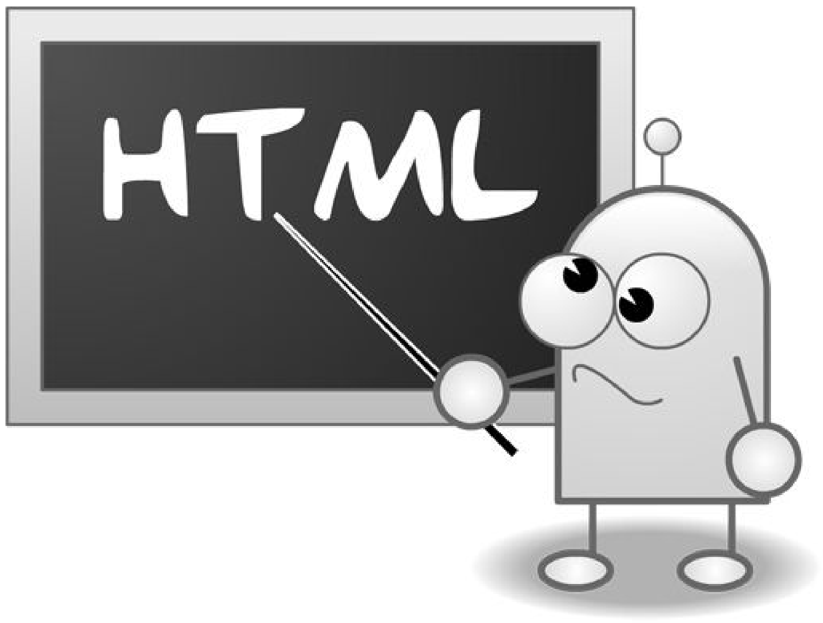
TECH TALENT SOUTH

LEARN TO CODE & DO <BIG> SOMETHING </BIG>

KIDS CODE TAKE HOME PACKET



HTML Lesson

Languages for Coding

- **HTML: HyperText Markup Language**
  - (What is on the web page)
- **CSS: Cascading Style Sheets**
  - (What the web page looks like)
- **JS: Java Script**
  - How the page interacts

Web Browser

- Browsers allow us to view the web pages

Folders and Files

- Create a folder to save your first web page in.
- Web folders and files should always be named with no spaces and ideally, with lower-case letters (example: tts_workshop)
- Open your code editor to create a new file
- File > Save.
- Save the web page into the folder with the same rules in mind (lower-case, no spaces.)
- If the page is the first page of your site, name it index.html

Rules to Coding in HTML:

1. HTML uses Tags to Annotate

2. HTML tags always have brackets <>
3. Opening Tags
    a. <>
4. Closing Tags
    a. </>
5. Web Pages have one head and one body:
    a. The head holds the information about web pages
    b. The body hold the words, pictures, and artwork

Announce to the page and browser what type of file we're using!

- The Doctype tells the browser what version of HTML is being used.

```
<!DOCTYPE html>
```

Type the following code:

```
<html>
<head>
        <title> This is my first web page </title>
</head>

<body>
Hello World! This is my first web page!
</body>
</html>
```

Let's check out what we've done!

- Save your file
- Open the file in your browser
- Awesome!

Let's add headers!

- **<h1> This is the biggest text on the page</h1>**
- **<h2>Then it gets a bit smaller</h2>**
- **<h3>Then even smaller</h3>**
- **<h4>So small, but still be featured </h4>**
- **<h5>Small </h5>**
- **<h6>Smallest</h6>**

1. Headers belong in the body

Time to add text!

- Use the <p></p> to include paragraphs

Now Let’s Add a List

- **Unordered Lists** look like this:
  - Apples
  - Oranges
  - Bananas
- **Ordered Lists** look like this:
  1. Apples
  2. Oranges
  3. Bananas

HTML for Unordered Lists

```
<ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
</ul>
```

HTML for Ordered Lists

```
<ol>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
</ol>
```

Let’s Create a Link!

- <a href=”https://WEBSITE LINK.com”> TEXT ABOUT LINK </a>

Opening a link a new Tab or Window

```
<a ref= “http://techtalentsouth.com” target=”_blank”><img
src=http://techtalentsouth.com/assets/img/logo.png/><a>
```

Footer Element

- Text that stays at the bottom of the page!
  <footer> Your name &copy, 2015</footer>

DIV ELEMENT

- The most common element you will see in websites is the <div> element. This is nested within the <body> and is usually a container for some section of a web page.

