Don't End The Week With Nothing

by Patrick McKenzie

HACKERMONTHLY Issue 47 April 2014

Curator
Lim Cheng Soon

Contributors
Rick Webb
Patrick McKenzie
Tom Martin
Csaba Okrona
Padraig Brady
Hadi Hariri
Julia Evans
Steve Pear
James Long
Leo Babauta

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrators
Lorenz Ruwwe
Joel Benjamin

Ebook Conversion
Ashish Kumar Jha

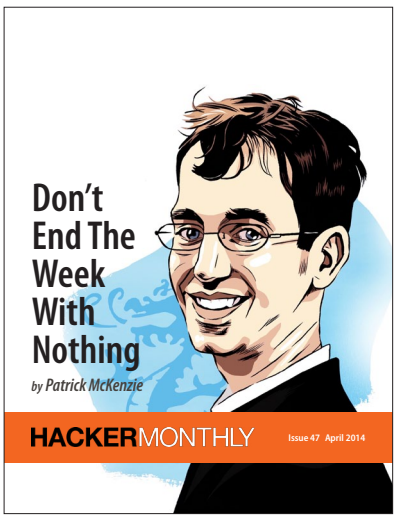
Printer
MagCloud

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 Illustration: Joel Benjamin

Contents

FEATURES

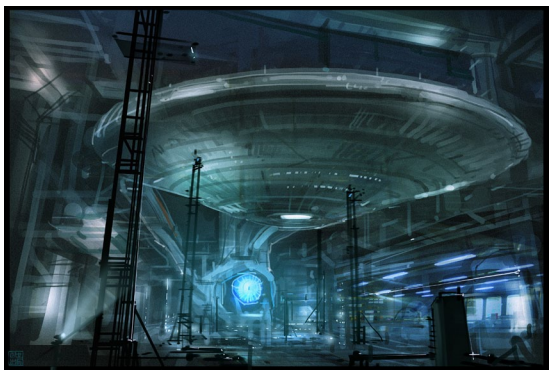


Illustration by Lorenz Hideyoshi Ruwwe

PROGRAMMING

SPECIAL

For links to Hacker News dicussions, visit *hackermonthly.com/issue-47*

The Economics of Star Trek

The Proto-Post Scarcity Economy

By RICK WEBB

When looking at the economics of Star Trek, there have been three broad approaches in the past:

1. Trying to shoehorn Star Trek's economics into the model of parecon. This is problematic because of the obviously hierarchical society of Starfleet, with Admirals, captains, commanders, chancellors, governors and whatnot, and the clear existence of a relatively strong Federation president, who is democratically elected. Plus we never once see a labor meeting, and it's pretty obvious personal freedom and enrichment are important to society.
2. Calling the Federation Communist, based on comments from Kirk in Star Trek IV on not having any money in the future and Picard's speech about the economics of the federation being significantly different than 21st century economics and people pursuing personal enrichment rather than the accumulation of wealth. The problem with this definition is it's lazy; just because they don't pursue the accumulation of wealth does not mean the Federation is communist. There is absolutely, obviously, still private property in the Federation: most obviously Joseph Sisko's restaurant in New Orleans and Chateau Picard, evidencing that not just small

possessions are allowed but that the land itself is still privately owned. One could argue that these aren't really Sisko and Picard's to own, but they are routinely referred to as "his" restaurant and vineyard so we gotta go with Occam's Razor here and assume they do, in fact, own them.

3. A sort-of guessing game based on the various mentions of Federation Credits and trying to glean the system from every single mention of money or payments within the series. This is always a pain in the ass, especially given the original series sometimes did things that were pretty out there according to later firmly established canon, and later firmly rejected by Roddenberry himself before his death. Additionally, many of the assumptions about Federation Credits seem iffy: are they really currency? Do they have to be? Are they scrip? Rations? We simply don't know. And in any case, trying to define the entire economy of the Federation — and perhaps even learning something from it — should be more than a matter of resolving obscure trivia references (though of course it's fun).

None of them seem correct. None of them seem realistic. And yes, let's go for realistic here, why not?

Let's take a different approach here.

What we know

Let's start with the facts.

The Federation is clearly not a centrally planned economy, and therefore obviously not communist. Individual freedom of choice is very obvious. Everyone chooses their careers, and there are many mentions of this throughout the series — witness every single time someone waxes nostalgic about why they chose to enter Starfleet. Witness Bashir going on about why he wanted to be a doctor instead of a tennis player. Witness Wesley dropping out of Starfleet. Witness Vash being an archeologist and Kasidy Yates being a cargo ship captain.

Private ownership still exists — the biggest examples, to me, are Sisko's restaurant and Chateau Picard, but many other examples abound from all the trinkets everyone owns in their quarters. Crusher's family owns a (haunted) cottage on some old-Scottish settlement planet. The Maquis routinely refer to "our land," which they presumably owned, and while an individual tribe may have collectively owned the land through a corporation, like the Alaska Native Land Claims Settlement Act, or through a co-op, they clearly "owned" the land, just like anyone else owned land, while the Federation was the superseding government that could give that territory away to another sovereign party, much like the ceding of the Sudetenland or Guam.

