Ruby_FullCourse

This repository contains the documentation of the Ruby Full Course.

Made with <3 by *Hugo Correia*, *Duarte Cruz* and *André Oliveira*

Table of Contents

Documentation

1 - What is Ruby? What is it used for?

1.1 - Introduction to Ruby

- **History and Background**

  Ruby is a dynamic, open-source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write. Ruby was created as a language of careful balance. Its creator, Yukihiro “Matz” Matsumoto, blended parts of his favourite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming.

  He has often said that he is “trying to make Ruby natural, not simple,” in a way that mirrors life.

- **Design Philosophy**

  The foundational philosophy behind Ruby’s design is that programming languages should make programmers happy. It is a language designed for programmer productivity and fun, following the principles of good user interface design. Ruby is a language of careful balance. The language's syntax and structure aim to reduce mental overhead for developers, thereby fostering increased productivity. But Ruby is not only for experienced developers. It's also a great language for those who are new to programming or who are new to the Ruby language.

1.2 - Ruby's Purpose and Use-Cases

- **Web Development**

  Ruby on Rails is a popular web framework written in Ruby. It allows you to quickly and easily develop web applications. Ruby on Rails is opinionated software. It makes the assumption that there is a "best" way to do things, and it's designed to encourage that way - and in some cases to discourage alternatives. Its guiding principle is "Convention over Configuration" (CoC), which means that the programmer only needs to specify unconventional aspects of the application. Everything else is either handled automatically (such as the structure of the database), or by following conventions (such as naming classes and methods).

Here are some examples of popular websites that use Ruby on Rails:

  - GitHub (a popular social network centred around code)
  - Shopify (an e-commerce platform)
  - Airbnb (a marketplace for renting lodging)
  - Twitch (a streaming platform for gamers)

- **Scripting and Automation**

  Beyond web development, Ruby is also a great language for scripting and automating common tasks. It's often used as a "glue" language to connect other software together. Here are some examples of popular tools that use Ruby:

  - Vagrant (a tool for building and distributing development environments)
  - Chef (a tool for automating server configuration)
  - Puppet (a tool for automating software configuration)
  - Homebrew (a package manager for macOS)

- **Prototyping and Data Analysis**

  Ruby's readability and flexibility also make it well-suited for prototyping and exploratory data analysis. It's easy to write and read Ruby code, which makes it a great tool for researchers and data scientists who aren't necessarily professional developers. Here are some examples of popular tools that use Ruby:

  - Jupyter (a popular tool for data analysis and visualization)
  - Metasploit (a penetration testing framework)
  - Cucumber (a tool for running automated tests written in plain language)
  - Sass (a popular CSS pre-processor)

1.3 - Ruby's Strengths

- **Readability and Writability**

  Ruby's syntax is designed to be intuitive, mirroring natural language, which enhances its readability and writability. This simplicity and elegance make it particularly appealing for new programmers. The language's flexibility and forgiving nature lend themselves well to prototyping and exploratory data analysis.

- **Flexibility and Productivity**

  Ruby is a very flexible language. It's dynamic typing and duck typing make it easy to write code that is both concise and expressive. It's also a very productive language. It's easy to write and read Ruby code, which makes it a great tool for researchers and data scientists who aren't necessarily professional developers.

- **Community and Support and Ecosystem**

  Ruby has a large community of developers who are actively working to improve the language. It's also a very popular language, which means that there are many libraries and tools available for Ruby developers. Ruby is also a very popular language for web development, which means that there are many web frameworks available for Ruby developers.

- **Portability and Compatibility**

  Ruby is highly portable, and available on numerous platforms like Windows, macOS, Linux, and BSD. It also boasts compatibility with several programming languages, including C, C++, Java, Python, and Perl, enhancing its versatility in diverse environments.

1.4 - Ruby's Weaknesses

- **Performance**

  Compared to languages like C++ or Java, Ruby is often critiqued for its slower execution speed. This can be a significant drawback in scenarios where performance is a critical factor.

- **Scalability and Concurrency**

  Ruby's concurrency model presents challenges, particularly in managing multithreading effectively. This can lead to issues in scaling applications, especially those requiring high levels of parallel processing.

- **Memory Usage**

  Ruby's memory management is often seen as less efficient, leading to higher memory usage in certain applications. This can be a limiting factor in memory-intensive operations.

- **Type Safety**

  Due to its dynamic typing, Ruby may face issues with type safety, which can lead to runtime errors. This necessitates more thorough testing and can be problematic in large-scale or complex applications.

- **Ecosystem and Dependency Management**

  While Ruby has a vast ecosystem, dependency management can sometimes be complex, especially when dealing with larger applications or numerous external libraries. This can make application maintenance and upgrades more challenging.

1.5 - Memory Management and Garbage Collection

- **Gargabe Collector**

- *Generational GC:* Continues to improve performance by focusing more frequently on collecting younger objects, which are more likely to be garbage.
    - *Incremental GC:* Reduces long pauses in program execution by breaking down the garbage collection process into smaller steps.
    - *Memory Compaction:* Introduced in Ruby 2.7 and improved in Ruby 3, memory compaction reduces memory fragmentation by moving objects to contiguous memory spaces.

- Memory Usage

  Ruby's memory management is often seen as less efficient, leading to higher memory usage in certain applications. This can be a limiting factor in memory-intensive operations.

    - *Memory Efficiency:* Historically, Ruby has been criticised for its relatively high memory usage. This is partly due to Ruby's object model, where even integers are treated as objects, consuming more memory than in languages where integers are primitive types.
    - *Improvements in Version 3:* Ruby 3 has made efforts to enhance memory efficiency. For instance, the introduction of memory compaction helps reduce fragmentation, which can lead to more efficient memory usage.
    - *Development Practices:* Developers can also take steps to improve memory efficiency, such as avoiding unnecessary object creation and using symbols instead of strings. This can help reduce memory usage and improve performance.

- Memory Management

    - *Automation with GC:* Ruby's Garbage Collector (GC) automatically manages most of the object lifecycle, freeing up memory for objects that are no longer accessible.

    - *Manual Interventions:* In specific situations, developers might need to intervene, such as manually releasing resources or explicitly calling GC.start. However, this should be approached with caution.

    - *Monitoring and Diagnostics:* Ruby 3 provides tools and libraries for monitoring memory usage and diagnosing memory-related issues, making it easier for developers to optimise memory usage in their applications.

Conclusion

While Ruby 3 has significantly improved in terms of memory efficiency and GC performance compared to previous versions, there is still room for optimisation, particularly in applications that are highly memory-intensive. Understanding how Ruby handles memory is crucial for writing efficient applications and solving issues related to memory usage.

1.6 - Ruby's Paradigms

Ruby is a multi-paradigm programming language, meaning that it supports several different programming paradigms. These include:

- Object-Oriented

  Ruby is purely object-oriented in that everything is an object, including primitive data types like numbers, booleans, and strings. This allows for a consistent and intuitive modelling of the real world. Ruby supports inheritance, polymorphism, encapsulation, and also accessor methods (attr_accessor, attr_reader, attr_writer) to facilitate the management of object states.

- Functional

  Although not a functional language in the strict sense, Ruby incorporates several features from functional languages, such as first-class functions and the closure of scopes (closures). Ruby also supports the passing of blocks of code to methods, which allows for functional patterns like map, reduce, and select.

- Imperative

  Ruby is an imperative language in that it uses statements to change the state of the program. It also supports the use of loops and conditionals to control the flow of execution.

- Procedural

  Ruby supports procedural programming, which is a programming paradigm based on the concept of the procedure call. It is a sequence of instructions that perform a specific task, packaged as a unit. Procedural programming is a subset of imperative programming.

- Reflective

  Ruby possesses metaprogramming, which enables programs to manipulate their structure and behaviour at runtime. This is powerful for the creation of DSLs (Domain-Specific Languages) and metaclasses, which are the classes of classes. Ruby also supports reflection, which allows for the inspection of objects and classes at runtime. This is useful for debugging and testing.

    - Metaprogramming

      Metaprogramming is a programming technique in which computer programs can treat other programs as their data. It means that a program can be designed to read, generate, analyse or transform other programs, and even modify itself while running.

- Scripting

  Ruby is often utilised as a scripting language due to its expressive syntax and task automation capability. It can interact with the operating system and other programs, facilitating the execution of repetitive tasks and process management.

By incorporating these paradigms, Ruby provides a rich and flexible programming experience, making it an excellent choice for object-oriented software development, automation, data manipulation, web development, and much more. The Ruby community continues to expand the language's capabilities, providing a wide range of gems (libraries) to support these and other programming paradigms.

1.7 - Ruby's Parallelism and Concurrency

Ruby is a flexible and dynamic programming language that supports various programming paradigms. Here, we delve into its concurrency and parallelism capabilities:

- Concurrency

  Ruby provides concurrency mechanisms through threads and fibres. Threads in Ruby allow for the simultaneous execution of code, enabling the program to perform multiple tasks concurrently. However, the standard Ruby implementation (MRI — Matz's Ruby Interpreter) employs a Global Interpreter Lock (GIL), also known as a Global VM Lock (GVL), which restricts native thread execution to one at a time. This means that even on a multicore system, only one thread can execute at a given moment.