ALT ATTRIBUTE ON IMAGES

- <a href=“http://techtalentsouth.com” target=“_blank”><imgsrc="http://techtalentsouth.com/assets/img/logo.png" alt=”Tech Talent South" /></a>
    - Assists visually impaired

VIDEO ELEMENT

- <video src=http://pointclearmedia.com/mediasamples/waterfall.mov controls loop></video>

CSS Lesson

.CSS FILE AND LINK

- <link href=“images/style.css” rel=“stylesheet”>

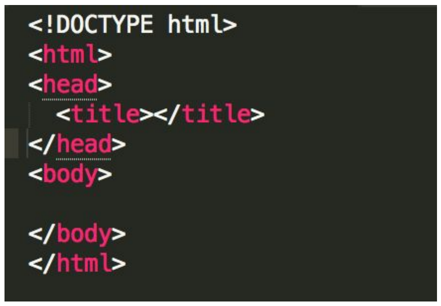
CSS SYNTAX

- <style>h1 {color:blue;font-size:12px;}</style>

Directions:

1. Create 1 new file called "style.css"
2. If you are using Sublime for your text editor, you can skip this step. For everyone else, go into your program's settings and make sure they're set to "plain text" and not "rich text" and that if you have an option for "smart quotes," be sure it is turned off for both HTML files and your CSS file.

3. Create the basic HTML structure in your index.html file:

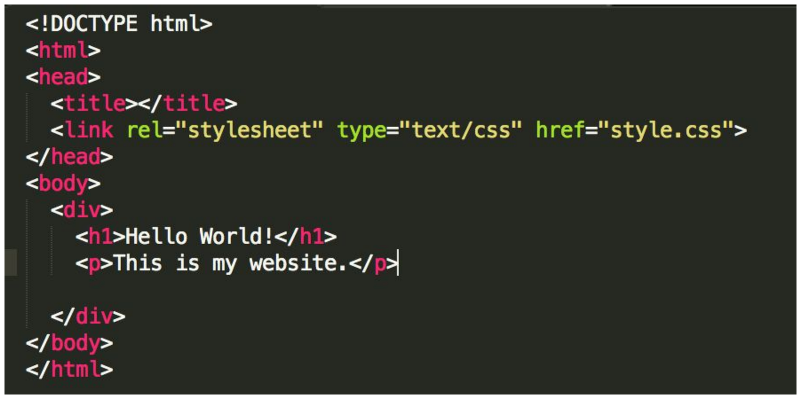


```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>

</body>
</html>
```

4. Make a link to your stylesheet by putting this code on the next line under your <title> tags: <link rel="stylesheet" type="text/css" href="style.css">

5. On the line below your opening <body> tag, make a <div> tag. Close it just before your </body> tag.

6. Add a set of <h1>s and <p>s between those tags and put content in them like this:



```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <div>
    <h1>Hello World!</h1>
    <p>This is my website.</p>

  </div>
</body>
</html>
```

7. Copy and paste that whole structure into your about.html file.

8. Update your <title> tags on index.html to say "Home" and on about.html to say "About."

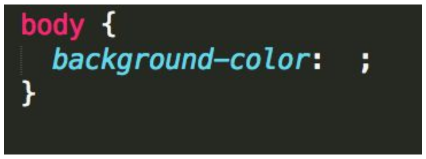
9. Change the "Hello World!" to "About" in your about.html document.

10. Save both files and open both pages in your web browser. You should be able to see those titles appear on the top of the browser tab. If not, something is wrong and you need to work with your team to troubleshoot. Remember, a second set of eyes can often find that missing semicolon or that little typo that you're not seeing.

    You've just set up your basic HTML files and now it is time to style them.

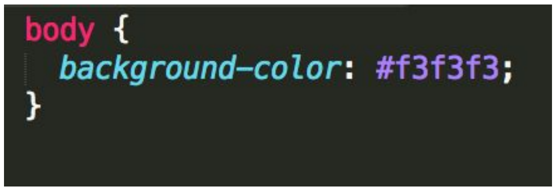
11. Remember what the <body> tag does? It's what houses all of the visible content on your page. Since we want to change the background color of our website, we'll do it by changing the background color of the <body> tag. This means anything that appears

within the <body> tags will be affected. In your style.css file, write this code:



```
body {
  background-color:  ;
}
```

12. The only thing you're missing now is the actual color. One common way of assigning colors is by using a hex code. A hex code looks like this: #3d93da. Go to http://www.color-hex.com/color-palettes/ and click on a color palette you like. Copy the hex code from one of the colors (keep this tab open, you'll be coming back to it). Paste the hex code into your code like this:

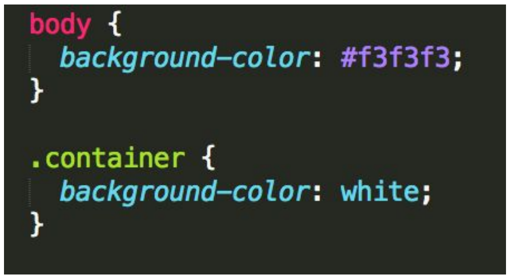


```
body {
  background-color: #f3f3f3;
}
```

(Use whatever color you'd like to. We're using #f3f3f3 in this example because it is a widely-used light gray. Two other important hex codes you should memorize are white: #ffffff and black: #000000 though the words "white" and "black" also work.)