Any alternative situation (the government owning the land and renting it to the settlers?) is never alluded to and in any case the words stated ("our land") clearly indicate private ownership is still very much part of the cultural zeitgeist. Then we have J. J. Abram's Star Trek and it's pretty unlikely that, what? The Federation owned that shack Kirk grew up in, that sweet Corvette or that roadhouse bar? Those items sure looked privately owned. Some spaceships were privately owned. Finally, let's not forget Star Trek: Generations when Kirk says in the Nexus "This is my house. I sold it years ago."

Next: *The Federation is not true post scarcity economy*: famines routinely still exist, transportation lines are vital in moving around goods within the Federation. Transportation is a whole grey area in most post-scarcity economic works, at least the few I've read. The Federation might have enough food, but at any time some planet may well be starving or in need of medicine that needs to come from somewhere else.

It seems pretty clear cut that jobs are optional. They explicitly state on many occasions that the Federation is based on a philosophy of self-improvement and cultural enrichment, and in any case we sure do run into a lot of "artists" in the Federation. I particularly love those hippies in TOS. The Federation seems a bit like Williamsburg — a lot of artists who don't need to work. Or maybe more like the UK at the height of its social programs supporting artists. Let a million J. K. Rowlings bloom. It's a bit weird to me that we've never seen people who sit around and literally do nothing, but then why would we? And, of course, we've certainly seen more than a few societies that are all chilled out and not doing much (Risa, etc.).

Next: *The Federation doesn't use money*. This is basically absolute. Kirk says it in Star Trek IV. Picard says it several times. Quark mocks it to Rom. Roddenberry put it down as a hard and fast rule. No theory of Star Trek economics can be real while ignoring this fact. It has to be addressed. It is the basis of all confusion and, honestly, interest in figuring it out at all.

Money still exists, so do banks. Crusher buys fabric at Farpoint. DS9 makes mention of the Bank of Bolias, on a Federation planet. Nog loans Jake latinum.

We also know there exists such a thing as the Federation Credit. This presumably causes some confusion since they are routinely referred to like money (Kirk mentions that the Federation has invested 122,200 credits in Spock), and things are purchased for credits (Uhura buys a tribble, Quark occasionally accepts them at his bar).

This would seem to be a giant contradiction to the lack of existence of money. We'll get to that in a bit.

There is still a ruling class, or classes — it is not perfectly-egalitarian in a communist manner. We have admirals and presidents and governors and colony leaders. There are enlisted personnel in Starfleet and officers. Some are elected, some are appointed. Some Federation members were even hereditary nobilities.

There is still commerce (and even Vulcan commerce), trade, trading vessels, and, we can assume, corporations, in some form (though this may not be 100% definite — Dytallix is mined for the Federation. It isn't 100% clear it is in the Federation).

Some thought exercises

Let's do a couple thought exercises.

First: if you eat a meal at Sisko's Creole Kitchen, do you pay? It seems almost definite that you don't pay. If you paid, with anything, including Federation Credits, that would be money. You could barter, but it seems if the entire economy was a barter economy, we'd hear it. No, it seems almost certain that you go to eat at Sisko's, you don't pay, and Joseph Sisko doesn't pay for his supplies, and his suppliers probably don't pay for theirs.

Next: Can everyone have anything? Anything at all? Is the Federation a perfect post scarcity society? The answer seems almost certainly no. If you went to a replicator, or a dealer, or the Utopia Planatia Fleet Yards and asked for 10 million star ships, the answer would be no. More concretely, when the Borg attacked, and during the Dominion War, the Federation suffered from a serious starship shortage.

Next: Imagine there's some level of welfare benefits in every country, including America. That's easy. That's true. Imagine that, as the economy became more efficient and wealthy, the society could afford to give more money in welfare benefits, and chooses to do so. Next, imagine that this kept happening until society could afford to give the equivalent of something like $10 million US dollars at current value to every man, woman and child. And imagine that, over the time that took to happen, society got its shit together on education, health, and the dignity of labor. Imagine if that self-same society frowned upon the conspicuous display of consumption and there was a large amount of societal pressure, though not laws, on people that evolved them into not being obsessed with wealth. Is any of that so crazy? Is it impossible?

I think that is basically what's going on on Star Trek.

A Theory of Star Trek Economics

I believe the federation is a proto-post scarcity society evolved from democratic capitalism. It is, essentially, European socialist capitalism vastly expanded to the point where no one has to work unless they want to.

It is massively productive and efficient, allowing for the effective decoupling of labor and salary for the vast majority (but not all) of economic activity. The amount of welfare benefits available to all citizens is in excess of the needs of the citizens. Therefore, money is irrelevant to the lives of the citizenry, whether it exists or not. Resources are still accounted for and allocated in some manner, presumably by the amount of energy required to produce them (say Joules). And they are indeed credited to and debited from each citizen's "account." However, the average citizen doesn't even notice it, though the government does, and again, it is not measured in currency units — definitely not Federation Credits. There is some level of scarcity — the Federation cannot manufacture a million starships, for example. This massive accounting is done by the Federation government in the background (witness the authority of the Federation President over planetary power supplies).

