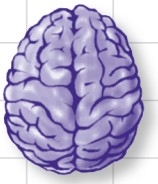
A Brain-Friendly Guide

Head First JavaScript

Improve your user experience with web page interactivity



Load important JavaScript concepts directly into your brain

Whip up working JavaScript code in a snap

Slice and dice HTML with help from the DOM

Stop fearing event handling—it won't hurt a bit

Bend your mind around dozens of puzzles and exercises

O'REILLY®

Michael Morrison

Head First JavaScript

by Michael Morrison

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Series Editor: Brett D. McLaughlin

Design Editor: Louise Barr

Cover Designers: Louise Barr, Steve Fehler

Production Editor: Sanders Kleinfeld

Proofreader: Colleen Gorman

Indexer: Julie Hawks

Page Viewers: Masheed Morrison (wife), family, and pet fish

Printing History:

December 2007: First Edition.

My family knows how to celebrate a book release...

...but my koi fish couldn't care less.

No rocks, stick figures, cube puzzles, or macho moviegoers were harmed in the making of this book. Just me, but I can handle it...I'm wiry.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

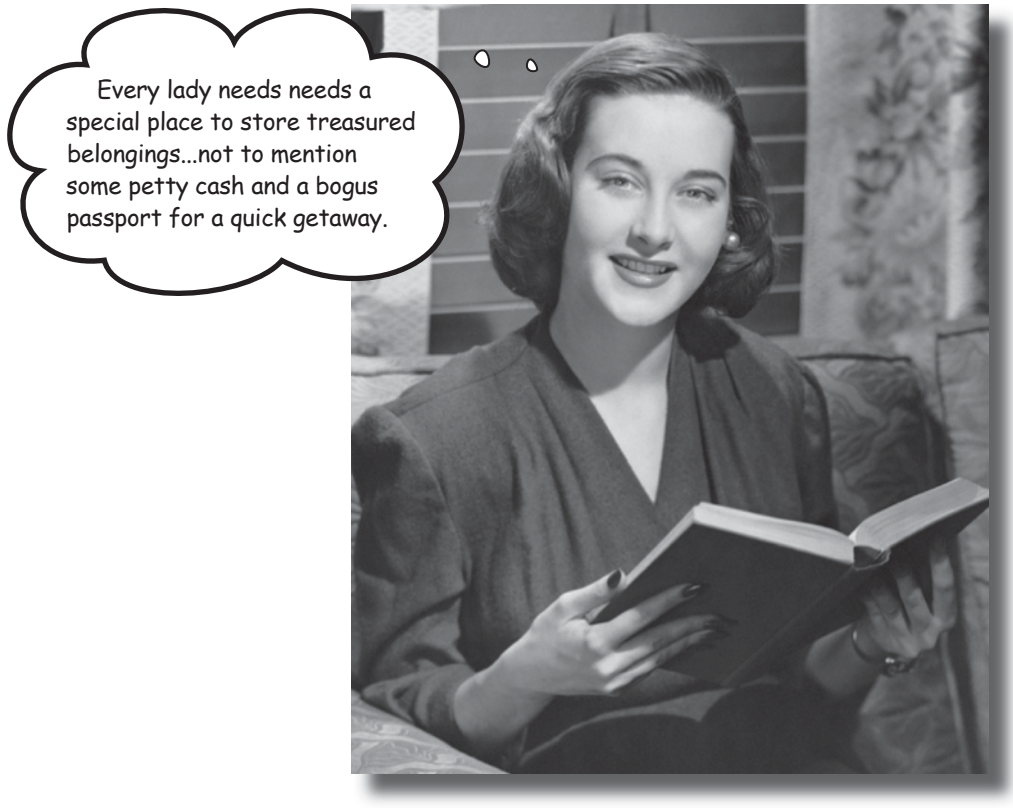
ISBN-10: 0-596-52774-8

ISBN-13: 978-0-596-52774-7

[M]

2 storing data

Everything Has Its Place



In the real world, people often overlook the importance of having a place to store all their stuff. Not so in JavaScript. You simply don't have the luxury of walk-in closets and three-car garages. In JavaScript, **everything has its place**, and it's your job to make sure of it. The issue is **data**—how to ***represent it***, how to ***store it***, and how to ***find it*** once you've put it somewhere. As a JavaScript storage specialist, you'll be able to take a cluttered room of JavaScript data and impose your will on it with a flurry of virtual labels and storage bins.

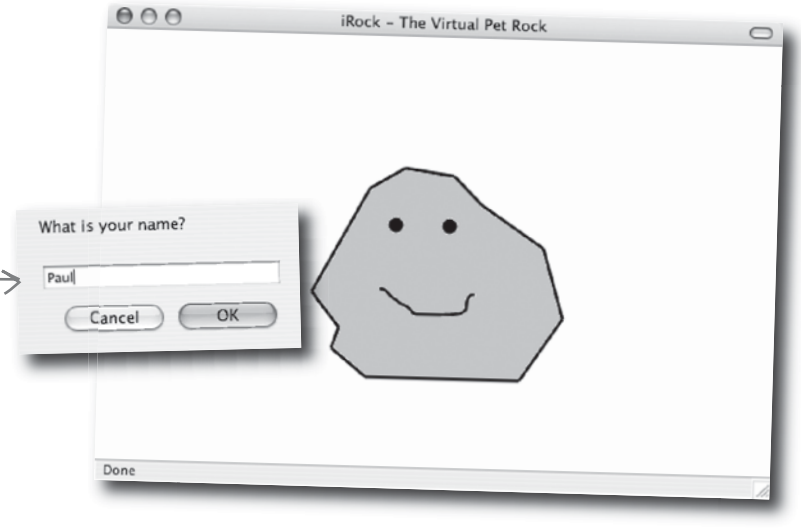
Your scripts can store data

Just about every script has to deal with data in one way or another, and that usually means storing data in memory. The JavaScript interpreter that lives in web browsers is responsible for setting aside little areas of storage for JavaScript data. It's your job, however, to spell out exactly **what the data is** and **how you intend to use it**.

The information associated with a house search must all be stored within the script that performs the calculations.

Scripts use stored data to carry out calculations and **remember** information about the user. Without the ability to store data, you'd never find that new house or really get to know your iRock.

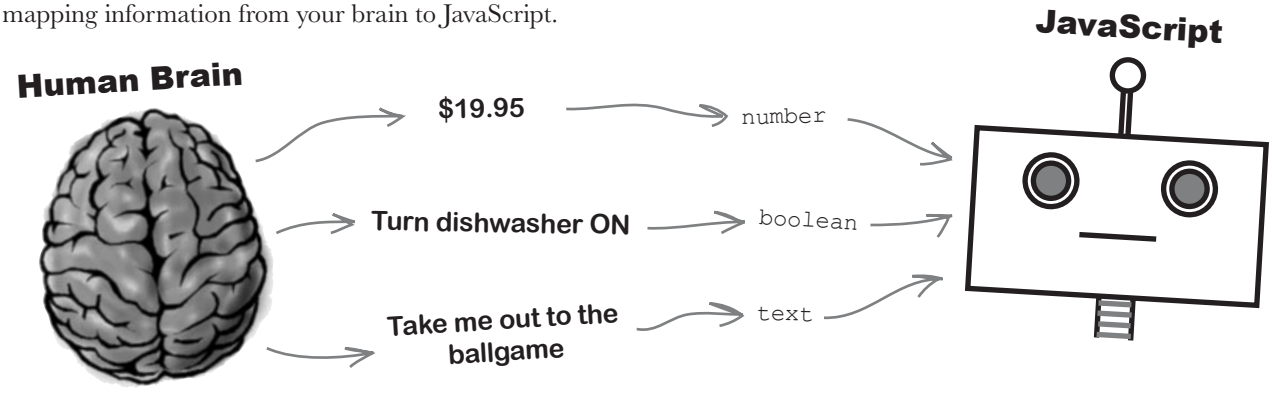
The user's name entered into the iRock page is stored away so that the script can show you a personalized greeting.



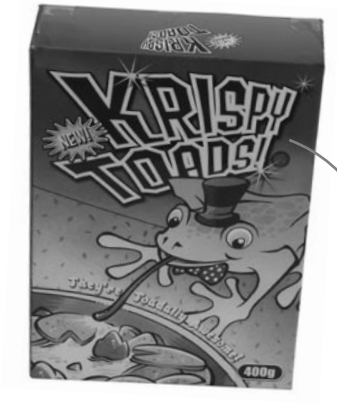
Think of the different real world pieces of information you deal with on a daily basis. How are they alike? Different? How would you organize those different pieces of data?

Scripts think in data types

You organize and categorize real world data into types without even thinking about it: names, numbers, sounds, and so on. JavaScript also categorizes script data into **data types**. Data types are the key to mapping information from your brain to JavaScript.



JavaScript uses three basic data types: text, number, and boolean.



Text

Text data is really just a sequence of characters, like the name of your favorite breakfast cereal. Text is usually words or sentences, but it doesn't have to be. Also known as **strings**, JavaScript text always appears within quotes (`""`) or apostrophes (`''`).

Number

Numbers are used to store numeric data like the weights and quantities of things. JavaScript numbers can be either integer/whole numbers (2 pounds) or decimals (2.5 pounds).

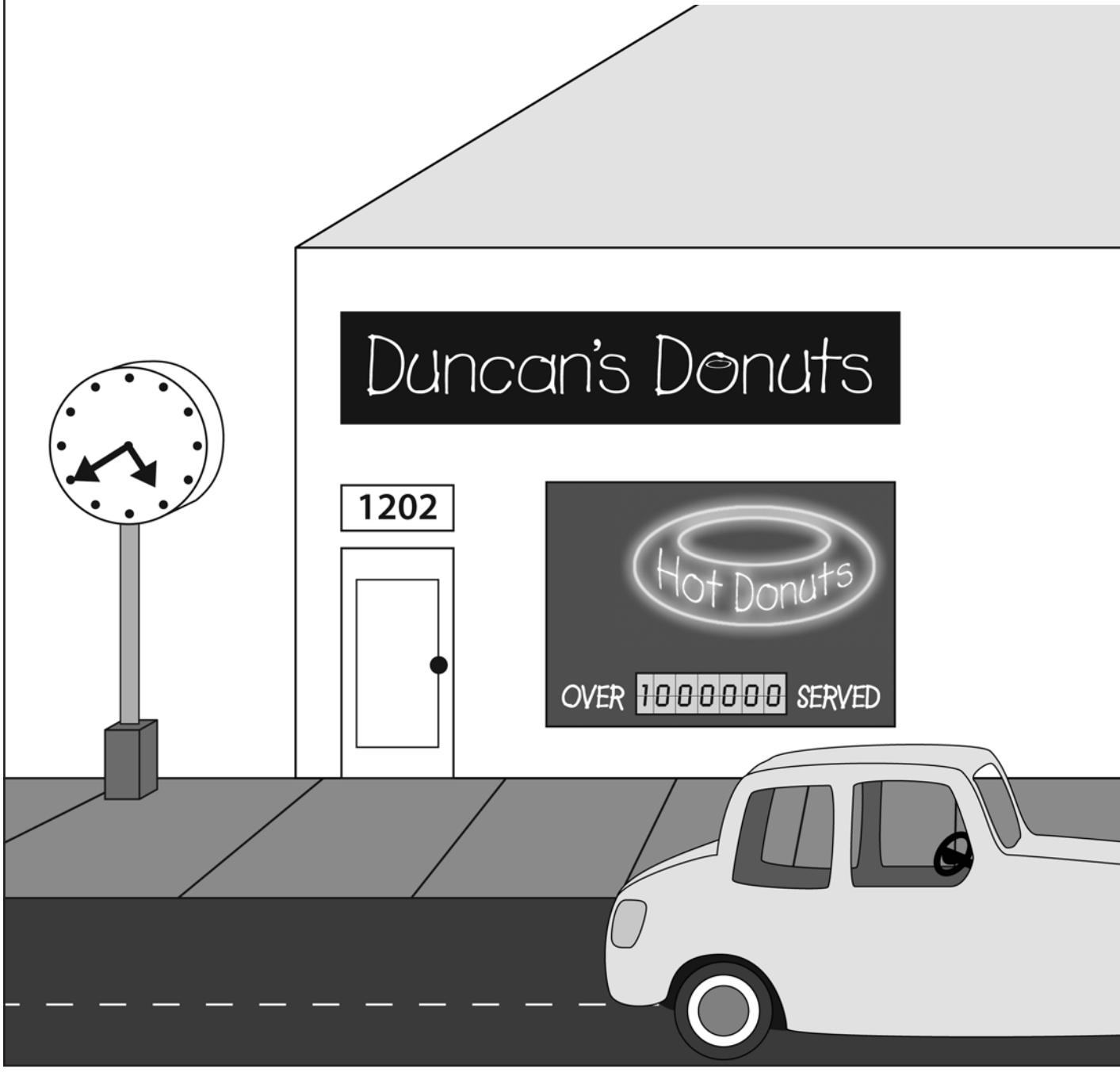
Boolean

Boolean data is always in one of two possible states—**true** or **false**. So you can use a boolean to represent anything that has two possible settings, like a toaster with an On/Off switch. Booleans show up all the time and you can use them to help in making decisions. We'll talk more about that in Chapter 4.

Data types **directly affect** how you work with data in JavaScript code. For example, alert boxes only display text, not numbers. So numbers are converted to text behind the scenes before they're displayed.

Sharpen your pencil

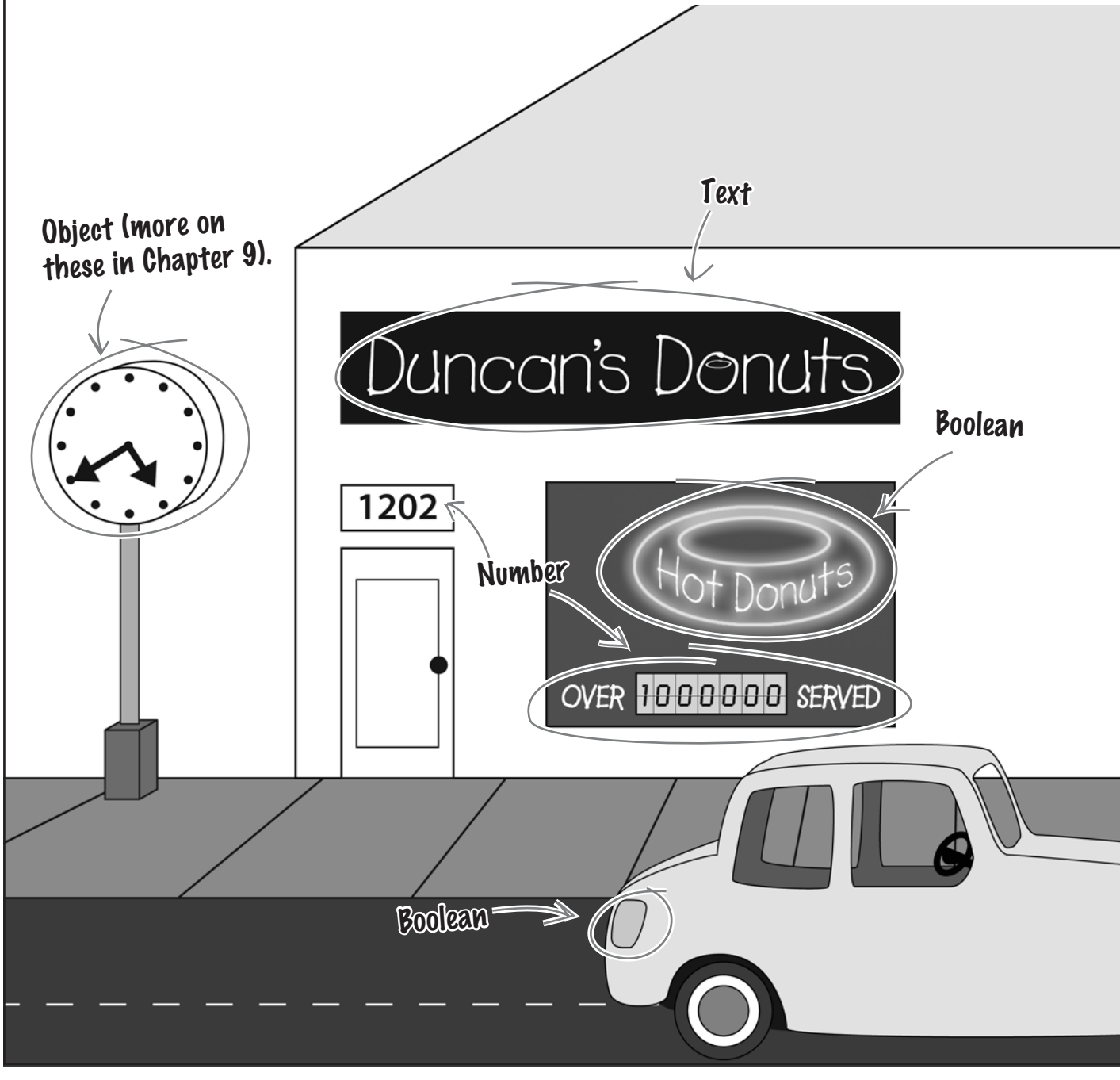
Find everything that could be represented by a JavaScript data type, and write down what type that thing should be.




