HTML/CSS Javascript and Ruby Tutorial

This tutorial aims to provide a thorough explanation of three key elements related to web development: HTML/CSS, Javascript, and Ruby. Programmers with little to no web development experience should find this tutorial helpful, however, those without prior programming experience should first consult an introductory tutorial first, in order to learn the basic concepts of computer programming.

Written By: Amna Khan, Bryan DAgostino, Boming Zhang, Derek DiCillo, and Summer Cui.

HTML/CSS

HTML

The HTML section will focus primarily on the uniqueness of the language and how it is used in web development. This section will cover the several aspects of HTML and how to create a webpage-using HTML.

HTML means **H**yper **T**ext **M**arkup **L**anguage; it is a language for defining webpages. Moreover, a markup language is a set of markup *tags* that describe the content in the document. An HTML document is made up of HTML *tags* and plain *text*; it is also referred to as *web pages*.

HTML Versions

There are many versions of HTML, and the latest version being HTML5. The difference between an older version and the new HTML5 is: HTML5 contains new types of *tags* and far more powerful. It is consistent, and its objective is to develop the capability of the browser to be an application platform by using HTML, CSS, and JavaScript.

HTML Document Setup

Before the start of writing the language, an HTML document needs to be created. In order to set up an HTML document, open up a New File in any text editor and save it as “yourname.html”.

HTML Tags

HTML tags are enclosed by *angle brackets*, and they usually come in pairs. For example:

```
<html> </html>
```

In this case, `<html>` is known as the start tag and `</html>` is known as the end tag. Start and end tags can be referred to as opening and closing tags, respectively. The basic structure of HTML tags is:

```
<tagname> text </tagname>
```

Check here for a detailed list of HTML5 tags.

The <!DOCTYPE> Declaration

Different versions of HTML has different declarations for its document type, this is because the web browser can only show an HTML page properly if it can identify the HTML type and version used.

The purpose of the <!DOCTYPE> declaration is primarily to display the web page properly. In the beginning of the HTML document, the document type is declared. An example of how to declare a document type is:

```
<!-- In this case, it is for an HTML5 document -->
<!DOCTYPE html>
```

HTML Elements

An HTML element is precisely what is between the start and end tags. For example:

```
<b> This is an HTML element </b>
```

HTML Page Structure

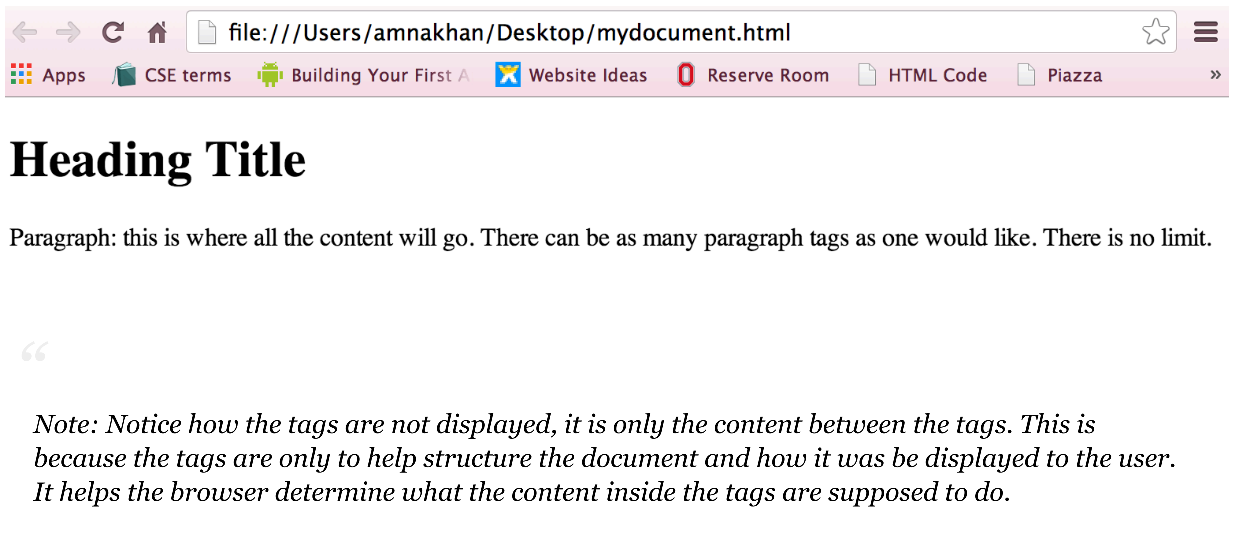
A basic webpage will contain an html, body, heading, and paragraph tag. An example of the page structure should look as follows:

```
<!-- Assuming the document type declaration is already inserted -->
<html>
    <body>
        <h1> Heading Title </h1>
        <p>
            Paragraph: this is where all the content will go.
            There can be as many paragraph tags as one would like.
        </p>
    </body>
</html>
```