Because the welfare benefit is so large, and social pressure is so strong against conspicuous consumption, the average citizen never pays any attention to the amounts allocated to them, because it's perpetually more than they need. But if they go crazy and try and purchase, say, 10 planets or 100 starships, the system simply says "no."

Citizens have no financial need to work, as their benefits are more than enough to provide a comfortable life, and there is, clearly, universal health care and education. The Federation has clearly taken the plunge to the other side of people's fears about European socialist capitalism: yes, some people might not work. So What? Good for them. We think most still will.

However, if they so choose they can also get a job. Many people do so for personal enrichment, societal pressure or through a desire to promote social welfare. Are those jobs paid? I would assume that yes, those jobs are "paid," in the sense that your energy allocation is increased in the system, though, again, your allocation is large enough that you wouldn't even really notice it. Why do I say this? The big challenge here is how does society get someone to do the menial jobs that cannot be done in an automated manner. Why would anyone? There are really only two options: there is some small, incremental increase in your hypothetical maximum consumption, thus appealing to the subconscious in some primal way, or massive societal pressure has ennobled those jobs in a way that we don't these days. I opt for the former

since it grounds everything in market economics, albeit on a bordering-on-infinitesimal manner, and that stands to reason, since that's how people talk in Star Trek. They talk about individual fulfillment, buying, selling, etc. No one was ever guilt-tripped into joining Starfleet, save by maybe their family.

There is almost zero mention of central planning. It's a capitalistic society, its benefits are just through the roof. Also, market economics = crowdsourced. That is, it's not centrally planned. It's democratic. It's the only mechanism we know of to allocate resources that isn't centrally planned. The alternative is that all allocations are done algorithmically through a computer and the economy is completely decoupled from market forces, but that's still basically central planning, and infinitely more complex than assuming there is still some semblance of market underpinning, much like we stayed on the gold standard for far longer than we needed to and we still have pennies even though we don't need them. It's a vestige of the past. It's the constitutional monarchy.

Either way, presumably, you take whatever job you want, and your benefits allocations are adjusted accordingly. But by and large you just don't care, because the base welfare allocation is more than enough. Some people might care, some people might still care about wealth, such as Carter Winston. More power to them. They can go try and be "rich" in some non-Federation-issued currency. But most people just don't care. After all, if you were effectively "wealthy" why would you take a job to become wealthy? It pretty much becomes the least likely reason to take a job.

So, behind the scenes there is a massive internal accounting and calculation going on — the economics still happen. They just aren't based on a currency unit, and people don't acquire things based upon a currency value. People just acquire things from replicators, from restaurants such as Sisko's or coffee shops like Cosimo's, or, presumably, get larger things from dealerships or factories. This could still be called "buying," as a throwback.

Two points here: first, the accounting is done in energy units, so that there is no need for currency. And why not? Resource allocation is mainly about energy anyhow, doubly so if it's only robots building most things. And secondly, if you never had money, never saw it, and it didn't physically exist to measure things, you'd pretty much tell people, like a certain 20th century oceanographer, that you don't have money in the 24th century, regardless of some automated accounting. This jibes with Federation people knowing what money is — because other societies have it — but saying they don't use it. Because they don't.

However, you could still buy and sell things. You could take a thing from a replicator and go to someone else and "buy" something else with it. Why couldn't you? It's a free society. It's essentially barter. Kirk may well have sold his house for a year's supply of Romulan ale.

Or Federation Credits.

It is tempting to argue here that the massive accounting system uses a unit called the Federation Credit, but I don't believe that's the case. If it were, the credit would be too much like money because a) accounting is done in it, b) it is issued by a governing body (like a fiat currency) and c) it is fungible, i.e. you can already buy things with it and if you could buy things with it AND a and b were true, it would pretty much be a currency. This would fly in the face of Roddenberry's absolute diktat that the Federation has no currency.

I'm going to make a bold new theory here. Federation Units are "Federation" the same way that American Cheese is American. It is simply descriptive. Currency was invented long before capitalism as a means to disintermediate trades: you wanted my grain, I didn't want your cows, I wanted farmer Ted's grapes. Rather than make every trade a 3, 4 or 5 way trade, we made a little certificate we all agreed was worth something to us and us only. This need would still occasionally crop up in the Federation, even without money. I believe the Federation Unit is a private currency, developed by third parties to facilitate complex trades or trades outside the Federation. I believe that the Federation Unit is not actually underwritten or issued by the Federation. I think it is more akin to the Calgary Dollar or the Chiemgauer. Or bitcoin. This would solve so many problems. It would make it unequivocally true that the Federation doesn't use money. It would give people a unit to use as reference when they say things are expensive. It would be a thing citizen's could acquire, if they wanted to, through barter originally, then allowing them to use them to purchase things (like Tribbles or Holosuites) from people who elected to take them, since taking them is optional (witness Quark's vacillations on whether he accepts them or not). It would make a nice proxy for talking about investment levels, such as when Kirk said how much the Federation had invested in Spock.