HARRY'S
BARBER SHOP
1204
SPECIAL
$5.99
DON'T
WALK
WALK
THE LEFT BANK
1206
TODAY'S RATE
2.5%
WEBVILLE
E4RL5

Sharpen your pencil Solution

Your job was to find everything that JavaScript could represent, and figure out the type JavaScript would use.




Text
HARRY'S
BARBER SHOP
Text
THE LEFT BANK
Number
1206
TODAY'S RATE
2.5%
Number
1204
Number
SPECIAL
$5.99
Boolean
DON'T
WALK
WALK
Boolean
Number (different number for each state).
WEBVILLE
E4RL5
Text. When numbers and characters are mixed, the data is ALWAYS considered text.

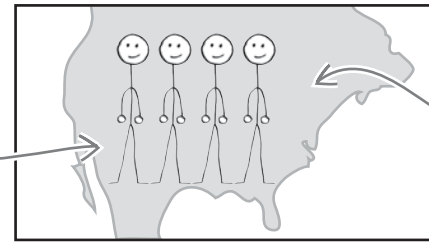
Constants stay the SAME, variables can CHANGE

Storing data in JavaScript isn't just about **type**, it's also about **purpose**. What do you want to do with the data? Or more specifically, will the data **change** throughout the course of your script? The answers determine whether you code your data type in JavaScript as a *variable* or a *constant*. ***A variable changes throughout the course of a script***, while ***a constant never changes its value***.

Variable data can change—constant data is fixed.

Constant

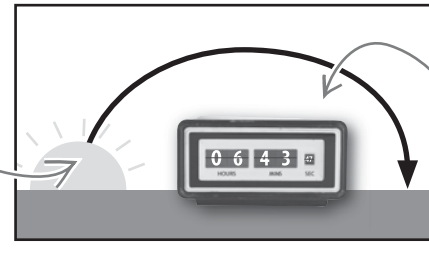
Land area of 3.5 million square miles—a constant (unless you wait around long enough for the Earth's tectonic plates to shift).



Variable

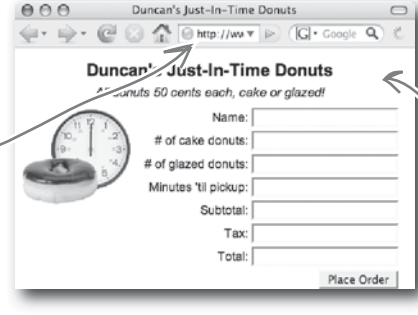
Population of 300 million people—a variable since the U.S. population is still on the rise.

24 hours in a day—a constant as far as humans are concerned, even though the moon is slowly leaving us.



Sunrise at 6:43am—a variable since the sunrise changes every day.

URL of web page is www.duncansdonuts.com—a constant, unless the donut biz takes a dramatic downturn.



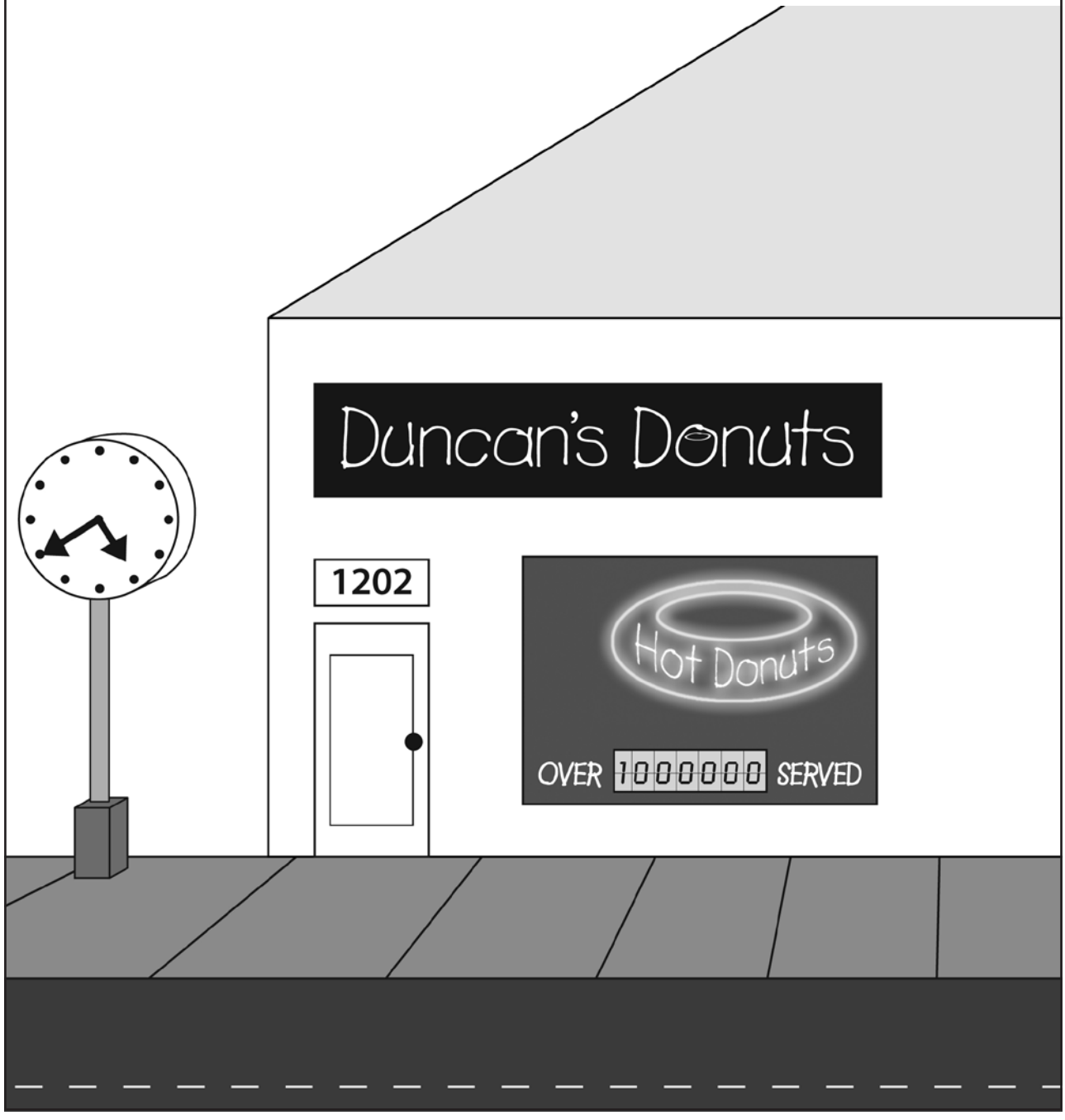
324 total page hits—a variable since users are constantly visiting the page and changing the hit count.

BRAIN POWER

What other information types could involve both variables and constants?

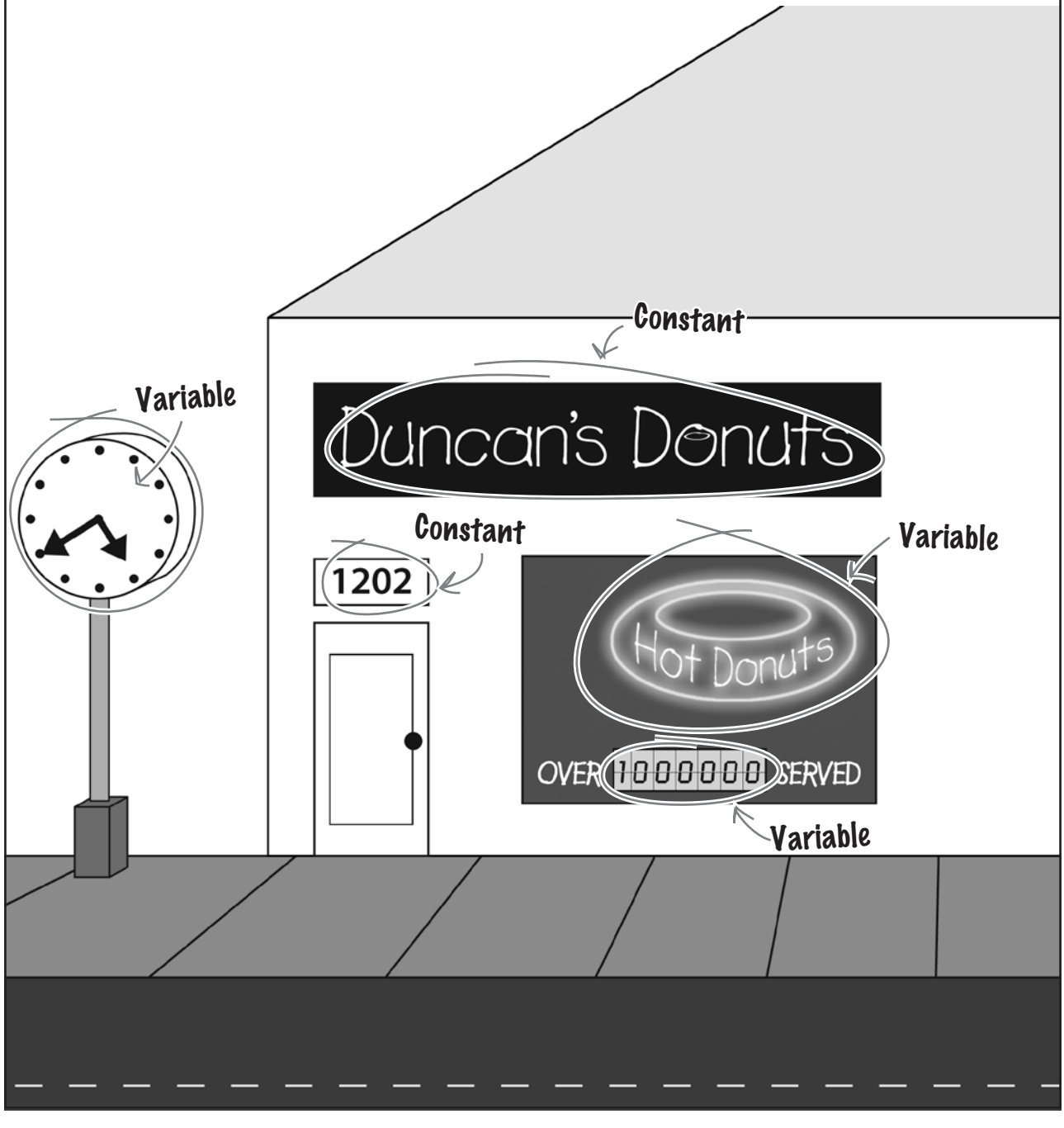
Sharpen your pencil

Circle all of the data at Duncan's Donuts, and then identify each thing you circled as being either a variable or a constant.

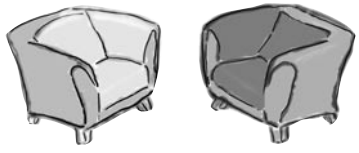


Sharpen your pencil Solution

Your job was to find all the variables and constants.



Fireside Chats



Tonight's talk: **Variable and Constant square off over data storage.**

Variable:

When it comes to storing data, I offer the most in flexibility. You can change my value all you want. I can be set to one value now and some other value later—that's what I call freedom.

Sure, but your mule-headed resistance to change just won't work in situations where data has to take on different values over time. For example, a rocket launch countdown has to change as it counts down from 10 to 1. Deal with that!

Yeah, sure, whatever. How do you get off calling variation a bad thing. Don't you realize that change can be a good thing, especially when you've got to to store information entered by the user, perform calculations, anything like that?

I suppose we'll just have to agree to disagree.

Constant:

And I call that flip-flopping! I say pick a value and stick to it. It's my ruthless consistency that makes me so valuable to scripters...they appreciate the predictability of data that always stays the course.

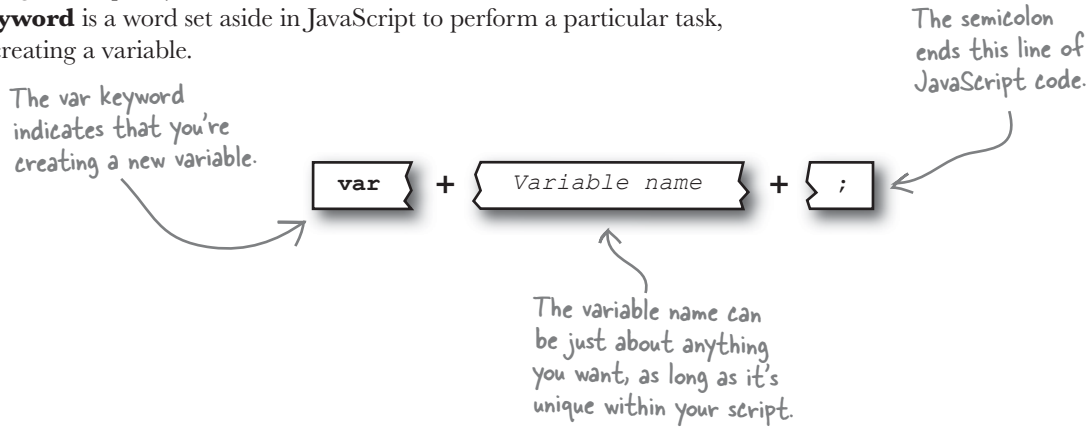
Oh, so you think *you're* the only data storage option for mission critical applications, huh? Wrong! How do you think that rocket ever got to the launch pad? Because someone was smart enough to make the launch date a constant. Show me a deadline that's a variable and I'll show you a project behind schedule.

I say the more things change, the more they stay the same. And really, why change in the first place? Settle on a good value from the start and leave it alone. Think about the comfort in knowing that a value can never be changed, accidentally or otherwise.