> *Note: The best way to understand HTML is to see each start and end tag as a block. Moreover, pay close attention how the blocks are nested. It is important that all other tags besides the document type declaration are inside the `html` tags.*

The Output on the Web Browser

Opening the HTML document in a web browser, the output of the document should display as follows:



Heading Title

Paragraph: this is where all the content will go. There can be as many paragraph tags as one would like. There is no limit.

> *Note: Notice how the tags are not displayed, it is only the content between the tags. This is because the tags are only to help structure the document and how it was be displayed to the user. It helps the browser determine what the content inside the tags are supposed to do.*

CSS

The CSS section will focus primarily on the unique aspects of the language and how it is used in web development corresponding with HTML. This section will cover the several aspects of CSS and how to stylize a webpage using CSS.

CSS means **C**ascading **S**tyle **S**heets; it primarily is used to style the HTML elements. There are three ways to stylize a webpage using CSS: external style sheet, internal style sheet, and inline style.

Originally, HTML was not made to style and format a document but to display content in a heading and paragraphs. In the later versions, fonts and styles were introduced using internal inline style and internal style sheet but it became messy and long if changes needed to be made in the document. This is when the external style sheet was introduced.

In an HTML document, the inline style and internal style sheet are more work if there needs to be a single change in the overall style of the webpage whereas the external style sheets uses the idea of *single point of control*.

Inline Style

Inline style will only change the font and/or format of one specific block. An example of an inline style is as follows:

```
<p style=”color:blue;”> This is the text in a certain paragraph that will be blue. </p>
```

Internal Style Sheet

Internal style sheet will change the overall document but it can become long and messy if it is in the HTML document. An example of an internal style sheet is as follows:

```
<head>
    <style>
        p {color: blue;}
        body {background-image:url("imgURL.png");}
    </style>
</head>
```

> “
>
> *Note: This will change all the paragraphs to the color blue. One can use classes and id’s to change only a specific paragraph, this is also a better way to organize styles. This will be introduced later.*

External Style Sheet

In an HTML document, the internal style tag will display as follows:

```
<style> style and format rules go here </style>
```

However, to link a CSS file in the HTML document, it will display as follows:

```
<link rel="stylesheet" type=”text/css” href=”yourname.css" />
```

> “
>
> *Note: This will link the CSS file to the HTML Document. Hence, A CSS file will need to created naming it `“yourname.css”`. For better understanding, notice how the style tags or the link tag and element to the CSS file will be inserted inside the head tags in the HTML document.*

The Basics

To get the flow of how CSS works, here is an example of how to stylize the body:

```
body {
    background-color: #333333;
    color: #999999;
    font: verdana;
}
```

> “

Note: This will change the style of the entire body in the HTML document. There are many ways to format an HTML document. One is able to change an entire element in the HTML document like a table or body, this is just one example for a body element.

ID and Class

To style an HTML element, CSS permits specifying your own selectors known as “id” and “class”. An id attribute is for stylizing a single element whereas a class is used for a group of elements.

An an HTML document, an example of how an id and class will be displayed is as follows:

```
<div = id=”thisisid”>
    <p class=”thisisclass”> This is an example paragraph. </p>
</div>
```

So, the CSS may look something like this:

```
#thisisid {
    background-color: #ffffff;
}
    .thisisclass {
    color: red;
    font-weight: bold;
}
```

“

Note: Each id should appear only once in an html file. Classes, however, may be used repeatedly to add style to multiple elements.