13. Save your style.css file. Refresh both of your HTML pages in the browser. Did your background color change? If not, you need to troubleshoot. (Possible trouble spots: the link to your stylesheet might be incorrect. The css in your stylesheet might be incorrect.)

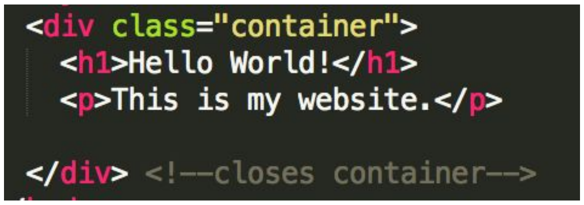
14. Since a colored background makes text hard to read, we want to put any of our actual text content (our <h1> and <p> tags) in a box with a white background. In your style.css, make a "class" called "container" like this:



```
body {
  background-color: #f3f3f3;
}

.container {
  background-color: white;
}
```

Notice the difference between the words body and container? See that dot before container? That's what makes it a class. Another difference is that you are applying styles *directly* to the existing HTML body tag while you are creating a new class of style called container that you will be able to *apply to* HTML tags.

15. Now that we have a class with a white background, we need to assign it to an HTML tag somewhere in our HTML documents. Good thing we planned ahead by adding a <div> back in step 8. Now we get to assign our new .container class to that <div> tag in both HTML documents like this:

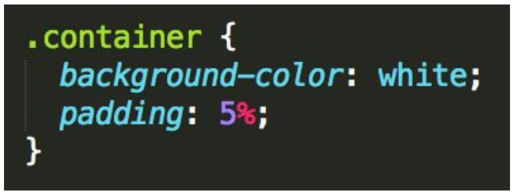

```
<div class="container">
  <h1>Hello World!</h1>
  <p>This is my website.</p>

</div> <!--closes container-->
```

Save and refresh. Did your white box show up? If not, you need to troubleshoot. (Again, remember how we didn't have to do anything to the HTML document when we added a background color to body?)

So what is a <div>? A <div> is a generic HTML box that you can use to make different sections of your website and apply different styles to those sections. It functions very similarly to your <h1> and <p> tags, but it doesn't have any pre-existing formatting for words in it.

If you are using Chrome as your web browser, you can install the Pesticide extension (here: https://goo.gl/VpRA6w) and it will allow you to turn on and off outlines around all of the boxes in your web page design. In fact, Pesticide will do this for *any* site on the internet that uses HTML and CSS.

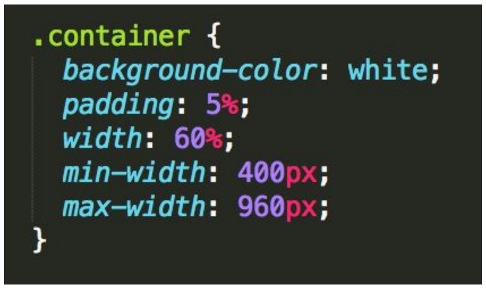
16. Now let's make our <div> with the .container class look better. In your style.css file, we'll add padding to the box. Padding gives space between the words in the box and the inside edge of the box. For example, if a baby pterodactyl fell out of its nest and you were preparing a shoebox for it to sleep in, you'd probably line the inside of that box with a little padding to keep your little prehistoric ward comfortable. In web design, we like to do the same for our word and image content. We add padding to the inside of the box to keep the words from touching the edge. So add 5% padding to your .container class like this:


```
.container {
  background-color: white;
  padding: 5%;
}
```

Save the .css file and refresh. Do you see the change? Have you ptsucessfully ptadded your pterodactyl? If not, you need to ptroubleshoot.

17. Let's be honest. Our .container looks a lot better, *but* most websites show a little more background and the content area is generally not as wide as the whole page. We'll achieve this by giving our .container div a width of 60%. And (for this example, but not necessarily in real web design) since we don't want our .container div to ever be smaller than 400px or wider than 960px, give it a minimum width of 400px and a maximum width

of 960px, like this:

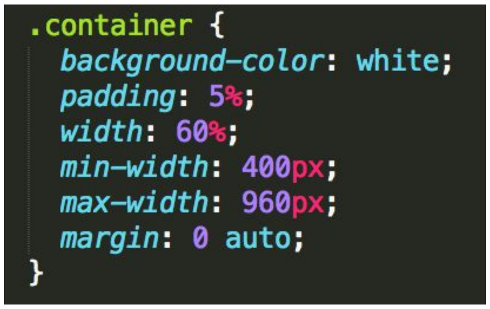


```css
.container {
  background-color: white;
  padding: 5%;
  width: 60%;
  min-width: 400px;
  max-width: 960px;
}
```

18. Save and refresh. When you resize your browser window, do you see the white box change shape but remain fixed at the 400px and 960px limits? If not, it's time to troubleshoot.