- **Parallelism**

  Achieving true parallelism, which involves running multiple threads or processes simultaneously across different CPU cores, is more challenging in Ruby due to the GIL in the standard implementation. Nevertheless, it is possible to attain parallelism by utilising processes instead of threads. The parallel gem, for example, allows the creation of multiple processes to execute code concurrently, circumventing the GIL limitations.

Moreover, alternative Ruby implementations, such as JRuby (Ruby on the JVM) and Rubinius, have been designed to support native threads without a GIL, enabling real parallelism on multicore environments.

While Ruby is not traditionally known for its performance in parallelism and concurrency due to the GIL in the standard implementation, there are methods to achieve these behaviours through multiple processes, Ractors, or by using an alternative Ruby implementation.

1.8 - Ruby's Future

As the landscape of programming languages continues to evolve, Ruby's future is shaped by its adaptability, community engagement, and ongoing development efforts. The language is poised to address contemporary programming challenges while staying true to its ethos of programmer happiness and productivity.

- **Matz’s Vision and Ruby 3x3**

  Yukihiro “Matz” Matsumoto, the creator of Ruby, has set forth a vision for the language known as "Ruby 3x3", which aims to make Ruby three times faster by the release of version 3. This goal is partly realized through advancements such as the introduction of Just-In-Time (JIT) compilation in MRI.

- **Ractor for Concurrency**

  Ruby 3 introduced Ractor, an actor-like concurrency abstraction that enables developers to better utilize modern multicore processors. This addition marks a significant step towards enhancing Ruby's performance in concurrent computing environments.

- **Type Checking and Sorbet**

  To address concerns about type safety, the Ruby community has seen initiatives like Sorbet, a static type checker for Ruby. While Ruby remains a dynamically typed language, tools like Sorbet provide optional type-checking to improve code robustness.

- **Guilds for Parallelism**

  The concept of Guilds, although not yet implemented, has been proposed as a way to achieve true parallelism in Ruby. This could revolutionize the way Ruby handles thread safety and concurrency, making it a more competitive choice for high-performance applications.

- **Active Development of Frameworks and Gems**

  Ruby on Rails, along with other Ruby frameworks and an extensive library of gems, continues to evolve. The dedication to enhancing these tools ensures that Ruby remains a top choice for web development and beyond.

- **Community Engagement and Conferences**

  The active Ruby community, with conferences like RubyConf and RailsConf, plays a crucial role in shaping the language's future. These events serve as platforms for sharing knowledge, discussing new features, and collaborating on open-source projects.

- **Emerging Trends and Integration**

  Ruby developers are also integrating emerging technologies such as machine learning, artificial intelligence, and serverless computing into the Ruby ecosystem, showcasing the language's flexibility and modern relevance.

In summary, Ruby's future looks promising, with concerted efforts to improve performance, concurrency, and type safety, while maintaining the language's unique philosophy. Its vibrant community and ecosystem ensure that Ruby will continue to adapt and thrive in the changing technological landscape.

2 - Ruby's Installation

2.1 - Windows

- **Step 1: Download Ruby**

  The first step is to download Ruby. You can download the latest version of Ruby from the RubyInstaller website. You can also download Ruby from the RubyInstaller GitHub

- **Step 2: Install Ruby**

  Once you have downloaded Ruby, you can install it by double-clicking on the downloaded file. This will open a window where you can select the components you want to install. You can also choose to install Ruby in a different location by clicking on the "Browse" button.

- **Step 3: Verify the Installation**

  Once you have installed Ruby, you can verify the installation by opening a command prompt and typing the following command:

  ```
  ruby -v
  ```

  This will display the version of Ruby that you have installed.

2.2 - Linux

- **Step 1: Install Ruby**

  The first step is to install Ruby. You can install Ruby by running the following command:

  ```
  sudo apt-get install ruby-full
  ```

- **Step 2: Verify the Installation**

  Once you have installed Ruby, you can verify the installation by opening a terminal and typing the following command:

  ```
  ruby -v
  ```

  This will display the version of Ruby that you have installed.

2.3 - Mac

- **Step 1: Install Ruby**

  The first step is to install Ruby. You can install Ruby by running the following command:

  (Normally, Ruby is already installed on Mac)

  ```
  brew install ruby
  ```

- **Step 2: Verify the Installation**

  Once you have installed Ruby, you can verify the installation by opening a terminal and typing the following command:

  ```
  ruby -v
  ```

  This will display the version of Ruby that you have installed.

---

3 - Ruby Basic Setup (after installation)

3.1 - Choosing an IDE

- **Visual Studio Code**

  Visual Studio Code is a free, open-source, cross-platform text editor developed by Microsoft. It has a large community of developers who are actively working to improve the editor. It's also a very popular editor, which means that there are many extensions available for Visual Studio Code.

  - **Installation**

    You can download Visual Studio Code from the Visual Studio Code website. You can also download Visual Studio Code from the [Visual Studio Code GitHub](

  - **Extensions**

    Visual Studio Code has a large number of extensions available. You can find a list of extensions on the Visual Studio Code Marketplace.

- **Atom**

  Atom is a free, open-source, cross-platform text editor developed by GitHub. It has a large community of developers who are actively working to improve the editor. It's also a very popular editor, which means that there are many extensions available for Atom.

  - **Installation**

    You can download Atom from the Atom website.

  - **Extensions**

    Atom has a large number of extensions available. You can find a list of extensions on the Atom Packages website. To run ruby code in Atom, you will need to install the script package.

- **Ruby Mine (JetBrains)**

  - **Installation**

    You can download Ruby Mine from the Ruby Mine website.

  - **Extensions**

    Ruby Mine has a large number of extensions available. You can find a list of extensions on the Ruby Mine Plugins website.

3.2 - Manage Ruby Gems

Upon installation of Ruby, it's essential to understand the management of Ruby gems, which are integral to the language's ecosystem. Gems facilitate the addition of features and functionalities to Ruby applications without the need to reinvent the wheel. They encompass a wide array of libraries and software snippets that can be easily installed and managed through the command-line tool gem.

RubyGems, the package manager for Ruby, simplifies the process of creating, sharing, and implementing gems. It provides a standardised format for the distribution and packaging of Ruby applications and libraries. Whether it's for web frameworks like Rails, database adapters, or any other purpose, gems can be found for almost any functionality one might need in a Ruby application. This system not only enhances productivity but also fosters a thriving community of developers who contribute to a rich repository of Ruby gems.

The use of gems is not limited to utilising existing libraries; developers are also encouraged to create and publish their own gems, thus contributing to the growth and richness of the Ruby ecosystem.

- **Installing Gems**

  Gems can be installed using the gem install command. For example, to install the rails gem, you would run the following command:

  ```
  gem install rails
  ```

- **Listing Installed Gems**

  Gems can be listed using the gem list command. For example, to list all the gems installed on your system, you would run the following command:

  ```
  gem list
  ```

- **Updating Gems**

  Gems can be updated using the gem update command. For example, to update the rails gem, you would run the following command:

  ```
  gem update rails
  ```

- **Uninstalling Gems**

  Gems can be uninstalled using the gem uninstall command. For example, to uninstall the rails gem, you would run the following command:

  ```
  gem uninstall rails
  ```

- **Searching for Gems**

  Gems can be searched using the gem search command. For example, to search for the rails gem, you would run the following command:

  ``bash gem search rails

  ```

  Or you can search for gems using the RubyGems website.

---

### 3.3 - Run Ruby Code

- **Running Ruby Code in Visual Studio Code**

  To run Ruby code in Visual Studio Code (IDE used in this course), simply open the file containing the code and click on the "Run" button in the top right corner of the editor.

- **Running Ruby Code on the Shell**

  ```
  ruby file_name.rb
  ```

---

## 4 - Ruby Syntax

### 4.1 - Reserved Words

The following list shows the reserved words in Ruby. These reserved words may not be used as constant or variable names. They can, however, be used as method names.

| Keyword | Keyword | Keyword | Keyword |
|---|---|---|---|
| BEGIN | do | next | then |
| END | else | nil | true |
| alias | elsif | not | undef |
| and | end | or | unless |
| begin | ensure | redo | until |
  ```

Keyword	Keyword	Keyword	Keyword
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	_FILE_
defined?	module	super	_LINE_

4.2 - Comments

- **Single Line Comments**

  Single-line comments are created using the `#` character. Anything after the `#` character is ignored by the interpreter.

  ```
   # This is a single-line comment
  ```

- **Multi-Line Comments**

  Multi-line comments are created using the `=begin` and `=end` keywords. Anything between these keywords is ignored by the interpreter.

  ```
   =begin
  This is a
  multi-line
  comment
  =end
  ```

4.3 - Variables

- **Variable Declaration and Assignment**

  Variables are declared using the `=` operator. The variable name is on the left side of the `=` operator, and the value is on the right side of the `=` operator.

  ```
   # Variable declaration and assignment
  variable = value
  ```

- **Variable Scope**

  Variables have a scope that determines where they can be accessed. There are three types of variable scope: global, local, and instance.

  - **Global Variables**

    Global variables are accessible from anywhere in the program. They are declared using the `$` character.

    ```
     # Global variable
    $variable = value
    ```

  - **Local Variables**

    Local variables are only accessible from within the block in which they are declared. They are declared using the `local` keyword.