Priority

Now the three different ways to stylize an HTML document have been discussed: inline style, internal style sheet, and external style sheet, all these can be combined together and used all at once. In addition, each one has a priority level. The inline (inside the HTML element) is given first priority. Second, the internal style sheet (in the head tag) has the next priority. Then, the external style sheet (CSS file created) has priority. Lastly, the browser default has priority. Priority can be overridden using the `!important` flag after the style. For example:

```
body {
    color: red !important;
}
```

This style will receive top priority. Priority within a stylesheet is also determined by specificity, which is related to the number and type of handles associated with each style. Inline styles are given top priority, then styles with the most ids specified, then those with the most classes specified, then those with the most tags specified. For example (in order of decreasing priority):

```
/* Highest priority */
body {
    color: red !important;
}

#element1 #sub-element1 {
    color: blue;
}

.container .alert {
    color: green;
}

/* Lowest priority */
div h1 {
    color: yellow;
}
```

The Flow of CSS

There are many parts to CSS and ways to format a document such as background color, text color, font, font-family, size, margin, padding, and so much more. Elements can be changed using CSS for a specific class and id or an element. Keep in mind, it is important to pay close attention to the priority of each type of CSS being used and whether it may be a class, an id, or a tag.

JavaScript

Once again, this tutorial assumes a certain amount of proficiency in computer programming. The JavaScript section will thus focus mainly on the differences between JavaScript and other common languages. There are many quirks and best practices unique to JavaScript. They will be covered here.

The Basics

Objects

JavaScript is an **Object-Based** language, which means it supports the creation of objects, but does not use classes like an **Object-Oriented** language would.

Several objects are implicitly defined by the browser, such as `Document` and `Window`. These objects can be used to interact with the core elements of the web page. For a detailed list of their associated members, check here for the `Document` object or here for the `Window` object.

Object members can be accessed using the `.` operator much like in Java. For example, the following code snippet would display a popup window using the `alert` method of the `window` object.

```
window.alert("Hello, World!");
```

Variables

JavaScript assigns variables on the basis of **dynamic typing**, meaning the type of a variable is determined at runtime. What this means for the programmer is that data types are not specified in the code. Additionally, the type of a variable may change throughout the course of execution. For example:

```
var x = 10 // x is and integer
...
... // Some code
...
x = "Ten" // Now x is a string
```

Notice how the `int` keyword commonly used in other languages does not appear. Instead, the `var` keyword is used. This keyword just specifies that `x` is a local variable. Were the `var` keyword not used, `x` would be defined as a global variable by default. It is considered best practice to make variables local unless global scope is required for the script to function.

Another feature unique to JavaScript is the existence of the `undefined` type. This type is assigned by default to a variable that has not yet been assigned a value. For example:

```
typeof x; // #=> undefined
```

> *Note: `typeof` is an operator, which will be explained later on in the tutorial.*

Another distinction made when dealing with variables is whether they are **primitive** or **reference** types. Primitive types in JavaScript include booleans, numbers, strings, null, and undefined. Reference types include arrays and objects. Reference types behave as they do in most popular languages such as Java in regards to argument passing (pass by reference) and equality checking (check equality based on reference location when using the `==` operator).

Operators

This tutorial will not cover basic use of operators such as `+`, `-`, `*`, `/`, etc... Instead, it will focus on those operators which are unique to JavaScript or behave differently than expected. For a more in-depth explanation of the operators, check here.

Assignment Operators

> *JavaScript has all of the shorthand assignment operators common in most other languages. i.e.* `+=`*,* `-=`*,* `*=`*,* `/=`*, etc...*

Comparison Operators

> *JavaScript has all of the common comparison operators. i.e.* `==`*,* `!=`*,* `<`*,* `<=`*, etc...*

A comparison operator unique to JavaScript is the `===` operator. This operator functions similarly to the `==`, but while the `==` operator will attempt to use **type coercion** to test the equality of two variables, the `===` operator requires that the two variables are of the same type (`!==` works similarly). For example:

```
1 == "1"    // #=> true
1 === "1"   // #=> false
```

Arithmetic Operators

> *JavaScript has all of the common arithmetic operators. i.e.* `+`*,* `-`*,* `*`*,* `/`*, etc...*
> *JavaScript also includes the common unary operators. i.e.* `++` *and* `--`*.*

When performing arithmetic operations, the JavaScript interpreter will use type coercion where appropriate in order to complete the operation. This means that the type of certain variables may be automatically changed in some cases. For example:

```
var x = 1;
var y = "Hello";

x + y; // #=> "1Hello"
```

> *Note: In this example, the type of* `x` *is promoted to* `string` *in order to perform the operation.* `x` *will return to a number upon completion of the operation.*