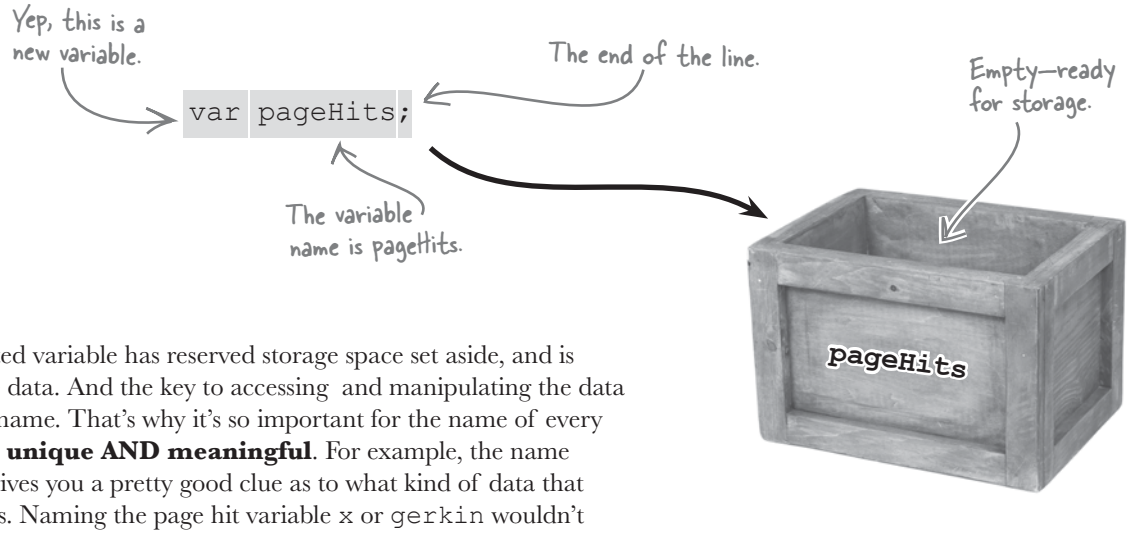
Actually, I've disagreed with you all along.

Variables start out without a value

A variable is a **storage location** in memory with a **unique name**, like a label on a box that's used to store things. You create a variable using a special JavaScript keyword called `var`, and the name of the new variable. A **keyword** is a word set aside in JavaScript to perform a particular task, like creating a variable.



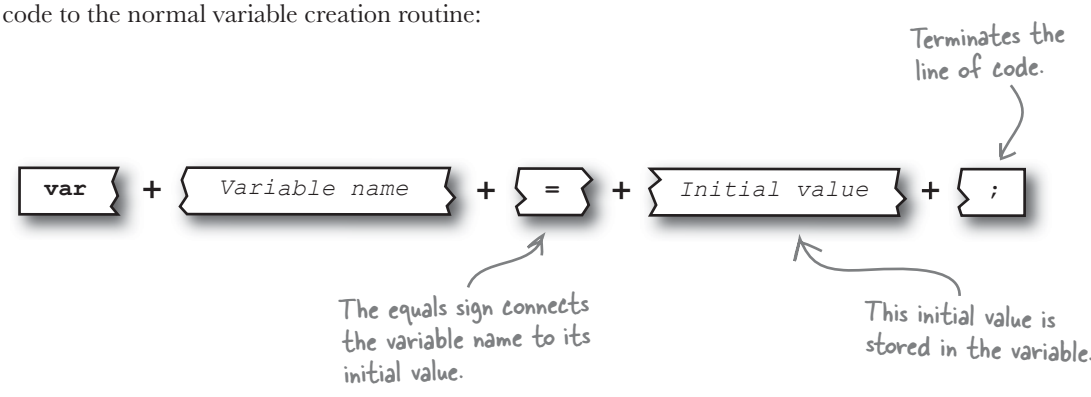
When you create a variable using the `var` keyword, that variable's initially empty.... it has no value. It's fine for a variable to start off being empty as long as you don't attempt to read its value before assigning it a value. It'd be like trying to play a song on your MP3 player before loading it with music.



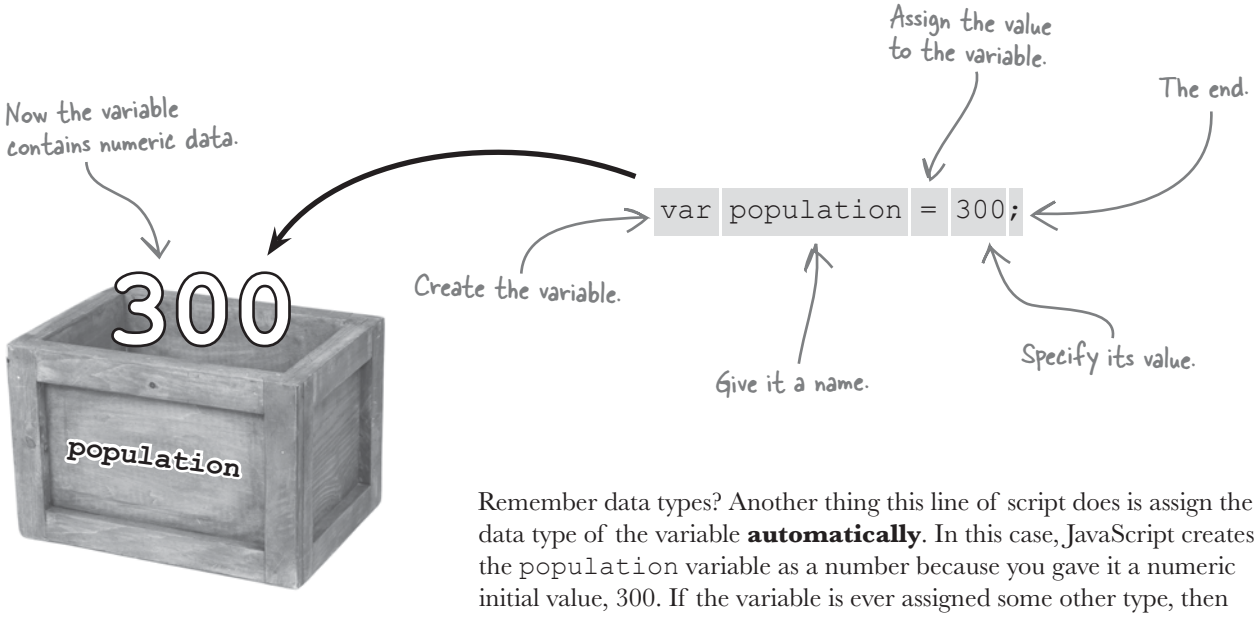
A newly-created variable has reserved storage space set aside, and is ready to store data. And the key to accessing and manipulating the data it stores is its name. That's why it's so important for the name of every variable to be **unique AND meaningful**. For example, the name `pageHits` gives you a pretty good clue as to what kind of data that variable stores. Naming the page hit variable `x` or `gerkin` wouldn't have been nearly as descriptive.

Initialize a variable with "="

You don't have to create variables without an initial value. In fact, it's usually a pretty good idea to give a variable a value when you first create it. That's called **initializing** a variable. That's just a matter of adding a tiny bit of extra code to the normal variable creation routine:



Unlike its blank counterpart, an initialized variable is immediately ready to be used... it already has a value stored in it. It's like buying a preloaded MP3 player—ready to play right out of the box.



Remember data types? Another thing this line of script does is assign the data type of the variable **automatically**. In this case, JavaScript creates the `population` variable as a number because you gave it a numeric initial value, 300. If the variable is ever assigned some other type, then the type of the variable changes to reflect the new data. Most of the time JavaScript handles this automatically; there will be cases where you will need to be explicit and even convert to a different data type...but we'll get to all that a bit later.

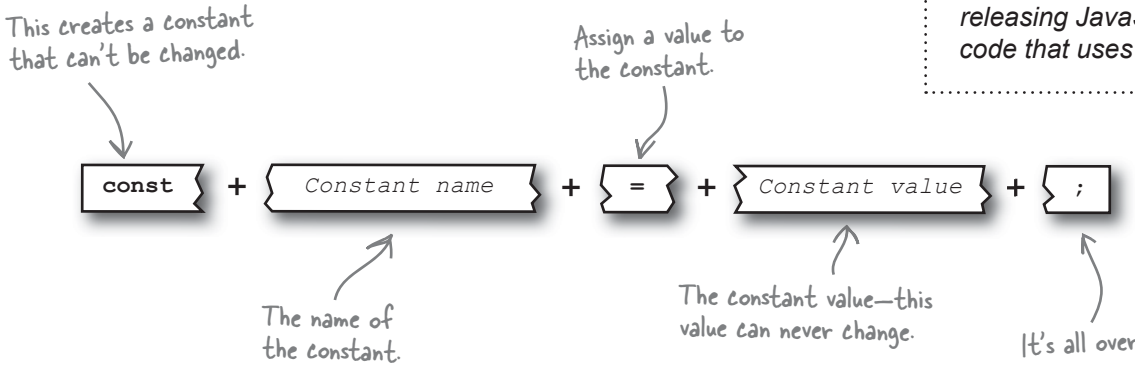
Constants are resistant to change

Initializing a variable is all about setting its **first** value—there's nothing stopping that value from being changed **later**. To store a piece of data that can never change, you need a constant. Constants are created just like initialized variables, but you use the `const` keyword instead of `var`. And the "initial" value becomes a **permanent** value...constants play for keeps!

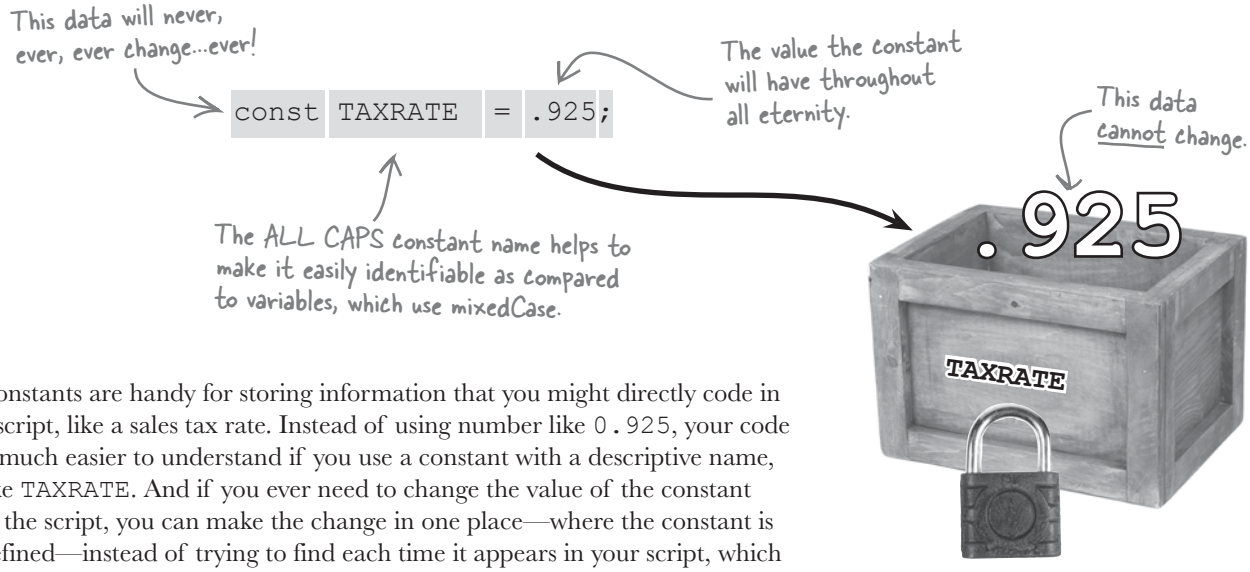
Watch it!

Not all browsers support the const keyword.

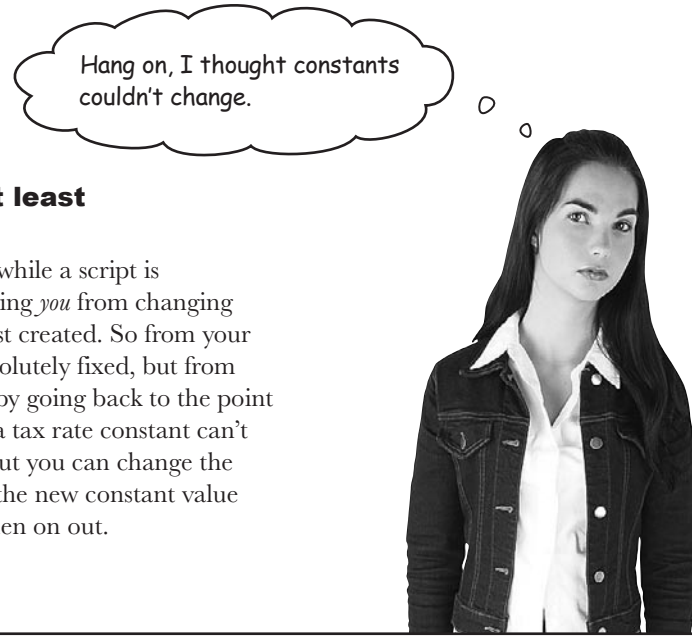
The `const` keyword is fairly new to JavaScript, and not all browsers support it. Be sure to double check your target browsers before releasing JavaScript code that uses const.



The biggest difference between creating a constant and a variable is you have to use the `const` keyword instead of `var`. The syntax is the same as when you're initializing a variable. But, constants are often named using all capital letters to make them **STANDOUT** from variables in your code.



Constants are handy for storing information that you might directly code in a script, like a sales tax rate. Instead of using number like `0.925`, your code is much easier to understand if you use a constant with a descriptive name, like `TAXRATE`. And if you ever need to change the value of the constant in the script, you can make the change in one place—where the constant is defined—instead of trying to find each time it appears in your script, which could get really complicated.



Constants can't change, at least not without a text editor.

It's true that constants can't change while a script is **running**...but there's nothing stopping *you* from changing the value of a constant where it's first created. So from your script's perspective, a constant is absolutely fixed, but from your perspective, it can be changed by going back to the point where you created the constant. So a tax rate constant can't change while the script is running, but you can change the rate in your initialization code, and the new constant value will be reflected in the script from then on out.

Decide whether each of the following pieces of information should be a variable or a constant, and then write the code to create each, and initialize them (if that's appropriate).


The current temperature, which is initially unknown


The conversion unit from human years to dog years (1 human year = 7 dog years)


The countdown for a rocket launch (from 10 to 0)


The price of a tasty donut (50 cents)

.....

.....

.....

.....

Your job was to decide whether each of the following pieces of information should be a variable or a constant, and then write the code to create them, and initialize them when appropriate.


The current temperature, which is initially unknown


The conversion unit from human years to dog years (1 human year = 7 dog years)


The countdown for a rocket launch (from 10 to 0)


The price of a tasty donut (50 cents)

```
var temp;
```

The temperature changes all the time and the value is unknown, so a blank variable is the ticket.

```
const HUMANTODOG = 7;
```

This conversion rate doesn't change, so it makes perfect sense as a constant.

```
var countdown = 10;
```

The countdown has to count from 10 to 1, so it's a variable, and it has to be initialized to the start count (10).

```
var donutPrice = 0.50;  or  const DONUTPRICE = 0.50;
```

If the donut price changes, it makes sense as a variable that's initialized to the current price.

...or maybe the donut price is fixed, in which case a constant set to the price works better.

there are no Dumb Questions

Q: If I don't specify the data type of JavaScript data, how does it ever know what the type is?

A: Unlike some programming languages, JavaScript doesn't allow you to explicitly set the type of a constant or variable. Instead, the type is **implied** when you set the value of the data. This allows JavaScript variables a lot of flexibility since their data types can change when different values are assigned to them. For example: if you assign the number 17 to a variable named `x`, the variable is a number. But if you turn around and assign `x` the text "seventeen", the variable type changes to string.

