of the Language :

##### Readability

##### i. Overall Simplicity

- o A manageable set of features and constructs
- o Minimal feature multiplicity
- o Minimal operator overloading

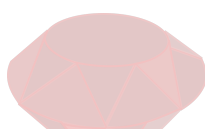
the Ruby programming language. Its use of English-like syntax and common language sounds so native and logical that some parts of the code read like English declarations.

The simple and readable syntax enable Ruby developers to do more with less code. That readability also makes Ruby almost self-documenting. This relieves the developers' burden of writing out separate comments or help text. The programmers and managers can view each other's code easily and quickly learn what's going on in a project.

##### ii. Orthogonality :

every programmer today knows the orthogonality principle ( which would better be termed the orthogonal completeness principle ) Suppose we have an imaginary pair of axes with a set of comparable language entities on one and a set of attributes or capabilities on the other. when we talk of orthogonality , we usually mean that the space defined by these axes is as "full" as we can logically make it .

part of the Ruby way is to strive for this orthogonality. An array is in some ways similar to a hash , so the operations on each of them should be similar. The limit is reached when we enter the areas where they are different .



Matz has said that "naturalness" is to be valued over orthogonality. But to fully understand what is natural and what is not may take some thinking and some coding.

Ruby strives to be friendly to the programmer. For example , there are aliases or synonyms for many method names; `size` and `length` will both return the number of entries in an array. The variant spellings `indexes` and `indices` both refer to the same method. Some consider this sort of thing to be an annoyance or "anti-feature" but I consider it a good design

### iii. Control Statement

Ruby can control the execution of code using Conditional branches. A conditional Branch takes the result of a test expression and executes a block of code depending whether the test expression is true or false. If the test expression evaluates to the constant false or nil, the test is false; otherwise, it is true. Note that the number zero is considered true, whereas many other programming languages consider it false.

Another feature and idiom combined is using trailing if statements to increase the readability of the code. One of the driving ideas in Ruby is to write code that reads as natural language. For this, we go

Ruby loves it when you write things concisely. The `if` and `unless` keywords can also be placed after the line of code you may or may not want to execute. When used in this way they are called statement modifiers:

### iv. Data Types and Structures

Adequate predefined data types

### v. Syntax Considerations

a. Identifier forms: flexible composition

b. Special words and methods of forming



compound statements

c. Form and meaning: self-descriptive constructs,  
meaningful keywords

b. Writability

i. Simplicit and Orthogonality

Few constructs, a small number of primitives, a small set of  
rules for combining them

Support for Abstraction:

Data abstraction is fundamental to most object oriented language - wherein the classes are designed to encapsulate data and provide methods to control how that data is modified (if at all), or helper methods to derive meaning of that data.

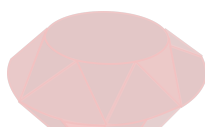
Ruby's Array class is an example of Data Abstraction. It provides a mechanism to manage an array of Objects, and provides operations that can be performed on that array, without you having to care how internally it is organized .

```
[1,3,4,5,2,10]arr =
```

```
p arr.class # Prints Array
```

```
[1,2,3,4,5,10] p arr.sort # Prints
```

Procedural abstraction is about hiding implementation details of procedure from the user. In the above example, you don't really need to know what sorting algorithm sort method uses internally, you just use it assuming that nice folks in Ruby Core team picked a best one for you.



At the same time, Ruby may not know how to compare two items present in the Array always. For example, below code would not run as Ruby does not know how to compare strings and numbers.

It allows us to hook into implementation and help with comparison, even though underlying sorting algorithm remains same

(as it is abstracted from the user)

```
| [1,3,4,5,"a","c","b", 2,10].sort { |i,j
```

```
if i.class == String and j.class == String
```

```
i <=> j
```

```
elsif i.class == Fixnum and j.class == Fixnum
```

```
i <=> j
```

```
else
```

```
0
```

```
end
```

```
}
```

```
[1, 3, 4, 5, 2, 10, "a", "b", "c"]=>#
```

Expressivity :

a.A set of relatively convenient ways of specifying operations

b.Strength and number of operators and predefined



functions

Reliability

Type Checking

Ruby developers often wax enthusiastic about the speed and agility with which they are able to write programs, and have relied on two techniques more than any other to support this: tests and documentation.

After spending some time looking into other languages and language communities, it's my belief that as Ruby developers, we are missing out on a third crucial tool that can extend our design capabilities, giving us richer tools with which to reason about our programs. This tool is a rich type system.

-Truly executable documentation

Types declared for methods or fields are enforced by the type checker. Annotated classes are easy to parse by developers and documentation can be extracted from type annotations.

-Stable specification

Tests which assert the input and return values of methods are brittle, raise confusing errors, and bloat test suites; documentation gets out of sync. Type annotations change with your implementation and can help maintain interface stability.

-Meaningful error messages

Type checkers are valuable in part because they bridge the gap between the code and the meaning of a program. Error messages which inform you not only that you made a mistake, but how (and potentially how to fix it) are possible with the right tools.

-Type driven design

Considering the design of a module of a program through its types can be an interesting exercise. With advancements in type checking and inference for dynamic programming languages, it may be possible to rely on these tools to help guide our program design.