Logical Operators

> *JavaScript has all of the basic logical operators as well as bitwise logical operators. i.e.* `&&`*,* `||`*,* `!`*, etc...*

Unique Operators

- `delete` - Removes a property from an object
- `in` - Determines whether or not a given object contains a given property
- `instanceof` - Deteremines whether or not a given object is an instance of another given object
- `new` - Creates a new object based on the object's constructor
- `typeof` - Returns the type of the given object in string format
- `void` - Ignores an expressions return value

Arrays

Arrays are defined in the same way as Java, however, one key difference is that they may hold multiple different data types. For example:

```
var arr = [1, 3.14, "Hello"];
// or
var arr = new Array(1, 3.14, "Hello");
```

Another distinction of JavaScript arrays is that they are dynamic structures. The length of the array can be adjusted manually by editing the `Array.length` property. Because arrays are reference types, they also have numerous methods which allow for convenient manipulation. For example:

```
var arr = new Array();
arr.push(1, 2, 3);
var x = arr.pop(); // #=> 3
```

The above code uses the `push` and `pop` methods to add and remove elements from the array dynamically. For a comprehensive list of all of the methods and properties associated with JavaScript arrays, look here.

Control Flow

The control flow implemented in JavaScript is almost identical to that of Java, so this section of the tutorial will only cover basic examples of each. For a more in-depth explanation of JavaScript control flow, check here.

If/Then

```
var x = 5
if (x < 5) {
    console.log("x is less than five!");
} else if (x == 5) {
    console.log("x equals five!");
} else {
    console.log("x is greater than five!");
}
```

Switch/Case

```
var x = 3
switch (x) {
    case 1:
        console.log("x equals one!");
        break;
    case 2:
        console.log("x equals two!");
        break;
    case 3:
        console.log("x equals three!");
        break;
    default:
        console.log("I don't know what x is!")
        break;
}
```

For Loops

```
for (var i = 0; i < 10; i++) {
    console.log("i is now " + i);
}
```

While Loops

```
var x = 10
while (x > 5) {
    console.log("x is now " + x);
    x--;
}
```

> *Note: Do...While loops are also valid in JavaScript*

Functions

Functions are one of the fundamental building blocks in JavaScript.

For example, to define a simple function called half:

```
function half(num) {
    return num / 2;
}
```

If an object is passed as a parameter and the object's properties are changed by the function, then the change is visible outside the function. For example:

```
function myFunc(theObject) {
  theObject.type = "Husky";
}

var dog = {type: "Golden Retriever"}, x, y;

x = dog.type;     // x is "Golden Retriever"
myFunc(dog);
y = dog.type;     // y is "Husky"
```

A function can be anonymous, for example:

```
var square = function(num) {return num * num};
var x = square(5) //x is 25
```

Passing a function as an argument to another function:

```
function map(fun,b) {
  var result = [], i;
  for (i = 0; i != b.length; i++)
    result[i] = fun(b[i]);
  return result;
}
```

For the following code:

```
map(function(x) {return x * x * x}, [0, 1, 2, 5, 10]);
```

“

> *it returns [0, 1, 8, 125, 1000].*

Scope

If a variable is defined inside a function, it cannot be accessed from anywhere outside the function because the variable is defined only in the scope of the function. Also, a function defined inside another function can access all variables defined in its parent function.

Closure

Closure features are powerful in JavaScript which allows nesting of functions.

```
var pet = function(type) {
  var getType = function() {
    return type;
  }
  return getType;
}
var myPet = pet("Husky");
```

The outer function defines a variable called type, and the inner function has access to the type variable of the outer function. The inner function is returned thereby is visible to outer scopes. Finally when this is called:

```
myPet();
```

> *it returns "Husky"*

More about Closures can be found here

Events

In JavaScript, tasks can be performed when certain events happen. There are 3 ways to register event handlers for a DOM element:

Inline (link in HTML itself)

```
<a href="catVideos.html" onclick="beHappy()"> Dog pics </a>
```

> *Note: This way should be avoided. This makes the markup bigger and less readable. Concerns of content/structure and behavior are not well-separated, making a bug harder to find.*

Direct (link in JavaScript)

```
var e = document.getElementById("dog");
e.onclick = woof;
```

> “
>
> *Note: The problem with this method is that only one handler can be set per element and per event.*

Chained (In JavaScript)

```
var e = document.getElementById("dog");
e.addEventListener("click", woof, false);
```

> “
>
> *Note: This is the method you should use in modern web pages.*

More details about addEventListener can be found here

Comments

JavaScript supports:

```
// Single line comments
```

and

```
/*  Multi-line
comments */
```

Ruby

The Ruby section of this tutorial will focus on introducing the fundamentals of Ruby as well as its roll in web application development.