Foreign Reserves

Additionally, I believe that the Federation acts like any current sovereign nation state and holds foreign reserves of currencies of other nations. It's assumed that not all foreign trade is done through barter. The federation itself probably holds foreign reserves in foreign currency just as China holds US dollars and England keeps a reserve of Euros. Sisko at one point tells Quark he could have charged rent for the

bar, but he chose not to. Presumably that would have been paid in latinum. Presumably the Federation would have just held onto it as foreign reserves. All evidence, in fact, points to the fact that the Federation operates as a nation and uses foreign reserves exactly as we do now. The Chinese government holds US Dollars but you don't hear a Chinese person say "we use dollars." This is a bit confusing by the episode in which the Federation offers 1.5 million Federation Credits for use of the Barzanian wormhole, but it doesn't have to be contradictory. Federation Credits had value to the Barzanians, so the Federation could simply procure them from the issuer with its foreign reserves of other currencies at market rate.

The Individual Can Have Money

An individual of the Federation can procure latinum by barter for goods, labor or, presumably Federation Credits, if they had them. I assume that there's probably some black market value for Federation Credits just like any other currency, sovereign issued or not (you can buy a Lewes Pound on eBay right now for $7.98). Perhaps it's more legitimate and the Units are traded on a commodities exchange. It really doesn't matter. As a Federation Citizen I can have gold pressed latinum, Federation Credits, Frangs, Darseks, Isiks, Leks, or Quatloos in my wallet. I can have a wallet. I can buy things with Self Sealing Stem Bolts if I want. But none of that is in conflict with the fact that the Federation has no unit of currency, has no money, and my society is predominantly concerned with societal good and self-improvement.

Then there's the matter of Quark's bar. What's up with that? He never seems to charge anyone for drinks, but is obsessed with money, and you can buy holosuites in latinum or Federation Credits, and you can bet on the Dabo table with Latinum. At first I thought there was a whole complex thing where Quark doesn't charge Starfleet personnel because he made the mental calculation it was cheaper to give them drinks for free and keep accepting free rent from Sisko, but then I realized that doesn't really work because he charges them for the Holosuites and Dabo tables. Then I realized: Quark's is like any other casino. The drinks are free: they are a loss leader against the higher profits of the Dabo Table and Holosuites.

The Proto Post Scarcity Economy

The thing I love most about this theory is that it seems plausible for our future. Tom Paris said that a new world economy takes shape in the 22nd century. That might be a smidge optimistic but we already have a world economy, in one sense, so the new one could be something only incrementally different from this one. Money went the way of the dinosaur, he said, and Ft. Knox

was turned into a museum. Most of us are already off the gold standard, and it's certainly not inconceivable in another 180 years we don't use paper money at all, and a single currency has dominated the planet — the Dollar is already close — and it slowly fades into the background.

From there, perhaps a cultural shift takes place as we realize that "everyone in a job" isn't the same as a full economy, and we start to look for models beyond capitalism that aren't all communist hoo-ha.

I sort of love that Star Trek forces us to think about a society that has no money but still operates with individual freedom and without central planning. I love that democracy is still in place. I love that people can still buy and sell things. It's real. It's a more realistic vision of post-capitalism than I have seen anywhere else. Scarcity still exists to some extent, but society produces more than enough to satisfy everyone's basic needs. The frustrating thing is that we pretty much do that now, we just don't allocate properly. And allocating properly cannot be done via central planning.

The only real "out there" requirement in all of this is a governmental layer higher than the nation, and indeed, higher than the planet. This doesn't seem insane, I suppose, if we were to suddenly find ourselves not alone in the universe. And indeed we already have some measure of international government now. Moreover, the Federation clearly adheres to the "laws made as close to home as possible" routine, since as far as we can tell the Federation president really only has authority over Starfleet, Foreign Relations, power allocation, and accounting. Virtually every other law we encounter in the Federation happens at the individual planet or colony level.

It's interesting to me because these are things we're going to have to reckon with, I believe, in my lifetime. If robots do all the dirty work, and the US is hugely rich, does every single person really need a job? Are we going to let all of that money pile up in the 0.1% ruling elite, or can it be distributed to everyone? Does wealth being distributed to the people in an equal manner mean communism absolutely? Of course it doesn't. The US isn't communist. The UK isn't communist. Denmark isn't communist. What happens when the surplus is more than enough? ■

Rick Webb is co-founder of The Barbarian Group and served as its COO. Since then, he's worked at Tumblr and Soundcloud. He is currently a venture partner at Quotidian Ventures, an early-stage fund operating in New York City.

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

Illustration by Lorenz Hideyoshi Ruwwe.

Don't End The Week With Nothing

By PATRICK MCKENZIE

Usually I concentrate more on the needs of established software businesses, but recently I've been asked for some advice by people who are still in the trenches working at a traditional day job.