    ```
     # Local variable
    local variable = value
    ```

  - **Instance Variables**

    Instance variables are accessible from anywhere within the class in which they are declared. They are declared using the `@` character.

    ```
     # Instance variable
    @variable = value
    ```

  - **Class Variables**

    Class variables are accessible from anywhere within the class in which they are declared. They are declared using the `@@` character.

    ```
     # Class variable
    @@variable = value
    ```

- **Variable Naming Conventions**

Variables can be named using any combination of letters, numbers, and underscores. They cannot start with a number or contain spaces. It is common practice to use lowercase letters for variable names.

```
 # Valid variable names
variable
variable1
variable_1
```

```
 # Invalid variable names
1variable
variable name
```

- **Variable Types**

  Variables can be of any type. The type of a variable is determined by the value assigned to it.

  ```
   # Integer variable
  variable = 5
  ```

  ```
   # Float variable
  variable = 3.14
  ```

  ```
   # String variable
  variable = "Hello"
  ```

- **Variable Casting**

  Variables can be cast to a different type using for example the `to_i`, `to_f`, and `to_s` methods. More advanced casting methods are also available like `to_a` (to array), `to_h` (to hash), `to_sym` (to symbol), and `to_r` (to rational).

  ```
   # Integer variable to float
  variable = 5
  variable.to_f
  ```

  ```
   # Float variable to integer
  variable = 3.14
  variable.to_i
  ```

  ```
   # Integer variable to string
  variable = 5
  variable.to_s
  ```

- **Variable Interpolation**

  Variable interpolation is a way to insert the value of a variable into a string. It is done by placing the variable name inside of `#{}`.

  ```
   # Variable interpolation
  variable = "World"
  puts "Hello #{variable}"
  ```

- **Variable Deletion**

  Variable deletion is a way to remove a variable from memory. It is done by using the `delete` keyword.

  ```
   # Variable deletion
  variable = "Hello"
  variable.delete
  ```

- **Pseudo-Variables**

  Pseudo-variables are variables that are automatically created by the interpreter. They are not declared by the programmer, but they can be used in the program.

  - **self**

    The `self` pseudo-variable refers to the current object. It is used to access the current object's instance variables and methods.

    ```
     # self pseudo-variable
    self.variable
    ```

  - **true**

The `true` pseudo-variable refers to the boolean value `true`.

```
# true pseudo-variable
true
```

- **false**

  The `false` pseudo-variable refers to the boolean value `false`.

  ```
  # false pseudo-variable
  false
  ```

- **nil**

  The `nil` pseudo-variable refers to the value `nil`. It is used to represent "nothing" or "no value".

  ```
  # nil pseudo-variable
  nil
  ```

- **__FILE__**

  The `__FILE__` pseudo-variable refers to the name of the current file.

  ```
  # __FILE__ pseudo-variable
  __FILE__
  ```

- **__LINE__**

  The `__LINE__` pseudo-variable refers to the current line number.

  ```
  # __LINE__ pseudo-variable
  __LINE__
  ```

4.4 - Data Types

There are several different data types in Ruby. Each data type has a specific purpose and is used in various ways in Ruby programming.

Data Type	Description	Example
Integer	Whole numbers, both positive and negative	`5`, `-20`
Float	Numbers with decimal points	`3.14`, `-0.001`
String	Sequence of characters	`"Hello"`, `'Ruby'`
Symbol	Lightweight, immutable strings, often used as keys	`:name`, `:user_id`
Array	Ordered collection of items (can be mixed types)	`[1, 'two', :three]`
Hash	Collection of key-value pairs	`{'a' => 1, 'b' => 2}`
Boolean	Logical data type representing true or false	`true`, `false`
Nil	Represents "nothing" or "no value"	`nil`
Range	Sequence of values with a start and end point	`(1..5)`, `(a...z)`
Regexp	Regular expression for pattern matching	`/[A-Z]/`, `/^\d+$/`

Integer

Integers are whole numbers, both positive and negative. They can be represented in decimal, hexadecimal, octal, and binary formats.

```
# Integers
5
-20
0x5 # 5 in hexadecimal
0b101 # 5 in binary
0o10 # 8 in octal
```

- **Integer Methods**

  Integers have several methods that can be used to perform operations on them. Some of these methods are:

  - `abs` - Returns the absolute value of the integer.
  - `even?` - Returns true if the integer is even, false otherwise.
  - `odd?` - Returns true if the integer is odd, false otherwise.
  - `times` - Executes the given block the specified number of times.

  ```
  # Integer methods
  5.abs
  5.even?
  5.odd?
  5.times { puts "Hello" }
  ```

  Heres the list of all the methods available for integers: Integer Methods or Integer FIle

Float

Floats are numbers with decimal points. They can be represented in decimal and scientific notation formats.

```
# Floats
3.14
-0.001
3.14e-2 # 0.0314 in scientific notation
```

- **Float Methods**

  Floats have several methods that can be used to perform operations on them. Some of these methods are:

  - `abs` - Returns the absolute value of the float.
  - `ceil` - Returns the smallest number greater than or equal to the float.
  - `floor` - Returns the largest number less than or equal to the float.
  - `round` - Rounds the float to the nearest value.

  ```
  # Float methods
  3.14.abs
  3.14.ceil
  3.14.floor
  3.14.round
  ```

  Heres the list of all the methods available for floats: Float Methods or Float File

  (some methods are the same as for integers so they are not repeated here)

String

Strings are sequences of characters. They can be represented using single or double quotes.

```
# Strings
"Hello"
'Ruby'
```

- **String Methods**

  Strings have several methods that can be used to perform operations on them. Some of these methods are:

  - `capitalize` - Returns a copy of the string with the first character converted to uppercase and the remainder to lowercase.
  - `downcase` - Returns a copy of the string with all uppercase letters replaced with their lowercase equivalents.
  - `empty?` - Returns true if the string is empty, false otherwise.
  - `length` - Returns the length of the string.
  - `reverse` - Returns a copy of the string with the characters in reverse order.
  - `upcase` - Returns a copy of the string with all lowercase letters replaced with their uppercase equivalents.

```
# String methods
"Hello".capitalize
"Hello".downcase
"Hello".empty?
"Hello".length
"Hello".reverse
"Hello".upcase
```

Heres the list of all the methods available for strings: String Methods or String File

Symbol

Symbols are lightweight, immutable strings. They are often used as keys in hashes.

```
# Symbols
:name
:user_id
```

- **Symbol Methods**

  Symbols have several methods that can be used to perform operations on them. Some of these methods are:

  - `to_s` - Returns a string representation of the symbol.
  - `id2name` - Returns a string representation of the symbol.

  ```
  # Symbol methods
  :name.to_s
  :name.id2name
  ```

  Heres the list of all the methods available for symbols: Symbol Methods or Symbol File

Array

Arrays are ordered collections of items. They can contain items of any type.

```
# Arrays
[1, 'two', :three]
```

- **Array Methods**

  Arrays have several methods that can be used to perform operations on them. Some of these methods are:

  - `each` - Iterates over the array, passing each item to the given block.
  - `first` - Returns the first item in the array.
  - `last` - Returns the last item in the array.
  - `length` - Returns the length of the array.
  - `reverse` - Returns a copy of the array with the items in reverse order.

  ```
  # Array methods
  [1, 2, 3].each { |item| puts item }
  [1, 2, 3].first
  [1, 2, 3].last
  [1, 2, 3].length
  [1, 2, 3].reverse
  ```

  Heres the list of all the methods available for arrays: Array Methods or Array File

Hash

Hashes are collections of key-value pairs. They are often used to store data in a structured way.

```
# Hashes
{'a' => 1, 'b' => 2}
```

- **Hash Methods**

  Hashes have several methods that can be used to perform operations on them. Some of these methods are:

  - `each` - Iterates over the hash, passing each key-value pair to the given block.
  - `empty?` - Returns true if the hash is empty, false otherwise.
  - `length` - Returns the length of the hash.

    - `keys` - Returns an array containing the keys of the hash.
    - `values` - Returns an array containing the values of the hash.

  ```
  # Hash methods
  {'a' => 1, 'b' => 2}.each { |key, value| puts "#{key}: #{value}" }
  {'a' => 1, 'b' => 2}.empty?
  {'a' => 1, 'b' => 2}.length
  {'a' => 1, 'b' => 2}.keys
  {'a' => 1, 'b' => 2}.values
  ```

  Heres the list of all the methods available for hashes: Hash Methods or Hash File

Boolean

Booleans are logical data types representing true or false.

```
# Booleans
true
false
```

- **Boolean Methods**

  Booleans have several methods that can be used to perform operations on them. Some of these methods are:

    - `!` - Returns the opposite boolean value.
    - `!=` - Returns true if the boolean values are not equal, false otherwise.
    - `&&` - Returns true if both boolean values are true, false otherwise.
    - `||` - Returns true if either boolean value is true, false otherwise.

  ```
  # Boolean methods
  !true # false
  true != false # true
  true && false # false
  true || false # true
  ```

  Heres the list of all the methods available for booleans: Boolean Methods or Boolean File

Nil

Nil represents "nothing" or "no value".

```
# Nil
nil
```

- **Nil Methods**

  Nil has several methods that can be used to perform operations on it. Some of these methods are:

    - `nil?` - Returns true if the object is nil, false otherwise.

  ```
  # Nil methods
  nil.nil? # true
  ```

  Heres the list of all the methods available for nil: Nil Methods or Nil File

Range

Ranges are sequences of values with a start and end point. They can be represented using the `..` and `...` operators.

```
# Ranges
(1..5) # 1, 2, 3, 4, 5
(1...5) # 1, 2, 3, 4
```

- **Range Methods**

  Ranges have several methods that can be used to perform operations on them. Some of these methods are:

    - `each` - Iterates over the range, passing each value to the given block.
    - `first` - Returns the first value in the range.
    - `last` - Returns the last value in the range.

