*James Greig*

# 100 Mile Bike Courier

HACKERMONTHLY Issue 32 January 2013

# stripe

Accept payments online.

*Now you can hack on DuckDuckGo*

# DuckDuckHack

*Create instant answer plugins for DuckDuckGo*

**Curator**
Lim Cheng Soon

**Contributors**
James Greig
Joel Runyon
Gabriel Weinberg
Barry Steyn
Luc Gommans
Greg Lehey
Kalid Azad
Carlos Bueno
Dan Ariely
Alex MacCaw

**Proofreaders**
Emily Griffin
Sigmarie Soto

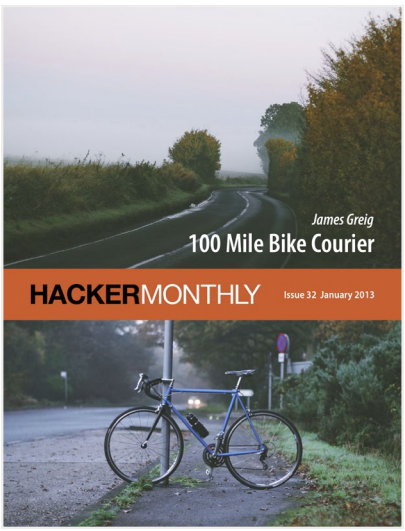
**Printer**
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

**Advertising**
ads@hackermonthly.com

**Contact**
contact@hackermonthly.com

**Published by**
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.




Cover Photographs by James Greig

# Contents



For links to Hacker News dicussions, visit *hackermonthly.com/issue-32*

# 100 Mile Bike Courier

*Why I cycled a hundred miles to meet my first customer*

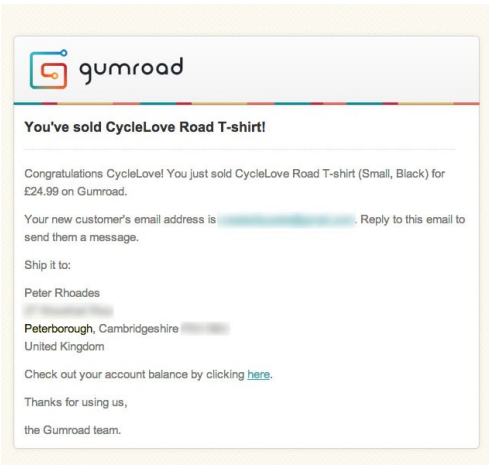
*By* JAMES GREIG

When I started selling CycleLove t-shirts, I made a quiet promise to myself: that I would deliver my first order in person, and by bike.

It seemed like a simple way of celebrating this small but important milestone in my new venture.

CycleLove is the first thing in a long time that I really care about; something I quit my job to do. And so making the effort to meet my first customer seemed like the very least I could do.

As I'm based in London, I figure this as-yet-nameless-customer won't be too far away, so you can imagine my mixed emotions when this email lands in my inbox a few weeks later:



gumroad

**You've sold CycleLove Road T-shirt!**

Congratulations CycleLove! You just sold CycleLove Road T-shirt (Small, Black) for £24.99 on Gumroad.

Your new customer's email address is [redacted]. Reply to this email to send them a message.

Ship it to:

Peter Rhoades
[redacted]
Peterborough, Cambridgeshire [redacted]
United Kingdom

Check out your account balance by clicking here.

Thanks for using us,

the Gumroad team.

Plugging the Peterborough postcode into Google Maps quickly confirms my hunch: to keep my promise, I must cycle over a hundred miles to deliver a t-shirt.

For a few days I chew the idea over in my head. I've completed the 120-odd miles of the Dunwich Dynamo overnight and raced 80 miles in Scotland on the Etape Caledonia. How hard could a hundred-odd miles on my own be? (First mistake: cycling long distances alone plays tricks on the mind).

Still unsure if I have the required huevos to do the ride, I email Peter to explain it may be a while before the t-shirt is delivered.

He fires back a quick response:

*"I've purchased this tee for my brother's birthday (in November) so there's a bit of time to play with ;)"*

This is the sucker punch. The t-shirt is a birthday present *and* I've got two months to deliver it. I resolve to stick to my plan and deliver it by bike, whatever it takes.

## Best-laid plans (or not)

I set a few ground rules for myself:

1. No training — this is not to be a race.
2. No visible lycra — I am not, and do not intend to become, a mamil.
3. No fancy gear, cleats or hi-vis — I will make the journey as a human on a bike, not a "cyclist."

The first half of the route I already know well from my training for the Etape Caledonia — the quiet country roads and softly rolling hills between Epping Forest and Cambridge. The second half between Cambridge and Peterborough is unknown territory, and I decide to keep it that way. (Second mistake: always know your route).

Now it's a matter of waiting for good weather. October passes. I am ill on and off. November rolls in, temperatures drop, and I start to regret waiting so long. Had I left it for too late? Would anyone join me on the ride?

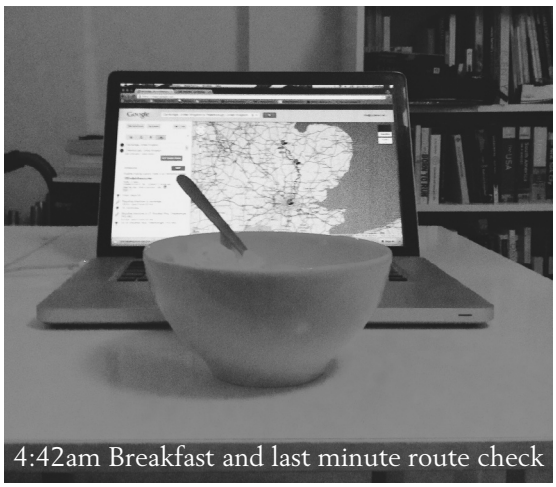
With time running out, I spot a full day of sunshine in the forecast, and the date is set. Sunday 11th November is the day I will become a one-hundred mile bike courier.

I arrange with Pete to meet him and his brother Rob in a pub that Sunday afternoon, pack as light a bag as possible with an SLR camera, and set my alarm for 4.30am.

It's a cold, dark morning, but I'm excited to start my journey after weeks of planning.

Sunday at 5am marks an overlap of worlds. Revelers are heading home whilst workers head out. A young couple is scaling a fence into Brockwell Park, whilst somber figures wait alone at bus stops. Traffic on London Bridge is reduced to less than a trickle. Drunken kids hang like monkeys from scaffolding in Leyton, howling with laughter.

Giro/Vulpine/Specialized/Brixton Cycles/ Canon/Knog/Her Majesty's Treasury



4:42am Breakfast and last minute route check

5:35am London Bridge

6:56am Epping Forest

After ten miles or so, I'm sweating and stop to remove a layer. (Third mistake: cities generate heat. The countryside doesn't, as I will soon remember).

As the light imperceptibly increases, I finally break out of Epping Forest into open countryside, and the temperature plummets towards freezing. I feel obliged to make use of the light by taking photos, even though it means removing both sets of gloves.

By this point, I'm feeling the cold however fast I ride, and I stop to put everything I've got in my bag back on.

I'm also cursing my decision not to wear overshoes. (Fourth mistake: extremities suffer first). Luckily I have some spare socks, intended for the journey home, and put them on over my long wooly socks. A mouth full of chocolate and a sausage roll is no comfort to my toes but settles a now-hungry stomach.

Switching my camera-phone lens around, a blue face peers back at me from the screen. I wish I had the Michelin Man's insulation to keep me warm.

7:09am Winding roads and freezing fog

8:29am Cold feet

I may be riding alone, but I know that a few people are following my progress on Twitter, and I realize they can offer the support I need. I'm not sure if my phone battery will last the distance if I keep refreshing my feed, but it's worth the risk. Sure enough, Twitter offers me some badly needed words of encouragement:

*Oh. How are your feet doing then? :) How much longer have you got?*
*— Discerning Cyclist (@discerningcyclist)*

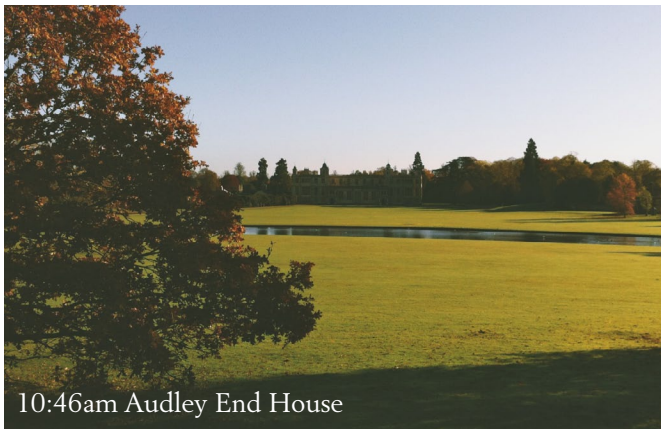
*Keep going. Hope you warm up and have a great ride.*
*— Belinda Scott (@Condorbee)*

Belinda is right. The only way to get warm is to speed up.

I know this part of England well from training rides, and familiar place names are now flashing by: Roydon. Much Hadham. Little Hadham. Patmore Heath.

A handful of lycra-clad cyclists are already out for their Sunday spin, and I bob my head in acknowledgement, not really caring if they respond or not.

10:22am Newport


10:46am Audley End House


11:35am Shouldn't be here

Two hours later, I roll over the M11 into Newport, twenty miles south of Cambridge, and can't help but think: maybe this isn't going to be so tough after all? (Fifth mistake: it's not over until it's over, James).

After a short stop in Newport, I head out north from the village.

Suddenly cars are buzzing by uncomfortably close. The road surface is rough like three-day stubble, and my speed drops.

Confused I pull into the first layby to check my location. I'm on the fucking A1301! (Sixth mistake: avoid fast roads at all cost).

For the first time on the ride, I'm angry. What the hell am I doing? Why didn't I plan my route better? Why didn't I do some training?

Of course it's too late to be asking these kinds of questions, so I ease gently back onto my bike, and force myself to press on.

11:47am Where does this go, iOS6?

12:38pm Lunch, Cambridge

13:53pm Chance encounter with a recumbent

Fortunately, signs of civilization have started to appear in the form of cycle paths. They're not marked on my map (thanks, Apple), but I'm sure they must be headed to my checkpoint of Cambridge. An hour later I'm there, tucking into what tastes like the best fish and chips in the world.

68 miles completed. 38 to go.

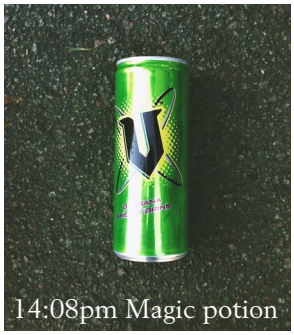
With a full belly, I start off slowly on the second leg of the journey. I'm under no pretentions that it's going to be easy — especially considering I have no idea how flat or hilly the road ahead is.