Role in Web Application

When developing a web application, Ruby can be used for different purposes. It can be used for writing

SMTP server, FTP daemon, or Web server, or used for CGI programming for defining how information is exchanged between the web server and a custom script. Moreover, with Ruby on Rails, a web application framework that uses Ruby, programmers can build an entire web application with MVC design pattern.

Environment Setup

Ruby can be used in various ways across multiple platforms. Ruby has an interactive environment called iRb which is a shell extension that can be used for experimentation. iRb is a tool that is installed along with Ruby. If Ruby is already installed, iRb can be accessed by typing irb in the command prompt and the interactive Ruby environment will start. The interactive session may be used to test segments of code without creating an entire project.

If Ruby is not installed, there are many methods to install Ruby. It is not recommended to use a third party installer other than the programs found in the official documentation. The correct Ruby installation instructions can be found for Windows/Mac OSX/Linux/Unix here. There are many supported methods so the best option is to choose the method that is most comfortable to skill level.

Once Ruby is installed its time to begin writing programs. Ruby programs can be developed directly from a default text editor on any operating system. Some of the most popular Ruby editors are VIM, RubyWin, and Sublime Text Editor.Now that Ruby has been installed and an editor has been chosen lets dive deeper into Ruby.

Variables

Variables in Ruby are dynamically typed, which means the user does not need to declare variable types in the program and the type of a variable is known during the run-time. This is similar to variables in JavaScript, but the syntax of declaring a new variable is slightly different. In Ruby, the value of an uninitialized variable is nil, which is a null value.

Local Variables

A local variable is started with a lowercase letter or `_`, and it only exists in a specific scope. For example:

```
def methodA
    str = “a new string”
    puts str
end
methodA  // #=> Output “a new string”
puts str // #=> Error: undefined local variable or method ‘str’
```

Instance Variables vs. Class Variables

An instance variable is started with `@` and it exists only with an instance of a specific class. A class variable is started with `@@` and it is static and shared among all instances of a specific class. For example:

```
class Rectangle
    @@color = "blue"
    def initialize (w, h)
        @width = w
        @height = h
    end
    def area
        @width * @height
    end
    def print_color
        puts @@color
    end
    def change_color (new_color)
        @@color = new_color
    end
end
rec1 = Rectangle.new(3, 2)
rec2 = Rectangle.new(10, 5)
puts rec1.area #=> Output “6”
puts rec2.area #=> Output “50”
rec1.print_color #=> Output “blue”
rec1.change_color("red") #=> @@color = “red”
rec2.print_color #=> Output “red”, since @@color is shared by all instances of Rectangle
```

Constants

A variable started with Uppercase letter is consider as a constant in Ruby by convention. For example:

```
CONST = 1
CONST = 2 // #=> warning: already initialized constant CONST
```

Operators

Arithmetic Operators

Ruby includes the basic operators such as `+`, `-`, `*`, `/`, but this tutorial is not going to cover these simple operators. Instead this tutorial will discuss the operators that are more unique within Ruby. The two Arithmetic Operators that may not be as intuitive are the modulus and the exponent. The modulus `%` operator divides the left hand operand by the right and returns the remainder.

```
5 % 2 // #=> 1
```

The exponent `**` operator is self explanatory yet it is denoted differently than most languages.

```
5 ** 2 // #=> 25
```

Ruby does not support the `++` or `--` operators to increment or decrement variables. Assignment Operators Ruby supports the common assignment operators that other languages support. The assignment operators in Ruby are: `+=`, `-=`, `/=`, `*=`, `%=`, `**=`.

Parallel Assignment

Ruby supports parallel assignment of variables. This enables more than one variable to be initialized with a single line of code.

```
x, y, z = 5, 10, 20
// #=> x=5
// #=> y=10
// #=> z=20
```

Comparison Operators

Ruby supports the basic comparison operators such as `==`, `!=`, `<`, `>`, `<=`, `>=`, as well as more complex operators such as `.eql?`, `<=>`, and `===`.