    This kind of flexibility is important for designing websites that look good on multiple screen types (like full monitors, laptops, tablets, and phones). Normally, instead of including min-width and max-width in our main css, we'd use @media queries (like these: https://css-tricks.com/snippets/css/media-queries-for-standard-devices/) in the bottom of our style.css document. We won't be getting into media queries in this lesson.

19. Now normally, we see the main content box of a website appear centered on the page, right? So let's do that by adding margin: 0 auto; to our .container class like this:



```css
.container {
  background-color: white;
  padding: 5%;
  width: 60%;
  min-width: 400px;
  max-width: 960px;
  margin: 0 auto;
}
```

    Save and refresh. See the change? If not, it's time to troubleshoot.

20. Let's do three more things to make our .container prettier. Do these one-at-a-time and save and refresh between each one. This will help you to catch any mistake you make.
    a. Add a border like this: border: 2px solid #000000;
    b. Give it rounded corners like this: border-radius: 15px;
    c. Apply a drop shadow like this: box-shadow: 5px 8px 15px #a9a9a9;

21. Play with the border property. Do you want it to be thicker? Thinner? Adjust your pixel size accordingly. Do you want it to be dotted or dashed? Change the word "solid" to "dotted" and see what happens (check out this page for more options: https://developer.mozilla.org/en-US/docs/Web/CSS/border-style) Change the color to one of the colors from the palette you chose back in step 15 from this site: http://www.color-hex.com/color-palettes/ Remember to save and refresh with each

change you make so you can see if it worked. (If you installed the Pesticide extension, now would be an excellent time to turn it *off*.)

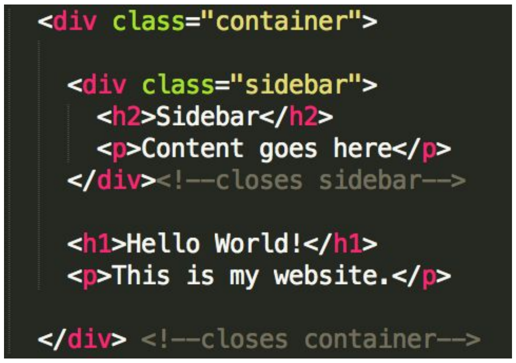
22. Play with the rounded corners. Right now all 4 corners have a radius of 15px. You can actually give each corner its own radius like this: border-radius: 0px 5px 20px 50px; The assignments go clockwise: top left, top right, bottom right, bottom left. Can you shape it like a tombstone? Can you shape it like a tab with a flat top and rounded corners on the bottom?

23. Play with drop shadow. Right now you have a horizontal shift (left-to-right) of 5px to the right, a vertical shift (top-to-bottom) of 8px downward, a shadow depth of 15px and a shadow color of #a9a9a9. Can you change the direction of the shadow? Can you change how big or small it is?

24. Now let's take a look at the power of the <div> by introducing a sidebar into your design. We want to make a sidebar that sits inside your .container, over on the right side. So add this new code to your index.html page just inside your .container <div> and above your <h1>:

```
<div class="sidebar">
        <h2>Sidebar</h2>
        <p>Content goes here</p>
</div><!--closes sidebar-->
```

25. Your code should look like this:

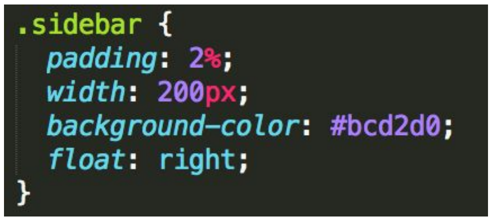


```
<div class="container">

  <div class="sidebar">
    <h2>Sidebar</h2>
    <p>Content goes here</p>
  </div><!--closes sidebar-->

  <h1>Hello World!</h1>
  <p>This is my website.</p>

</div> <!--closes container-->
```

Save and refresh. Do you see your new content on top of your previous content? Is it still on the left side of the page? If not, troubleshoot.

BTW: See the code that says <!--closes sidebar-->? That's an HTML comment. Anything you put between the "<!--" and the "-->" will *not* appear on your page. Even though it *says* "closes sidebar", it's not actually *closing* anything. In fact, your comment could say <!--help i'm being attacked by a pack of rabid donkeys--> and be just as effective. The closing </div> is the code that does the closing. Following your closing </div> tags with a comment explaining which <div> you've closed comes in very handy for troubleshooting when your document has a lot of <div> tags in it -- or for when you revisit your code after a year's time has passed. Notice how we've got comments after both of our closing <div> tags? Ahh, isn't that lovely?