Each town and village on the route now becomes a target.

I memorize a few at a time and chant them to myself on repeat, like a mantra, before stopping to learn a new batch of place names.

I'm taking another break for food when, out of nowhere, a recumbent bike appears, and I find myself smiling again. Looking at the photo now at full size, its owner seems to be wearing green wellington boots. Chapeau!

It's now several hours since lunch, and my energy levels are flagging. Even the smallest of hills seem to grind on relentlessly.



14:08pm Magic potion


The B1040

Thankfully I have the cure for the problem and am soon tripping on an awesome wave of sugar and caffeine. Time is running out though — I need to get to my destination before the hit wears off — and as if to heighten the sense of urgency, the sun is dipping towards the horizon.

At this point I get lucky, hitting the smoothest, straightest and flattest stretch of road I've ever seen…

If you're reading this B1040, I love you. Let's do it again sometime soon.

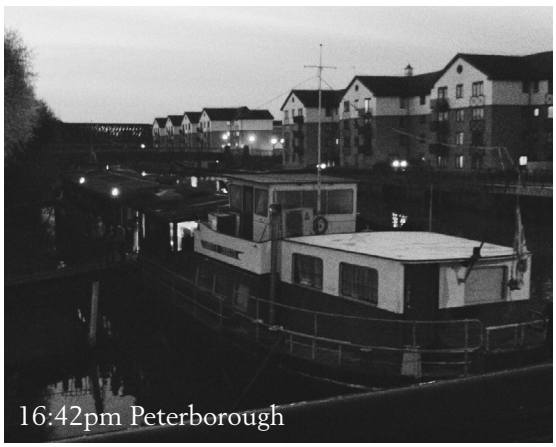
With fifteen or so miles covered at top speed, I'm feeling more positive. The first road sign for Peterborough appears, and there's no doubt in my mind that I'm going to make it.

I allow myself the luxury of stopping to get out my camera and document the last rays of sun for the day.

As the light fades, I hit the outskirts of Peterborough, knowing that Pete and his brother Rob will already be at our rendezvous point, a floating bar moored on the River Nene in the center of town.

By the time I arrive there, the novelty of the location escapes me — all I can think about is getting my lips around a cold beer.

Sitting down with the pint that Pete has just bought me, I grab the packaged CycleLove t-shirt out of my bag. I search for the best words to explain to his brother who this sweaty and exhausted-looking man sitting across from him at the table is.



16:42pm Peterborough

16:57pm My first customer: Rob

15:52pm Twilight

I turn to Rob and find myself saying…

"Well, I've got this blog about cycling… and…" ■

*Distance ridden: 105 miles*
*Total time: 11.5 hours*
*Resting time: 2.5 hours approx.*
*Photos taken: 112*
*Ales drunk: 2*
*Happy customers: 1*

---

James Greig trained as a graphic designer at Glasgow School of Art in Scotland before returning to London in 2009 to pursue his career as a graphic designer. In the spring of 2012 he quit his job to travel across America, and began working (almost) full-time on CycleLove.

Reprinted with permission of the original author.
First appeared in *hn.my/cycle* (cyclelove.com)

# An Unexpected Ass Kicking

*By* JOEL RUNYON

I SAT DOWN AT yet another coffee shop in Portland determined to get some work done, catch up on some emails and write another blog post.

About 30 minutes into my working, an elderly gentleman at least 80 years old sat down next to me with a hot coffee and a pastry. I smiled at him and nodded and looked back at my computer as I continued to work.

"Do you like Apple?" he asked as he gestured to the new Macbook Air I had picked up a few days prior.

"Yea, I've been using them for a while," I said, wondering if I was going to get suckered into a Mac vs. PC debate in a Portland coffee shop with an elderly stranger.

"Do you program on them?"

"Well, I don't really know how to code, but I write quite a bit and spend a lot of time creating online projects and helping clients run their businesses," I replied.

"I've been against Macintosh company lately. They're trying to get everyone to use iPads and when people use iPads they end up just using technology to consume things instead of making things. With a computer you can make things. You can code, you can make things and create things that have never before existed and do things that have never been done before."

"That's the problem with a lot of people," he continued, "They don't try to do stuff that's never been done before, so they never do anything**, but if they try to do it, they find out there's lots of things they can do that have never been done before.**"

I nodded my head in agreement and laughed to myself — mostly because that would be something that I would say and because of the coincidence that out of all the people in the coffee shop I ended up talking to, it was this guy. What a way to open a conversation.

The old man turned back to his coffee, took a sip and then looked back at me.

## “Nothing is withheld from us which we have conceived to do.”

“In fact, I’ve done lots of things that haven’t been done before,” he said half-smiling.

Not sure if he was simply toying with me or not, I let my curiosity get the better of me.

“Oh, really? Like what types of things?” I asked, all the while half-thinking he was going to make up something fairly non-impressive.

“I invented the first computer,” he said.

Um, Excuse me?

“I created the world’s first internally programmable computer. It used to take up a space about as big as this whole room and my wife and I used to walk into it to program it.”

“What’s your name?” I asked, thinking that this guy was either another crazy homeless person in Portland or legitimately who he said he was.

“Russell Kirsch,” he answered.

Sure enough, after .29 seconds, I found out he wasn’t lying to my face. Russell Kirsch indeed invented the world’s first internally programmable computer [hn.my/russell] as well as a bunch of other things, and he definitely lives in Portland. As he talked, I began googling him.

He read my mind and volunteered, “Here, I’ll show you.”

He stood up and directed me to a variety of websites and showed me through the archives of what he’d created while every once in a while dropping some minor detail like, “I also created the first digital image. It was a photo of my son.”

At this point, I learned better than to call Russell’s bluff, but sure enough, a few more Google searches [hn.my/newborn] showed that he did just what he said he did.

As he showed me through the old history archives of what he did (and while any hope of productivity vacated my mind), I listened to his stories and picked his brain.

At some point in the conversation, I said to him, "You know Russell, that's really impressive."

"I guess. I've always believed that nothing is withheld from us which we have conceived to do. Most people think the opposite — that all things are withheld from them which they have conceived to do and they end up doing nothing."

"Wait," I said, pausing at his last sentence, "What was that quote again?"

"Nothing is withheld from us which we have conceived to do."

"That's good. Who said that?"

"God did," he said.

"What?"

"God said it and there were only two people who believed it. You know who?"

"Nope. Who?" I asked.

"God and me, so I went out and did it," he replied.

Want to mess with your mind? Without the man in the photo, the photo of this man wouldn't exist.

Well then, I thought as he finished showing me the archives, I'm not going to argue with the guy who invented the computer. After about 20 minutes of walking me through his contributions to technology, he sat down, finished his coffee, glanced at his half-eaten pastry now-cold, checked his watch and announced, "Well, I have to go now."

With that, we shook hands, he got up, walked to his car and drove off as I just sat there trying to figure out what exactly had just happened. As I sat there thinking, two things he said reverberated in the back of my mind:

- Nothing is withheld from us which we have conceived to do.
- Do things that have never been done.

The first meaning: If you have conceived something in your mind, have decided to do it and are willing to put in the work, then nothing can stop you.

The second is fairly self-explanatory but carries the extra weight of coming from the guy who invented the very thing that's letting me type these words out on the internet. ■

---

Joel Runyon is the author of the Blog of Impossible Things & Impossible HQ, where he writes about pushing your limits and doing the impossible through physical challenges, adventure and service.

Reprinted with permission of the original author. First appeared in *hn.my/kick* (joelrunyon.com)

# Traction Mistakes

*By* GABRIEL WEINBERG

Most startups don't fail at building a product. They fail at acquiring customers.

The biggest mistakes I see over and over again when startups try to get traction are as follows (in order of importance).

### 1 They don't pursue traction in parallel with product development.

The benefits of parallel customer acquisition cannot be understated. First and foremost, you can use initial customer development to inform your product roadmap and literally prevent yourself from a) building something people don't really want, and b) building something people want but not enough to form a business around it. Second, you can launch with a nice base of initial users. Third, you're prepared to scale to the next step because you've been testing messaging and distribution channels since the beginning and thus have a great idea of where to focus post-launch.

### 2 They didn't spend enough time pursuing traction.

I believe distribution is equally important as product. That means quite literally you should be spending 50% of your time on it. For tech people, you should probably bias it to 75% so you end up getting to equal in the end.

At 50%, it competes with being at the top of your mind, which means you can actually make real headway and be creative. Otherwise, it becomes an afterthought and progress drops off exponentially.

Note I'm not a fan of doing this 50/50 split between a tech and non-tech co-founder. You will maximize success probabilities if each co-founder does both. The other problem with the split is the non-tech guy ends up picking up all sorts of other stuff (QA, paperwork, etc.) and 50% starts to become 25%.

### 3 They were biased towards or against certain traction verticals.

There are many verticals that startups have used to get traction. Usually in a given growth stage, one ends up mattering the most, but which one is a bit unpredictable.

The biggest bias here is availability bias. Startups generally just don't think of things like billboards and infomercials because they're out of their vision. Another large bias is a negative bias towards things people find icky, e.g. sales, affiliates — but they don't have to be icky at all. A third bias is the general bias against schlep — business development is in this category for sure.

The point is that you should consider all traction verticals in the pursuit of traction. I'm not saying actually act on all of them at all, but at least consider them.

### 4 They didn't take a systematic approach to getting traction.

People have established processes for product development, but less so for distribution. The usual approach is to build the product, then frantically try to figure out how startups promote things, then haphazardly attempt various obvious things in serial (try to get press coverage, buy some Adwords, Facebook ads, etc.).

You can do better, however. What I like to see is an educated guess at a few traction verticals that are likely to work based on product type, market approach and stage of company. List them all out in order of potential usefulness.

Then approach the most promising verticals (say five) with small but effective tests using something like quant based marketing (i.e. with numbers). If one or two out of the initial five seem promising, focus hard on them. If they turn out not to work, then back up and pick the next set of verticals.

### 5 They didn't take advantage of micro-opportunities.

Startup micro-opportunities are little moments that pop up every now and again that can give you a nice blip in traction if you move fast on them. Two common examples are responding to stuff in the press or memes in an interesting way and trying out new tactics within traction verticals that appear (e.g. Pinterest ads if they come out with them).

To take advantage of these opportunities you have to be watching, flexible and creative. What this means in practice is that you generally need #2, i.e. to be spending enough time on pursuing traction to recognize one when you see it.