The `.eql?` operators acts similarly to `==` but `.eql?` is not the same. For instance when you compare integer variables `.eql?` will return the same result as `==`, but comparing string variables may return different results. For example:

```
a = “hello”
b = “hello”
a == b   // #=> true
a.eql? b // #=> false
```

The `.eql?` operator is comparing the reference of a variable. Therefore a and b have the same data value but they have different reference values and therefore are not equivalent.

Another comparison operator that is unique to Ruby is the `<=>` operator. This operator is a combined comparison operator. `<=>` returns 0 if the first operand equals the second, 1 if the first operand is greater than the second and -1 if the first operand is less than the second.

```
1 <=> 1 // #=> 0
1 <=> 2 // #=> -1
2 <=> 1 // #=> 1
```

The last comparison operator to discuss is the `===` operator. `===` is used to test equality within a clause of a case statement. For example `===` is used to test if a value is within a desired range.

```
(1...5) === 3 // #=> true
(0...6) === 7 // #=> false
```

Logical Operators

Ruby supports all of the common logical operators such as `and`, `or`, `&&`, `||`, `!`, and `not`.

Data Types

Ruby has a variety of data types. Some data types are similar to other programing languages and do not need further explanation so this tutorial will cover those data types that need more explanation. The data types that Ruby uses are: Boolean, Number, String, Array and Hashes.

Arrays

Arrays in Ruby are defined much like other programing languages, however, arrays in Ruby may hold multiple data types. For example:

```
myarray = [5, 123, ”hello”, 54, ”world”]
// or
myarray = Array.[](4, 5, ”hello”, ”world)
// or
myarray = Array.new(4) // Creates empty array of size 4
```

A unique feature for arrays is that arrays will dynamically change in size as elements are added or removed. Since elements within arrays are referred to by index there are many methods to accessing elements. For example:

```
myarray = Array.[](4,5,”hello”,”world)
myarray[2]  // #=> “hello”
myarray[-3] // #=> 5
```

For more information on other methods used to access Ruby arrays visit here

Hashes

A hash is a collection of key-value pairs. Hashes are similar to Ruby Arrays except the hash performs indexing to keys unlike an array which uses an integer index. There are various ways to create a new hash. An empty has may be created or the hash may be initialized with default values. Examples for creating hashes can be seen below:

```
months = Hash.new  // Creates an empty hash
months = Hash[“Jan” => 1, “Feb” => 2]
// or
months = { "Jan" => 1, "Feb" => 2 }
```

Hashes in Ruby also contain various methods. For example to obtain all values in the hash the code would look like:

```
months = Hash[“Jan”=>1, “Feb”=>2]
result = months.values
```

For a comprehensive list of methods for hashes visit here

Control Flow

The control flow for Ruby is similar to the control flow of other programing languages. In this section of the tutorial examples will be shown for each type. For more detailed information visit here

While/Until Loops

```
i = 0
while i<10 do
    puts i
    i +=1
end

j=0
while j<3
    puts j
    j += 1
    break if i == 2
end
```

> *Note: The do after the condition of a while loop is optional and does not affect output.*

The `until` loops vary from loops of other programing languages. In Ruby the `until` loop performs like a while loop but executes until a true condition is met. For example:

```
i = 0
until i == 3
    puts i
    i += 1
end

// or

puts i += 1 until i == 3
```

> *Note: Even though the examples above appear different they perform the same task.*

For Loops

```
for i in 1..5 do
    puts i
end

// or

for j in 1..5 do puts j end
```

> “
>
> *Note: The do is optional in the first example but if the for loop is written on a single line it is necessary.*

If/Then

```
if 12 > 10 then
    print “12 is greater than 10”
end

// or

print “12 is greater than 10 if 12 > 10
```

If/Elsif

```
if animal == “cat”
    print “I own a cat”
elsif animal == “dog”
    print “I own a dog”
else
    print “I do not own a cat or dog”
end
```

Unless

The `unless` statement is an alternative to the `if/else` statement. Below is a comparison between equivalent `if/else` statement and `until` statement.

```
if i < 5
    print “Integer is small”
else
    print “Integer is large”
end

or

unless i > 5
    print “Integer is small
else
    print “Integer is large”
end
```

Case Statement

```
score = 95
result = case score
    when 0..60 then “fail”
    when 61..80 then “average”
    when 81..99 then “above average”
    when 100 then “perfect”
end
puts result
```

Ranges

Ruby supports ranges of various data types since ranges may be used in a variety of ways. Ruby ranges may be ranges as sequences, ranges as conditions, or ranges as intervals.