    - `length` - Returns the length of the range.
    - `to_a` - Returns an array containing the values of the range.

  ```
  # Range methods
  (1..5).each { |value| puts value } # 1, 2, 3, 4, 5
  (1..5).first # 1
  (1..5).last # 5
  (1..5).length # 5
  (1..5).to_a # [1, 2, 3, 4, 5]
  ```

  Heres the list of all the methods available for ranges: Range Methods or Range File

Regexp

Regular expressions are used for pattern matching. They can be represented using the `/` character.

```
# Regular expressions
/[A-Z]/ # Matches any uppercase letter
/^\d+$/ # Matches any number
```

- **Regexp Methods**

  Regular expressions have several methods that can be used to perform operations on them. Some of these methods are:

    - `=~` - Returns the index of the first match in the string.
    - `match` - Returns a MatchData object containing information about the match.

  ```
  # Regular expression methods
  "Hello".match(/[A-Z]/) # MatchData object
  "Hello" =~ /[A-Z]/ # 0
  ```

  Heres the list of all the methods available for regular expressions: Regexp Methods or Regexp File

4.5 - Operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numbers. They can be used with integers and floats.

Operator	Description	Example
+	Addition	`5 + 2` will give 7
-	Subtraction	`5 - 2` will give 3
*	Multiplication	`5 * 2` will give 10
/	Division	`5 / 2` will give 2
%	Modulus	`5 % 2` will give 1
**	Exponentiation	`5 ** 2` will give 25

Assignment Operators

Assignment operators are used to assign values to variables. They can be used with any data type. The following table shows the assignment operators available in Ruby.

Operator	Description	Example
=	Assigns a value to a variable	`variable = value`
+=	Adds a value to a variable and assigns the result to the variable	`variable += value`
-=	Subtracts a value from a variable and assigns the result to the variable	`variable -= value`
*=	Multiplies a variable by a value and assigns the result to the variable	`variable *= value`

Operator	Description	Example
/=	Divides a variable by a value and assigns the result to the variable	`variable /= value`
%=	Divides a variable by a value and assigns the remainder to the variable	`variable %= value`
**=	Raises a variable to a power and assigns the result to the variable	`variable **= value`

Comparison Operators

Comparison operators are used to compare two values. They return a boolean value indicating whether the comparison is true or false. The following table shows the comparison operators available in Ruby.

Operator	Description	Example
==	Returns true if the values are equal, false otherwise	`5 == 5` will give true
!=	Returns true if the values are not equal, false otherwise	`5 != 5` will give false
>	Returns true if the left value is greater than the right value, false otherwise	`5 > 2` will give true
<	Returns true if the left value is less than the right value, false otherwise	`5 < 2` will give false
>=	Returns true if the left value is greater than or equal to the right value, false otherwise	`5 >= 2` will give true
<=	Returns true if the left value is less than or equal to the right value, false otherwise	`5 <= 2` will give false
<=>	Returns -1 if the left value is less than the right value, 0 if they are equal, and 1 if the left value is greater than the right value	`5 <=> 2` will give 1
===	Returns true if the values are equal, false otherwise	`5 === 5` will give true
.eql?	Returns true if the values are equal and of the same type, false otherwise	`5.eql?(5)` will give true
equal?	Returns true if the values are the same object, false otherwise	`5.equal?(5)` will give false

Logical Operators

Logical operators are used to combine boolean values. They return a boolean value indicating whether the combination is true or false. The following table shows the logical operators available in Ruby.

Operator	Description	Example
!	Returns the opposite boolean value	`!true` will give false
!=	Returns true if the boolean values are not equal, false otherwise	`true != false` will give true
&&	Returns true if both boolean values are true, false otherwise	`true && false` will give false
||	Returns true if either boolean value is true, false otherwise	`true || false` will give true
and	Returns true if both boolean values are true, false otherwise	`true and false` will give false
or	Returns true if either boolean value is true, false otherwise	`true or false` will give true

Parallel Assignment

Parallel assignment is a way to assign multiple values to multiple variables at the same time. It is done by using the `=` operator.

```
# Parallel assignment
a, b = 1, 2
```

Ternary Operator

The ternary operator is a way to assign a value to a variable based on a condition. It is done by using the `?` and `:` operators.

```
# Ternary operator
variable = condition ? value1 : value2
```

Bitwise Operators

Bitwise operators are used to perform operations on binary numbers. They can be used with integers. The following table shows the bitwise operators available in Ruby.

Operator	Description	Example
&	Performs a bitwise AND operation	`5 & 2` will give 0
|	Performs a bitwise OR operation	`5 | 2` will give 7
^	Performs a bitwise XOR operation	`5 ^ 2` will give 7
~	Performs a bitwise NOT operation	`~5` will give -6
<<	Performs a bitwise left shift operation	`5 << 2` will give 20
>>	Performs a bitwise right shift operation	`5 >> 2` will give 1

Defined? Operator

The defined? operator is used to check if a variable is defined. It returns a boolean value indicating whether the variable is defined or not.

```
# Defined? operator
defined? variable
```

Range Operators

Range operators are used to create ranges. They can be used with integers and floats. The following table shows the range operators available in Ruby.

Operator	Description	Example
..	Creates an inclusive range	`(1..5)` will give 1, 2, 3, 4, 5
...	Creates an exclusive range	`(1...5)` will give 1, 2, 3, 4

Operators Precedence

There are several other operators available in Ruby. The following table shows some of them.

Operator	Description	Example
::	Scope resolution operator	`Math::PI` will give 3.141592653589793
[][]=	Element reference, element set	`array[0] = 1`
**=	Exponentiation assignment	`variable **= value`
! ~ +@ -@	Not, complement, unary plus and minus	`!true` will give false
* / %	Multiplication, division, modulus	`5 * 2` will give 10
+ -	Addition, subtraction	`5 + 2` will give 7
<< >>	Bitwise left shift, bitwise right shift	`5 << 2` will give 20
&	Bitwise AND	`5 & 2` will give 0

Operator	Description	Example
^ |	Bitwise XOR, bitwise OR	`5 ^ 2` will give 7
> >= < <=	Comparison	`5 > 2` will give true
<=> == != === =~ !~	Equality and pattern matching	`5 == 5` will give true
&&	Logical AND	`true && false` will give false
||	Logical OR	`true || false` will give true
.. ...	Range creation	`(1..5)` will give 1, 2, 3, 4, 5
? :	Ternary operator	`variable = condition ? value1 : value2`
= %= { /= -= +=	= &= >>= <<= *= &&=	
defined?	Check if a variable is defined	`defined? variable`
not	Logical negation	`not true` will give false
or and	Logical composition	`true or false` will give true

4.6 - Conditionals

If Statement

The if statement is used to execute a block of code if a condition is true. It is done by using the `if` keyword.

```
# If statement
if condition
  # Code to be executed if condition is true
end
```

The if statement can also be used with the `else` keyword to execute a block of code if the condition is false.

```
# If statement with else
if condition || condition
  # Code to be executed if condition is true
else
  # Code to be executed if condition is false
end
```

The if statement can also be used with the `elsif` keyword to execute a block of code if the condition is false and another condition is true.

```
# If statement with elsif
if condition1 && condition2
  # Code to be executed if condition1 is true
elsif condition2
  # Code to be executed if condition2 is true
end
```

The if statement can also be used with the `unless` keyword to execute a block of code if the condition is false.

```
# If statement with unless
unless condition
  # Code to be executed if condition is false
end
```

The if statement can also be used with the `case` keyword to execute a block of code if the condition is true.

```
# If statement with case
case condition
when condition1
  # Code to be executed if condition1 is true
when condition2
  # Code to be executed if condition2 is true
end
```

Unless Statement

The unless statement is used to execute a block of code if a condition is false. It is done by using the `unless` keyword.

```
# Unless statement
unless condition
  # Code to be executed if condition is false
end
```

The unless statement can also be used with the `else` keyword to execute a block of code if the condition is true.

```
# Unless statement with else
unless condition
  # Code to be executed if condition is false
else
  # Code to be executed if condition is true
end
```

The unless statement can also be used with the `elsif` keyword to execute a block of code if the condition is true and another condition is false.

```
# Unless statement with elsif
unless condition1
  # Code to be executed if condition1 is false
elsif condition2
  # Code to be executed if condition2 is true
end
```

If modifier

The if modifier is used to execute a block of code if a condition is true. It is done by using the `if` keyword.

```
# If modifier
code if condition
```

The if modifier can also be used with the `unless` keyword to execute a block of code if a condition is false.