There's absolutely nothing wrong with day jobs. Most people have them. They're an honest living. Some people really enjoy the particular one that they have. If your day job is right for you, that is wonderful and I will not second-guess your decision.

Many people nurse dreams of entrepreneurship because their day job is not quite right for them. Here's my story.

I used to be a salaryman at a Japanese mega corporation. The social expectation is that the company insulates the employee from all risk, and in turn, the employee swears themselves body and soul to the company.

I worked 70 to 90 hour weeks for three years. This isn't particularly out of the ordinary for white collar employees in Japan. It didn't strike me while I was a salaryman that I was going to continue doing it until retirement, largely because the amount of work was killing me, and I worked after those hours on my own projects.

Somebody asked me recently how I managed to stay motivated to work for the 91st through 95th hours every week. Answer: because I wanted to end the week with something.

Applied Capitalism For Fun And Profit

I'm a capitalist. A friend of mine is a devoted Marxist. I think we mutually agree that, considering any particular employee, it is in that employee's personal interest to stop selling hours of labor and start renting access to his accumulated capital as soon as humanly possible.

I don't mean just monetary capital — having $100,000 in your 401k is awesome but that's not the type which is really interesting to me, simply because rates of return on that sort of capital are so low. There are many types of capital that are no less real just because you can't conveniently reduce them to a number.

Human capital: the skills you've built up over time and the value you're able to create as a result of them.

Social capital: the ability to call on someone who trusts you and have them do something in your interest, like e.g. recommend you to a job.

Reputational capital: the way your name rings out in rooms you aren't even in, simply when your topic of expertise comes up. (Hopefully in a good way!)

A lot of day jobs structurally inhibit capital formation. If I were a Marxist I'd say "And this is an intended consequence of Capital's desire to keep Labor subservient to it," but I honestly think it's true even without anybody needing to twirl their mustache.

There's a great line from Jack Welch to the effect of "You work for a week, collect your paycheck on Friday, and then you and the company are even." Corporate America has embraced it with a vengeance. I'm too young to remember an America where "company loyalty" wasn't a punch line.

If company loyalty were a bankable proposition (and it might still be at some places — I know a smaller company or two where "we treat our employees like family" means exactly what it says on the tin), you'd get a wee bit of capital every week you worked. That's one more week towards your boss' good impression of you. One more week towards your pension. One more week towards that gold watch.

Japanese salarymen still have that sort of arrangement.

At some point at my ex-employer, I realized that I couldn't possibly work at a salaryman job until retirement, because it was going to be the death of me. (I won't belabor that period of my life because it was pretty rough, but suffice it to say if you pull 6 months of 90 hour weeks, towards the end of it the periodic blackouts start to get a little distressing.)

Once I came to the conclusion that I'd probably quit, and therefore discounted the till-your-death-do-us-part slow accumulation of firm-specific capital, I realized something which is fundamentally true of a lot of day jobs. Nothing I did at the job mattered, in the long run.

Sure, in the short run, I was writing XML files and Java classes which, knock on wood, successfully let my employers ship an examination management system to their client (a major university). I was a really effective Turing machine which accepted emails and tickets as input and delivered (occasionally) working code and Excel files as output. But no matter how much I spun, nothing about my situation ever changed. I worked my week, got to the end of it, and had nothing to show. The next week there would be more emails and more tickets, exactly like the week before. The week after that would be more of the same. And absolutely nothing about my life would change. I'd end the week with nothing.

Don't end the week with nothing. Prefer to work on things you can show. Prefer to work where people can see you. Prefer to work on things you can own.

Prefer Working On Things You Can Show

One of the reasons developers have embraced OSS so much is because it gives you portable capital between companies: if your work is sitting on GitHub, even if you leave one job, you can take it with you to your next job. Previously this happened pretty widely but generally under the table. (Is there any programmer who does not have a snippets folder or their own private library for scratching that one particular itch?) One of the great wrinkles that OSS throws into this is that OSS is public by default, and that's game changing.

Why? Because when your work is in public, you can show it to people. That's often the best way to demonstrate that you're capable of doing work like it.

Telling people you can do great work is easy: any idiot can do it, and many idiots do. Having people tell people you do great work is an improvement. It suffers because measuring individual productivity on a team effort is famously difficult, and people often have no particular reason to trust the representations of the people doing the endorsements.

(Quick: if you had credible evidence that a mid-level engineering manager at a company you've never heard of in Nagoya thought I was a really effective employee, would that make you markedly more likely to hire me? Right, without the context of knowing him, that recommendation is almost useless.)

Work you can show off, though, is prima facie evidence of your skills. After your portfolio includes it, your ability to sell your skills gets markedly better. Given that most people's net worth is almost 100% invested in their personal capital (i.e. if you're a young engineer the net present value of all future salary absolutely swamps everything in your bank account), this is a fairly radical improvement in your present situation for not a very radical change in how you go about things.