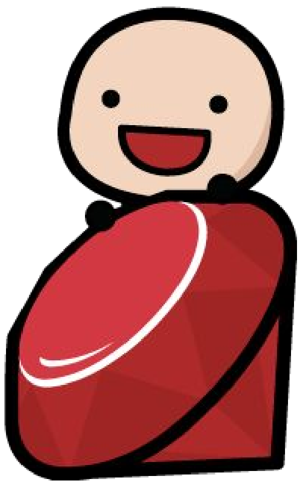
26. Now we get to make a new class called .sidebar in our css file. Build in some padding for this new class, a width of 200px, a background color, and lastly, tell it to float:right; like this:



```
.sidebar {
  padding: 2%;
  width: 200px;
  background-color: #bcd2d0;
  float: right;
}
```

Save and refresh. Has your sidebar moved over to the right? If not, troubleshoot.

27. **Reflect:** Since your styles live in one place (your style.css file) and you *link* to the content of that style file from both your index.html and about.html files, you only need to update your styles in one file instead of every HTML file on your website. Imagine if you had 10 more HTML pages... products.html, contact.html, internetcats.html... or 100 more... Updating the background color on every page individually would be a ridiculous waste of time. Go ahead and change the background color in style.css and see if both of your pages change. Change something about your `.container` class. It will update on every page and every `<div>` you have assigned to that class. Make more classes (remember how to make a class? See step 17.). Assign them to your `<h1>` tags and see what happens. Modify the style of your `<p>` tags like we did with `<body>`. Experiment!



RUBY!

What is Programming?

Programming is language. We use language to communicate with our friends. We use programming to communicate with our computers.

A program is nothing more than a set of instructions for the computer to execute. These instructions outline the plan or project, like a blueprint for a house. Without a blueprint, the builders wouldn't know what the architect wants, just like the computer doesn't know how to create what you want without a program.

Learning a programming language is a lot like learning any language. First you start with the basic building blocks--the A, B, C's. Then you start learning how to put those blocks into words, sentences, and eventually commands that will tell your computer what to do.

Why do we need to speak a strange language to communicate with computers?

Computers are a bit like dogs. They are great companions, but you need to give them commands to tell them what you want them to do. There are a lot of different programming languages, just like there are a lot of foreign languages. However, just as there are similarities in German, French or Spanish, there are similarities across programming languages, like JavaScript, Ruby or Python.

Integers, Floats and Strings

Before we can understand how to write programs, we need to understand data. Data is simply information that you can input (give), output (get), store or manipulate with a computer. The two fundamental types of data used in almost every programming language are numbers and strings.

Numbers come in two tasty flavors. Integers, which are whole numbers without a decimal point, and floats, numbers that contain a decimal.

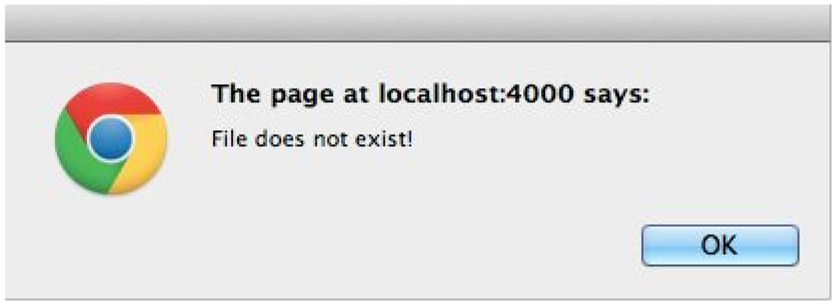
For example, these are integers:
1, 7, 0, 13, 2000

And these are floats:
1.2, 3.14, 5.12345, 0.35

Every time you see a whole number like 8 or 19 you will know they are integers, and every time you see a number with a tasty sprinkle, or decimal, they will look like this: 3.4 or 8.1 and you will know they are (root beer) floats.

Simple, right? Ok, let's keep going.

The second type of data, or information given to a computer, are called strings. What's a string? "Anything between quotes is a string." You probably see strings all the time without even realizing it! For example, have you ever seen an alert message on your computer saying something like this:



The page at localhost:4000 says:

File does not exist!

Somewhere inside a program or web application, an engineer wrote the sentence *"File does not exist!"* and put it in quotes to create a string. When you were alerted with the pop up box, that string was printed to the screen.

Strings are a little bit like backpacks or lunch pails, they are great for storing all the stuff we care about in an easy-to-carry container. Except with a string, the straps are the quotes. We use strings to store words, sentences, and even files.

Here are some examples of strings:

"I'm a string!"
"And_so_am_I"

"9"
"This long paragraph is even a string.\nAnd it has these strange \n things that we'll explain later."