Unless modifier

The unless modifier is used to execute a block of code if a condition is false. It is done by using the `unless` keyword.

```
# Unless modifier
code unless condition
```

Ternary Operator

The ternary operator is a way to assign a value to a variable based on a condition. It is done by using the `?` and `:` operators.

```
# Ternary operator
variable = condition ? value1 : value2
```

4.7 - Loops

While Loop

The while loop is used to execute a block of code while a condition is true. It is done by using the `while` keyword.

```
# While loop
while condition
  # Code to be executed while condition is true
end
```

- **While Modifier**

  The while modifier is used to execute a block of code while a condition is true. It is done by using the `while` keyword.

  ```
  # While modifier
  code while condition
  ```

Until Loop

The until loop is used to execute a block of code until a condition is true. It is done by using the `until` keyword.

```
# Until loop
until condition
  # Code to be executed until condition is true
end
```

- **Until Modifier**

  The until modifier is used to execute a block of code until a condition is true. It is done by using the `until` keyword.

  ```
  # Until modifier
  code until condition
  ```

For Loop

The for loop is used to iterate over a collection of items. It is done by using the `for` keyword.

```
# For loop
for item in collection
  # Code to be executed for each item in collection
end
```

- **For Modifier**

  The for modifier is used to iterate over a collection of items. It is done by using the `for` keyword.

  ```
  # For modifier
  code for item in collection
  ```

- **break Statement**

  The break statement is used to exit a loop. It is done by using the `break` keyword.

  ```
  # break statement
  for item in collection
    break if condition
    # Code to be executed for each item in collection
  end
  ```

- **next Statement**

  The next statement is used to skip to the next iteration of a loop. It is done by using the `next` keyword.

  ```
  # next statement
  for item in collection
    next if condition
    # Code to be executed for each item in collection
  end
  ```

- **redo Statement**

  The redo statement is used to repeat the current iteration of a loop. It is done by using the `redo` keyword.

```
 # redo statement
for item in collection
  redo if condition
  # Code to be executed for each item in collection
end
```

- **retry Statement**

  The retry statement is used to repeat the current iteration of a loop. It is done by using the `retry` keyword.

  ```
   # retry statement
  for item in collection
    retry if condition
    # Code to be executed for each item in collection
  end
  ```

Each Loop

The each loop is used to iterate over a collection of items. It is done by using the `each` keyword.

```
 # Each loop
collection.each do |item|
  # Code to be executed for each item in collection
end
```

- **Each Modifier**

  The each modifier is used to iterate over a collection of items. It is done by using the `each` keyword.

  ```
   # Each modifier
  collection.each { |item| code }
  ```

Times Loop

The times loop is used to execute a block of code a specified number of times. It is done by using the `times` keyword.

```
 # Times loop
number.times do
  # Code to be executed number of times
end
```

- **Times Modifier**

  The times modifier is used to execute a block of code a specified number of times. It is done by using the `times` keyword.

  ```
   # Times modifier
  number.times { code }
  ```

Begin/End Loop

The begin/end loop is used to execute a block of code until a condition is true. It is done by using the `begin` and `end` keywords.

```
 # Begin/End loop
begin
  # Code to be executed until condition is true
end while condition
```

4.8 - Methods

Method Declaration

Methods are declared using the `def` keyword. The method name is on the left side of the `def` keyword, and the method body is on the right side of the `def` keyword.

```
 # Method declaration
def method_name
  # Method body
end
```

Method Call

Methods are called using the `.` operator. The method name is on the left side of the `.` operator, and the method arguments are on the right side of the `.` operator.

```
 # Method call
object.method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as –

```
 # Method call with parameters
object.method_name(parameter1, parameter2)
# or
object.method_name parameter1, parameter2
```

Method Arguments

Methods can take arguments. Arguments are passed to the method using the `()` operator. The arguments are separated by commas.

```
 # Method arguments
def method_name(argument1, argument2)
  # Method body
end
```

Methods can set default values for their arguments. The default values are set using the `=` operator. The default values are used when no value is passed to the method.

```
 # Method arguments with default values
def method_name(argument1 = value1, argument2 = value2)
  # Method body
end
```

Finally, methods can take a variable number of arguments. The variable number of arguments are passed to the method using the `*` operator. The variable number of arguments are stored in an array.

```
 # Method arguments with variable number of arguments
def method_name(*arguments)
  # Method body
end

#Calling the method
method_name(argument1, argument2, argument3, ...)
```

Method Return Value

The return statement in ruby is used to return one or more values from a Ruby Method.

Basic Syntax:

```
 # Method return value
def method_name
  # Method body
  return [expression[',' expression...]]
end
```

Methods can return a value. The return value is the last expression evaluated in the method body.

```
# Method return value
def method_name
  # Method body
  return value
end
```

Methods can also return multiple values. The return values are separated by commas.

```
# Method return value
def method_name
  # Method body
  return value1, value2
end
```

Method Scope

Methods have a scope that determines where they can be accessed. There are three types of method scope: global, local, and instance.

- **Global Methods**

  Global methods are accessible from anywhere in the program. They are declared using the `$` character.

  ```
  # Global method
  $method_name
  ```

- **Local Methods**

  Local methods are only accessible from within the block in which they are declared. They are declared using the `local` keyword.

  ```
  # Local method
  local method_name
  ```

- **Instance Methods**

  Instance methods are accessible from anywhere within the class in which they are declared. They are declared using the `@` character.

  ```
  # Instance method
  @method_name
  ```

Method Aliases

Methods can be aliased using the `alias` keyword. The alias name is on the left side of the `alias` keyword, and the method name is on the right side of the `alias` keyword.

```
# Method alias
alias alias_name method_name
```

Method Chaining

Methods can be chained together using the `.` operator. The method name is on the left side of the `.` operator, and the method arguments are on the right side of the `.` operator.

```
# Method chaining
object.method_name.name_method
```

Example:

```
# Method chaining example
"Hello".upcase.reverse
```

Method Overloading

Methods can be overloaded using the `def` keyword. The method name is on the left side of the `def` keyword, and the method arguments are on the right side of the `def` keyword.

```
 # Method overloading
def method_name(argument1, argument2)
  # Method body
end

def method_name(argument1, argument2, argument3)
  # Method body
end
```

Method Overriding

Methods can be overridden using the `def` keyword. The method name is on the left side of the `def` keyword, and the method arguments are on the right side of the `def` keyword.

```
 # Method overriding
def method_name(argument1, argument2)
  # Method body
end

def method_name(argument1, argument2, argument3)
  # Method body
end
```

Method Access Control

Methods can be accessed using the `public`, `protected`, and `private` keywords. The method name is on the left side of the `public`, `protected`, and `private` keywords, and the method arguments are on the right side of the `public`, `protected`, and `private` keywords.

```
 # Method access control
public method_name
protected method_name
private method_name
```

Method Reflection

Methods can be reflected using the `method` keyword. The method name is on the left side of the `method` keyword, and the method arguments are on the right side of the `method` keyword.

```

# Method reflection
method(:method_name)
```

Exanple:

```
 # Method reflection example
puts "Hello".method(:upcase).call() # HELLO
```

Method Metaprogramming

Methods can be metaprogrammed using the `define_method` keyword. The method name is on the left side of the `define_method` keyword, and the method arguments are on the right side of the `define_method` keyword.

```
 # Method metaprogramming
define_method(:method_name) do |argument1, argument2|
  # Method body
end
```

Method Decorators

Methods can be decorated using the `method_added` keyword. The method name is on the left side of the `method_added` keyword, and the method arguments are on the right side of the `method_added` keyword.

```
 # Method decorators
def method_added(method_name)
  # Method body
end
```

Method Hooks

Ruby Hook Methods are called in reaction to something you do. They are usually used to extend the working of methods at run time. These methods are not defined by default, but a programmer can define them according to imply them on any object or class or module and they will come into picture when certain events occur. These methods are called automatically when certain events occur.

There are several Ruby Hook Methods, but majorly, the followings have major roles to play:

1. Included
2. Prepended
3. Extended
4. Inherited
5. method_missing
6. More methods are available on the Ruby Docs

- **Included**

  This method is used to include a method or attribute or module to another module. The method makes the underlined module available to the instances of the class. The following example explains the usage and working of the include method.

  Example:

  ```
   # Declaring a module to greet a person
  module Greetings

    def self.included(person_to_be_greeted)

      puts "The #{person_to_be_greeted} is welcomed with an open heart !"
    end
  end

  # Class where the module is included
  class Person

    include Greetings # implementation of the include statement
  end

  # Output
  The Person is welcomed with an open heart !
  ```

- **Prepended**

  This method was brought by Ruby 2.0. This is slightly different from what we observed above. Prepended method provides another way of extending the functioning of modules at different places. This uses the concept of overriding. The modules can be overridden using methods defined in the target class.

  Example:

  ```
   # Code as an example for prepend method
  module Ruby

  def self.prepended(target)# Implementation of prepend method
    puts "#{self} has been prepended to #{target}"
  end

  def Type
    "The Type belongs to Ruby"
  end
  end

  class Coding

  prepend Ruby # the module Ruby is prepended
  end

  # Method call
  puts Coding.new.Type

  # Output
  Ruby has been prepended to Coding
  ```

- **Extended**

  This method is a bit different from both the include and prepend method. While include applies methods in a certain module to instance of a class, extend applies those methods to the same class.

Example:

```
 # Code as an example for prepend method
module Ruby

def self.prepended(target)# Implementation of prepend method
  puts "#{self} has been prepended to #{target}"
end

def Type
  "The Type belongs to Ruby"
end
end

class Coding

prepend Ruby # the module Ruby is prepended
end

# Method call
puts Coding.new.Type

# Output
Ruby has been prepended to Coding
The Type belongs to Ruby
```

- **Inherited**

  Inheritance as a concept is one of the most important concepts of Object Oriented Programming and is common in almost every programming language. In ruby, we deal in objects that are inspired from the real life, and thus, Oops operations play a very important role there. The inherited method is called whenever a subclass of a class is implemented. It is a method of making a child class from a parent class.

  Example:

```
 # Making the parent Vehicle class
class Vehicle

  def self.inherited(car_type)
    puts "#{car_type} is a kind of Vehicle"
  end

end

# Target class
class Hyundai < Vehicle #Inhereting the Vehicle class
end

# Output
Hyundai is a kind of Vehicle
```

- **method_missing**

  method_missing method which is one of the most widely used in Ruby. This comes to action when one tries to call a method on an object that does not exist.

  Example:

```
 # The main class
class Ruby

def method_missing(input, *args) # method_missing function in action
  "#{input} not defined on #{self}"
end

def Type
  "The Type is Ruby"
end
end

var = Ruby.new

# Calling a method that exists
puts var.Type

# Calling a method that does not exist
puts var.Name

# Output
The Type is Ruby
Name not defined on #<Ruby:0x0000000001e2f6c0>
```

Recursive Methods

Recursive methods are methods that call themselves. They are used to solve problems that can be solved by breaking them down into smaller problems.

```
 # Recursive method
def method_name(argument)
  # Method body
  method_name(argument)
end
```

Example:

```
 # Recursive method example
def factorial(n)
  if n == 0
    1
  else
    n * factorial(n - 1)
  end
end

puts factorial(5) # 120
```

4.9 - Blocks

Block Syntax

Blocks are used to group statements together. They are declared using the `do` keyword. The block body is on the right side of the `do` keyword.

```
 # Block syntax
do
  # Block body
end
```

Example:

```
 # Block syntax example
[1, 2, 3].each do |item|
  puts item
end
```

Block Call

Blocks are called using the `yield` keyword. The block body is on the right side of the `yield` keyword.

```
 # Block call
yield
```

Example:

```
 # Block call example
def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}

method_name { puts "Hello" } # Hello
```

Block Arguments

Blocks can take arguments. Arguments are passed to the block using the `|` operator. The arguments are separated by commas.

```

# Block arguments
do |argument1, argument2|
  # Block body
end
```

Example:

```
 # Block arguments example
[1, 2, 3].each do |item, index|
  puts "#{index}: #{item}"
end
```

Block Return Value

Blocks can return a value. The return value is the last expression evaluated in the block body.

```
 # Block return value
do
  # Block body
  return value
end
```

Example:

```
 # Block return value example
[1, 2, 3].each do |item|
  return item
end
```

Block Scope

Blocks have a scope that determines where they can be accessed. There are three types of block scope: global, local, and instance.

- **Global Blocks**

  Global blocks are accessible from anywhere in the program. They are declared using the `$` character.

  ```
   # Global block
  $do
    # Block body
  end
  ```

- **Local Blocks**

Local blocks are only accessible from within the block in which they are declared. They are declared using the `local` keyword.

```
 # Local block
local do
  # Block body
end
```

- **Instance Blocks**

  Instance blocks are accessible from anywhere within the class in which they are declared. They are declared using the `@` character.

```
 # Instance block
@do
  # Block body
end
```

Block Aliases

Blocks can be aliased using the `alias` keyword. The alias name is on the left side of the `alias` keyword, and the block name is on the right side of the `alias` keyword.

```


# Block alias


alias alias_name do
  # Block body
end
```

Block Chaining

Blocks can be chained together using the `.` operator. The block name is on the left side of the `.` operator, and the block arguments are on the right side of the `.` operator.

```
 # Block chaining
do.name do |argument1, argument2|
  # Block body
end
```

Begin and End Blocks

Begin and end blocks are used to group statements together. They are declared using the `begin` and `end` keywords. The block body is on the right side of the `begin` and `end` keywords.

```
 # Begin and end block
begin
  # Block body
end
```

Example:

```
 # Begin and end block example
begin
  puts "Hello"
end
```

4.10 - Lambdas

Lambda Syntax

Lambdas are used to group statements together. They are declared using the `->` operator. The lambda body is on the right side of the `->` operator.

```
 # Lambda syntax
-> do
  # Lambda body
end
```

Example:

```
 # Lambda syntax example

-> { puts "Hello" }

# or

lambda { puts "Hello" }
```

Lambda Call

Lambdas are called using the `call` keyword. The lambda body is on the right side of the `call` keyword.

```
 # Lambda call
lambda.call
```

Example:

```
 # Lambda call example
-> { puts "Hello" }.call

# or

lambda { puts "Hello" }.call
```

Lambda Arguments

Lambdas can take arguments. Arguments are passed to the lambda using the `|` operator. The arguments are separated by commas.

```
 # Lambda arguments
-> (argument1, argument2) do
  # Lambda body
end
```

Example:

```
 # Lambda arguments example
-> (item, index) { puts "#{index}: #{item}" }.call

# or

lambda { |item, index| puts "#{index}: #{item}" }.call
```

Lambda Return Value

Lambdas can return a value. The return value is the last expression evaluated in the lambda body.

```
 # Lambda return value
-> do
  # Lambda body
  return value
end
```

Example:

```
 # Lambda return value example
-> { return "Hello" }.call

# or

lambda { return "Hello" }.call
```

Lambda Scope

Lambdas have a scope that determines where they can be accessed. There are three types of lambda scope: global, local, and instance.

- **Global Lambdas**

  Global lambdas are accessible from anywhere in the program. They are declared using the `$` character.

  ```
   # Global lambda
  $->
    # Lambda body
  end
  ```

- **Local Lambdas**

  Local lambdas are only accessible from within the lambda in which they are declared. They are declared using the `local` keyword.

  ```
   # Local lambda
  local ->
    # Lambda body
  end
  ```

- **Instance Lambdas**

  Instance lambdas are accessible from anywhere within the class in which they are declared. They are declared using the `@` character.

  ```
   # Instance lambda
  @->
    # Lambda body
  end
  ```

Lambda Aliases

Lambdas can be aliased using the `alias` keyword. The alias name is on the left side of the `alias` keyword, and the lambda name is on the right side of the `alias` keyword.

```
 # Lambda alias
alias alias_name ->
  # Lambda body
end
```

Lambda Chaining

Lambdas can be chained together using the `.` operator. The lambda name is on the left side of the `.` operator, and the lambda arguments are on the right side of the `.` operator.

```
 # Lambda chaining
->.name (argument1, argument2) do
  # Lambda body
end
```

Lambdas vs Methods vs Block

Lambdas, methods, and blocks are similar in many ways. They all have a name, arguments, and a body. However, there are some differences between them. The following table shows the differences between lambdas, methods, and blocks.

Lambda	Method	Block
Lambdas are used to group statements together.	Methods are used to group statements together.	Blocks are used to group statements together.
Lambdas are declared using the `->` operator.	Methods are declared using the `def` keyword.	Blocks are declared using the `do` keyword.
Lambdas are called using the `call` keyword.	Methods are called using the `.` operator.	Blocks are called using the `yield` keyword.
Lambdas can take arguments.	Methods can take arguments.	Blocks can take arguments.
Lambdas can return a value.	Methods can return a value.	Blocks can return a value.
Lambdas have a scope that determines	Methods have a scope that determines	Blocks have a scope that determines

Lambda	Method	Block
where they can be accessed.	where they can be accessed.	where they can be accessed.
Lambdas can be aliased using the `alias` keyword.	Methods can be aliased using the `alias` keyword.	Blocks can be aliased using the `alias` keyword.
Lambdas can be chained together using the `.` operator.	Methods can be chained together using the `.` operator.	Blocks can be chained together using the `.` operator.

- **Lambdas**

Example:

```
 # Lambdas example
-> { puts "Hello" }.call
```

- **Methods**

Example:

```
 # Methods example
def method_name
  puts "Hello"
end

method_name
```

- **Blocks**

Example:

```


# Blocks example


do
  puts "Hello"
end
```

4.11 - Modules

Module Declaration

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

```
 # Module declaration
module ModuleName
  # Module body
end
```

Example:

```
 # Module declaration example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

Greetings.say_hello # Hello
```

Module Call

Modules are called using the `include` keyword. The module name is on the left side of the `include` keyword, and the module arguments are on the right side of the `include` keyword.