For what it's worth, I've made all these mistakes myself. My first company was a disaster in this regard. My second one swung like a pendulum in the other direction and I spent too much time getting traction and not enough on product development to build a long-term sustainable company. At DuckDuckGo I've tried to avoid these mistakes as best I can. ■

---

Gabriel Weinberg is the founder of DuckDuckGo, a search engine. He is also an active angel investor, based out of Valley Forge, PA.

Reprinted with permission of the original author.
First appeared in *hn.my/mistrack* (gabrielweinberg.com)

# JavaScript: Function Invocation Patterns

*By* BARRY STEYN

JavaScript has been described as a Functional Oriented Language (this as opposed to Object Oriented Language). The reason is because functions in JavaScript do more than just separate logic into execution units; functions are first class citizens that also provide scope and the ability to create objects. Having such a heavy reliance upon functions is both a blessing and a curse: It's a blessing because it makes the language lightweight and fast (the main goal of its original development), but it is a curse because you can very easily shoot yourself in the foot if you don't know what you are doing.

One concern with JavaScript functions is how different invocation patterns can produce vastly different results. This article explains the four patterns, how to use them and what to watch out for. The four invocation patterns are:

1. Method Invocation
2. Function Invocation
3. Constructor Invocation
4. Apply And Call Invocation

## Function Execution

JavaScript (like all languages these days) has the ability to modularize logic in `functions` which can be invoked at any point within the execution. Invoking a function suspends execution of the current function, passing controls and parameters to the invoked function. In addition, a parameter called `this` is also passed to the function. The invocation operator is a pair of round brackets `()`, that can contain zero or more expressions separated by a comma.

Unfortunately, there is more than one pattern that can be used to invoke functions. These patterns are not *nice-to-know*: They are absolutely essential

to know. This is because invoking a function with a different pattern can produce a vastly different result. I believe that this is a language design error in JavaScript, and had the language been designed with more thought (and less haste), this would not have been such a big issue.

## The Four Invocation Patterns

Even though there is only one invocation operator `()`, there are four invocation patterns. Each pattern differs in how the `this` parameter is initialized.

### Method Invocation

When a function is part of an object, it is called a method. Method invocation is the pattern of invoking a function that is part of an object. For example:

```
var obj = {
    value: 0,
    increment: function() {
        this.value+=1;
    }
};
obj.increment();
//Method invocation
```

Method invocation is identified when a function is preceded by `object.`, where `object` is the name of some object. JavaScript will set the `this` parameter to the object where the method was invoked on. In the example above, `this` would be set to `obj`. JavaScript binds `this` at execution (also known as late binding).

### Function Invocation

Function invocation is performed by invoking a function using `()`:

```
add(2,3); //5
```

When using the function invocation pattern, `this` is set to the global object. This was a mistake in the JavaScript language! Blindly binding `this` to the global object can destroy its current context. It is noticeable when using an inner function within a method function. An example should explain things better:

```
var value = 500; //Global variable
var obj = {
  value: 0,
  increment: function() {
    this.value++;

    var innerFunction = function()
      {
        alert(this.value);
      }
    innerFunction();
    //Function invocation pattern
  }
}
obj.increment();
//Method invocation pattern
```

What do you think will be printed to screen? For those that answered 1, you are wrong (but don't be too hard on yourselves, this is because JavaScript does not do things very well). The real answer is 500. Note that `innerFunction` is called using the function

invocation pattern, therefore this is set to the global object. The result is that innerFunction (again, it is important to note that it is invoked with function pattern) will not have this set to current object. Instead, it is set to the global object, where value is defined as 500. I stress that this is bad language design; the increment function was invoked with the method invocation pattern, and so it is natural to assume the this should always point to the current function when used inside it.

There is an easy way to get round this problem, but it is in my opinion a hack. One gets around this problem by assigning a variable (by convention, it is named that) to this inside the function (aside: This works because functions in JavaScript are closures):

```
var value = 500; //Global variable
var obj = {
  value: 0,
  increment: function() {
    var that = this;
    that.value++;
    var innerFunction = function()
      {
        alert(that.value);
      }
    innerFunction();
    //Function invocation pattern
  }
}
obj.increment();
```

If this could be bound to the current object whose scope it is called in, function and method invocations would be identical.

## Constructor Invocation

Warning: This is another JavaScript peculiarity! JavaScript is not a classical object oriented language. Instead, it is a prototypical object oriented language, but the creators of JavaScript felt that people with classical object orientation experience may be unhappy with a purely prototype approach. This resulted in JavaScript being unsure of its prototypical nature and the worst thing happened: it mixed classical object orientation syntax with its prototypical nature. The result: a mess!

In classical object orientation, an object is an instantiation of a class. In C++ and Java, this instantiation is performed by using the new operator. This seems to be the inspiration behind the constructor invocation pattern....

The constructor invocation pattern involves putting the new operator just before the function is invoked. For example:

```
var Cheese = function(type) {
    var cheeseType = type;
    return cheeseType;
}

cheddar = new Cheese("cheddar");
//new object returned, not the
//type.
```

Even though `Cheese` is a function object (and intuitively, one thinks of functions as running modularized pieces of code), we have created a new object by invoking the function with `new` in front of it. The `this` parameter will be set to the newly created object and the `return` operator of the function will have its behavior altered. Regarding the behavior of the `return` operator in constructor invocation, there are two cases:

1. If the function returns a simple type (number, string, Boolean, null, or undefined), the return will be ignored and instead `this` will be returned (which is set to the new object).
2. If the function returns an instance of `Object` (anything other than a simple type), then that object will be returned instead of returning `this`. This pattern is not used that often, but it may have utility when used with closures.

For example:

```
var obj = {
    data : "Hello World"
}
var Func1 = function() {
    return obj;
}
var Func2 = function() {
    return "I am a simple type";
}

var f1 = new Func1();
//f1 is set to obj

var f2 = new Func2();
//f2 is set to a new object
```

We might ignore this, and just use object literals to make objects, except that the makers of JavaScript have enabled a key feature of their language by using this pattern: object creation with an arbitrary prototype link. This pattern is unintuitive and also potentially problematic. There is a remedy which was championed by Douglas Crockford: augment `Object` with a create method that accomplishes what the constructor invocation pattern tries to do. I am happy to note that as of JavaScript 1.8.5, `Object.create` is a reality and can be used. Due to legacy, the constructor invocation is still used often, and for backward compatibility, will crop up quite frequently.

### Apply And Call Invocation

The apply pattern is not as badly thought out as the preceding patterns. The `apply` method allows manual invocation of a function with a means to pass the function an array of parameters and explicitly set the `this` parameter. Because functions are first class citizens, they are also objects and hence can have methods (functions) run on it. In fact, every function is linked to `Function.prototype`, and so methods can very easily be augmented

to any function. The apply method is just an augmentation to every function as, I presume, it is defined on `Function.prototype`.

`Apply` takes two parameters: the first parameter is an object to bind the `this` parameter to, the second is an array which is mapped to the parameters:

```
var add = function(num1, num2) {
    return num1+num2;
}

array = [3,4];
add.apply(null,array); //7
```

In the example above, `this` is bound to null (the function is not an object, so it is not needed) and array is bound to `num1` and `num2`. More interesting things can be done with the first parameter:

```
var obj = {
    data:'Hello World'
}

var displayData = function() {
    alert(this.data);
}

displayData(); //undefined
displayData.apply(obj);
//Hello World
```

The example above uses `apply` to bind `this` to `obj`. This results in being able to produce a value for `this.data`. Being able to explicitly assign a value to `this` is where the real value of apply comes about. Without this feature, we might as well use `()` to invoke functions.

JavaScript also has another invoker called `call`, that is identical to apply except that instead of taking an array of parameters, it takes an argument list. If JavaScript would implement function overriding, I think that `call` would be an overridden variant of `apply`. Therefore one talks about `apply` and `call` in the same vein.

## Conclusion

For better or worse, JavaScript is about to take over the world. It is therefore very important that the peculiarities of the language be known and avoided. Learning how the four function invocation methods differ and how to avoid their pitfalls is fundamental to anyone who wants to use JavaScript. I hope this post has helped people when it comes to invoking functions. ■

---

Barry is an entrepreneur in the tech space and loves building programs. He also loves the building process that is inherent with entrepreneurism.

Reprinted with permission of the original author.
First appeared in *hn.my/invo* (doctrina.org)

# How Does SSL Work?

*By* LUC GOMMANS

SSL (AND ITS successor, TLS) is a protocol that operates directly on top of TCP (although there are also implementations for datagram-based protocols, such as UDP). This way, protocols on higher layers (such as HTTP) can be left unchanged while still providing a secure connection. Underneath the SSL layer, HTTP is identical to HTTPS.

When using SSL/TLS correctly, all an attacker can see on the cable is which IP and domain you are connected to, roughly how much data you are sending, and what encryption and compression is used. He can also terminate the connection, but both sides will know that the connection has been interrupted by a third party.

## High-level description of the protocol

After building a TCP connection, the SSL handshake is started by the client. The client (which can be a browser as well as any other program, such as Windows Update or PuTTY) sends a number of specifications: which version of SSL/TLS it is running, what ciphersuites it wants to use, and what compression methods it wants to use. The server checks for the highest SSL/TLS version that is supported by them both, picks a ciphersuite from one of the client's options (if it supports one), and optionally picks a compression method.

After this basic setup is done, the server sends its certificate. This certificate must be trusted by either the client itself or a party that the client trusts. For example, if the client trusts Thawte, then the client can trust the certificate from Google.com because Thawte cryptographically signed Google's certificate.

After the client verifies the certificate and is certain this server really is who he claims to be (and not a man in the middle), a key is exchanged. This can be a public key, a "PreMasterSecret" or simply nothing, depending on the chosen ciphersuite. Both the server and the client can now compute the key for the symmetric encryption whynot PKE?. The client tells the server that all communication will be encrypted from now on, and sends an encrypted and authenticated message to the server.

The server verifies that the MAC (used for authentication) is correct and that the message can be correctly decrypted. It then returns a message, which the client verifies as well.

The handshake is now finished and the two hosts can communicate securely.

To close the connection, a `close_notify` "alert" is used. If an attacker tries to terminate the connection by finishing the TCP connection (injecting a FIN packet), both sides will know the connection was improperly terminated. The connection cannot be compromised by this though, merely interrupted.

## Some more details

### Why can you trust Google.com by trusting Thawte?

Consider that a website wants to communicate with you securely. In order to prove its identity and make sure that it is not an attacker, you must have the server's public key. However, you can hardly store all keys from all websites on earth; the database would be huge and updates would have to run every hour!