Ranges as Sequences

This is the most common and widely used form for a range. Ranges as sequences in Ruby can be created using `..` or `...` operators. The `..` operator creates an inclusive range while the `...` operator creates a range that excludes the largest value.

```
(1..5)     // #=> 1, 2, 3, 4, 5
(1...5)    // #=> 1, 2, 3, 4
(‘a’..’c’) // #=> ‘a’, ‘b’, ‘c’
```

Ranges as Conditions

Ranges in Ruby may be used as conditional expressions. For example a case statement may use a range as conditions to determine output.

```
score = 95
result = case score
    when 0..60 then “fail”
    when 61..80 then “average”
    when 81..99 then “above average”
    when 100 then “perfect”
end
puts result
```

Ranges as Intervals

This type of range is generally used for interval testing. Interval testing is done by checking to see if a given value is represented by a range.

```
if ((1..5) === 2)
    print “Included”
end

if ((‘a’..’c’) === ‘b’)
    print “Included”
end
```

Classes

Classes in Ruby are similar to other object based programing languages such as Javascript. Classes are blueprints for other objects that will be created. A class consists of data members specific to that class. For example if we wanted to define a class for vehicles we may want to know the vehicles horsepower, type of fuel, and fuel capacity. There may also need to be functions to calculate gas mileage based on driving habits. The class described may be defined as:

```
Class Vehicle{
    def horsepower=(value)
        @horsepower = value
    end
    def fuel_capacity=(value)
        @fuel_capacity = value
    end
    def type_fuel=(value)
        @type_fuel = value
    end
    def gas_mileage(.....)
    end
end
```

Ruby classes can be created with any desired number of variables or methods. A class may include public, protected or private methods. Public methods may be accessed by everyone. Protected methods can only be invoked by objects of the defining class and subclasses. Private methods can only be called in the context of the current object. The example below shows a class with all three types of methods.

```
class Access
    def method1
    end
    protected
        def method2
        end
    private
        def method3
        end
end
```

> *Note: `method1` is public, `method2` is protected, and `method3` is private.*

Methods

The way methods in Ruby work is similar to other languages, such as JavaScript. The programmer can declare a method with a method name, started with lowercase letter and bundle some statements within the method for later reuse. The syntax of declaring a method in Ruby is as follows:

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
    statements
end
```

A method can be declared without any parameter. For example,

```
def person_info
    puts “Tom is 5 years old.”
end
person_info // #=> Output: “Tom is 5 years old.”
```

A method can be declared with parameters. Note that the parentheses around the parameters are not necessary. For example:

```
def person_info (name, age)
    puts “#{name} is #{age} years old.”
end
person_info “Tom”, 5 // #=> Output “Tom is 5 years old.”
```

Moreover, the parameters in a method can be given default values. When the method is called without one or more parameters passing in, the method will use the default value for those parameters. For example:

```
def person_info (name = “Tom”, age = 5)
    puts “#{name} is #{age} years old.”
end
person_info            // #=> Output: “Tom is 5 years old.”
person_info “John”, 10 // #=> Output: “John is 10 years old.”
```

Modules

In Ruby modules are used to group together methods, classes and constants. Modules provide a namespace which helps prevent name clashes. Modules also implement the mixin facility (which is Ruby’s version of multiple inheritance). Below is an example of a Ruby module which includes methods.

```
module Week
    first_day = “Sunday”
    def Week.weeks_per_month
        prints “Four Weeks in a Month”
    end
    def Week.weeks_per_year
        prints “52 Weeks in a year”
    end
end
```

For more detailed information about modules or if multiple inheritance is not understood check out this page for more information.

Blocks

In Ruby, a block includes a set of statements and can be declared using the following syntax:

```
block_name {statements}
```

Blocks can be associated with iterators of data structure such as Array and Hash, or associate yield statement within methods.

Here is an example passing a block as a parameter to an iterator of an array that calculate the sum of all the integers in the array.

```
a = [1,2,3,5,7,9]
sum = 0
a.each {|i| sum+=i} #=> sum = 42
```

Here is another example applying a block to a method that contains a yield statement. In the method, whenever yield executed, it will pass the parameters to the associated block and start to execute the block. And after the block finish executing, it will return to the method and continue.