```
 # Module call
include ModuleName
```

Example:

```
 # Module call example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

include Greetings

say_hello # Hello
```

Module Constants

Module constants are named just like class constants, with an initial uppercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

```
 # Module constants
ModuleName::CONSTANT_NAME
```

Example:

```
 # Module constants example
module Greetings
  CONSTANT_NAME = "Hello"
end

puts Greetings::CONSTANT_NAME # Hello
```

Module Methods

Module methods are named just like class methods, with an initial lowercase letter. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

```
 # Module methods
ModuleName.method_name
```

Example:

```
 # Module methods example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

Greetings.say_hello # Hello
```

Module Variables

Module variables are named just like class variables, with an initial `@` character. The method definitions look similar, too: Module methods are defined just like class methods.

As with class methods, you call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

```
 # Module variables
ModuleName.@variable_name
```

Example:

```
 # Module variables example
module Greetings
  @variable_name = "Hello"
end

puts Greetings.@variable_name # Hello
```

Module Aliases

Modules can be aliased using the `alias` keyword. The alias name is on the left side of the `alias` keyword, and the module name is on the right side of the `alias` keyword.

```
 # Module alias
alias alias_name ModuleName
```

Example:

```
 # Module alias example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

alias Greetings2 Greetings

Greetings2.say_hello # Hello
```

Module Chaining

Modules can be chained together using the `.` operator. The module name is on the left side of the `.` operator, and the module arguments are on the right side of the `.` operator.

```
 # Module chaining

ModuleName.name ModuleName
```

Example:

```
 # Module chaining example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

Greetings.say_hello # Hello
```

Module Inheritance

Modules can be inherited using the `<` operator. The module name is on the left side of the `<` operator, and the module arguments are on the right side of the `<` operator.

```
 # Module inheritance
ModuleName < ModuleName
```

Example:

```
# Module inheritance example
module Greetings
  def self.say_hello
    puts "Hello"
  end
end

module Greetings2 < Greetings
  def self.say_hello2
    puts "Hello2"
  end
end

Greetings2.say_hello # Hello

Greetings2.say_hello2 # Hello2
```

Module Hooks

All the method hooks can be applied here. Check the Method Hooks section for more information.

4.12 - Date and Time

Date

The Date class is used to represent dates. It is done by using the `Date` keyword.

```
 # Date
require 'date'

Date
```

Example:

```
 require 'date'
# Date example
Date.today # 2021-10-20
```

Time

The Time class represents dates and times in Ruby. It is a thin layer over the system date and time functionality provided by the operating system. This class may be unable on your system to represent dates before 1970 or after 2038.

```
 # Time
Time.new
```

Example:

```
 # Time example
time = Time.new

puts time.inspect # Mon Jun 02 12:03:08 -0700 2008
```

- **Componentes of a Date & Time**

  The Time class has several components that can be accessed using the `.` operator. The following table shows the components of a Time object.

  Example for the following table:

  ```
   # Time example
  time = Time.new

  puts time.inspect # Mon Jun 02 12:03:08 -0700 2008
  ```

Component	Description	Example
year	The year	`time.year` will give 2008
month	The month	`time.month` will give 6
day	The day	`time.day` will give 2
wday	The day of the week	`time.wday` will give 1 (Sunday - 0, Monday - 1, ..7)
yday	The day of the year	`time.yday` will give 154
hour	The hour	`time.hour` will give 12
min	The minute	`time.min` will give 3
sec	The second	`time.sec` will give 8
usec	The microsecond	`time.usec` will give 247476
zone	The time zone	`time.zone` will give "UTC" (UTC is the abbreviation for Coordinated Universal Time)

- **Time.utc & Time.gm & Time.local Functions**

  The Time class has several functions that can be used to create Time objects. The following table shows the functions of the Time class.

Function	Description	Example
Time.utc	Creates a Time object in UTC time zone	`Time.utc(2008, 6, 2, 12, 3, 8)` will give 2008-06-02 12:03:08 UTC
Time.gm	Creates a Time object in UTC time zone	`Time.gm(2008, 6, 2, 12, 3, 8)` will give 2008-06-02 12:03:08 UTC
Time.local	Creates a Time object in local time zone	`Time.local(2008, 6, 2, 12, 3, 8)` will give 2008-06-02 12:03:08 +0100

Following is the example to get all the components in an array in the following format

```
[ sec, min, hour, day, month, year, wday, yday, isdst, zone ]
```

- **Timezones and Daylight Savings Time**

  You can use a Time object to get all the information related to Timezones and daylight savings as follows

Function	Description	Example
time.utc?	Returns true if time represents a time in UTC (GMT)	`time.utc?` will give false
time.zone	Returns the name of the time zone used for time	`time.zone` will give "UTC"
time.isdst	Returns true if time occurs during Daylight Saving Time in its time zone	`time.isdst` will give false
time.utc_offset	Returns the offset in seconds between the timezone of time and UTC	`time.utc_offset` will give 0
time.localtime	Returns a new Time object representing time in local time zone	`time.localtime` will give Mon Jun 02 12:03:08 +0100 2008
time.gmtime	Returns a new Time object representing time in UTC	`time.gmtime` will give Mon Jun 02 11:03:08 UTC 2008
time.getlocal	Returns a new Time object representing time in local time zone	`time.getlocal` will give Mon Jun 02 12:03:08 +0100 2008
time.getutc	Returns a new Time object representing time in UTC	`time.getutc` will give Mon Jun 02 11:03:08 UTC 2008

- **Time and Date Formatting**

  You can use a Time object to get all the information related to Timezones and daylight savings as follows

Function	Description	Example
time.to_s	Returns a string representing time	`time.to_s` will give "Mon Jun 02 12:03:08 +0100 2008"

Function	Description	Example
time.ctime	Returns a string representing time	`time.ctime` will give "Mon Jun 02 12:03:08 +0100 2008"
time.localtime	Returns a string representing time	`time.localtime` will give "Mon Jun 02 12:03:08 +0100 2008"
time.strftime	Formats time according to the directives in the given format string	`time.strftime("%Y-%m-%d %H:%M:%S")` will give "2008-06-02 12:03:08"

- **Time Formatting Directives**

  The following table shows the formatting directives that can be used with the `strftime` function.

Directive	Description	Example
%a	The abbreviated weekday name ("Sun")	`time.strftime("%a")` will give "Mon"
%A	The full weekday name ("Sunday")	`time.strftime("%A")` will give "Monday"
%b	The abbreviated month name ("Jan")	`time.strftime("%b")` will give "Jun"
%B	The full month name ("January")	`time.strftime("%B")` will give "June"
%c	The preferred local date and time representation	`time.strftime("%c")` will give "Mon Jun 02 12:03:08 2008"
%d	Day of the month (01..31)	`time.strftime("%d")` will give "02"
%H	Hour of the day, 24-hour clock (00..23)	`time.strftime("%H")` will give "12"
%I	Hour of the day, 12-hour clock (01..12)	`time.strftime("%I")` will give "12"
%j	Day of the year (001..366)	`time.strftime("%j")` will give "154"
%m	Month of the year (01..12)	`time.strftime("%m")` will give "06"
%M	Minute of the hour (00..59)	`time.strftime("%M")` will give "03"
%p	Meridian indicator ("AM" or "PM")	`time.strftime("%p")` will give "PM"
%S	Second of the minute (00..60)	`time.strftime("%S")` will give "08"
%U	Week number of the current year, starting with the first Sunday as the first day of the first week (00..53)	`time.strftime("%U")` will give "22"
%W	Week number of the current year, starting with the first Monday as the first day of the first week (00..53)	`time.strftime("%W")` will give "22"
%w	Day of the week (Sunday is 0, 0..6)	`time.strftime("%w")` will give "1"
%x	Preferred representation for the date alone, no time	`time.strftime("%x")` will give "06/02/08"
%X	Preferred representation for the time alone, no date	`time.strftime("%X")` will give "12:03:08"
%y	Year without a century (00..99)	`time.strftime("%y")` will give "08"
%Y	Year with century	`time.strftime("%Y")` will give "2008"
%Z	Time zone name	`time.strftime("%Z")` will give "UTC"
%%	Literal "%" character	`time.strftime("%%")` will give "%"

- **Time Arithmetic**

  You can perform simple arithmetic with time as follows

```
 # Time arithmetic
now = Time.now          # Current time

past = now - 10         # 10 seconds ago. Time - number => Time

future = now + 10       # 10 seconds from now Time + number => Time

diff = future - now     # => 10  Time - Time => number of seconds

now > future            # => false  Time > Time => true or false

now < future            # => true  Time < Time => true or false

puts diff              # => 20.0
```

4.13 - File I/O

Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class IO provides all the basic methods, such as read, write, gets, puts, readline, getc, and printf.

This chapter will cover all the basic I/O functions available in Ruby. For more functions, please refer to Ruby Class IO.

Puts Statement

The puts statement is used to print on the screen. It adds a new line character at the end of the output.

Example:

```
 # Puts statement example
puts "Hello" # Hello
```

Print Statement

The print statement is used to print on the screen. It does not add a new line character at the end of the output.

Example:

```
 # Print statement example
print "Hello"
print " Daniel"

# Output
Hello Daniel
```

Get Statement

The get statement is used to get input from the user. It does not add a new line character at the end of the output.

Example:

```
 # Get statement example
puts "Enter your name: "
name = gets

puts "Hello #{name}, how are you?"