The solution to this is Certificate Authorities, or CA for short. When you installed your operating system or browser, a list of trusted CAs probably came with it. This list can be modified at will; you can remove whom you don't trust, add others, or even make your own CA (though you will be the only one trusting this CA, so it's not much use for public website). In this CA list, the CA's public key is also stored.

When Google's server sends you its certificate, it also mentions it is signed by Thawte. If you trust Thawte, you can verify (using Thawte's public key) that Thawte really did sign the server's certificate. To sign a certificate yourself, you need the private key, which is only known to Thawte. This way an attacker cannot sign a certificate himself and incorrectly claim to be Google.com. When the certificate has been modified by even one bit, the sign will be incorrect and the client will reject it.

### So if I know the public key, the server can prove its identity?

Yes. Typically, the public key encrypts and the private key decrypts. Encrypt a message with the server's public key, send it, and if the server can tell you what it originally said, it just proved that it got the private key without revealing the key.

This is why it is so important to be able to trust the public key: anyone, including an attacker, can generate a private/public key pair. You don't want to end up using the public key of an attacker!

If one of the CAs that you trust is compromised, an attacker can use the stolen private key to sign a certificate for any website they like. When the attacker can send a forged certificate to your client, signed by himself with the private key from a CA that you trust, your client doesn't know that the public key is a forged one, signed with a stolen private key.

### But a CA can make me trust any server they want!

Yes, and that is where the trust comes in. You have to trust the CA not to make certificates as they please. When organizations like Microsoft, Apple and Mozilla trust a CA though, the CA must have audits during which another organization periodically checks on them to make sure things are still running according to the rules.

Issuing a certificate is done if, and only if, the registrant can prove they own the domain that the certificate is issued for.

### What is this MAC for message authentication?

Every message is signed with a so-called Message Authentication Code, or MAC for short. If we agree on a key and hashing cipher, you can verify that my message comes from me, and I can verify that your message comes from you.

For example, with the key "correct horse battery staple" and the message "example," I can compute the MAC "58393." When I send this message with the MAC to you (you already know the key), you can perform the same computation and match up the computed MAC with the MAC that I sent.

An attacker can modify the message, but does not know the key. He cannot compute the correct MAC, and you will know the message is not authentic.

By including a sequence number when computing the MAC, you can eliminate replay attacks. SSL does this.

**You said the client sends a key, which is then used to setup symmetric encryption. What prevents an attacker from using it?**

The server's public key does. Since we have verified that the public key really belongs to the server and no one else, we can encrypt the key using the public key. When the server receives this, he can decrypt it with the private key. When anyone else receives it, they cannot decrypt it.

This is also why key size matters: The larger the public and private key, the harder it is to crack the key that the client sends to the server.

## How to crack SSL

**In summary:**

- Try if the user ignores certificate warnings;
- The application may load data from an unencrypted channel (e.g., http), which can be tampered with;
- An unprotected login page that submits to HTTPS may be modified so that it submits to HTTP;
- Unpatched applications may be vulnerable for exploits like BEAST and CRIME;
- Resort to other methods, such as a physical attack.

**In detail:**

There is no simple and straight-forward way; SSL is secure when done correctly. An attacker can try if the user ignores certificate warnings though, which would break the security instantly. When a user does this, the attacker doesn't need a private key from a CA to forge a certificate; he merely has to send a certificate of his own.

Another way would be by a flaw in the application (server- or client-side). An easy example is in websites: if one of the resources used by the website (such as an image or a script) is loaded over HTTP, the confidentiality cannot be guaranteed anymore. Even though browsers do not send the HTTP Referer header when requesting non-secure resources from a secure page, it is still possible for someone eavesdropping on traffic to guess where you're visiting from. For example, if they know images X, Y, and Z are used on one page, they can guess you are visiting that page when they see your browser request those three images at once. Additionally, when loading JavaScript, the entire page can be compromised. An attacker can execute any script on the page, modifying for example to whom the bank transaction will go.

When this happens (a resource being loaded over HTTP), the browser gives a mixed-content warning: Chrome, Firefox, Internet Explorer 9.

Another trick for HTTP is when the login page is not secured, and it submits to an https page. "Great," the developer probably thought, "now I save server load and the password is still sent encrypted!" The problem is sslstrip, a tool that modifies the insecure login page so that it submits somewhere so that the attacker can read it.

There have also been various attacks in the past few years, such as the TLS renegotiation vulnerability, sslsniff, BEAST, and very recently, CRIME. All common browsers are protected against all of these attacks though, so these vulnerabilities are no risk if you are running an up-to-date browser.

Last but not least, you can resort to other methods to obtain the information that SSL denies you to obtain. If you can already see and tamper with the user's connection, it might not be that hard to replace one of his/her .exe downloads with a keylogger or simply to physically attack that person. Cryptography may be rather secure, but humans and human error are still a weak factor. According to this paper [hn.my/breach] by Verizon, 10% of the data breaches involved physical attacks (see page 3), so it's certainly something to keep in mind. ■

---

Luc Gommans is a software development student from Eindhoven, The Netherlands. He is interested in coding, computer networking, security, and is a regular contributor to *security.stackexchange.com*

Reprinted with permission of the original author.
First appeared in *hn.my/ssl* (stackexchange.com)

# Hacking ls -l

*By* GREG LEHEY

Once upon a time, files were small. The first edition of Unix had a maximum file size of 64 KB, and even today we see the effect of the ancient 2 GB limit in the Linux `O_LARGEFILE` flag to open. But the truth is much larger. I back up my systems to disk, and looking at them is something like:

```
=== grog@eureka (/dev/pts/14) ~ 29 -> ls -l /src/dump/boskoop/
total 168169
-rw-r--r--  1 grog  wheel   4173914809 Jul 20  2006 boskopp.tar.gz
-rw-r--r--  1 root  wheel  10273920512 Mar 18  2012 delicious-image
-rw-r--r--  1 root  wheel  28968755200 Mar 16  2012 root.tar
```

What are those values? How big are the files? Your eyes go funny just trying to count the digits. How much easier would this be:

```
=== grog@eureka (/dev/pts/14) ~ 32 -> ls -l, /src/dump/boskoop/
total 168169
-rw-r--r--  1 grog  wheel   4,173,914,809 20 Jul  2006 boskopp.tar.gz
-rw-r--r--  1 root  wheel  10,273,920,512 18 Mar  2012 delicious-image
-rw-r--r--  1 root  wheel  28,968,755,200 16 Mar  2012 root.tar
```

Then again, I have the source, so I can do it. But the "how" is interesting. There are a number of steps.

1 How do you get `printf` to print the commas? Does it even work? Clearly a case for RTFM, which tells me:

```
`''   Decimal conversions (d, u,
or i) or the integral portion of
a floating point conversion (f or
F) should be grouped and sepa-
rated by thousands using the non-
monetary separator returned by
localeconv(3).
```

What's that character? It looks like a quote (`) or apostrophe ('), but so does the character after it, and they don't look the same. But they are: it's just in what passes for bold font on an xterm. But further up other confusing characters (zero and space) are explained, so this one's a candidate, too.

**On my to-do list: Update man page to explain that the character is an apostrophe.**

2 Find the code and do a quick-and-dirty modification. `ls` is `/bin/ls`, so the source should be in `/usr/src/bin/ls/`, and it is. It was relatively trivial to find the place: it's in `print.c`. For test purposes, I just added an apostrophe (') to the format, which of course would always print the commas:

```
--- print.c     (revision 241498)
+++ print.c     (working copy)
@@ -612,7 +612,7 @@
-   (void)printf("%*jd ", (u_int)
width, bytes);
+   (void)printf("%*j'd ", (u_int)
width, bytes);
```

But that came up with an unexpected problem:

```
cc -O2 -pipe  -DCOLORLS -std=gnu99
-fstack-protector -Wsystem-headers
-Werror -Wall -Wno-format-y2k -W
-Wno-unused-parameter  -Wwrite-
strings -Wswitch -Wshadow  -Wre-
dundant-decls -Wold-style-defini-
tion -Wno-pointer-sign -c /usr/
src/bin/ls/print.c
cc1: warnings being treated as
errors
/usr/src/bin/ls/print.c: In func-
tion 'printsize':
/usr/src/bin/ls/print.c:615:
warning: unknown conversion type
character ''' in format
/usr/src/bin/ls/print.c:615: warn-
ing: too many arguments for format
*** [print.o] Error code 1
```

What's that? Who's right, the man page or the compiler? In this case, the man page is right. The compiler tries to second-guess what should be in a format, and it's wrong. But it only does that if the format is a string literal. The next attempt was:

```
@@ -611,8 +611,10 @@
-   } else
-     (void)printf("%*jd ", (u_int)width, bytes);
+   } else {
+     const char *format = "%*j'd ";
+     (void)printf(format, (u_int)width, bytes);
+   }
  }
```

And that worked. Well, it compiled anyway.

**To-do list: Fix compiler's format parsing.**

---

**3** So, run `ls -l` again. No change. It seems that printf is ignoring the format specifier. Back to RTFM:

```
`''            Decimal conversions
(d, u, or i) or the integral por-
tion of a floating point conversion
(f or F) should be grouped and
separated by thousands using the
non-monetary separator returned by
localeconv(3).
```

Locales rearing their ugly head again. OK, how do I find out what my non-monetary separator is? `localeconv()` is a library function, so I can't use that to look. What commands are there? It proves that there's only `locale(1)` and `mklocale(1)`. `locale(1)` seems the obvious one to choose:

```
DESCRIPTION       The locale util-
ity is supposed to provide most
locale specific information to the
standard output.
```

That "supposed" didn't exactly fill me with confidence. But still, all I wanted to do was print the contents of my current locale. How do you do that? Run `locale(1)` with no options:

```
=== grog@eureka (/dev/pts/7) ~ 20
-> locale
LANG=
LC_CTYPE="C"
LC_COLLATE="C"
LC_TIME="C"
LC_NUMERIC="C"
LC_MONETARY="C"
LC_MESSAGES="C"
LC_ALL=
```

Not quite what I was looking for. I wanted to know what values I had set, and for that I needed keywords. The `-k` option looked like a possibility:

```
-k      Print the names and values
of all selected keywords.
```

But that's the wrong way: it wants me to tell it which keywords, and I want it to tell me all keywords. There doesn't seem to be a way to get it to show all of them.

**To-do list: Modify `locale(1)` to print all keywords if no argument is passed to the `-k` option.**

4 I carried on searching in `localeconv(3)`, which gave me the contents of struct lconv, conveniently with comments that are missing from the header file `/usr/include/locale.h`.