9 is such a joker. **Did you notice we put the *number nine* as a string? This is very different than the actual number nine, but we will get to that later.**

NUMBERS

Do you know basic math? Great, so does Ruby! **Ruby uses the same adding, subtracting, multiplication and division that you do.**

```
2 + 2 => 4
9 - 3 => 6
2 * 3 => 6
4 / 2 => 2
```

Ruby can also perform logical operations, such as greater than > and less than <.

```
4 > 2 => true
7 < 2 => false
3 >= 3 => true
0 <= 1 => true
```

So, what is IRB anyway?

IRB stands for Interactive Ruby Shell. An Interactive Ruby Shell is like a little secret fort located inside your computer. **You can use your IRB environment to play around with Ruby and learn different commands.** Open up IRB and try typing some of these simple math calculations to see what the computer returns. Or try your own!

- Go to Cloud9 or whichever cloud-based environment you are using.
- Open up a new Ruby project
- Title it *numbers.rb*
- SAVE!
- Go to terminal and type *irb*
- Try playing with numbers on IRB!

Curious about the => arrows? Ruby engineers like to call these hash rockets. You will see that typing 3 + 2 or any other thing into IRB will always return a value, signified by the pointer =>.

A bit more math

Programming languages like Ruby can perform a lot of mathematical equations and expressions. Now that we can see how Ruby performs addition, subtraction, multiplication and division, we'll see how she handles exponents and the oh-so-cool modulo!

Exponents

Exponents tell a number how many times it should be multiplied. For example, 2 times 2 equals 4, 4 times 2 equals 8. So, 2^3 means 2 times 2 times 2, or 8. Don't worry about exponents if they are unfamiliar to you. They are not necessary to learn programming! Ruby uses two stars to signify an exponent math expression:

```
2 ** 3 => 8
3 ** 2 => 9
10 ** 3 => 1000  #10 times 10 times 10!
# Also, the pound symbol is used for comments
# (and ignored by Ruby)
```

Modulo

In addition to these standard math operations, the computer has something called the modulo operator, which is represented using a percent symbol: %. The modulo's job is to find the remainder after dividing one number with another. The remainder is what is left over, or what remains when you divide one big number by a smaller number. Let's look at an example.
9 divided by 3 would result in a modulo of 0, because there is no remainder. Since 3 divides evenly into 9 (three times) there are no numbers left over, and that is why 9 modulo 3 is 0. If we try it again with a different set of numbers, say 9 modulo 2, we would have a remainder (or a modulo operator) of 1. Because 9 divided by 2 equals 8, leaving a remainder of 1.
Try these examples

```
8 % 2 => 0
9 % 2 => 1
9 % 5 => 4
```

If exponents and modulos are too complex to understand right now, don't worry. Again, they are not a requirement for programming. It's important at this stage to simply understand that the computer can do simple math for you.

PRACTICE

Try these problems in your head, or on paper. Then see how they work in the Ruby shell (IRB).

1) 2 + 3 + 5
2) 10 - 3
3) 9 / 3
4) 4 * 2
5) 4 ** 2

Now for some harder ones.

6) What is the result of 11 % 5 ?
7) What is 14 % 3 ?

8) A wizard carries two numbers (one even and one odd) in each hand. He won't open his hands to show you, but he will let you use modulo. Your task is to find out which hand holds an even number, and which holds the odd.
Hint: A number divided by two, with no remainder, is even.

STRINGS

Congratulations! You are halfway through learning the two most fundamental 'blocks' of programming code--Numbers and Strings.

Remember that a string is simply a piece of data (usually words) wrapped in quotes.

In Ruby, we can perform some math-like operations using our friends, the strings. For example, we can multiply a string like so:

"repeat " * 3
=> "repeat repeat repeat "
In the above example we have a string: "repeat " that we multiply * by three. When Ruby performs the * 3 method it actually just copies the string three times and pastes the result together. Sort of like this:

"repeat " + "repeat " + "repeat "
=> "repeat repeat repeat "
In fact, you can write either of these lines of code and the result will be the same. Much in the same way that writing 3 + 3 + 3 = 9 yields the same result as 3 * 3 = 9. When the computer adds strings together we call it concatenation.

"Cat and " + "Dog"
=> "Cat and Dog"

In the last example, there are two strings, one is "Cat and " and the second string is "Dog". By giving the command to add the two strings +, we are able to join both strings together using the magic of concatenation. (Concatenation is just a fancy old Latin word for join together).

Ready to try it out?

Open Terminal/IRB and try multiplying and adding a few strings yourself to get the hang of it.

Now try adding a number with a string. It didn’t work did it? Remember that joker "9" string from our previous example in chapter 1? Ruby doesn’t see this as the actual number 9, instead it sees a string.