# Output
Enter your name:
Daniel
Hello Daniel
, how are you?
```

- **Chomp Method**

  The chomp method is used to remove the new line character from the end of the string.

  Example:

```
 # Chomp method example
puts "Enter your name: "
name = gets.chomp

puts "Hello #{name}, how are you?"

# Output
Enter your name:
Daniel
Hello Daniel , how are you?
```

Putc Statement

Unlike the puts statement, which outputs the entire string onto the screen, the putc statement can be used to output one character at a time.

Example:

```
 # Putc statement example
putc "Hello" # H
```

Opening and Closing Files

Ruby provides a whole set of I/O-related methods implemented in the Kernel module. All the I/O methods are derived from the class IO.

The class IO provides all the basic methods, such as read, write, gets, puts, readline, getc, and printf.

- **Opening Files**

  You can open a file using the `File.new` method. The `File.new` method takes two arguments: the name of the file and the mode in which you want to open the file.

  The following table shows the different modes in which you can open a file.

Mode	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.
w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

  Example:

  ```
   # Opening files example
  file = File.new("filename", "mode")

  # or

  file = File.new("filename", "mode") if File::exists?( "filename" )

  # or

  File.open("filename", "mode") do |file|
    # ...
  end
  ```

- **Sysread and Syswrite Method**

  The sysread and syswrite methods are used to read and write data from and to a file. They are similar to the read and write methods, but they do not buffer the data.

  Example:

```
 # Sysread and syswrite method example
file = File.new("filename", "mode")

if file
  content = file.sysread(20) # First 20 characters of the file, the file pointer is moved to the 21st character
  puts content # Hello, how are you?
  file.syswrite("Im fine, thank you!") # Write to the file from the 21st character

  content = file.sysread(20) # Next 20 characters of the file
  puts content # Im fine, thank you!
else
  puts "Unable to open file!"
end

# Output
Hello, how are you?
```

- **each_byte Method**

  The each_byte method is used to read a file byte by byte. It returns an enumerator object.

  Example:

  ```
   # Each_byte method example
  aFile = File.new("input.txt", "r+")
  if aFile
    aFile.syswrite("ABCDEF")
    aFile.each_byte {|ch| putc ch; putc ?. }
  else
    puts "Unable to open file!"
  end

  # Output
  A.B.C.D.E.F.
  ```

- **IO.readlines Method**

  The IO.readlines method is used to read a file line by line. It returns an array of lines.

  Example:

  ```
   # IO.readlines method example
  arr = IO.readlines("input.txt")
  puts arr[0] # This is line one
  puts arr[1] # This is line two

  # Output
  This is line one
  This is line two
  ```

- **IO.foreach Method**

  The IO.foreach method is used to read a file line by line. It returns an enumerator object.

  Example:

  ```
   # IO.foreach method example
  IO.foreach("input.txt"){|block| puts block}

  # Output
  This is line one
  This is line two
  ```

Renaming and Deleting Files

- **Rename Method**

  The rename method is used to rename a file. It takes two arguments: the old name of the file and the new name of the file.

  Example:

  ```
   # Rename method example
  File.rename( "test1.txt", "test2.txt" )
  ```

- **Delete Method**

  The delete method is used to delete a file. It takes one argument: the name of the file.

Example:

```
# Delete method example
File.delete("test2.txt")
```

File Modes and Ownership

The following table shows the different modes in which you can open a file.

Mode	Description
r	Read-only mode. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Read-write mode. The file pointer will be at the beginning of the file.
w	Write-only mode. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Read-write mode. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Write-only mode. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Read and write mode. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- **File Ownership**

  The File class provides several methods that can be used to get information about the owner of a file. The following table shows the methods that can be used to get information about the owner of a file.

Method	Description
File.owned?	Returns true if the named file exists and the effective used id of the calling process is the owner of the file.
File.grpowned?	Returns true if the named file exists and the effective group id of the calling process is the owner of the file.
File.owned?	Returns true if the named file exists and the effective used id of the calling process is the owner of the file.
File.owned?	Returns true if the named file exists and the effective used id of the calling process is the owner of the file.
File.chmod	Changes permission bits on the named file to the bit pattern represented by mode_int.

  Example

```
# File ownership example
File.chmod(0755, "test.txt")
```

Other File Methods

Check the Ruby Class IO for more information.

4.14 - Exceptions

Exception Handling

Exception handling is used to handle errors that occur during the execution of a program. It is done by using the `begin`, `rescue`, and `end` keywords.

```
# Exception handling
begin
  # Code that might raise an exception
rescue
  # Code that will execute when an exception is raised
else
  # Code that will execute if no exception is raised
ensure
  # Code that will always execute
end
```

Everything from begin to rescue is protected. If an exception occurs during the execution of this block of code, control is passed to the block between rescue and end.

For each rescue clause in the begin block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

Example

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"

# Output
# #<IO:0x401b3944>==#<IO:0x401b3944>
```

Retry Statement

The retry statement is used to repeat the execution of the begin block.

```
# Retry statement
begin
  # Code that might raise an exception
rescue
  # Code that will execute when an exception is raised
  retry # This will cause the program to repeat the begin block
end
```

Example

```
begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end

# Output
File opened successfully # existant_file
```

The following is the flow of the process

- An exception occurred at open.
- Went to rescue. fname was re-assigned.
- By retry went to the beginning of the begin.
- This time file opens successfully.
- Continued the essential process.

NOTE – Notice that if the file of re-substituted name does not exist this example code retries infinitely. Be careful if you use retry for an exception process.

Raise Statement

The raise statement is used to raise an exception. It is done by using the `raise` keyword.

```
 # Raise statement
raise

# or

raise "Error Message"

# or

raise ExceptionType, "Error Message"

# or

raise ExceptionType, "Error Message" condition
```

The first form simply re-raises the current exception (or a RuntimeError if there is no current exception). This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new RuntimeError exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument.

The fourth form is similar to the third form but you can add any conditional statement like unless to raise an exception.

Example

```
 #!/usr/bin/ruby

begin
   puts 'I am before the raise.'
   raise 'An error has occurred.'
   puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'

# Output

I am before the raise.
I am rescued.
I am after the begin block.
```

Ensure Statement

Sometimes, you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block and you need to make sure it gets closed as the block exits.

The ensure clause does just this. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception, the ensure block will get run.

```
 begin
   #.. process
   #..raise exception
rescue
   #.. handle error
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example

```
 begin
   raise 'A test exception.'
rescue Exception => e
   puts e.message
   puts e.backtrace.inspect
ensure
   puts "Ensuring execution"
end

# Output
A test exception.
["main.rb:2:in `block in <main>'", "main.rb:1:in `<main>'"]
Ensuring execution
```

Else Statement

If the else clause is present, it goes after the rescue clauses and before any ensure.

The body of an else clause is executed only if no exceptions are raised by the main body of code.

```
 begin
   #.. process
   #..raise exception
rescue
   # .. handle error
else
   #.. executes if there is no exception
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example

```
 begin
   # raise 'A test exception.'
   puts "I'm not raising exception"
rescue Exception => e
    puts e.message
    puts e.backtrace.inspect
else
    puts "Congratulations-- no errors!"
ensure
    puts "Ensuring execution"
end

# Output
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

Catch and Throw

Ruby provides the catch and throw keywords to handle situations where you want to terminate the execution of a block of code.

The catch and throw keywords are used together. The catch keyword is used to define a block of code that can be terminated using the throw keyword.

```
 # Catch and throw
catch :lablename do
  # Code that might throw an exception
  throw :lablename
end
```

Example

```
 def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # process information
end
promptAndGet("Name:")


# Output

Name: Daniel
Age: 12
Sex: !
Name: Daniela
```

Exception Class Methods

Ruby's standard classes and modules raise exceptions. All the exception classes form a hierarchy, with the class Exception at the top. The next level contains seven different types

- NoMemoryError
- ScriptError
- Interrupt
- NoMemoryError
- SignalException
- StandardError
- SystemExit

There is one other exception at this level, Fatal, but the Ruby interpreter only uses this internally.

Both ScriptError and StandardError have a number of subclasses, but we do not need to go into the details here. The important thing is that if we create our own exception classes, they need to be subclasses of either class Exception or one of its descendants.

Example

```
 class FileSaveError < StandardError
   attr_reader :reason
   def initialize(reason)
      @reason = reason
   end
end
```

Now, look at the following example, which will use this exception

```
 File.open(path, "w") do |file|
begin
   # Write out the data ...
rescue
   # Something went wrong!
   raise FileSaveError.new($!)
end
end

# Output

FileSaveError: Permission denied - data.txt
```

The important line here is raise FileSaveError.new($!). We call raise to signal that an exception has occurred, passing it a new instance of FileSaveError, with the reason being that specific exception caused the writing of the data to fail.

The $! global variable contains the last exception that was raised, so we pass this to the constructor of FileSaveError so that the exception object will contain all the information about the original exception.

4.15 - Object Oriented Programming

Object Oriented Programming

Classes

- Constructor
- Getters and Setters
- Access Control

Objects

Methods

Instance Variables

Class Variables

Class Methods

Inheritance

Polymorphism

Mixins

4.16 - Regular Expressions

Regular Expressions

Regular Expressions Methods

Regular Expressions Patterns