Thus my first piece of advice: if you have the choice between multiple jobs, all else being equal, pick the one where you are able to show what you've worked on. This could mean working on a language stack where work byproducts are customarily OSSed (e.g. Rails) versus one which isn't (e.g. C#). This could mean working on particular projects within the organization which like external visibility (e.g. Android) rather than projects which don't (e.g. AdWords plumbing — presumably Google will pay you a lot of money to do that, but consider it compensation for not being able to talk about it). This could mean working in industries which default to being open rather than those which default to being closed.

OSS isn't nearly the only way to be able to show what you've worked on. In the creative industries, where the end product is customer-visible, people keep very close eye on whose name ends up in the credits. Academics spend lots of time worrying about citation counts and directed graphs.

More prosaically, establish an expectation early that you're simply going to talk about what you're doing. I think at Fog Creek / Stack Exchange they call this "producing artifacts" — conference presentations, blog posts, OSSed software, and the like, centered around the work. Even at very open companies there exists a lot of secret sauce, but most of the valuable work of the company is not particularly sensitive, and much of it has widely generalizable lessons. Write about those lessons as you learn them. If at all possible, publish what you write. Even if it is published to an audience of no one, you will be able to point people back to it later.

Some of my most effective writing in terms of career growth was back in 2006 through 2008, when I was struggling through not understanding anything I was doing, and where I — quite literally — had less readers than my younger brother's blog on writing superhero novels. Why was toiling in Internet-obscurity still valuable? Because I was able to point to particular experiments that I started in 2008, and then point to the follow-ups in 2009 and 2010, which showed those experiments were really successful. The failures and false starts aren't extremely interesting to most people, but having some successes under your belt credibly demonstrates that you're capable of either reproducing them in the future or experimenting your way to new successes in your new environment.

If you cannot build things you can show at work, you should build things you can show outside of work. Companies in our industry are gradually becoming more reasonable about IP assignment clauses — there's less of the "we own everything you think of at any point in your employment" nonsense these days. Even at my very straight-laced Japanese megacorp, they were willing to write an exception into the employment contract for a) OSS work that I did outside of company hours and b) Bingo Card Creator. I offered them this in exchange: "If you let me continue working on these, I'm going to learn lots of skills which I can put to the use of the company. Normally you invest lots of money sending engineers to conferences and professional training. This is even better for you: I'll learn more with no operating expenditure and no decrease in billing efficiency." That's an offer you can make to substantially any employer.

I prefer being upfront with people rather than doing the "It is easier to ask for forgiveness than ask for permission" route a lot of folks suggest. Sure, you can just roll the dice and pretend your employer is unlikely to notice your side project. Unfortunately, the odds of them noticing your side project go up sharply if the side project is ever successful, and then your lack of forthrightness about it give you unbounded liability extending far into the future. Just ask. The worst they can say is no.

You might consider asking in the context of a more general compensation discussion than just "Hey boss, can I work on OSS?" That way, if they say "No side projects," you'll say "OK, in lieu of the side projects, I'll need more money." It's easier to be sticklers for the stock agreements when there's absolutely no cost to the company to insist on the usual boilerplate, but minor concessions on the boilerplate are often easier than concessions on things which actually appear on the company's books.

Prefer To Work Where People Can See You

I used to phrase this as "work in public," but when people think about folks who work in public, they think of rock stars and figure "Well, I'll never be a rock star."

Vanishingly few people in our industry have the profile of rock stars. They can still have substantial profile among the audience of "people professionally relevant to them." That might be as tightly scoped as "people with hiring authority for front-end developers in my metro area," which might be a set of, what, a couple of dozen folks?

How do you develop that profile? I'd suggest, all things being equal, working at places and on projects which have above-average visibility.

Many engineering projects are deep in the bowels of late-stage industrial capitalism. Then there's writing the Facebook mobile app. I have no clue what engineers actually worked on the Facebook mobile app, but I'm betting that if I were a Silicon Valley hiring manager in iOS or Android development I'd a) know their names and b) have them at or near the top of my personal poach list.

Side note: A poach list is my informal name for "people who, if I had infinite money and they had no other commitments, I'd hire to work on a particular project." I have several mental poach lists — the best people I know on Rails programming, on A/B testing, on

writing email, etc. When people ask me for advice on what to do about those topics, I often say "You know who is really great at this? `<%= poach_list.pop() %>` You cannot possibly waste your time taking them out to coffee." Brokering coffee dates cannot possibly work out poorly for the people who go to them. (My interest? Helping people out is fun, and — funny enough — people often seem to remember when you get them a job or a key employee.)

You don't have to optimize for "sexy" projects. You know, sexy projects: I don't know how to describe them but I know it when I see it. Most engineering work isn't intrinsically sexy. I would, however, **optimize for impact and visibility**.

Don't try to make a career out of optimizing the SQL queries to display a preference page on a line of business app at a company that no one has ever heard of. That is not the straightforward path to having other people learn you are capable of doing meaningful work. Instead, work at higher profile companies/organizations — AmaGooFaceSoft, startups or small companies with anomalously high profile (locally, nationally, whatever), or in positions where by your nature you're exposed to lots of people.