Q: If the data type of JavaScript data is taken care of automatically, why should I even care about data types?

A: Because there are plenty of situations where you can't rely solely on JavaScript's automatic data type handling. For example, you may have a number stored as text that you want to use in a calculation. You have to convert the text type to the number type in order to do any math calculations with the number. The reverse is true when displaying a number in an alert box—it must first be converted to text. JavaScript will perform the number-to-text conversion automatically, but it may not convert exactly like you want it to.

Q: Is it OK to leave a variable uninitialized if I don't know what it's value is up front?

A: Absolutely. The idea behind initialization is to try to head off problems where you might try to access a variable when it doesn't have a value. But, there are also times where there's no way to know the value of a variable when you first create it. If that happens, just make sure that the variable gets set before you try to use it. And keep in mind that you can always initialize a variable to a "nothing" value, such as `""` for text, `0` for numbers, or `false` for booleans. This helps eliminate the risk of accidentally accessing uninitialized data.

Q: Is there any trick to knowing when to use a variable and when to use a constant?

A: While it's easy to just say constants can't change and variables can, there's a bit more to it than that. In many cases you'll start out using variables for everything, and only realize that there are opportunities to make some of those variables into constants later. Even then, it's rare that you'll be able to turn a variable into a constant. More likely, you'll have a fixed piece of text or number that is used in several places, like a repetitive greeting or conversion rate. Instead of duplicating the text or number over and over, create a constant for it and use that instead. Then if you ever need to adjust or change the value, you can do it in one place in your code.

Q: What happens to script data when a web page is reloaded?

A: Script data gets reset to its initial values, as if the script had never been run before. In other words, refreshing a web page has the same effect on the script as if the script was being run for the first time.

Data types are established when variable's and constant's values are set.

BULLET POINTS

- Script data can usually be represented by one of the three basic data types: **text**, **number**, or **boolean**.
- A **variable** is a piece of data that can **change** over the course of a script.
- A **constant** is a piece of information that **cannot change**.
- The `var` keyboard is used to **create variables**, while `const` is used to **create constants**.
- The **data type** of a piece of JavaScript data is established when you **set the data** to a certain value, and for variables the type can change.

What's in a name?

Variables, constants, and other JavaScript syntax constructs are identified in scripts using unique names known as **identifiers**. JavaScript identifiers are like the names of people in the real world, except they aren't as flexible (people can have the same name, but JavaScript variables can't). In addition to being unique within a script, identifiers must abide by a few naming laws laid down by JavaScript:

An identifier must be at least one character in length.

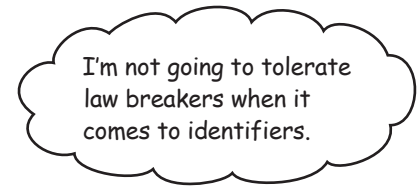
The first character in an identifier must be a letter, an underscore (_), or a dollar sign ($).

Each character after the first character can be a letter, an underscore (_), a dollar sign ($), or a number.

Spaces and special characters other than _ and $ are not allowed in any part of an identifier.

When you create a JavaScript identifier for a variable or constant, you're naming a piece of information that typically has **meaning** within a script. So, it's not enough to simply abide by the laws of identifier naming. You should definitely try to add context to the names of your data pieces so that they are immediately **identifiable**.

Of course, there are times when a simple `x` does the job—not every piece of data in a script has a purpose that is easily described.



Sheriff J.S. Justice, dedicated lawman.

Identifiers should be descriptive so that data is easily identifiable, not to mention legal...

Legal and illegal variable and constant names

Not legal: can't start with a number.

`firstName`

Legal: all letters, so everything is fine.

`top100`

Legal: numbers don't appear at the beginning, so this is A-OK.

`ka_chow`

Legal: letters and underscores are all good.

Not legal: can't start with a special character other than _ or $.

`_topSecret`

Legal: Starting with an underscore isn't a problem at all—some people even use this technique to name variables that have a special meaning.

`$total`

Legal: although it looks a little strange, starting with a dollar sign is perfectly legal.

Exercise

The pastry wizards over at Duncan's Donuts are trying to decide on a promotional cap design. Unfortunately, they don't realize that some of the designs violate JavaScript's rules for naming identifiers. Mark an X over the names on the caps that won't cut it in JavaScript.

Exercise Solution

Your job was to mark an X over the caps that have variable names that won't cut it in JavaScript.

Exclamation points aren't allowed anywhere in an identifier.

Sorry, spaces aren't allowed either.

The pound symbol is only going to invoke the wrath of Sheriff Justice.

Variable names often use CamelCase

Although there aren't any JavaScript laws governing how you style identifier names, the JavaScript community has some **unofficial** standards. One of these standards is using **CamelCase**, which means mixing case within identifiers that consist of more than one word (remember, you can't have spaces in a variable name). Variables usually use **lower** camel case, in which the first word is all lowercase, but additional words are mixed-case.

```
num_cake_donuts
```

Separating multiple words with an underscore in a variable identifier isn't illegal, but there's a better way.

The first letter of each word is capitalized.

```
NumCakeDonuts
```

Better... this style is known as camel case, but it still isn't quite right for variables.

The first letter of each word except the first is capitalized.

```
numCakeDonuts
```

Ah, there it is—lower camel case is perfect for naming variables with multiple words.

lowerCamelCase is used to name multiWord variables.

JavaScript Magnets

The identifier magnets have gotten separated from the variables and constants they identify at Duncan's Donuts. Match up the correct magnet to each variable/constant, and make sure you avoid magnets with illegal names. Bonus points: identify each data type.

The number of cups of coffee sold today

The name of the employee of the month

The amount of flour that goes into a single batch of donuts

The business tax number used to file sales tax

The status of the alarm system

The record holder for most eclairs eaten in a sitting

```
employee*of*the*Month
cups-o-coffee
Employee of the Month
alarmStatus
Tax#
FLOURPERBATCH
alarm_status
eclairRECORDHOLDER
numCups
TAXNUM
eclairRecord
eclairWinner!
employeeOfMonth
ALARM-STATUS
flour quantity
#OfCups
```

JavaScript Magnets Solution

The identifier magnets have gotten separated from the variables and constants they identify at Duncan's Donuts. Match up the correct magnet to each variable/constant, and make sure you avoid magnets with illegal names. Bonus points: identify each data type.

The number of cups of coffee sold today

Number → `numCups`

The name of the employee of the month

Text → `employeeOfMonth`

The amount of flour that goes into a single batch of donuts

`FLOURPERBATCH` → Number

The record holder for most eclairs eaten in a sitting

`eclairRecord` → Text

The status of the alarm system

`alarmStatus` → Boolean

The business tax number used to file sales tax

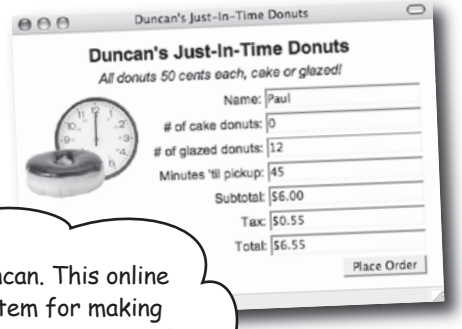
`TAXNUM` → Number

All these leftovers are illegal names in JavaScript.

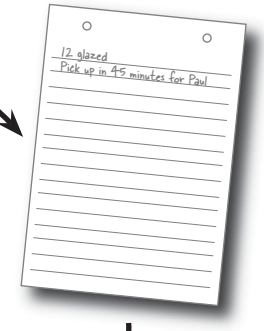
- `cups-o-coffee`
- `Employee of the Month`
- `alarm_status`
- `eclairWinner!`
- `eclairRECORDHOLDER`
- `employee*of*the*Month`
- `Tax#`
- `ALARM-STATUS`
- `flour quantity`
- `#OfCups`

The next big thing (in donuts)

You may know about Duncan's Donuts, but you haven't met Duncan or heard about his big plan to shake up the donut market. Duncan wants to take the "Hot Donuts" business to the next level...he wants to put it online! His idea is **just-in-time donuts**, where you place an order online and enter a specific pick-up time, and have a **hot** order of donuts waiting for you at the precise pick-up time. **Your job is to make sure the user enters the required data, as well as calculate the tax and order total.**



JavaScript captures the user input and calculates the tax and total.



Hey, I'm Duncan. This online ordering system for making hot donuts is going to ROCK!

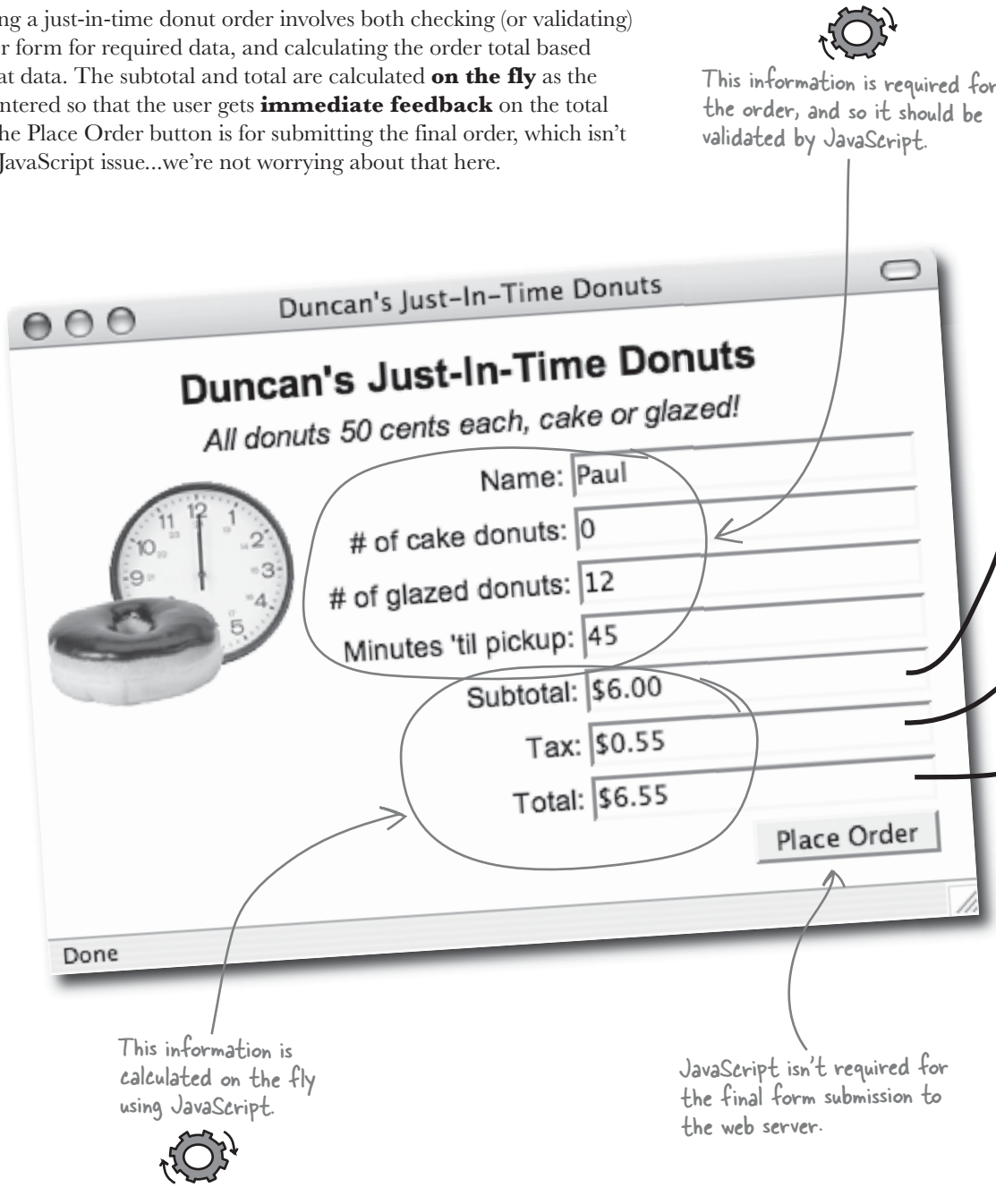
The Donut Blaster 3000.

x 12

Hot and on time!

Plan the Duncan's Donuts web page

Processing a just-in-time donut order involves both checking (or validating) the order form for required data, and calculating the order total based upon that data. The subtotal and total are calculated **on the fly** as the data is entered so that the user gets **immediate feedback** on the total price. The Place Order button is for submitting the final order, which isn't really a JavaScript issue...we're not worrying about that here.



The subtotal is calculated by multiplying the total number of donuts by the price per donut:

(# of cake donuts + # of glazed donuts) x price per donut

The tax is calculated by multiplying the subtotal by the tax rate:

subtotal x tax rate

The order total is calculated by adding the subtotal and the tax:

subtotal + tax

It looks like Duncan has a fair amount of data to keep track of in his form. Not only does he have to keep up with the various pieces of information entered by the user, but there are also several pieces of data that get calculated in JavaScript code.

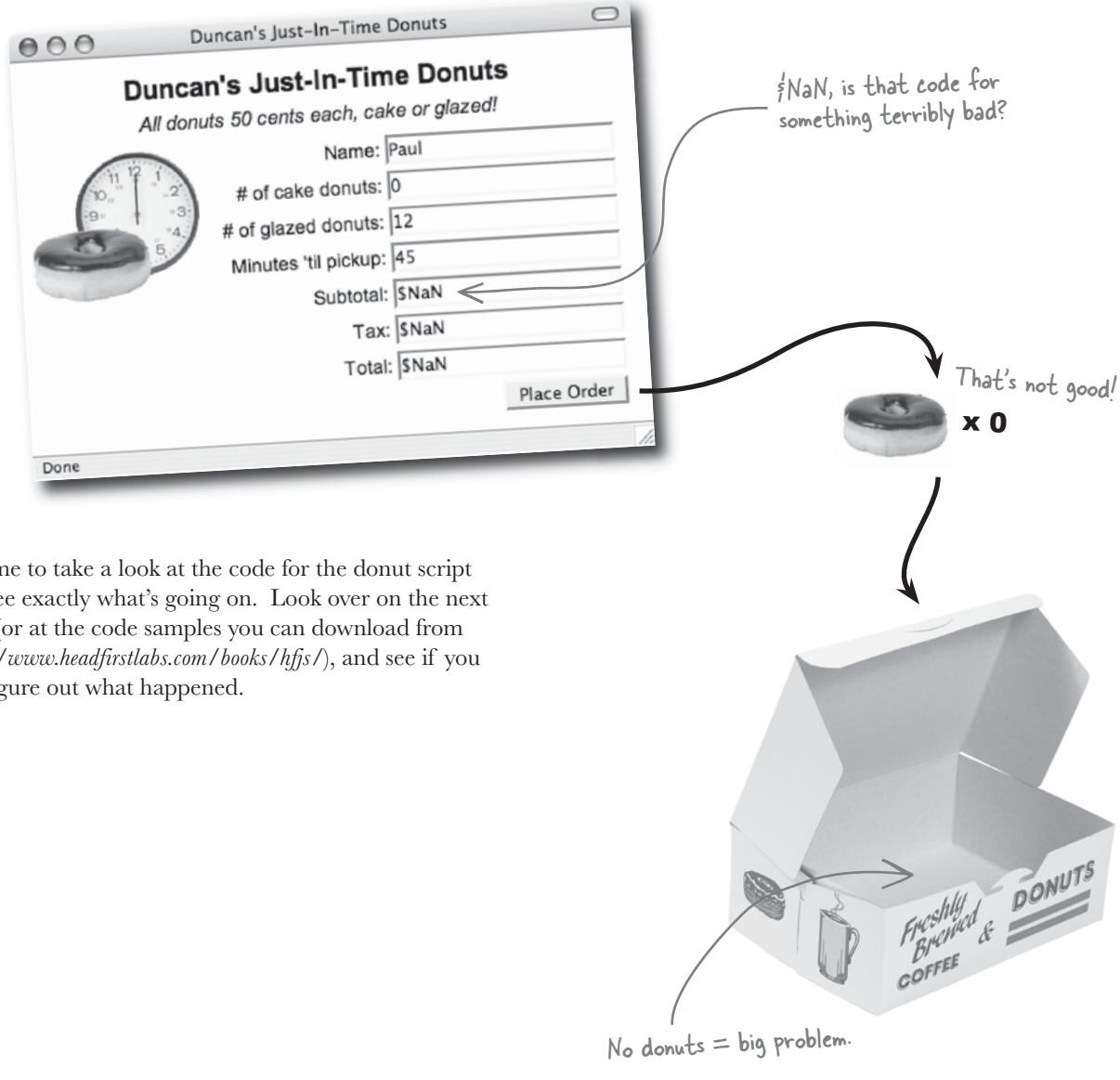
With a little help from JavaScript, each order is filled just in time...genius!