To Ruby, anything that is inside the quote isn’t just a word or a number anymore. Everything inside the quotes is a string. So when we try to add a string with a number, Ruby gives us an error:

"9" + 9
=> TypeError: can't convert Fixnum into String

In the above example, we would see "9" + 9 and think the answer is 18. But Ruby doesn't see it like that. Ruby sees "STRING" + NUMBER. And you can't add strings and numbers, because they are different types of data.
This doesn't mean we can't do this sort of equation, it just means we need to use a trick to make sure Ruby understands what we want. Of course, there are lots of interesting methods or actions we can perform to get Ruby to do what we want.

We'll look at some of those actions a bit later, but for now, **you could solve the above problem by using the to integer method, or to_i. This would convert the string to a number (more specifically, an integer). And with two numbers, Ruby can do the math:**
"9".to_i + 9
=> 18

Remember those \n characters from the first chapter? This is what the computer uses to note a new line in your string. So the following string:

print "One line.\nAnother line.\nAnd another.\n"
Would print like this:
One line.
Another line.
And another.

In this way, Ruby can store sentences or even whole files inside just one string. Now that you have a better understanding of strings…

PRACTICE!

What is the result of each of the following?
1) puts "What is the result" + " of " + "this operation?"
2) "This string minus" - "That string"
3) "1234.55".to_i
4) "1234.55".to_f
5) "Not a number".to_i
6) puts "1\n2\n3\n"
7) Why might it be useful that the to_i (to integer) method return zero for strings that can't be represented as numbers?
8) What do you think the length method does?
"Count".length
9) How about the split method?
"Count".split("")
10) What do you think slice does?

"Count".slice(2)

VARIABLES

At their simplest, variables are merely storage containers. They help us hold information. You create your own variables, starting with a lowercase letter followed by an equal sign and its value (usually a number or a string, but sometimes more complex code can be stored in a variable). You've seen this before in math.

x = 12
The value of x is?
=> 12

In Ruby, a variable name is defined once you write (almost) anything left of the equal sign. The equal sign (as in most programming languages) is used to *assign* the value to the right, to the variable. Here are a few more examples of creating variables.

myVar **=** "my string variable"
a_long_var_name **=** 42
myCat **=** "whiskers"

y = 5 => y points to memory address AB1, which contains the value 5

x = y => x points to memory address AB1

y = 7 => y now points to memory address CD1, which contains 7

x equals ?

The trick here is to understand that X does not equal Y. Remember, X only POINTS to the value stored in the memory address that Y points to (AB1), at the time it was assigned. No matter what happens later to Y, the only thing X needs to remember is the location of AB1 (which will always be 5 while this program is running).

When we change the value of Y to 7, we are actually telling the computer to create a new memory address that contains the value 7 and point to it. X does not change its original address, so the value of X remains 5.

Practice

1) Store the value of 54 / 3 into the variable x. What is the value of x?

2) Give the value of x (from problem 1) to y, a new variable. Now make x equal to itself divided by 3.
What's the value of x?
What's the value of y?

3) What if we performed some math on our variables? If we set x to 12, what's x divided by 3?

4) What's the value of x now?

MAD LIB!
Create a new Ruby on Rails workspace
Name it *madlib.rb*
SAVE!

---

madlib.rb

```
puts "Want to play a game of Mad Lib?"
puts "Great! Lets get started. When I ask for something, you give me that type of word!"

print "Family Member: "
a = gets.chomp.strip.downcase
print "Color: "
b = gets.chomp.strip.downcase
print "Noun: "
c = gets.chomp.strip.downcase
print "Something That Flies: "
d = gets.chomp.strip.downcase
print "Past Tense Verb: "
e = gets.chomp.strip.downcase
print "Plural Noun: "
f = gets.chomp.strip.downcase
print "Plural Noun: "
g = gets.chomp.strip.downcase
print "Color: "
h = gets.chomp.strip.downcase
print "Body Part Plural: "
i = gets.chomp.strip.downcase
print "Tool: "
j = gets.chomp.strip.downcase
print "Body Part: "
k = gets.chomp.strip.downcase
print "Imaginary Creature Plural: "
l = gets.chomp.strip.downcase

puts ""
puts "You're all done! Here is your Mad Lib!"
puts ""
```

puts "One night my " + a + " and I were driving when we saw a bright " + b + " " + c + " in the sky. At first I thought it was a " + d + " but I knew it couldn't be when a beam of light " + e + " out of it and that is the last thing we remember. Several hours later we awoke in our car on the side of the road and we had " + f + " on our arms and " + g + " on our stomachs. Under hypnosis I remembered " + h + " colored creatures with big " + i + " and they stuck a " + j + " in my " + k + ". Although I used to be a skeptic I am now a believer in " + l + "!"

IF/ELSE

"If you clean your room then you can play, or else you won't." - Mom

***If* and *else* statements help Ruby understand what you want her to do and when you want her to do it.**

So far, we've learned how Ruby can perform basic math on numbers, store words and sentences as strings, and place these types of information into memory stored in variables. Now that we know how to store information and interact with it, we need a way of telling the computer what to *do* with that information.