I have a few friends who are developer evangelists, which is a funny job created at API companies where your brief is basically "Go demo our product to a group of developers. Now, do that again, every day, for the next several years." Sentiment on the actual job is decidedly mixed.

An observation: every developer evangelist I know goes into a much better job right after they quit being an evangelist. This is not true of other engineering jobs with checkered reputations, like e.g. The Build Guy. Why do developer evangelists get upgrades but The Build Guy(s) do not? My bet is because evangelists literally spent years meeting thousands of people and showing them "Hey, I'm going to live code in front of you while also making my employers fat stacks of money. You run a company and could use both engineers and money. You should probably remember my name, you know, just in case." The Build Guy(s) suffers in under-appreciated solitude, except when maven bottoms out or RubyGems goes down and it is somehow The Build Guy's fault.

If you cannot gain exposure at your day job, try to get some exposure outside of it. Network actively. Go to local meetups of technical folks, but also go to the (often separate) events where the business side of your industry talks shops. Speak at conferences. Take the things you have created (see above) and actively show them to people to

solicit feedback. You don't have to have an audience of thousands for an audience to be worthwhile — for landing a new job, having an audience of one hiring manager is a darn sight better than having no audience at all. Blog and collect an email list. It's old and hackneyed advice but it freaking works, particularly when you can compound improvements over years.

Amy Hoy has a great metaphor for this: "stacking the bricks." [hn.my/bricks] Seen from the outside, you might say "That person with an impressive career? It's like they have a sheer wall made out of awesome. I could not hope to ever have a wall like that." Seen from the inside, it looks like one day of delivering a single good conference talk, a few weeks spent writing an OSS library, another day writing the definitive blog post on getting multiple Ruby versions playing together, a few months shipping a product used by many people, an hour recording a podcast. **Brick by brick, stone by stone, the wall gets higher.**

Prefer To Work On Things You Can Keep

The employer/employee relationship is generally "You give us an hour and, in return, we give you some consideration for that hour." As an employee, you very rarely get to keep hours, bank them against the future, or have them redound to your benefit years later.

I'm not generally a fan of the Silicon Valley model, but I'll say this in their defense: widespread employee ownership of the enterprise is one of the single best innovations in the history of capitalism. Non-managerial employees own plus or minus 20% of Twitter, Facebook, etc. They own plus or minus "rounding error" of almost all other publicly traded companies, with very rare exceptions.

I think that's an improvement on the "no shared stake" model of employment, but I don't think it is the last word in it. For one thing, it over-concentrates employee wealth with one company. As an employee, your short-term cash flow generation is tied to the continued health of your employer. If a large portion of your net worth is tied to their stock, you're magnifying the impact of a secular or firm-specific shock should one occur. (This is, relatedly, why I'm not a fan of buying the stock of an employer in a company-sponsored DRIP or IRA. You've got plenty of exposure to their future already without buying more of it with your own money.)

The explicit understanding among professional investors is that 90% of all shares of early-stage startups are worthless. It seems more than a little self-serving for professional investors to tell employees "While our general partners would laugh us out of the room if we suggested betting the entire fund on a single investment, even if we thought it was a sure thing, you are going the be the lucky ones and you should certainly have 99% of your net worth tied up in the illiquid shares of one particular company."

So if not hard assets directly awarded by employers, then what?

Well, obviously, sock away money like every financial advisor ever will tell you to. (Here's everything you need to know: buy broad market index funds in your tax-advantaged accounts. If that sounds too complicated, get a Vanguard target retirement fund where the number most closely matches the approximate time you'll retire.)

There's another, harder option with higher returns: the side project. You can "buy" them with sweat equity, one bead at a time. They provide you with many benefits, including the direct financial benefits (if you sell things to people for money, you get money, which can be useful), the compounded benefits of investing the financial benefits (my first $2,000 from Bingo Card Creator turned into Chipotle stock at an average price of $50 a share — don't buy stocks, buy index funds, but that decision worked out pretty decently for me).

There's also intangible — but no less real — benefits to having an artifact which is yours. This is one reason why, while I love OSS, I would suggest people not immediately throw their OSS on GitHub. That makes it very easy for developers to consume your code, but it does not make it easy for you to show the impact of that code to other people, particularly to non-technical stakeholders. To the extent that people's lives are meaningfully improved by your code, the credit (and observable citations) often goes to GitHub rather than going to you. If you're going to spend weeks or months of time writing meaningful OSS libraries, make a stand-alone web presence for them.

Example: my A/Bingo was once probably the best option for Rails A/B testing, by dint of being the only serious option for Rails A/B testing. It is a little old in the tooth now, but being The A/B Testing Guy got me several consulting gigs. The effort to make documentation, a quick start guide, a logo, and a branded web presence beats the heck out of having a junior engineer at a potential client just git clone my GitHub URL and never have my work exposed to a decision maker there at all.