**To-do list: Add comments to `/usr/include/locale.h`.**

The name of the struct member is `thousands_sep`, and `locale(1)` understands that:

```
=== grog@eureka (/dev/pts/9) ~/
fbbg/www/BGIS 48 -> locale -k
thousands_sep
thousands_sep=""
```

Not quite what I was hoping for, but at least it explains part of the problem. But why isn't it set? I have `LC_NUMERIC="C"`. Does that not allow commas? How do I find out? I still don't know. Round about this time, Callum Gibson was trying his own experiments and established that setting the variable `LC_ALL` changes things:

```
export LC_ALL=en_AU.ISO8859-1
```

That's not as obvious as it seems. The output of `locale(1)` looks like these environment variables, but the only one that seems to make any difference is `LC_ALL`. After that, my test version of `ls` finally worked:

```
=== grog@eureka (/dev/pts/14) ~ 32
-> /usr/obj/usr/src/bin/ls/ls -l
/src/dump/boskoop/
total 168169
-rw-r--r--  1 root  wheel
36,211,690,564
20 Mar 2012 boskoop.disk0-1.bz2
-rw-r--r--  1 root  wheel
16,596,907,252
24 Dec 2009 boskoop.disk0.bz2
-rw-r--r--  1 root  wheel
28,968,755,200
16 Mar 2012 root.tar
```

**To-do list: Review documentation of how to set locales; possibly fix.**

5 Next, I had to do things properly by adding an option for the commas, rather than printing them all the time. That's relatively trivial, but which option? ls doesn't have too many option characters left, and there's the consideration of compatibility with POSIX.2, the other BSDs and Linux. In many ways it's a lost cause, of course. The options for GNU ls vary wildly from those for BSD ls, including lots of long options such as `--show-control-chars`, a verbose way of representing FreeBSD's `-w` option. And others, such as `-T`, have completely unrelated meanings.

Still, it's good not to add more entropy than necessary, and I'm going to have to investigate this one.

**To-do list: Choose a good option character.**

For now, the most obvious one seems to be the comma (,) character. That works, but it's possible that POSIX doesn't like that, and it's liable to stir up a bikeshed when I commit.

6 So, finally I'm done. Or am I? No, there's more:

**To-do list: Update man page and `usage()` function.**

7 But then we're done! Well, no. Callum Gibson reported that it still didn't work for his program, so I wrote a little one that just called `printf` with the apostrophe (') format modifier. And it didn't work. We traced the problem to the difference in ls: at the start of the program there's a:

```
(void)setlocale(LC_ALL, "");
```

And this appears to be necessary. Is it adequately documented? There's something in `setlocale(3)` (obviously), but I managed to miss it. So:

**To-do list: Investigate `setlocale()` documentation.**

But then I'm really done — I hope. It's amazing how much work there is apart from just hacking the code. ■

---

Greg Lehey is an independent computer consultant specializing in UNIX. In the course of over 20 years in the industry he has performed most jobs you can think of, ranging from kernel support to product marketing, systems programming to operating, processing satellite data to programming gasoline pumps.

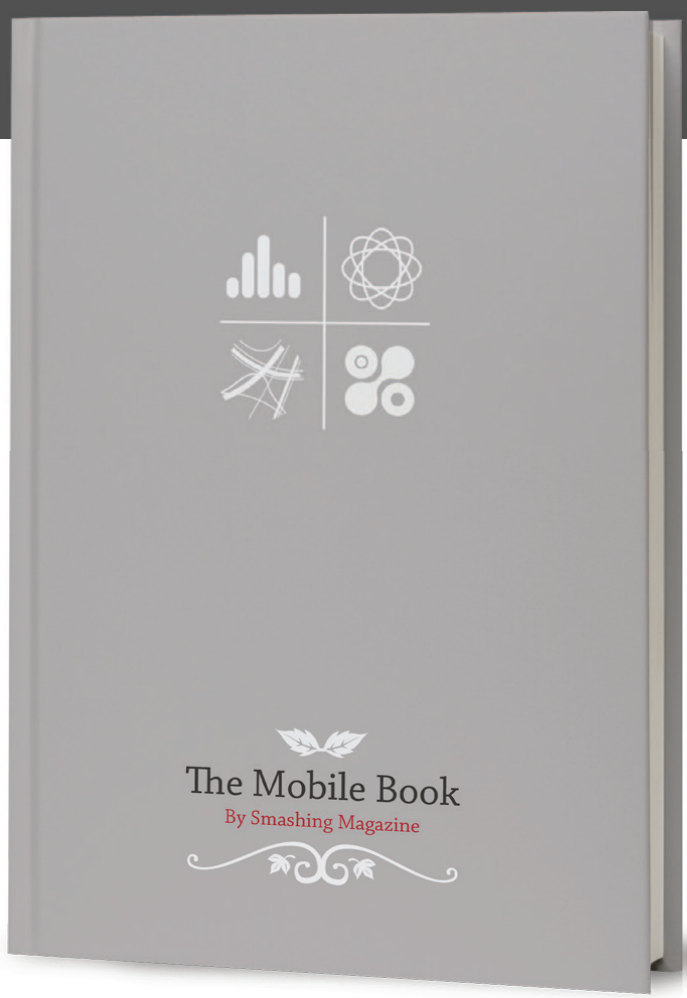
Reprinted with permission of the original author.
First appeared in *hn.my/ls* (lemis.com)

# THE NEW SMASHING MOBILE BOOK

+ eBook version

If you're looking for a good book on Mobile, this is the one. Brand new book on best design and coding practices for mobile, Responsive Web design and UX design for mobile.

**www.the-mobile-book.com**




SMASHING MAGAZINE

# An Intuitive Guide to Linear Algebra

*By* KALID AZAD

Despite two linear algebra classes, my knowledge consisted of "Matrices, determinants, Eigen something something."

Why? Well, let's try this course format:

- Name the course "Linear Algebra" but focus on things called matrices and vectors
- Label items with similar-looking letters (i/j), and even better, similar-looking-and-sounding ones (m/n)
- Teach concepts like Row/Column order with mnemonics instead of explaining the reasoning
- Favor abstract examples (2d vectors! 3d vectors!) and avoid real-world topics until the final week

The survivors are physicists, graphics programmers, and other masochists. We missed the key insight:

**Linear algebra gives you mini-spreadsheets for your math equations.**

We can take a table of data (a matrix) and create updated tables from the original. It's the power of a spreadsheet written as an equation.

Here's the linear algebra introduction I wish I had, with a real-world stock market example.

## What's In A Name?

"Algebra" means, roughly, "relationships." Grade-school algebra explores the relationship between unknown numbers. Without knowing x and y, we can still work out that (x + y)^2 = x^2 + 2xy + y^2.

"Linear Algebra" means, roughly, "line-like relationships." Let's clarify a bit.

Straight lines are predictable. Imagine a rooftop: move forward 3 horizontal

feet (relative to the ground), and you might rise 1 foot in elevation (the slope! Rise/run = 1/3). Move forward 6 feet, and you'd expect a rise of 2 feet. Contrast this with climbing a dome: each horizontal foot forward raises you a different amount.

Lines are nice and predictable:

- If 3 feet forward has a 1-foot rise, then going 10x as far should give a 10x rise (30 feet forward is a 10-foot rise)
- If 3 feet forward has a 1-foot rise, and 6 feet has a 2-foot rise, then (3 + 6) feet should have a (1 + 2) foot rise

In math terms, an operation F is linear if scaling inputs scales the output, and adding inputs adds the outputs:

$$F(ax) = a \cdot F(x)$$
$$F(x+y) = F(x) + F(y)$$

In our example, F(x) calculates the rise when moving forward x feet. F(10*3) = 10 * F(3) = 10 and F(3+6) = F(3) + F(6) = 3.

## Linear Operations

An operation is a calculation based on some inputs. Which operations are linear and predictable? Multiplication, it seems.

Exponents (F(x) = x^2) aren't predictable: 10^2 is 100, but 20^2 is 400. We doubled the input but quadrupled the output.

Surprisingly, regular addition isn't linear either. Consider the "add three" function:

$$F(x) = x + 3$$
$$F(10) = 13$$
$$F(20) = 23$$

We doubled the input and did not double the output. (Yes, F(x) = x + 3 happens to be the equation for an offset line, but it's still not "linear" because F(10) isn't 10 * F(1). Fun.)

Our only hope is to multiply by a constant: F(x) = ax (in our roof example, a=1/3). However, we can still combine linear operations to make a new linear operation:

$$G(x,y,z) = F(x+y+z) = F(x) + F(y) + F(z)$$

G is made of 3 linear subpieces: if we double the inputs, we'll double the output.

We have "mini arithmetic": multiply inputs by a constant, and add the results. It's actually useful because we can split inputs apart, analyze them individually, and combine the results:

$$G(x,y,z) = G(x,0,0) + G(0,y,0) + G(0,0,z)$$

If the inputs interacted like exponents, we couldn't separate them — we'd have to analyze everything at once.

## Organizing Inputs and Operations

Most courses hit you in the face with the details of a matrix. "Ok kids, let's learn to speak. Select a subject, verb and object. Next, conjugate the verb. Then, add the prepositions…."

No! Grammar is not the focus. What's the key idea?

- We have a bunch of inputs to track
- We have predictable, linear operations to perform (our "mini-arithmetic")
- We generate a result, perhaps transforming it again

Ok. First, how should we track a bunch of inputs? How about a list:

```
x
y
z
```

Not bad. We could write it (x, y, z) too — hang onto that thought.

Next, how should we track our operations? Remember, we only have "mini arithmetic": multiplications, with a final addition. If our operation F behaves like this:

$$F(x, y, z) = 3x + 4y + 5z$$

We could abbreviate the entire function as (3, 4, 5). We know to multiply the first input by the first value, the second input by the second value, etc., and add the result.

Only need the first input?

$$G(x, y, z) = 3x + 0y + 0z = (3, 0, 0)$$

Let's spice it up: how should we handle multiple sets of inputs? Let's say we want to run operation F on both (a, b, c) and (x, y, z). We could try this:

$$F(a, b, c, x, y, z) = ?$$

But it won't work: F expects 3 inputs, not 6. We should separate the inputs into groups:

```
1st Input  2nd Input
-------------------
a          x
b          y
c          z
```

Much neater.