One way we do that is with conditionals. This is why your parents say they love you unconditionally, because there is no *if* or *else*, there is only 100% love, no matter what. Ruby doesn't love anything unconditionally, Ruby needs to be convinced by conditions.

A conditional is something that depends on other factors. If this *something* happens, do *that*, otherwise, do something *else*. For example, if you are hungry, eat a sandwich, else, don't eat a sandwich!

Here's how an If Else conditional might look in Ruby.

```
# set an x variable to the value 5
x = 5

# start the if / else conditional
if x <= 10
  puts x
else
  puts "Number is greater than 10"
end
```

In a conditional, the computer checks to see that the code after the *if* is true. We call this code a block, which just means a small piece of code that has some function. In this case, our block is X <= 10.

So, *if* our block is *true* (if X is less than or equal to 10), the computer will perform the action in the next line below the if statement. Since X is equal to 5, it *is* less than 10, and the value 5 is *put* on the screen. If X were 11, the conditional (X <= 10) would realize this was a false statement, and the string "Number is greater than 10" would be outputted (shown) on the screen.

When the computer moves through an if-else conditional, it will follow the instructions under the *first* if statement that evaluates to true. It then ignores all other conditions in the if-else conditional.

What is a boolean?

We've just learned how a conditional checks to see if something is true or false. A true or false value is called a *boolean* in programming. We could write another conditional in a different way using booleans.

```
if false
  puts "false"
elsif true
  puts "true"
else
  puts "This won't print"
end
```

Here are the basic elements used to evaluate conditionals:

```
<   # less than
>   # greater than
<=  # less than or equal to
>=  # greater than or equal to
==  # equal to
!=  # not equal to
```

When checking if an object is *less than* or *greater than*, we use the same symbols found in math. When checking if some object is equal to another object, we use two equal signs. In Ruby, like many programming languages, one equal sign is used to assign or *give* a value to a variable. If we want to check that an object is *not equal* we use an exclamation mark before the equal sign. In Ruby, we could also simply use the word not.

Practice

What would the computer say? Try to do these in your head before going to IRB. Remember, the answers will be a boolean value, either true or false.

1) Is 3 > 5 ?

2) 3 < 5

3) 5 == 5

4) 10 >= 10

5) 10 <= 12

6) 10 != 10

7) 10.object_id == 10.object_id

What about strings?
8) "dog" == "cat"

9) "cat" == "cat"

10) "cat.object_id" == "cat.object_id"

Rock, Paper, Scissors

```
rps = ["rock", "paper", "scissors"]
# puts " "
puts "Let's play rock, paper, scissors! First to 5 wins."
puts " "
user_score = 0
comp_score = 0

until user_score >= 5 || comp_score >= 5
puts "What is your throw?"
user_throw = gets.chomp.downcase
comp_throw = rps.sample.downcase
puts "You threw #{user_throw} and I threw #{comp_throw}!"

if user_throw == comp_throw
        puts "It's a Tie!"
        puts "Your score: #{user_score}"
        puts "My score: #{comp_score}"
        puts " "
elsif user_throw == "rock" && comp_throw == "paper"
        puts "Paper beats rock. I win!"
        comp_score += 1
        puts "Your score: #{user_score}"
        puts "My score: #{comp_score}"
        puts " "

elsif user_throw == "rock" && comp_throw == "scissors"
        puts "Rock beats scissors. You win..."
        user_score += 1
        puts "Your score: #{user_score}"
        puts "My score: #{comp_score}"
        puts " "

elsif user_throw == "paper" && comp_throw == "rock"
```

```
            puts "Paper beats rock. You win..."
            user_score += 1
            puts "Your score: #{user_score}"
            puts "My score: #{comp_score}"
            puts " "

elsif user_throw == "paper" && comp_throw == "scissors"
            puts "Scissors beats paper. I win!"
            comp_score += 1
            puts "Your score: #{user_score}"
            puts "My score: #{comp_score}"
            puts " "

elsif user_throw == "scissors" && comp_throw == "rock"
            puts "Rock beats scissors. I win!"
            comp_score += 1
            puts "Your score: #{user_score}"
            puts "My score: #{comp_score}"
            puts " "

elsif user_throw == "scissors" && comp_throw == "paper"
            puts "Scissors beats paper. You win..."
            user_score += 1
            puts "Your score: #{user_score}"
            puts "My score: #{comp_score}"
            puts " "

else
            puts "You didn't make a valid throw! Try again!"
end
end

if comp_score == 5
            puts "Yes! I win the game! I am the rock, paper, scissors king!"
else
            puts "Congratulations. You win. I have to admit you are pretty good at this game."
end
```

Extra Kids Coding Resources

1. www.madewithcode.com
2. scratch.mit.edu
3. code.org/learn
4. www.youthdigital.com
5.