BRAIN POWER

What variables and constants will you need to carry out these calculations? What would you name them?

A first take at the donut calculations

Duncan tried to write the JavaScript for the calculations himself, but ran into problems. As soon as a user enters a number of donuts, the on-the-fly calculations immediately go haywire. They're coming up with values of `$NaN`, which doesn't make much sense. Even worse, orders aren't getting filled and customers aren't exactly thrilled with Duncan's technological "advancements."



It's time to take a look at the code for the donut script and see exactly what's going on. Look over on the next page (or at the code samples you can download from *http://www.headfirstlabs.com/books/hfjs/*), and see if you can figure out what happened.

This code is called to update the order by calculating the subtotal and total on the fly.

Since the data entered by the user looks OK, there must be something wrong with the constants.

This code submits the order to the server and confirms the order with the user.

The order is updated when either number of donuts changes.

The order is submitted when the Place Order button is clicked.

```
<html>
  <head>
    <title>Duncan's Just-In-Time Donuts</title>
    <link rel="stylesheet" type="text/css" href="donuts.css" />
    <script type="text/javascript">
      function updateOrder() {
        const TAXRATE;
        const DONUTPRICE;
        var numCakeDonuts = document.getElementById("cakedonuts").value;
        var numGlazedDonuts = document.getElementById("glazeddonuts").value;
        var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
        var tax = subTotal * TAXRATE;
        var total = subTotal + tax;
        document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
        document.getElementById("tax").value = "$" + tax.toFixed(2);
        document.getElementById("total").value = "$" + total.toFixed(2);
      }
      function placeOrder() {
        // Submit order to server...
        form.submit();
      }
    </script>
  </head>
  <body>
    <div id="frame">
      ...
      <form name="orderform" action="donuts.php" method="POST">
        ...
        <div class="field">
        # of cake donuts: <input type="text" id="cakedonuts" name="cakedonuts"
          value="" onchange="updateOrder();" />
        </div>
        <div class="field">
        # of glazed donuts: <input type="text" id="glazeddonuts"
          name="glazeddonuts" value="" onchange="updateOrder();" />
        </div>
        ...
        <div class="field">
        <input type="button" value="Place Order"
        onclick="placeOrder(this.form);" />
        </div>
      </form>
    </div>
  </body>
</html>
```

Sharpen your pencil

Write down what you think went wrong with Duncan's just-in-time donut script code.

.....

.....

Sharpen your pencil Solution

Write down what you think went wrong with Duncan's just-in-time donut script code.

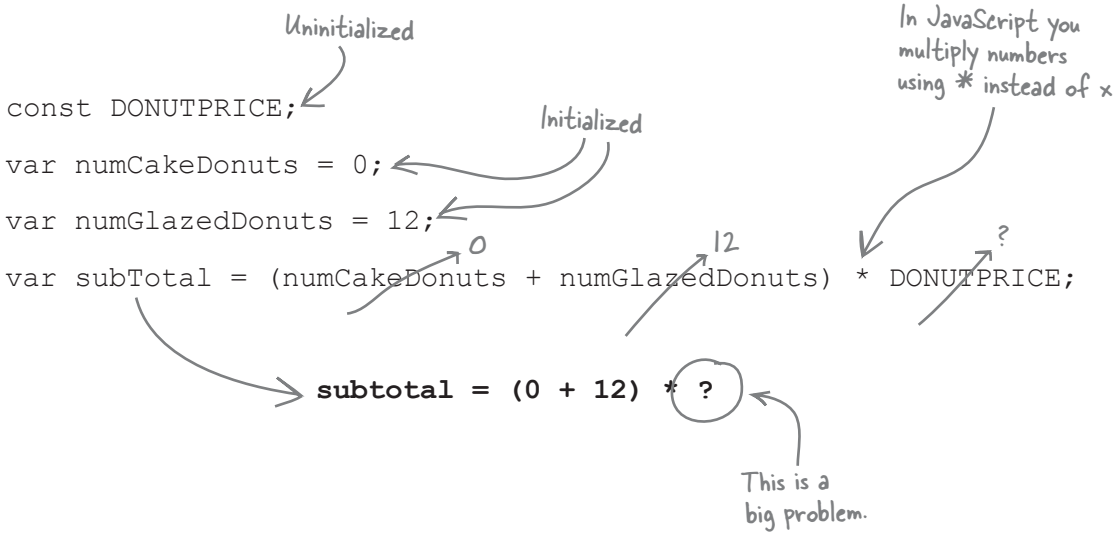
The two constants, TAXRATE and DONUTPRICE, aren't initialized, which means the calculations that depend on them can't be completed.

You shouldn't ever uninitialize a constant.

You can uninitialize a constant by never giving it a value, but it's **a very bad idea**. When you don't initialize a constant when you create it, that constant ends up in no man's land—it has no value, and even worse, **it can't be given one**. An uninitialized constant is essentially a **coding error**, even though browsers don't usually let you know about it.

Initialize your data...or else

When you don't initialize a piece of data, it's considered *undefined*, which is a fancy way of saying it has no value. That doesn't mean it isn't worth anything, it just means it doesn't contain any information... yet. The problem shows up when you try to use variables or constants that haven't been initialized.

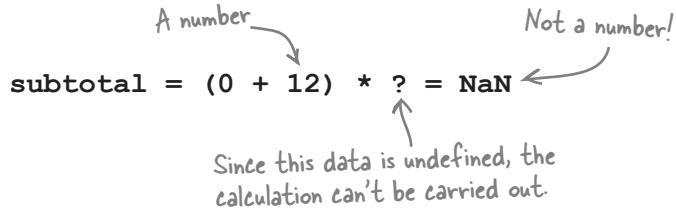


The `DONUTPRICE` constant is **uninitialized**, which means it has **no value**. Actually JavaScript has a special value just for this "non-value" state: `undefined`. It's sort of like how your phone's voice mail will report "no messages" when you don't have any messages—"no messages" is technically still a message but it's purpose is to represent the **lack of messages**. Same deal with `undefined`—it indicates a **lack of data**.

A piece of data is undefined when it has no value.

NaN is NOT a number

Just as `undefined` represents a special data **condition**, there's another important value used to indicate a special case with JavaScript variables: `NaN`. `NaN` means **Not a Number**, and it's what the `subTotal` variable gets set to since there isn't enough information to carry out the calculation. In other words, you treated a missing value as a number... and got `NaN`.



So solving the `NaN` problem requires initializing the `DONUTPRICE` constant when you create it:

```
const DONUTPRICE = 0.50;
```

NaN is a value that isn't a number even though you're expecting the value to be one.

there are no Dumb Questions

Q: What does it mean that identifiers must be unique within a script?

A: The whole point of identifiers is to serve as a unique name that you can use to *identify* a piece of information in a script. In the real world, it isn't all that uncommon for people to have the same name... but then again, people have the ability to deal with such "name clashes" and figure out who is who. JavaScript isn't equipped to deal with ambiguity, so it needs you to carefully distinguish different pieces of information by using *different* names. You do this by making sure identifiers within your script code are all unique.

Q: Does every identifier I create have to be unique, or unique only in a specific script?

A: Identifier uniqueness is really only important within a **single** script, and in some cases only within certain portions of a single script. However, keep in mind that scripts for big web applications can get quite large, spread across lots of files. In this case, it becomes more challenging to ensure uniqueness among all identifiers. The good news it that it isn't terribly difficult to maintain identifier uniqueness in scripts of your own, provided you're as descriptive as possible when naming them.

Q: I still don't quite understand when to use camel case and lower camel case. What gives?

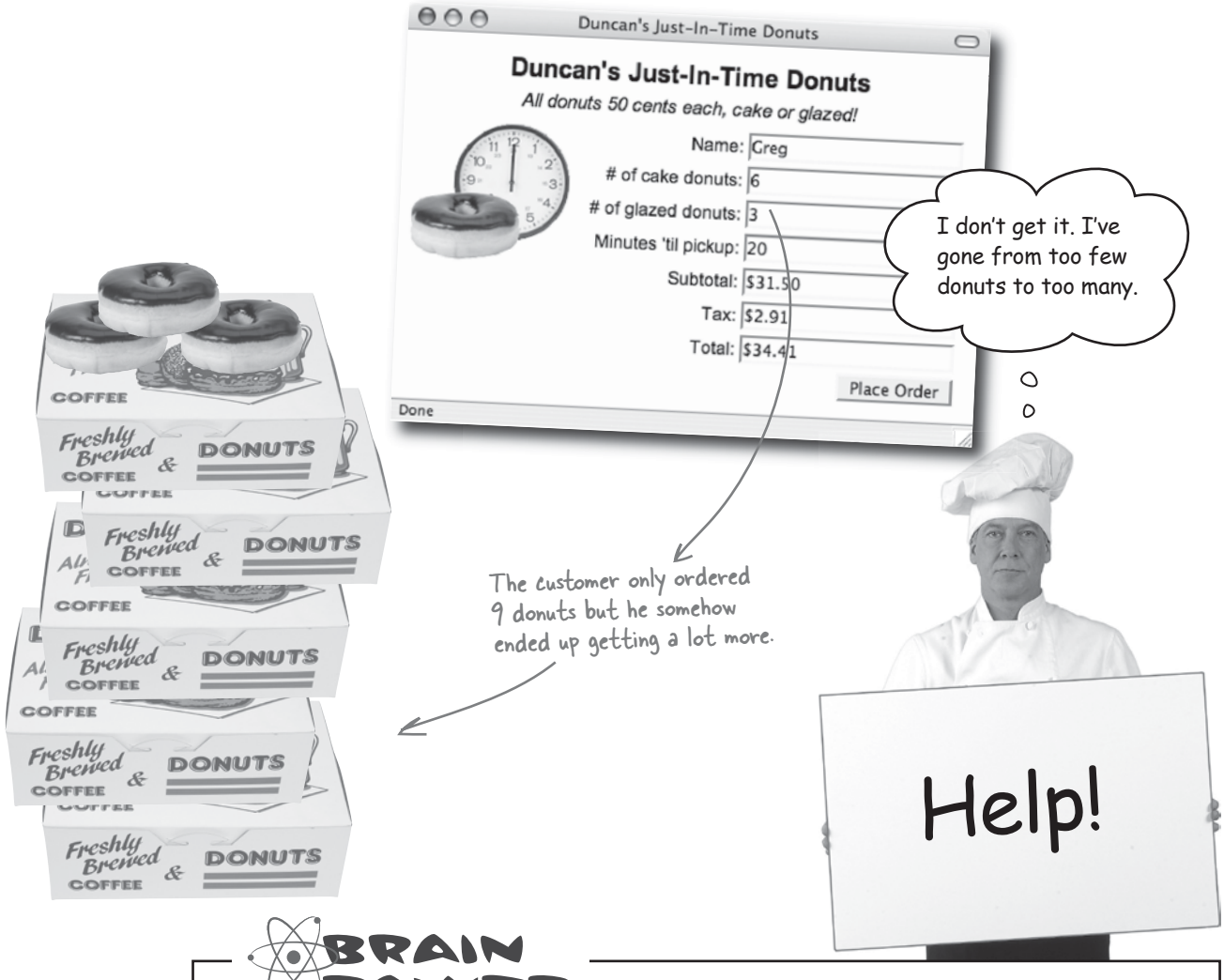
A: Camel case (with the first word capitalized) only applies to naming JavaScript objects, which we'll talk about in Chapter 9. Lower camel case applies to variables and functions, and is the same as camel case, except the first letter in the identifier is lowercase. So camel case means you would name an object `Donut`, while lower camel case means you would name a function `getDonut()` and a variable `numDonuts`. There isn't a cute name for constants—they're just all caps.

Q: Are text and boolean data considered NaN?

A: Theoretically, yes, since they definitely aren't numbers. But in reality, no. The purpose of `NaN` is to indicate that a number isn't what you think it is. In other words, `NaN` isn't so much a description of JavaScript data in general as it is an error indicator for number data types. You typically only encounter `NaN` when performing calculations that expect numbers but for some reason are given non-numeric data to work with.

Meanwhile, back at Duncan's...

Back at Duncan's Donuts, things have gone from bad to worse. Instead of empty boxes, now there are donuts everywhere—every order is somehow getting overcalculated. Duncan is getting overwhelmed with complaints of donut overload and pastry gouging.



BRAIN POWER

What could be wrong with how the donut quantity data is being handled?

You can add more than numbers

In JavaScript, **context** is everything. Specifically, it matters what **kind** of data you're manipulating in a given piece of code, not just what you're **doing** with the data. Even something as simple as adding two pieces of information can yield very different results depending upon the **type** of data involved.

```
1 + 2 = 3
```

Numeric Addition

Adding two numbers does what you might expect—it produces a result that is the **mathematical** addition of the two values.

```
"do" + "nuts" = "donuts"
```

Fancy word for "stick these things together".

String Concatenation

Adding two strings also does what you might expect but it's very different than mathematical addition—here the strings are attached **end-to-end.**

Knowing that strings of text are added differently than numbers, what do you think happens when an attempt is made to add two textual numbers?

```
"1" + "2" = ?
```

Addition, concatenation, what gives?

JavaScript doesn't really care what's in a string of text—it's all characters to JavaScript. So the fact that the strings hold numeric characters makes no difference... string concatenation is still performed, resulting in an **unexpected** result if the **intent** was numeric addition.

```
"1" + "2" = "12"
```

The result is a string that doesn't look like mathematical addition at all.

Since these are strings and not numbers, they are "added" using string concatenation.

Watch it!

Always make sure you're adding what you think you're adding.