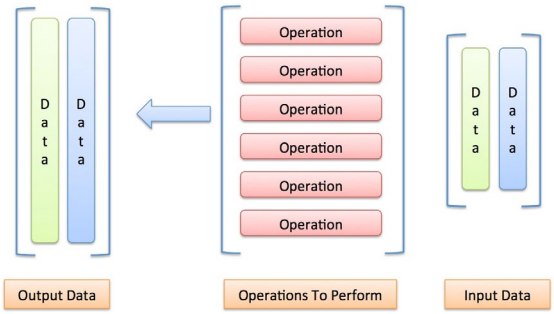
And how could we run the same input through several operations? Have a row for each operation:

```
F: 3 4 5
G: 3 0 0
```

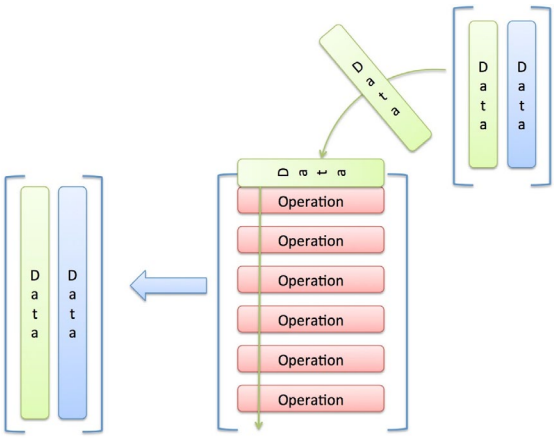
Neat. We're getting organized: inputs in vertical columns, operations in horizontal rows.

## Visualizing The Matrix

Words aren't enough. Here's how I visualize inputs, operations, and outputs:




Imagine "pouring" each input along each operation:




As an input passes an operation, it creates an output item. In our example, the input (a, b, c) goes against operation F and outputs 3a + 4b + 5c. It goes against operation G and outputs 3a + 0 + 0.

Time for the red pill. A matrix is a shorthand for our diagrams:

$$\text{Inputs} = A = \begin{bmatrix}\text{input1} & \text{input2}\end{bmatrix} = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

$$\text{Operations} = M = \begin{bmatrix}\text{operation1} \\ \text{operation2}\end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \\ 3 & 0 & 0 \end{bmatrix}$$

A matrix is a single variable representing a spreadsheet of inputs or operations.

### Trickiness #1: The reading order

Instead of an input => matrix => output flow, we use function notation, like y = f(x) or f(x) = y. We usually write a matrix with a capital letter (F), and a single input column with lowercase (x). Because we have several inputs (A) and outputs (B), they're considered matrices too:

$$MA = B$$

$$\begin{bmatrix} 3 & 4 & 5 \\ 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix} = \begin{bmatrix} 3a + 4b + 5c & 3x + 4y + 5z \\ 3a & 3x \end{bmatrix}$$

### Trickiness #2: The numbering

Matrix size is measured as RxC: row count, then column count, and abbreviated mxn. Items in the matrix are referenced the same way: aij is the ith row and jth column. Mnemonics are ok with context, and here's what I use:

- RC, like Roman Centurion or RC Cola
- Use an "L" shape. Count down the L, then across

Why does RC ordering make sense? Our operations matrix is 2×3 and our input matrix is 3×2. Writing them together:

```
[Operation Matrix] [Input Matrix]
[operation count x operation size]
[input size x input count]
[m x n] [p x q] = [m x q]
[2 x 3] [3 x 2] = [2 x 2]
```

Notice the matrices touch at the "size of operation" and "size of input" (n = p). They should match! If our inputs have 3 components, our operations should expect 3 items. In fact, we can only multiply matrices when n = p.

The output matrix has m operation rows for each input, and q inputs, giving a "m x q" matrix.

## Fancier Operations

Let's get comfortable with operations. Assuming 3 inputs, we can whip up a few 1-operation matrices:

- Adder: [1 1 1]
- Averager: [1/3 1/3 1/3]

The "Adder" is just a + b + c. The "Averager" is similar: (a + b + c)/3 = a/3 + b/3 + c/3.

Try these 1-liners:

- First-input only: [1 0 0]
- Second-input only: [0 1 0]
- Third-input only: [0 0 1]

And if we merge them into a single matrix:

```
[1 0 0]
[0 1 0]
[0 0 1]
```

Whoa — it's the "identity matrix", which copies 3 inputs to 3 outputs, unchanged. How about this guy?

```
[1 0 0]
[0 0 1]
[0 1 0]
```

He reorders the inputs: (x, y, z) becomes (x, z, y).

And this one?

```
[2 0 0]
[0 2 0]
[0 0 2]
```

He's an input doubler. We could rewrite him to 2*I (the identity matrix) if we were so inclined.

And yes, when we decide to treat inputs as vector coordinates, the operations matrix will transform our vectors. Here are a few examples:

- Scale: make all inputs bigger/smaller
- Skew: make certain inputs bigger/smaller
- Flip: make inputs negative
- Rotate: make new coordinates based on old ones (East becomes North, North becomes West, etc.)

These are geometric interpretations of multiplication, and how to warp a vector space. Just remember that vectors are examples of data to modify.

## A Non-Vector Example: Stock Market Portfolios

Let's practice linear algebra in the real world:

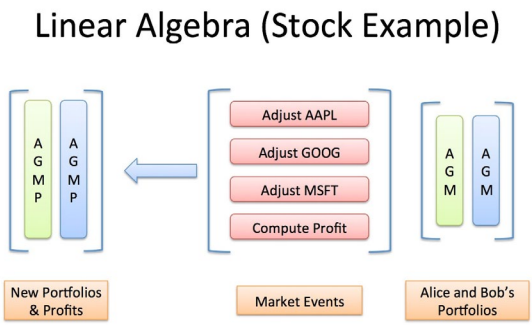
- Input data: stock portfolios with dollars in Apple, Google, and Microsoft stock
- Operations: the changes in company values after a news event
- Output: updated portfolios

And a bonus output: let's make a new portfolio listing the net profit/loss from the event.

Normally, we'd track this in a spreadsheet. Let's learn to think with linear algebra:

- The input vector could be ($Apple, $Google, $Microsoft), showing the dollars in each stock. (Oh! These dollar values could come from another matrix that multiplied the number of shares by their price. Fancy that!)
- The 4 output operations should be: Update Apple value, Update Google value, Update Microsoft value, Compute Profit

Visualize the problem. Imagine running through each operation:




The key is understanding why we're setting up the matrix like this, not blindly crunching numbers.

Got it? Let's introduce the scenario.

Suppose a secret iDevice is launched: Apple jumps 20%, Google drops 5%, and Microsoft stays the same. We want to adjust each stock value, using something similar to the identity matrix:

```
New Apple     [1.2  0      0]
New Google    [0    0.95   0]
New Microsoft [0    0      1]
```

The new Apple value is the original, increased by 20% (Google = 5% decrease, Microsoft = no change).

Oh wait! We need the overall profit:

Total change = (.20 * Apple) + (-.05 * Google) + (0 * Microsoft)

Our final operations matrix:

```
New Apple        [1.2  0      0]
New Google       [0    0.95   0]
New Microsoft    [0    0      1]
Total Profit     [.20  -.05   0]
```

Making sense? Three inputs enter, four outputs leave. The first three operations are a "modified copy" and the last brings the changes together.

Now let's feed in the portfolios for Alice ($1000, $1000, $1000) and Bob ($500, $2000, $500). We can crunch the numbers by hand, or use a Wolfram Alpha:

Input interpretation:

$$\begin{pmatrix} 1.2 & 0 & 0 \\ 0 & 0.95 & 0 \\ 0 & 0 & 1 \\ 0.2 & -0.05 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1000 & 1000 & 1000 \\ 500 & 2000 & 500 \end{pmatrix}^{\mathrm{T}}$$

Computed by Wolfram|Alpha

Result:

$$\begin{pmatrix} 1200. & 600. \\ 950. & 1900. \\ 1000. & 500. \\ 150. & 0. \end{pmatrix}$$

Computed by Wolfram|Alpha

*(Note: Inputs should be in columns, but it's easier to type rows. The Transpose operation, indicated by t (tau), converts rows to columns.)*

The final numbers: Alice has $1200 in AAPL, $950 in GOOG, $1000 in MSFT, with a net profit of $150. Bob has $600 in AAPL, $1900 in GOOG, and $500 in MSFT, with a net profit of $0.

What's happening? We're doing math with our own spreadsheet. Linear algebra emerged in the 1800s yet spreadsheets were invented in the 1980s. I blame the gap on poor linear algebra education.

## Historical Notes: Solving Simultaneous Equations

An early use of tables of numbers (not yet a "matrix") was bookkeeping for linear systems:

$$\begin{aligned} x + 2y + 3z &= 3 \\ 2x + 3y + 1z &= -10 \\ 5x + -y + 2z &= 14 \end{aligned}$$

becomes

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 5 & -1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ -10 \\ 14 \end{bmatrix}$$

We can avoid hand cramps by adding/subtracting rows in the matrix and output, vs. rewriting the full equations. As the matrix evolves into the identity matrix, the values of x, y and z are revealed on the output side.

This process, called Gauss-Jordan elimination, saves time. However, linear algebra is mainly about matrix transformations, not solving large sets of equations (It'd be like using Excel for your shopping list).

## Terminology, Determinants, and Eigenstuff

Words have technical categories to describe their use (nouns, verbs, adjectives). Matrices can be similarly subdivided.

Descriptions like "upper-triangular," "symmetric," and "diagonal" are the shape of the matrix, and influence their transformations.

The **determinant** is the "size" of the output transformation. If the input was a unit vector (representing area or volume of 1), the determinant is the size of the transformed area or volume. A determinant of 0 means matrix is "destructive" and cannot be reversed (similar to multiplying by zero: information was lost).

The **eigenvector** and **eigenvalue** are the "axes" of the transformation.

Consider a spinning globe: every location faces a new direction, except the poles.

An "eigenvector" is the input that doesn't change direction after going through the matrix (it points "along the axis"). And although the direction doesn't change, the size might. The eigenvalue is the amount the eigenvector is scaled up or down when going through the matrix.

## Matrices As Inputs

A funky thought: we can treat the operations matrix as inputs!

Think of a recipe as a list of commands (Add 2 cups of sugar, 3 cups of flour…).

What if we want the metric version? Take the instructions, treat them like text, and convert the units. The recipe is "input" to modify. When we're done, we can follow the instructions again.

An operations matrix is similar: commands to modify. Applying one operations matrix to another gives a new operations matrix that applies both transformations, in order.

If N is "adjust for portfolio for news" and T is "adjust portfolio for taxes" then applying both:

TN = X

means "Create matrix X, which first adjusts for news, and then adjusts for taxes". Whoa! We didn't need an input portfolio, we applied one matrix directly to the other.

The beauty of linear algebra is representing an entire spreadsheet calculation with a single letter. Want to apply the same transformation a few times? Use N^2 or N^3.

## Can We Use Regular Addition, Please?

Yes, because you asked nicely. Our "mini arithmetic" seems limiting: multiplications, but no addition? Time to expand our brains.

Imagine adding a dummy entry of 1 to our input: (x, y, z) becomes (x, y, z, 1).