Integrating traditional typing into a dynamic language like Ruby is inherently

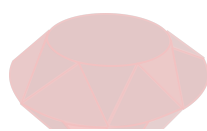
challenging. However, in searching for a way to integrate these design advantages into Ruby programs, I have come across a very interesting body of research about “gradual typing” systems. These systems exist to include, typically on a library level, the kinds of type checking and inference functionality that would allow Ruby developers to benefit from typing without the expected overhead[1]

In doing this research I was pleasantly surprised to find that four researchers from the University of Maryland’s Department of Computer Science have designed such a system for Ruby, and have published a paper summarizing their work. It is presented as “The Ruby Type Checker” which they describe as “...a tool that adds type checking to Ruby, an object-oriented, dynamic scripting language.” [2] Awesome, let’s take a look at it!

### The Ruby Type Checker

The implementation of the Ruby Type Checker (rtc) is described by the authors as “a Ruby library in which all type checking occurs at run time; thus it checks types later than a purely static system, but earlier than a traditional dynamic type system.” So right away we see that this tool isn’t meant to change the principal means of development relied on by Ruby developers, but rather to augment it. This is similar to how we think about Code Climate - as a tool which brings information about problems in your code earlier in your process.

Exception Handling :



In Ruby, exception handling is a process which describes a way to handle the error raised in a program. Here, error means an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. So these types of errors were handled by the rescue block. Ruby also provides

a separate class for an exception that is known as an Exception class which contains different types of methods.

The code in which an exception is raised, is enclosed between the begin/end block, so you can use a rescue clause to handle this type of exception.

Syntax:

```
begin
```

```
raise
```

```
# block where exception raise
```

```
rescue
```

```
# block where exception rescue
```

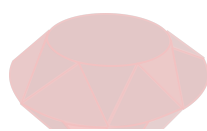
```
end
```

Example:

```
filter_none
```

```
brightness_4
```

Ruby program to create the user #



```
# defined exception and handling it
```

```
# defining a method
```

```
def raise_and_rescue
```

```
begin
```

```
puts 'This is Before Exception Arise'
```

```
# using raise to create an exception
```

```
raise 'Exception Created!'
```

```
puts 'After Exception'
```

```
using Rescue method #
```

```
rescue
```

```
puts 'Finally Saved!'
```

```
end
```

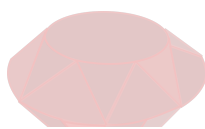
```
puts 'Outside from Begin Block!'
```

```
end
```

```
calling method #
```

```
raise_and_rescue
```

```
output:
```



This is Before Exception Arise!

Finally Saved!

Outside from Begin Block!

Aliasing :

The alias keyword

Ruby provides an alias keyword to deal with method and attribute aliases

1- class User

2- def fullname

'3- 'John Doe

4- end

5- alias username fullname

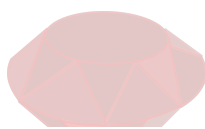
6- alias name username

7- end

8- u = User.new

9- p u.fullname # => "John Doe"

10- p u.username # => "John Doe"



```
11- p u.name # => "John Doe"
```

Here we define a `User#fullname` method and we define a `username` alias for this method.

Then, the `username` alias is aliased with a `name` alias.

So, a call to `name` or `username` will call the code defined within the `User#fullname` method and return the same result.

Note that it's possible to define an alias for an existing alias.

```
1- class User
```

```
2- def fullname
```

```
'3- 'John Doe
```

```
4- end
```

```
5- alias username fullname
```

```
6- alias name username
```

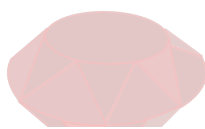
```
7- end
```

```
8- u = User.new
```

```
9- p u.fullname # => "John Doe"
```

```
10- p u.username # => "John Doe"
```

```
11- p u.name # => "John Doe"
```





Like the alias keyword, we define a `User#fullname` method and we define a `username` alias for this method.

Then the `username` alias is aliased with a `name` alias.

So, a call to `name`, `username` or `fullname` returns the same result.

We can see that the `alias_method` method takes a `String` or a `Symbol` as argument that allows

Ruby to identify the alias and the method to alias.

If `alias` and `Module#alias_method` share the same behavior, then what's the purpose of dealing with two things that do the exact same job?

The answer is that they don't exactly do the same thing.

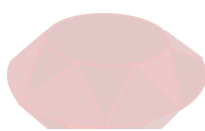
Readability and Writability :

Writability

With the inclusion of regular expressions, if the programmer can get used to them, programs can be written to process various types of data extremely easily. The OOP and implicit variable declaration also increase the ease at which code can be written. What is taken away from readability is given back in writability and speed at which it can be written.

Readability

Since there are compilers provided for free on the official site for the three main operating systems as well in Java, a language that works on as many platforms as possible, it is highly reliable that the same code that works on one computer will work on all others.





## 5. Cost of the Language :

While the compiler is open source and free as are several IDEs, there are some IDEs that do cost some money. The Ruby in Steel IDE, which is a Visual Studio plugin, costs \$249.00 to

license, which doesn't include the \$799 - \$3,799.00 for Microsoft's Visual Studio 2010. RubyMine, another IDE, costs either \$69 or \$149 depending on whether the purchaser is a company/organization or an individual developer. Unless the programmer or their company is dead set on one of these environments, Ruby is a completely free language to learn and use with lots of documentation on the internet. When I downloaded the Aptana IDE, a free IDE, to study Ruby with, it even came with a free public .pdf file of a book that teaches Ruby and is a great resource.

Not all valuable things sparkle like silver though, time is a precious resource that cannot be replaced, and as it is said "time is money." If the person learning already knows a programming language decently, it may take them a week to pick up Ruby if it is troubling them. If they know a scripting language and are familiar with regular expressions it could be less. They won't be an expert in the language in this time, but they will know enough to get most tasks completed.