Accidentally concatenating strings when you intend to add numbers is a common JavaScript mistake. Be sure to convert strings to numbers before adding them if your intent is numeric addition.

parseInt() and parseFloat(): converts text to a number

Despite the addition/concatenation problem, there are legitimate situations where you need to perform a mathematical operation on a number that you've got stored as a string. In these cases, you need to **convert** the string to a number **before** performing any numeric operations on it. JavaScript provides two handy functions for carrying out this type of conversion:

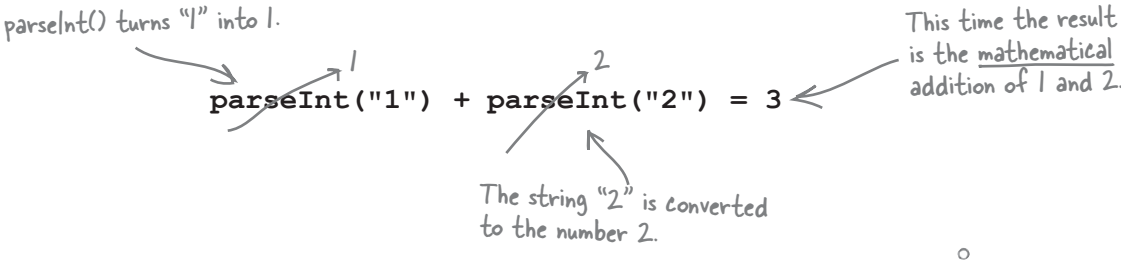
parseInt()

Give this function a string and it converts the string to an integer

parseFloat()

Give this function a string and it converts the string to a floating point (decimal) number

Each of these built-in functions accepts a string and returns a number after carrying out the conversion:



Keep in mind that the `parseInt()` and `parseFloat()` functions **aren't guaranteed** to always work. They're only as good as the information you provide them. They'll do their best at converting strings to numbers, but the idea is that you should be providing them with strings that **only** contain numeric characters.

```
parseFloat("$31.50") = NaN
```

This code is a problem because the $ character confuses the function.

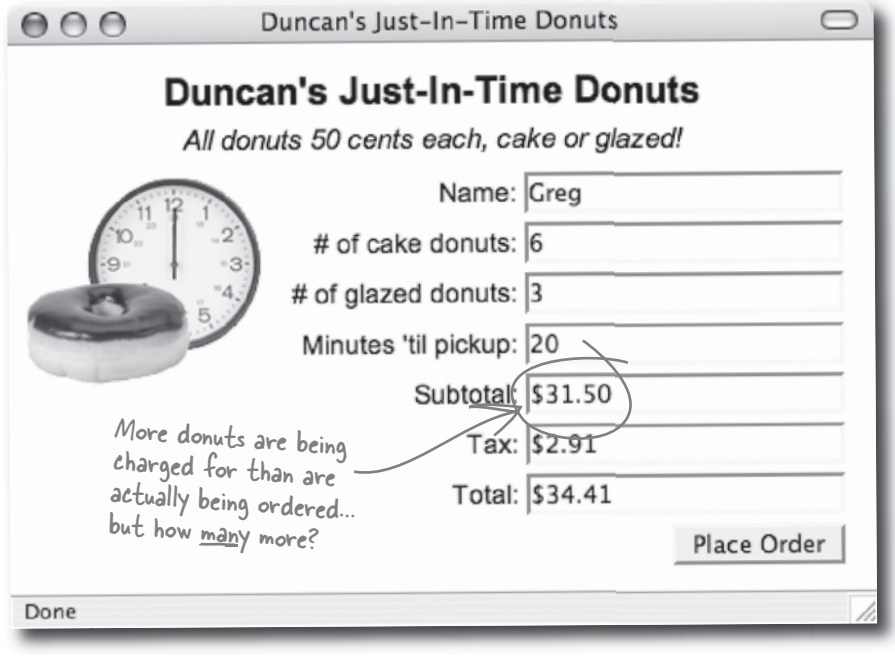
Surprise, surprise, the result is Not a Number.

Don't worry if this function stuff is still a little confusing.

You'll get the formal lowdown on functions a little later—for now all you really need to know is that functions allow you pass them information and then give you back something in return.

Why are extra donuts being ordered?

Take a closer look at the just-in-time donut order form. We should be able to figure out why so many donuts are being accidentally ordered...



We can divide the subtotal by the price for each donut...and the answer is how many donuts are getting ordered.

The order subtotal.

```
$31.50 / $0.50 = 63 donuts
```

The price per donut.

The total number of donuts actually ordered... hmmm.

Remember "1" + "2" = "12"? Looks kind of like that, doesn't it?

This looks a whole lot like the numeric string addition problem, especially when you consider that form data is always stored as strings regardless of what it is. Even though numbers are entered into the form fields, from a JavaScript perspective, they're really just text. So we just need to convert the strings to actual numbers to prevent a numeric addition from being misinterpreted as a string concatenation.

Sharpen your pencil

Using the pieces of code below to grab the contents of the donut quantity form fields, write the missing lines of code in Duncan's `updateOrder()` function so that the donut quantities are converted from strings to numbers.

```
document.getElementById("cakedonuts").value
```

This code gets the number of cake donuts entered by the user in the donut form.

This code grabs the number of glazed donuts entered into the donut form.

```
document.getElementById("glazeddonuts").value
```

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts =

    ..........
  var numGlazedDonuts =

    ..........
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

Sharpen your pencil Solution

Using the pieces of code below to grab the contents of the donut quantity form fields, write the missing lines of code in Duncan's `updateOrder()` function so that the donut quantities are converted from strings to numbers.

```
document.getElementById("cakedonuts").value
```

```
document.getElementById("glazeddonuts").value
```

Since both numbers are integers, parseInt() is used for the conversion.

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts =
    parseInt(document.getElementById("cakedonuts").value);
  var numGlazedDonuts =
    parseInt(document.getElementById("glazeddonuts").value);
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

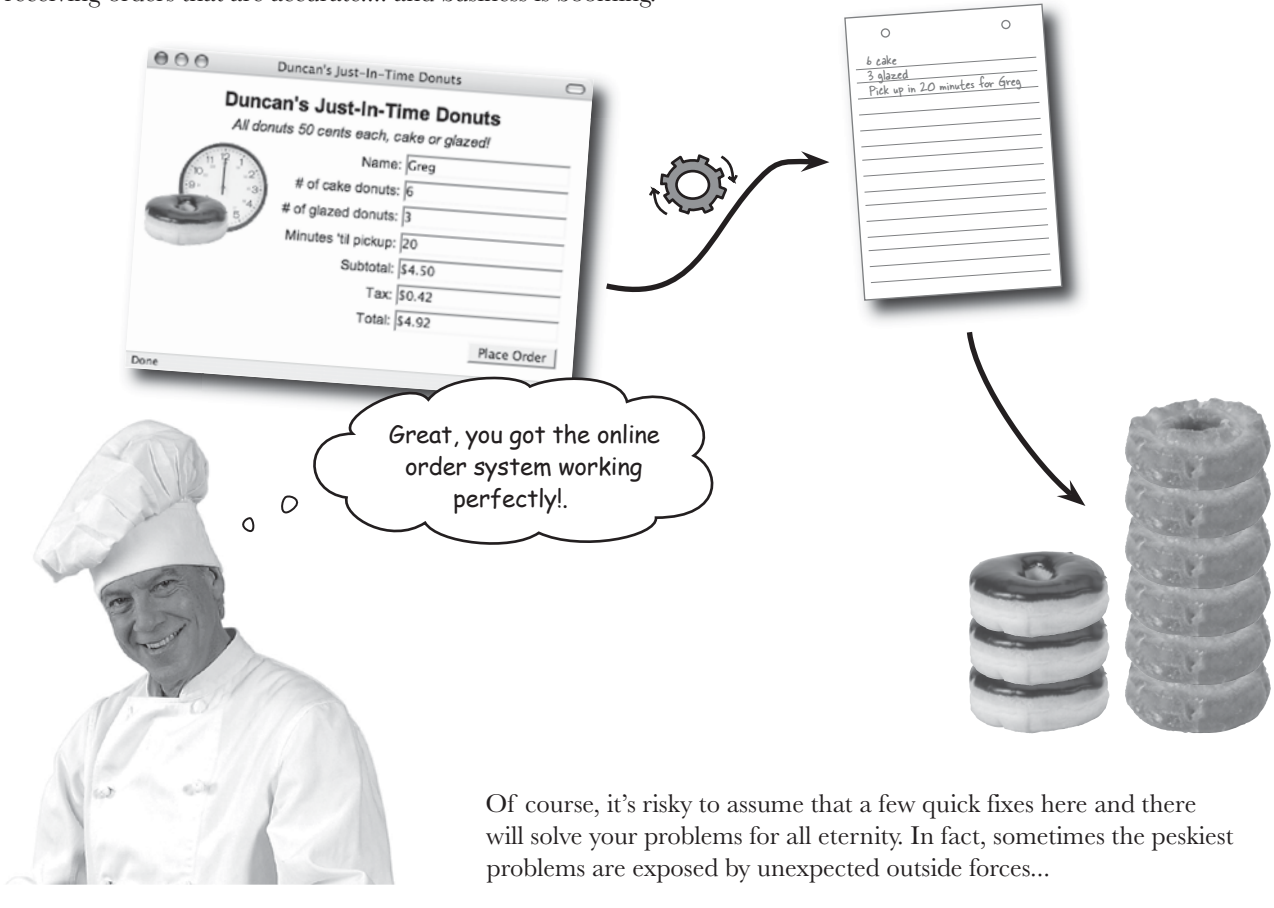
The toFixed() function rounds the dollar values to two decimal places.

BULLET POINTS

- Although not a strict JavaScript requirement, it's a good coding convention to name **constants** in **ALL UPPERCASE** and **variables** in **lowerCamelCase**.
- **Always initialize constants** when you create them, and initialize variables whenever possible.
- When a variable isn't initialized, it remains **undefined** until a value is eventually assigned to it.
- `NaN` stands for **Not a Number**, and is used to indicate that a piece of data is not a number when the expectation is that it should be.
- String concatenation is very different from mathematical addition, even though both use the familiar plus sign (+).
- The built-in `parseInt()` and `parseFloat()` functions are used to **convert strings to numbers**.

You figured out the problem...

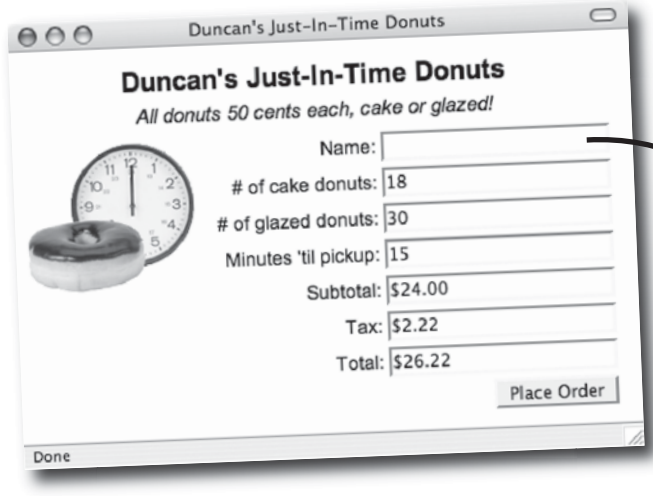
Duncan is thrilled with the JavaScript code fixes you made. He's finally receiving orders that are accurate.... and business is booming.



Of course, it's risky to assume that a few quick fixes here and there will solve your problems for all eternity. In fact, sometimes the peskiest problems are exposed by unexpected outside forces...

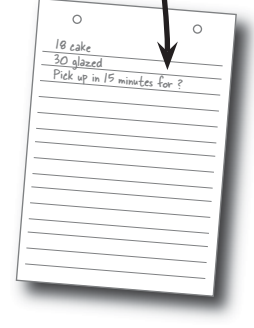
Duncan discovers donut espionage

Duncan's got a new problem: a weasel competitor named Frankie. Frankie runs the hotdog business across the street from Duncan, and is now offering a Breakfast Hound. Problem is, Frankie's playing dirty and submitting bogus donut orders with no names. So now we have orders with no customers—and that's not good.



Even though no name has been entered, the order is still accepted.

I'm not worried about my competitors, I just need to make the donut code smarter about how it accepts data.



Duncan is wasting precious time, energy, and donuts filling bogus orders... and he needs you to make sure all the form data has been entered before allowing an order to go through.

Use getElementById() to grab form data

In order to check the validity of form data, you need a way to grab the data from your Web page. The key to to accessing a web page element with JavaScript is the `id` attribute of the HTML tag:

```
<input type="text" id="cakedonuts" name="cakedonuts" />
```

The id attribute is what you use to access the form field in JavaScript code.

The cake donut quantity HTML input element.

JavaScript allows you to retrieve a web page element with its ID using a function called `getElementById()`. This function doesn't grab an element's data directly, but instead provides you with the HTML field itself, as a JavaScript object. You then access the data through the field's `value` property.

Technically, getElementById() is a method on the document object, and not a function.

document.getElementById()

Give this method the ID of an element on a web page and it gives you back the element itself, which can then be used to access web data

The getElementById() method belongs to the document object.

Don't sweat objects, properties, and methods right now.

JavaScript supports an advanced data type called an object that allows you to do some really cool things. In fact, the JavaScript language itself is really just a bunch of objects. We'll talk a lot more about objects later in the book—for now, just know that a method is a lot like a function, and a property is a lot like a variable.

```
document.getElementById("cakedonuts")
document.getElementById("cakedonuts").value
```

The ID is the key to accessing an element.

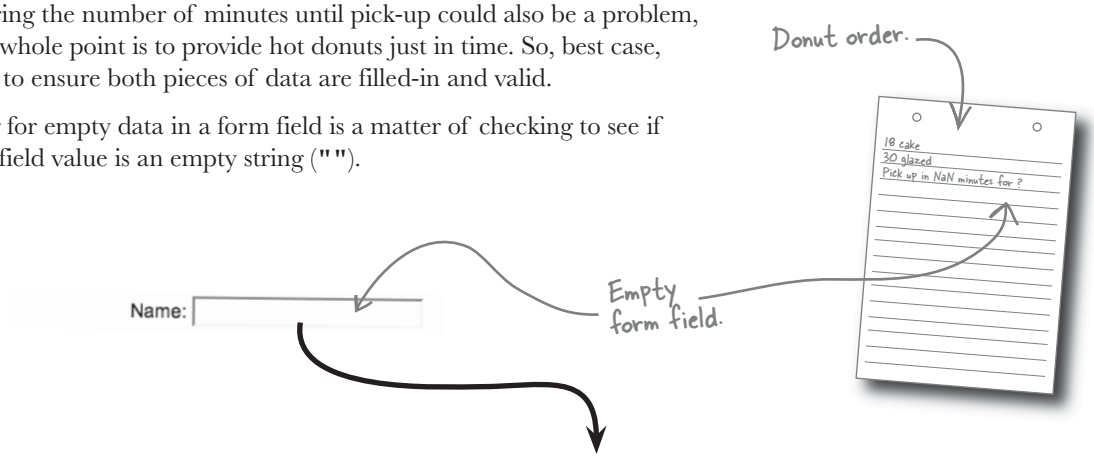
The value property gives you access to the data.

With this code in hand, you're now ready to check Duncan's form data to make sure the fields aren't empty before accepting an order.