Now our operations matrix has an extra, known value to play with! If we want x + 1 we can write:

```
[1 0 0 1]
```

And x + y - 3 would be:

```
[1 1 0 -3]
```

Huzzah!

Want the geeky explanation? We're pretending our input exists in a 1-higher dimension, and put a "1" in that dimension. We skew that higher dimension, which looks like a slide in the current one. For example: take input (x, y, z, 1) and run it through:

```
[1 0 0 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

The result is (x + 1, y + 1, z + 1, 1). Ignoring the 4th dimension, every input got a +1. We keep the dummy entry, and can do more slides later.

Mini-arithmetic isn't so limited after all.

## Onward

I've overlooked some linear algebra subtleties, and I'm not too concerned. Why?

These metaphors are helping me think with matrices, more than the classes I "aced." I can finally respond to "Why is linear algebra useful?" with "Why are spreadsheets useful?"

They're not, unless you want a tool used to attack nearly every real-world problem. Ask a businessman if they'd rather donate a kidney or be banned from Excel forever. That's the impact of linear algebra we've overlooked: efficient notation to bring spreadsheets into our math equations.

Happy math. ■

---

Kalid is a YC alum living in Seattle. He loves to simplify complex ideas, blog aha! moments at BetterExplained, and do math with *instacalc.com*

Reprinted with permission of the original author.
First appeared in *hn.my/linear* (betterexplained.com)

# Pascal's Apology

*By* CARLOS BUENO

Blaise Pascal once famously ended a letter with an apology: *I'm sorry that this was such a long letter, but I didn't have time to write you a short one.* Computer science has pretty much the same problem. It's a young field, and young fields are, by their nature, messy. As we race to generate new knowledge, we also generate excess jargon. There are multiple names for the same ideas, and the ideas themselves are often tangled together. What's worse is that because that's the way things are now, we assume that's the way they are always going to be. It's a complicated science because we haven't taken the time to make a simple one.

Let me show you what I mean:

> *A Von Neumann randomness extractor takes as input a Bernoulli sequence with p not necessarily equal to 1/2, and outputs a Bernoulli sequence with p equal to 1/2. More generally, it applies to any exchangeable sequence relying on the fact that for any pair, 01 and 10 are equally likely.*

And now you know about as much as you did before. Let me try that again. Imagine that you have a coin that is unfair. It's biased. If you flip it you'll get heads seventy percent of the time. If you wanted to make a fair, fifty-fifty choice, what do you do?

Well, one way is to get a coin that doesn't suck. Another way, discovered by John von Neumann, is to use the bias against itself. You can flip it two times. If you get a heads and then a tails, you say heads is your answer. If you get a tails and then a heads, you say tails. If you get anything else, you just start over. And that's it. This works because no matter what the bias happens to be, the odds of getting a heads and then tails will always be exactly the same as the odds of getting a tails and then heads. This is for the same reason that seven times three is equal to three times seven.

That's all very interesting, but why should you care? This algorithm was invented to ensure that you can get a clean source of random numbers even if you are stuck with buggy hardware, which happens all the time. Quality randomness is essential for cryptography. Cryptography is essential for secure communications, which is the basis of our entire modern life. Without this little coin trick there would be no Facebook, no Gmail, no Skyping grandma, no PIN numbers on your credit cards, no buying books online, no banking from your mobile phones. And, no mobile phones. It's fundamental, and it turns out that it's also easy to understand. You could teach it to a child. So, why don't we?

My overall complaint here, and the reason I wrote this book [laurenipsum.org], is a little nuanced. I'm not just saying, "Hey everyone! Guess what — computer science is hard!" Anything worth doing is going to be hard, and I can't change that. But some parts are harder than others. My complaint is that if the hard stuff is messy and complicated, and we allow the easy stuff to be messy and complicated too, then you can't really tell the difference.

That means, as teachers, we're probably starting in the wrong place.

I don't think it has to be this way. What do we expect children to understand about math? Negative numbers, zero, exponents, the square root of two, pi, etc. In those boring little facts I see hope, precisely because they are boring little facts. It wasn't always like that. Once upon a time, the existence of negative numbers was considered the most difficult question in the world. People died arguing about the hypotenuse, for God's sake. The fact that we can teach this to innocent children is evidence of progress — real, measurable, personally empowering progress. It's the kind of progress we haven't had time to make in computer science.

If the mathematicians are making fun of you for being too complicated, you know there's work to do.

I started writing Lauren Ipsum by looking for ideas that I understood well enough to explain to a nine-year-old child, without regard for how supposedly hard they were. If I found something I couldn't clearly explain, then I tried to break it down further to learn why. Sometimes it worked, sometimes not. I learned a lot from this.

Early on I decided that I wasn't actually writing a book about how to program. I was writing a book about how programmers think. Once you get past the first hump and really start to learn this stuff, you develop some mental habits to help you cope. So, I figured, let's write those down. Start there.

The book starts with a character called the Wandering Salesman. He's lost, but only mostly lost. He actually knows where he is, and he knows where he's going. He's still lost because he doesn't know how to get there. This is the essence of effective problem solving. It's about having a clear goal and knowing where you stand. Instead of giving in to that instinct to do the first thing that comes to mind, you try to imagine all the possible ways to solve the problem, compare them, and choose one.

This is a big part of what computer science actually studies. It's not about computers; they are just a tool. It's about how to generate those possible answers and those algorithms; how to characterize them so they can be compared; and how to choose.

As the hero, a little girl named Lauren, goes through her adventures, she learns not just that she is responsible for her own decisions, but that there may be better ways to make them. And, by the way, here are some tools to help you do that.

So far the response has been pretty good. It's a start. A small start. But I think this is a rich vein to explore. Think about it this way: if you leave this world as complicated as you found it and if the next generation takes just as long to learn what you've learned, then they'll never have time to do better than you.

So, wherever you can, don't just transmit knowledge. Simplify it. Take the time. Write the short letters. Because that's how we make progress. ■

---

Carlos Bueno is an engineer at Facebook. He writes occasionally about programming, performance, internationalization, and why everyone should learn computer science.

Reprinted with permission of the original author.
First appeared in *hn.my/pascal* (bueno.org)

# Understanding Ego Depletion

*By* DAN ARIELY

From your own experience, are you more likely to finish half a pizza by yourself on a) Friday night after a long work week, or b) Sunday evening after a restful weekend? The answer that most people will give, of course, is "a." And in case you hadn't noticed, it's on stressful days that many of us give in to temptation and choose unhealthy options. The connection between exhaustion and the consumption of junk food is not just a figment of your imagination.

And it is the reason why so many diets bite it in the midst of stressful situations, and why many resolutions derail in times of crisis.

How do we avoid breaking under stress? There are six simple rules.

## 1 Acknowledge the tension, don't ignore it.

Usually in these situations, there's an internal dialogue (albeit one of varying length) that goes something like this:

*"I'm starving! I should go home and make a salad and finish off that leftover grilled chicken."*

"But it's been such a long day. I don't feel like cooking." [Walks by popular spot for Chinese takeout] "Plus, beef lo mein sounds amazing right now."

*"Yes, yes it does, but you really need to finish those vegetables before they go bad, plus, they'll be good with some Dijon vinaigrette!"*

"Not as good as those delicious noodles with all that tender beef."

*"Hello, remember the no carbs resolution? And the eat vegetables every day one, too? You've been doing so well!"*

"Exactly, I've been so good! I can have this one treat…"

And so the battle is lost. This is the push-pull relationship between reason (eat well!) and impulse (eat that right now!). And here's the reason we make bad decisions: we use our self-control every time we force ourselves to make the good, reasonable decision, and that self-control, like other human capacities, is limited.

**2 Call it what it is: ego-depletion.** Eventually, when we've said "no" to enough yummy food, drinks, and potential purchases, and forced ourselves to do enough unwanted chores, we find ourselves in a state that Roy Baumeister calls "ego-depletion," where we don't have any more energy to make good decisions. So, back to our earlier question: when you contemplate your Friday versus Sunday night selves, which one is more depleted? Obviously, the former.

You may call this condition by other names (stressed, exhausted, worn out, etc.) but depletion is the psychological sum of these feelings, of all the decisions you made that led to that moment. The decision to get up early instead of sleeping in, the decision to skip pastries every day on the way to work, the decision to stay at the office late to finish a project instead of leaving it for the next day (even though the boss was gone!), the decision not to skip the gym on the way home, and so on, and so forth. Because when you think about it, you're not actually too tired to choose something healthy for dinner (after all, you can just as easily order soup and sautéed greens instead of beef lo mein and an order of fried gyoza), you're simply out of will power to make that decision.

**3 Understand ego-depletion.** Enter Baba Shiv (a professor at Stanford University) and Sasha Fedorikhin (a professor at Indiana University) who examined the idea that people yield to temptation more readily when the part of the brain responsible for deliberative thinking has its figurative hands full.

In this seminal experiment, a group of participants gathered in a room and were told that they would be given a number to remember and which they were to repeat to another experimenter in a room down the hall. Easy enough, right? Well, the ease of the task actually depended on which of the two experimental groups you were in. You see, people in group 1 were given a two-digit number to remember. Let's say, for the sake of illustration, that the number is 62. People in group two, however, were given a seven-digit

number to remember, 3074581. Got that memorized? Okay!

Now here's the twist: half way to the second room, a young lady was waiting by a table upon which sat a bowl of colorful fresh fruit and slices of fudgy chocolate cake. She asked each participant to choose which snack they would like after completing their task in the next room, and gave them a small ticket corresponding to their choice. As Baba and Sasha suspected, people laboring under the strain of remembering 3074581 chose chocolate cake far more often than those who had only 62 to recall. As it turned out, those managing greater cognitive strain were less able to overturn their instinctive desires.