```
def foo
    for i in 1..5 do
        yield i
    end
end
foo {|i| print i} // #=> Output: “12345”
```

I/O

I/O in Console

In Ruby, a programmer can use puts method or print method to print a string in console. The difference between them is that, puts will automatically attach a newline character at the end of the string so that after printing the string, the console will start with a new line; whereas print will not start the console with a new line after printing the string. For example:

```
puts “string1”
print “string2”
puts “string3”

// Output:
//  string1
//  string2string3
```

A programmer may use putc method to only one character from the console. For example:

```
putc “Hello” // #=> Output: “H”
```

Also, a programmer can use gets to read a string from the console. For example:

```
print “Please enter your name: ” // #=> User input: Tom[Enter]
name = gets // #=> “Tom\n”
```

I/O for Files

To open a file, a programmer can use File.new method or File.open method and pass in a filename and open mode as the parameter. The difference between this two methods is that File.open can be associate with a block while File.new cannot. The open mode includes the following:

- `r` : Read-only mode
- `r+` : Read-write mode
- `w` : Write-only mode, create a new file or overwrites it if a file with the given filename exists
- `w+` : Read-write mode, create a new file or overwrites it if a file with the given filename exists
- `a` : Write-only mode, appends the file instead of overwrites it if a file with the given filename exists
- `a+` : Read and write mode, appends the file instead of overwrites it if a file with the given filename

exists

To close a file, a programmer can call the close method on that file.

The way to read or write a file is similar to I/O in console, using puts, print, putc, gets methods. But note that these method will only work will only work with corresponding open mode. The following is an example that uses File I/O.

```
read_file = File.new(“read.txt”, “r”)
write_file = File.new(“write.txt”, “w”)
string = read_file.gets
write_file.puts string
read_file.close
write_file.close
```

After running the code above, a file with filename “write.txt” is created with the same contents as "read.txt".

Regular Expressions

A Regexp holds a regular expression, used to match a pattern against strings. Regexps are created using `/.../` and `%r{...}` literals.

Matching

Regular expression can be matched in two ways, with `=~` or match method. The `=~` operator is used for matching regular expression pattern with string. It returns the first occurrence index if there is a match, returns `nil` otherwise. The match method is also use for matching regular expression pattern with strings, but it returns a MatchData object if there is a match. For example:

```
/hello/ =~ “hello world”     // #=> matches literal “hello” in string “hello world” and
return 0
/hello/.match(“hello world”) // #=> matches literal “hello” in string   “hello world” and
return #<MatchData "hello">
```

Modifiers:

Modifiers control the matching process and provide a strong influence on how the regular expression engine processes characters. Regular expression modifiers are placed after the pattern, like so: /foo/modifiers

Modifiers i can be used to match pattern in case insensitive. For example:

```
/mAtCH/i =~ “abc matches” // #=> 4
/mAtCH/ =~ “abc matches”  // #=> nil
```

Modifier `o` can be used to match pattern performing interpolation only once. For example:

```
for i in 1..5
    if /#{i}/o =~ i.tostr
        puts i
    end
end // #=> Output “1” since the interpolation only performed once.
```

Modifier `x` can be used to match pattern ignoring whitespace. For example:

```
/mat ch/x =~ “abc matches” // #=> 4
/mat ch/ =~ “abc matches”  // #=> nil
```

Modifier `m` can be used to match multiple lines and use newline characters as regular expression. For example:

```
/abc.ef/m =~ "abc\nef" // #=> 0
/abc.ef/ =~ "abc\nef"  // #=> nil
```

Patterns

Ruby supports a wide array of commonly used characters within regular expressions.For example: `[]` pattern is used to match any single character in brackets. `$` is used to match pattern at end of lind. `*` is used to match 0 or more occurrences of the pattern it follows. The following is an example applying the patterns mentioned above:

```
/[Ss]*tring$/ ~= “this is a test sString” // #=> it matches sString in the end
```

Characters

There are some pre-defined character syntax can be used in regular expression pattern. `\d`, equivalent to `[0-9]`, will match a digit in string. `\s`, equivalent to `[\t\r\n\f]`, will match a space character in string. `\w`, equivalent to `[A-Za-z0-9]`, will match a word character. The following is an example applying the characters syntax mentioned above:

```
/\d\s*\w*/ ~= “1    abc” // #=> it matches 1 as digit in the front following 4 spaces, and
3 characters abc in the end
```

For more information regarding regular expressions visit this site.