Validate the web form's data

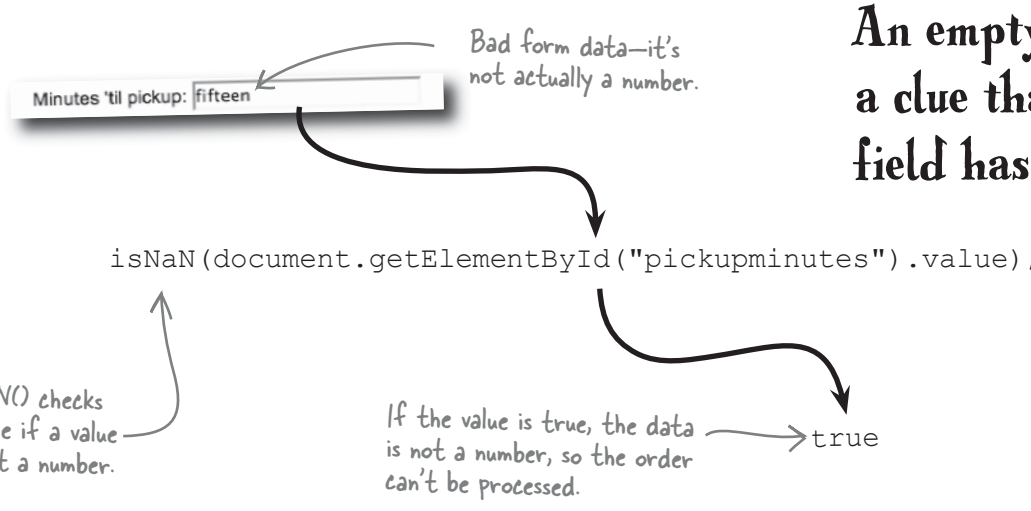
You need to check to make sure a name is entered into the donut form. Not entering the number of minutes until pick-up could also be a problem, since the whole point is to provide hot donuts just in time. So, best case, you want to ensure both pieces of data are filled-in and valid.

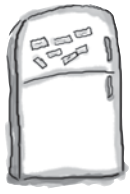
Checking for empty data in a form field is a matter of checking to see if the form field value is an empty string (`""`).



If the name field value is an empty string, then you know the order needs to be halted and the user should get asked to enter their name. The same thing goes for the minutes field, except it's also helpful to go a step further and look to see if the data in that field is a number. The built-in `isNaN()` function is what makes this check possible—you pass it a value and it tells you whether the value is not a number (`true`) or if it is a number (`false`).

An empty string is a clue that a form field has no data.





JavaScript Magnets

The `placeOrder()` function is where the name and pick-up minutes data validation takes place. Use the magnets to finish writing the code that checks for the existence of name and pick-up minutes data, along with making sure that the pick-up minutes entered is a number. You'll need to use each magnet, and some magnets more than once.

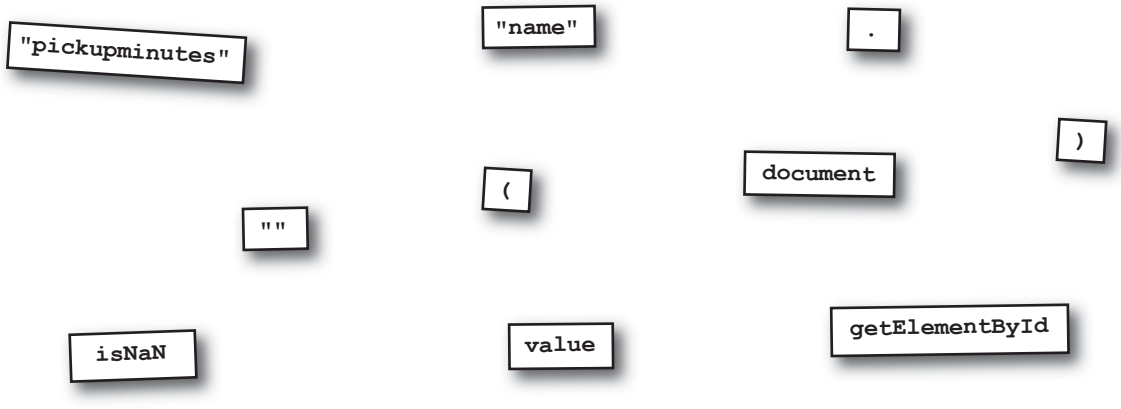
"if" is used to test for a condition and then take action accordingly—if this, then do something.

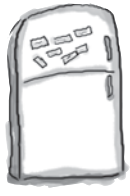
This is an equality test—is one thing equal to another thing?

This means one of two conditions can result in the action—if this OR that, then do something.

```
function placeOrder() {
  if (.....==.....)
    alert("I'm sorry but you must provide your name before submitting an order.");

  else if (.....  .....||
    .....)
    alert("I'm sorry but you must provide the number of minutes until pick-up" +
      " before submitting an order.");
  else
    // Submit the order to the server
    form.submit();
}
```





JavaScript Magnets Solution

The `placeOrder()` function is where the name and pick-up minutes data validation takes place. Use the magnets to finish writing the code that checks for the existence of name and pick-up minutes data, along with making sure that the pick-up minutes entered is a number. All of the magnets are used, and some are used several times.

This says, if the name value is empty, then pop up an alert...else do something different.

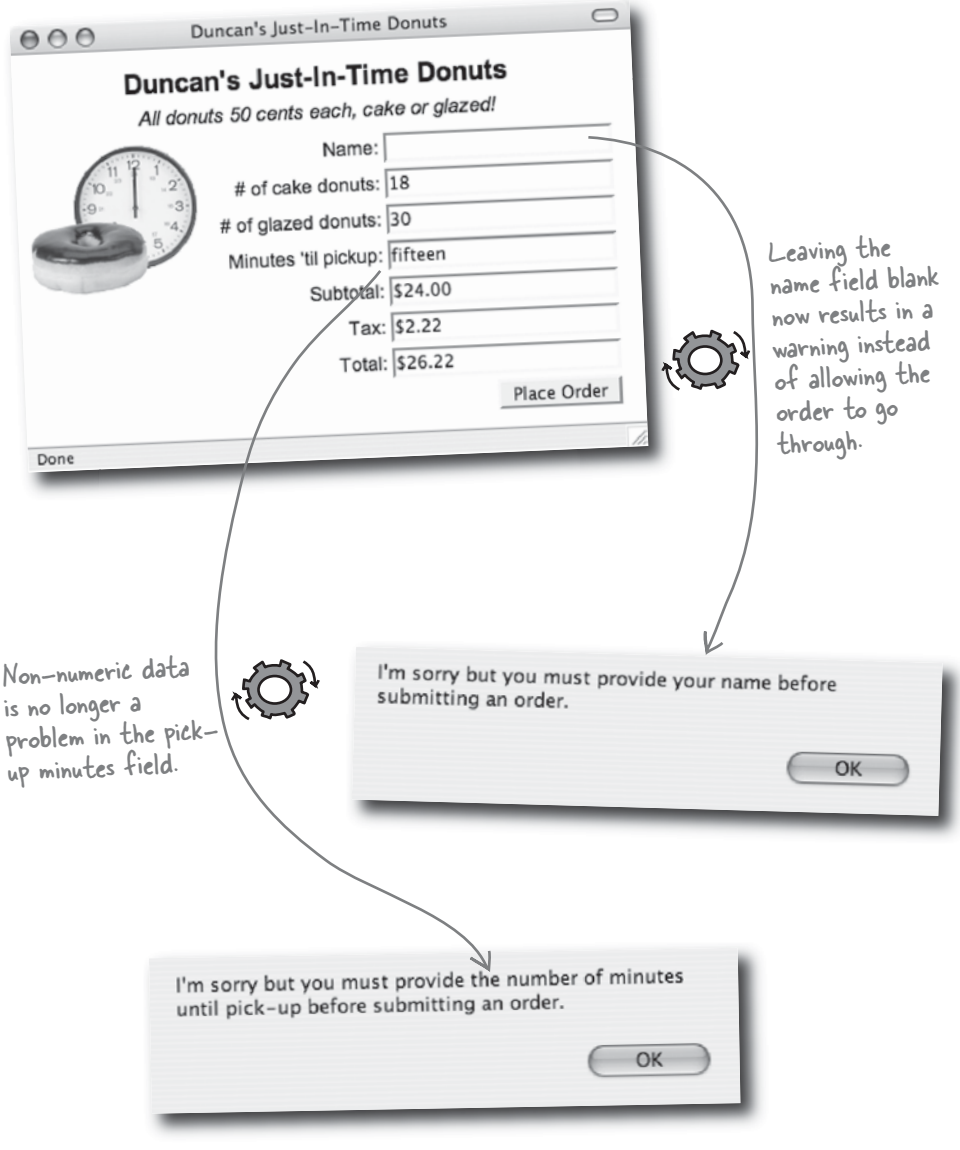
This checks the value of the name field to see if it's equals to "".

Here, we're saying if the value is empty, OR if the value is not a number.

```
function placeOrder() {
  if (document . getElementById ( "name" ) . value == "" )
    alert("I'm sorry but you must provide your name before submitting an order.");
  else if (document . getElementById ( "pickupminutes" ) . value == "" ||
    isNaN ( document . getElementById ( "pickupminutes" ) . value ) )
    alert("I'm sorry but you must provide the number of minutes until pick-up" +
      " before submitting an order.");
  else
    // Submit the order to the server
    form.submit();
}
```

You saved Duncan's Donuts... again!

The new and improved just-in-time donut form with data validation has put an end to Frankie's pastry espionage, and also made the page more robust for real customers. Using JavaScript to protect the integrity of data entered by the user is a win-win, especially in the cutthroat breakfast biz!



there are no Dumb Questions

Q: How does the plus sign (+) know to add or concatenate?

A: Like many things in JavaScript, functionality is determined by context. This means the plus sign takes a look at the two things being "added" and decides whether to numerically add them or concatenate them as text based upon their data types. You already know that "adding" two words means sticking them end-to-end. But problems can occur when you mistakenly assume that you're working with one type of data when it's actually another. That's another reason why it's always a good idea to check to make sure you provide numeric data when you intend numeric addition, and text for text.

Q: What happens when you attempt to add a string to a number?

A: Since number-to-string conversion is automatic in JavaScript, mixing the two data types in an addition always results in a string concatenation. So, the number first gets converted to a string, and then the two strings get concatenated. If you intended to add the two numbers, you need to explicitly convert the string to a number using `parseInt()` or `parseFloat()`.

Q: What happens if you use `parseInt()` to convert a string containing a decimal number?

A: Don't worry, nothing catches on fire. All that happens is that JavaScript assumes you don't care about the fractional part of the number, so it returns only the integer portion of the number.

Q: How does the `id` HTML attribute tie web elements to JavaScript code?

A: Think of the `id` attribute as the portal through which JavaScript code accesses HTML content. When people say JavaScript code runs on a web page, they don't literally mean the web page itself—they mean the browser. In reality, JavaScript code is fairly insulated from HTML code, and can only access it through very specific mechanisms. One of these mechanisms involves the `id` attribute, which lets JavaScript retrieve an HTML element. Tagging a web element with an ID allows the element to be found by JavaScript code, opening up all kinds of scripting possibilities.

Q: That's pretty vague. How specifically does JavaScript code access an HTML element?

A: The `getElementById()` method of the `document` object is the key to accessing an HTML element from JavaScript, and this method uses the `id` attribute of the element to find it on the page. HTML IDs are like JavaScript identifiers in that they should be unique within a given page. Otherwise, the `getElementById()` method would have a tough time knowing what web element to return.

Q: I know you said we'll talk more about them in Chapter 9, but objects have already come up a few times. What are they?

A: We're jumping ahead a little here, so don't tell anyone. Objects are an advanced JavaScript data type that can combine functions, constants, and variables into one logical entity. A method is just a function that is part of an object, while a property is a variable or constant in an object. On a practical level, JavaScript uses objects to represent just about everything—the browser window is an object, as is the web page document. That's why the `getElementById()` method must be called through the `document` object—it's a part of the object, which represents the entire web page. OK, back to Chapter 2...

Q: I still don't understand the difference between a web page element and its value. What gives?

A: Web page elements are exposed to JavaScript as objects, which means they have properties and methods you can use to manipulate them. One of these properties is `value`, which holds the value stored in the element. As an example, the *value* of a form field is the data entered into the field.

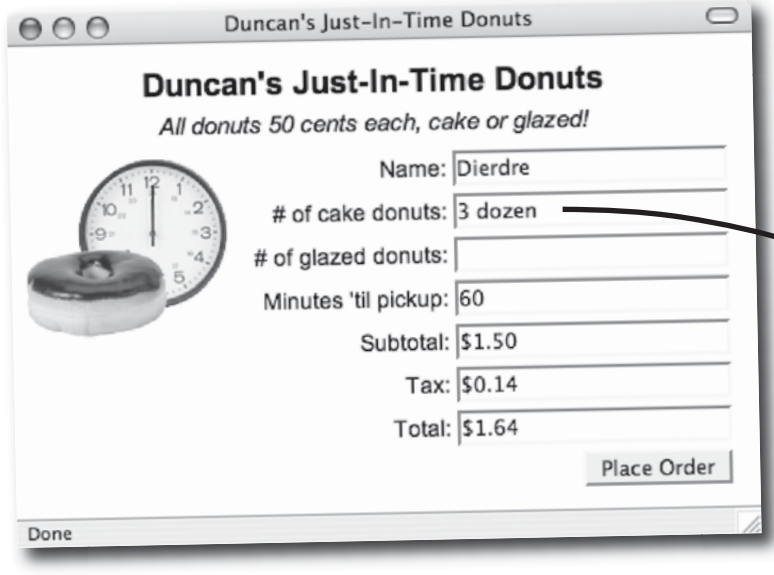
Q: Why is it necessary to know if a value is *not* a number? Wouldn't it make more sense to see if it *is* a number?

A: Good question. What it boils down to is why you care about a value being a number or not. In most cases the assumption is that you're dealing with a number, so it makes sense to check for the exception (the unexpected). By checking for `NaN`, you're able to make number-handling script code more robust, and hopefully alleviate a weird computation involving a non-number.

Strive for intuitive user input

Now that Duncan is no longer putting out fires, he really wants to improve the user experience of the just-in-time donut form. Just as the "hot donuts" sign is **intuitive** to people passing by his storefront, he wants the online form to be similarly intuitive. Duncan knows that donuts are typically ordered and served in dozens. Very few people order 12 or 24 donuts—they order 1 or 2 dozen donuts. He thinks the donut form should allow users to enter data in the most natural way possible.

Problem is, the current script doesn't take into account the user entering the word "dozen" when specifying the quantity of donuts.



"3 dozen" donuts gets converted into the number 3 thanks to the parseInt() function.

```
parseInt("3 dozen")
```

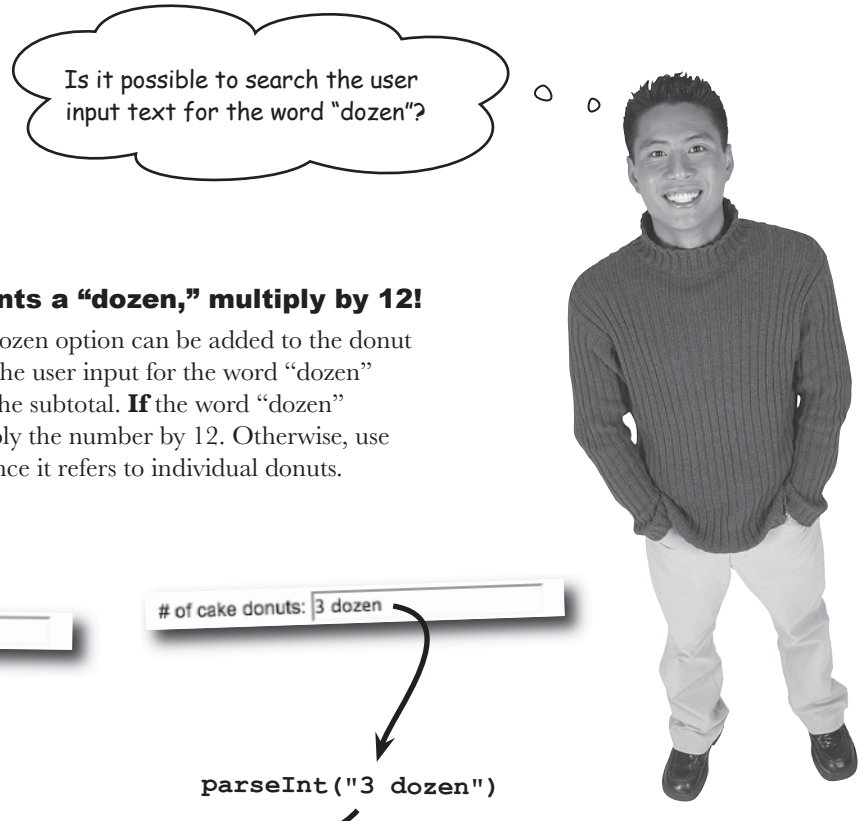
3

This is a number, not a string.