This simple experiment doesn't really show how ego-depletion works, but it does demonstrate that even a simple cognitive load can alter decisions that could potentially have an effect on our lives and health. So consider how much greater the impact of days and days of difficult decisions and greater cognitive loads would be.

## 4 Include and consider the moral implications.

Depletion doesn't only affect our ability to make good decisions, it also makes it harder for us to make honest ones. In one experiment that tested the relationship between depletion and honesty, my colleagues and I split participants into two groups and had them complete something called a Stroop task, which is a simple task requiring only that the participant name aloud the color of the ink a word (which is itself a color) is written in. The task, however, has two forms: in the first, the color of the ink matches the word, called the "congruent" condition, in the second, the color of the ink differs from the word, called the "incongruent" condition. Go ahead and try both tasks yourself…

**The congruent condition: color matches word.**

| | | | | |
|---|---|---|---|---|
| RED | BLUE | GREEN | RED | BLUE |
| GREEN | GREEN | RED | BLUE | GREEN |
| BLUE | RED | BLUE | GREEN | RED |
| GREEN | BLUE | RED | RED | BLUE |
| RED | RED | GREEN | BLUE | GREEN |
| BLUE | GREEN | BLUE | GREEN | RED |
| RED | BLUE | GREEN | BLUE | GREEN |
| BLUE | GREEN | RED | GREEN | RED |
| GREEN | RED | BLUE | RED | BLUE |
| BLUE | GREEN | GREEN | BLUE | GREEN |
| GREEN | RED | BLUE | RED | RED |
| RED | BLUE | RED | GREEN | BLUE |
| GREEN | RED | BLUE | RED | GREEN |
| BLUE | BLUE | RED | GREEN | RED |
| RED | GREEN | GREEN | BLUE | BLUE |

**The incongruent condition: color conflicts with word.**

| | | | | |
|---|---|---|---|---|
| RED | BLUE | GREEN | RED | BLUE |
| GREEN | GREEN | RED | BLUE | GREEN |
| BLUE | RED | BLUE | GREEN | RED |
| GREEN | BLUE | RED | RED | BLUE |
| RED | RED | GREEN | BLUE | GREEN |
| BLUE | GREEN | BLUE | GREEN | RED |
| RED | BLUE | GREEN | BLUE | GREEN |
| BLUE | GREEN | RED | GREEN | RED |
| GREEN | RED | BLUE | RED | BLUE |
| BLUE | GREEN | GREEN | BLUE | GREEN |
| GREEN | RED | BLUE | RED | RED |
| RED | BLUE | RED | GREEN | BLUE |
| GREEN | RED | BLUE | RED | GREEN |
| BLUE | BLUE | RED | GREEN | RED |
| RED | GREEN | GREEN | BLUE | BLUE |

As you no doubt observed, naming the color in the incongruent version is far more difficult than in the congruent. Each time you repressed the word that popped instantly into your mind (the word itself) and forced yourself to name the color of the ink instead, you became slightly more depleted as a result of that repression.

As for the participants in our experiment, this was only the beginning. After they finished whichever task they were assigned to, we first offered them the opportunity to cheat. Participants were asked to take a short quiz on the history of Florida State University (where the experiment took place), for which they would be paid for the number of correct answers. They were asked to circle their answers on a sheet of paper, then transfer those answers to a bubble sheet. However, when participants sat down with the experimenter, they discovered she had run into a problem. "I'm sorry," the experimenter would say with exasperation, "I'm almost out of bubble sheets! I only have one unmarked one left, and one that has the answers already marked." She explained to participants that she did her best to erase the marks but that they're still slightly visible. Annoyed with herself, she admits that she had hoped to give one more test today after that one, then asks a question: "Since you are the first of the last two participants of the day, you can choose which form you would like to use: the clean one or the premarked one."

So what do you think participants did? Did they reason with themselves that they'd help the experimenter out and take the premarked sheet, and be fastidious about recording their accidents accurately? Or did they realize that this would tempt them to cheat, and leave the premarked sheet alone? Well, the answer largely depended on which Stroop task they had done: those who had struggled through the incongruent version chose the premarked sheet far more often than the unmarked. What this means is that depletion can cause us to put ourselves into compromising positions in the first place.

And what about the people, in either condition, who chose the premarked sheet? Once again, those who were depleted by the first task, once in a position to cheat, did so far more often than those who breezed through the congruent version of the task.

What this means is that when we become depleted, we're not only more apt to make bad and/or dishonest choices, we're also more likely to allow ourselves to be tempted to make them in the first place. Talk about double jeopardy.

### 5 Evade ego-depletion.

There's a saying that nothing good happens after midnight, and arguably, depletion is behind this bit of folk wisdom. Unless you work the third shift, if you're up after midnight it's probably been a pretty long day for you, and at that point, you're more likely to make sub-optimal decisions, as we've learned.

So how can we escape depletion?

A friend of mine named Dan Silverman once suggested an interesting approach during our time together at the Institute for Advanced Study at Princeton, which is a delightful place for researchers to take a year off to think, plan, and eat very well. Every day, after a rich lunch, we were plied with nigh-irresistible desserts: cheesecake, chocolate tortes, profiteroles, beignets — you name it. It was difficult for all of us, but especially for poor Dan, who was forever at the mercy of his sweet tooth.

It was a daily dilemma for my friend. Dan, who was an economist with high cholesterol, wanted dessert. But he also understood that eating dessert every day was not a good decision. He contemplated this problem (along with his other academic interests), and concluded that when faced with temptation, a wise person should occasionally succumb. After all, by doing so, said person can keep him- or herself from becoming overly depleted, which will provide strength for whatever unexpected temptations lie in wait. Dan decided that giving in to daily dessert would be his best defense against being caught unawares by temptation and weakness down the road.

In all seriousness though, we've all heard time and time again that if you restrict your diet too much, you'll likely to go overboard and binge at some point. Well, it's true. A crucial aspect of managing depletion and making good decisions is having ways to release stress and reset, and to plan for certain indulgences. In fact, I think one reason the Slow-Carb Diet seems to be so effective is because it advises dieters to take a day off (also called a "cheat" day — see item 4 above), which allows them to avoid becoming so deprived that they give up entirely. The key here is planning the

indulgence rather than waiting until you have absolutely nothing left in the tank. It's in the latter moments of desperation that you throw yourself on the couch with the whole pint of ice cream, not even making a pretense of portion control, and go to town while watching your favorite TV show.

Regardless of the indulgence, whether it's a new pair of shoes, some "me time" where you turn off your phone, an ice cream sundae, or a night out — plan it ahead. While I don't recommend daily dessert, this kind of release might help you face down challenges to your will power later.

### 6 Know Thyself.

The reality of modern life is that we can't always avoid depletion. But that doesn't mean we're helpless against it. Many people probably remember the G.I. Joe cartoon catch phrase: "Knowing is half the battle." While this served in the context of PSAs of various stripes, it can help us here as well. Simply knowing you can become depleted, and moreover, knowing the kinds of decisions you might make as a result, makes you far better equipped to handle difficult situations when and as they arise. ■

---

Dan Ariely is also the author of several excellent books, including Predictably Irrational and, most recently, The Honest Truth About Dishonesty.

Reprinted with permission of the original author.
First appeared in *hn.my/ego* (danariely.com)

# Small

*By* ALEX MACCAW

LOOKING DOWN FROM his perch on the edge of space, Felix Baumgartner remarked:

> *Sometimes you have to be really high, to see how small you really are.*

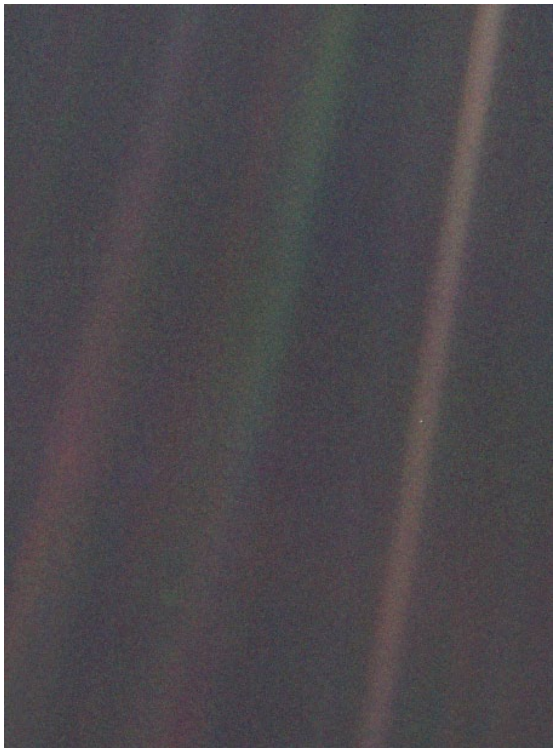
It turns out that this feeling is a well documented phenomena dubbed the *Overview effect*. When a person gazes upon Earth from outer space, they have a profound sense of perspective, a realization of fragility, that humanity and all life as we know it is completely dependent on a single planet and its thin atmosphere.

> *It suddenly struck me that that tiny pea, pretty and blue, was the Earth. I put up my thumb and shut one eye, and my thumb blotted out the planet Earth. I didn't feel like a giant. I felt very, very small.*
> *— Neil Armstrong*

So while the first astronauts to the moon went as technicians, they came back as humanitarians. In the words of William Anders, "We came all this way to explore the moon, and the most important thing is that we discovered the Earth."

> *The view of the Earth from the Moon fascinated me — a small disk, 240,000 miles away. It was hard to think that that little thing held so many problems, so many frustrations. Raging nationalistic interests, famines, wars, pestilence don't show from that distance.*
> *— Frank Borman, Apollo 8*

As Voyager 1 was approaching the edge of our Solar System, Carl Sagan convinced the team at NASA to rotate the probe and send one last photograph back. A photograph portraying the earth as a tiny blue dot contrasted against the emptiness of space.



This photograph wasn't taken for purely scientific reasons, but had a deeper significance which Sagan elaborated on in his book Pale Blue Dot:

> *There is perhaps no better a demonstration of the folly of human conceits than this distant image of our tiny world.*

Look again at that dot. That's here, that's home, that's us. On it everyone you love, everyone you know, everyone you ever heard of, every human being who ever was, lived out their lives. The Earth is a very small stage in a vast cosmic arena.

It's no coincidence that the word small is endemic to experiences of space. In all these quotes from astronauts, the word comes up time and time again. Compared against the vastness of space, all our quarrels, conceits and concerns fade away into insignificance.

It's for this reason, the Overview effect, that I am extremely excited about the prospects of Space Tourism. With more people viewing the Earth from afar, perhaps the world will gain a little more perspective, and a better sense of proportion. ■

---

Alex MacCaw is a JavaScript programmer, O'Reilly author and open source developer. He currently works at Stripe.

Reprinted with permission of the original author. First appeared in *hn.my/small* (alexmaccaw.com)

# MEET MANDRILL

By MailChimp

Mandrill is a new way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

MANDRILL.COM

# MEMSET® HOSTING

Rent your IT infrastructure from Memset and discover the incredible benefits of cloud computing.

## MINISERVER™ CLOUD COMPUTE

From £0.015p/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

## MEMSTORE™ CLOUD STORAGE

£0.07p/GByte/month or less
99.999999% object durability
99.995% availability guarantee
**RESTful API, FTP/SFTP and CDN Service**

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

CarbonNeutral® hosting

ISO 9001: Quality
ISO 14001: Environmental
ISO 27001: Security

Find out more about us at www.memset.com

or chat to our sales team on 0800 634 9270.