The script doesn't complain when the user enters the word "dozen" alongside a number... the `parseInt()` function ignores any text present after a number in a string. So, the word "dozen" is just discarded, and all that's kept is the number.

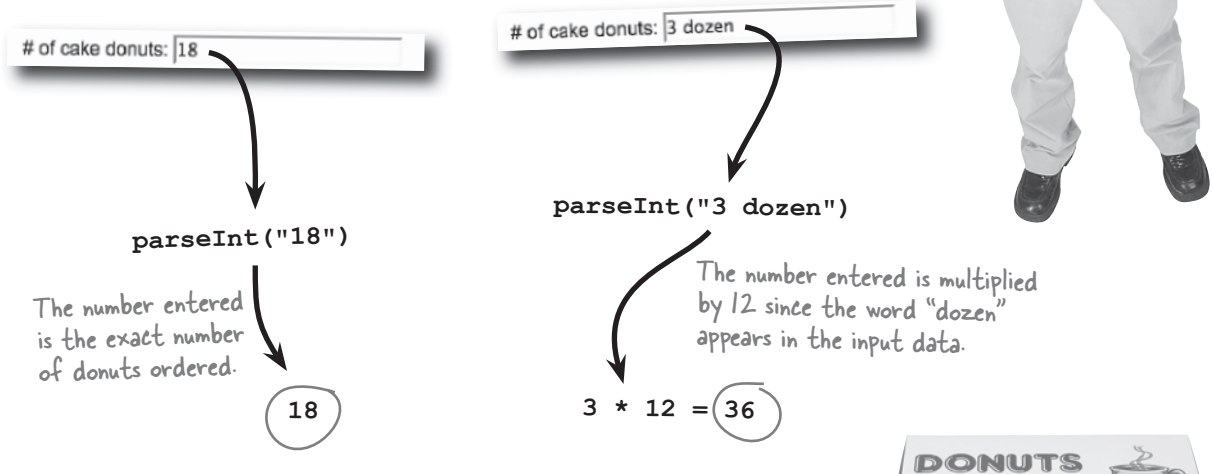
BRAIN POWER

Is it possible for the donut script to allow users to enter either a number or a number **and** the word "dozen" for ordering by the dozen? How?



If the user wants a "dozen," multiply by 12!

The order-by-the-dozen option can be added to the donut script by checking the user input for the word "dozen" before calculating the subtotal. **If** the word "dozen" appears, just multiply the number by 12. Otherwise, use the number as-is since it refers to individual donuts.



Ready Bake JavaScript

The custom `parseDonuts()` function is responsible for processing donut quantity input data. It first converts the data to a number, and then checks for the appearance of the word "dozen" in the input data. If "dozen" appears, the number of donuts is multiplied by 12. Get this recipe at *http://www.headfirstlabs.com/books/hfjs/*.

```
function parseDonuts(donutString) {
  numDonuts = parseInt(donutString);
  if (donutString.indexOf("dozen") != -1)
    numDonuts *= 12;
  return numDonuts;
}
```

Check to see if the word "dozen" appears in the input data.

Multiply the number of donuts by 12.

Parsing dozens of donuts

The `parseDonuts()` function is called in the `updateOrder()` function, which is when the subtotal and total are calculated from the user-entered data.

Initialize the two constants.

Get the number of donuts from the form field.

```
function updateOrder() {
  const TAXRATE = 0.0925;
  const DONUTPRICE = 0.50;
  var numCakeDonuts = parseDonuts(document.getElementById("cakedonuts").value);
  var numGlazedDonuts = parseDonuts(document.getElementById("glazeddonuts").value);
  if (isNaN(numCakeDonuts))
    numCakeDonuts = 0;
  if (isNaN(numGlazedDonuts))
    numGlazedDonuts = 0;
  var subTotal = (numCakeDonuts + numGlazedDonuts) * DONUTPRICE;
  var tax = subTotal * TAXRATE;
  var total = subTotal + tax;
  document.getElementById("subtotal").value = "$" + subTotal.toFixed(2);
  document.getElementById("tax").value = "$" + tax.toFixed(2);
  document.getElementById("total").value = "$" + total.toFixed(2);
}
```

If the number of donuts entered is not a number, set them to 0.

Calculate the subtotal, tax, and total.

Show the dollar amounts on the page.

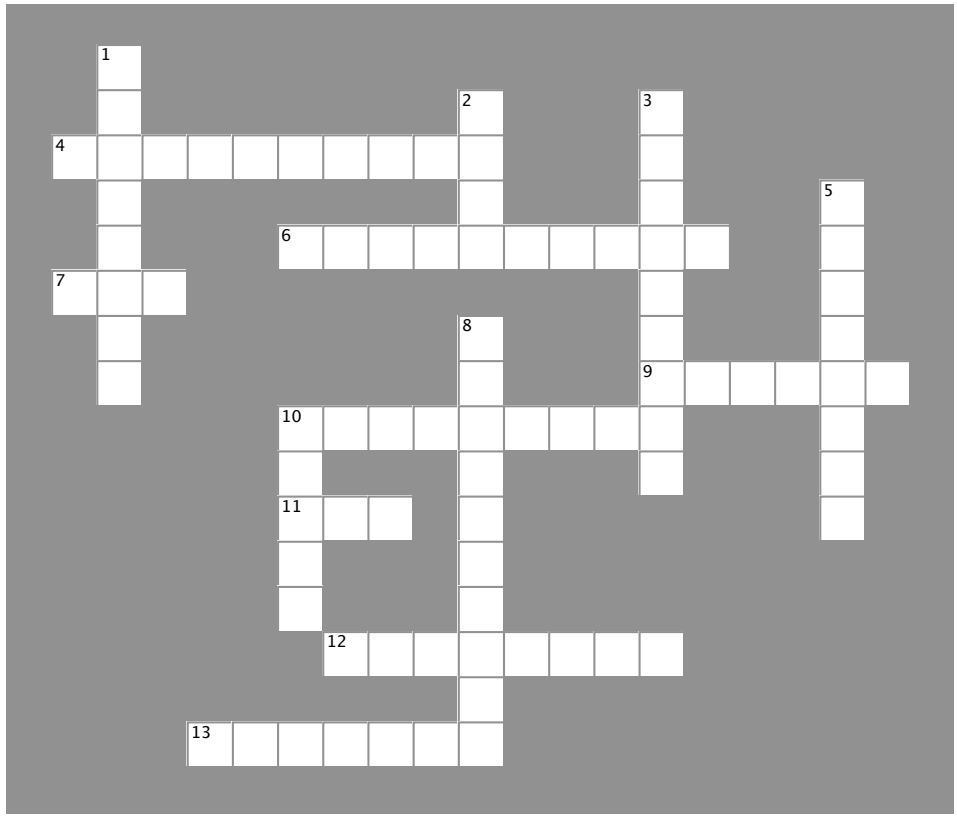
Round the dollar amounts to two decimal places (cents).

Just-in-time donuts a smashing success!

Life is good now that Duncan and his just-in-time hot donut idea has been fully realized in a JavaScript-powered page that carefully validates orders entered by the user.

JavaScriptcross

Data isn't always stored in JavaScript code. Sometimes it gets stored in the rows and columns of a crossword puzzle, where it waits patiently for you to uncover it.



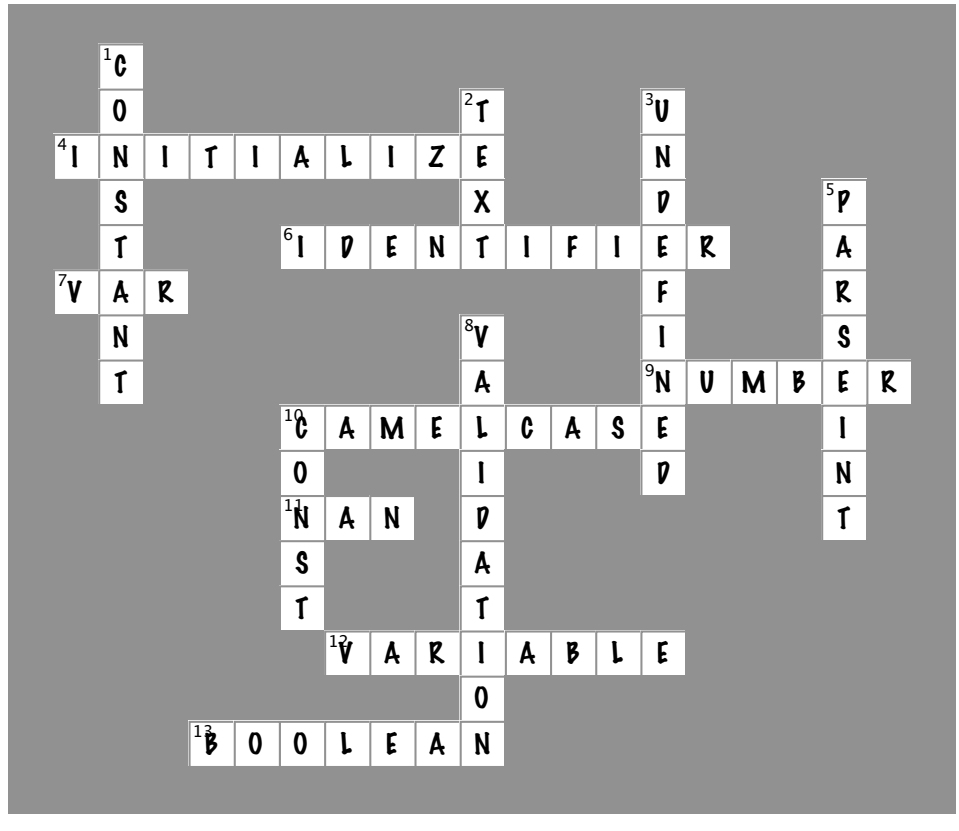
Across

4. When you set the value of a piece of data upon creating it, you ..... it.
6. The unique name used to reference a piece of data.
7. The JavaScript keyword used to create a variable.
9. 3.14, 11, and 5280 are all this data type.
10. A coding convention that involves naming identifiers with mixed case, as in ThisIsMyName.
11. It's not a num-bah.
12. A piece of information whose value can change.
13. An piece of data with an on/off value would be stored as this data type.

Down

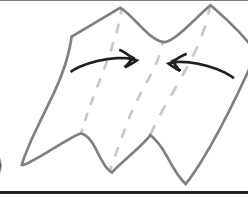
1. A piece of data whose value cannot change.
2. The data type used to store characters, words, and phrases.
3. When a value isn't set for a variable or constant, the data is considered .....
5. The built-in JavaScript function used to convert a string to an integer.
8. The process of checking to make sure user-entered data is accurate is called .....
10. The JavaScript keyword used to create a constant.

JavaScriptcross Solution



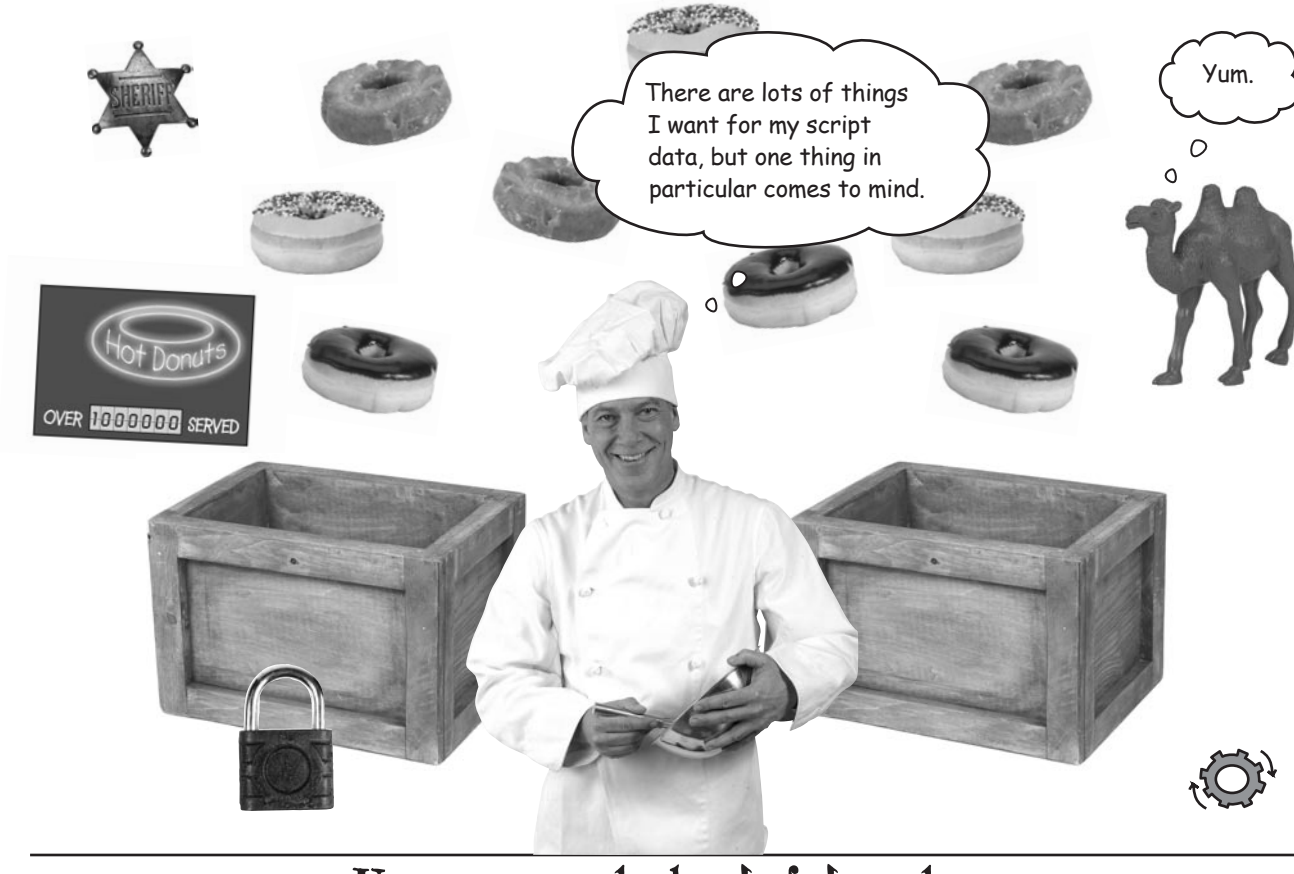
Page Bender

Fold the page vertically to line up the two brains and solve the riddle.



What do we all want for our script data?

It's a meeting of the minds!



User input is the kind of data that you shouldn't trust. It's just not safe to assume that users will enter data and check to make sure it is OK. A more secure storage solution involves using JavaScript.