If you want to learn more about the actual mechanics of building a side project, my blog covers it in a lot of detail. For a much briefer overview of it, I really recommend Jason Cohen's presentation at MicroConf 2013 [hn.my/microconf]. His formula is "Predictable acquisition of recurring revenue with an annual pre-pay option with a product which solves a demonstrable, enduring pain point for a business." That idea is developed at the above link for an hour, and a lot of the advice given is specific and wildly actionable. I highly recommend it.

Consumption Is Sometimes Valuable, But Creation Moves You Forward

I'll close with my usual advice to peers: reading this was valuable (knock on wood). Watching Jason's video is valuable. Rolling up your sleeves and actually shipping something is much, much more valuable. If you take no other advice from me ever, ship something. You'll learn more shipping a failure than you'll learn from reading about a thousand successes. And you stand an excellent chance of shipping a success — people greatly overestimate how difficult this is.

Just don't end the week with nothing.■

Patrick McKenzie is a small software developer. He made Bingo Card Creator and Appointment Reminder. He blogs at *kalzumeus.com/blog* and teaches people how to sell more software at *training.kalzumeus.com*

Reprinted with permission of the original author.
First appeared in *hn.my/week* (kalzumeus.com)
Illustration by Joel Benjamin.

A Quick Look at the Redis Source Code

By TOM MARTIN

HAVING BEEN WRITING predominantly Java and Scala for the last 7 years, my C skills are pretty rusty. In fact they're practically nonexistent. Apart from the occasional hack I've not had the occasion to write much C since University. There is a widely held opinion that reading other people's code is an excellent way to learn, particularly if those people are experts or if the codebase is held in high regard in terms of its quality. I've decided to take a look at one such codebase.

Redis [redis.io] is an open source data structure server written in ANSI C. "Data structure server" is another way of saying really, really neat key-value store. Not only can you store simple values like strings against keys but also hashes (or maps, or dicts even), lists, sets and sorted sets. We use Redis a lot at Top10 [top10.com], mostly for indexing hotels in (near) real time depending on their availability and price for the dates the user is searching on. I've also discovered that it has a pretty easily understandable code base, even for a C newbie like myself. The code is cleanly written, relatively small (around 45,000 lines of code), mostly single threaded, and with few dependencies. The dependencies are all included with the source making building it as simple as cloning the repo and typing make.

I decided to dive straight in to the code by adding a new command to Redis. Something simple that will give me an idea of how Redis handles a command and dispatches a response. A command `rand` that accepts a single integer argument `max` and returns a random integer between 0 and `max` (exclusively). Not an ideal use of a key value store but implementing it should be instructive. I certainly won't be submitting a pull request!

Disclaimer — as mentioned before I'm by no means an expert in C so take all the code and interpretation of code here on those terms. Also I'm linking to the unstable branch of Redis on github so the links may be just that — unstable. You'll probably get more out of this post if you clone the Redis source yourself and follow along in your favorite editor, particularly if you compile and run the code changes found here.

The command table is found near the very top of `src/redis.c`. It is an array of instances of the `redisCommand` struct. `redisCommand` is defined in `src/redis.h` but there's a very handy block comment explaining each of its fields above the declaration of `redisCommandTable`. Here is the definition of the `get` command:

```
{"get",getCommand,2,"r",0,NULL,1,1,1,0,0},
```

The first field "`get`" is the name of the command. The second is a pointer to the function that implements the command (you can see the implementation in `t_string.c`).

The third field is the command's arity (number of arguments it accepts). Specifying this means the command lookup and execution code can pre-validate a request before passing control by calling the function pointer. This reduces the error handling code necessary in each of the command functions. The argument count appears to include the name of the command itself so the `get` command accepts two arguments: its name and the name of the key whose value should be fetched.

The fourth field, set to "r", is specifying that the command is read only and doesn't modify any keys' value or state. There are a whole bunch of one-letter flags that you can specify in this string that are explained in detail in the nearby block comment. The field following this string should always be set to zero, and will be computed later. It's simply a bitmask representation of the information implied by the string.

The sixth field is `NULL` because it is only necessary when you need complex logic to tell Redis which arguments of the command are actually keys. A key implies a reference to a value stored in Redis as opposed to simple value parameters such as our `max` argument. This allows Redis to extract the values of the keys (and check that they exist) before calling the command implementation. If this field were used it would be a pointer to a function that would return an integer array of argument indexes (`zunionInterGetKeys` in `db.c` is an example of this). In the case of the get command though (and most other commands) this information can be conveyed with the next 3 integer fields. There is only one argument to getCommand and it is a key. Therefore the first argument that is a key is at index 1, the last argument that is a key

is at index 1, and the step increment to find all the keys is; 1,1,1.

The last two fields of a `redisCommand` represent metrics about the command, are set by Redis and should always be initialized to 0.

Let's add our `rand` command to the bottom of the table:

```
{"rand",randCommand,2,"rRl",0,NULL,0,0,0,0,0}
```

The command is called "`rand`", `randCommand` is the pointer to the implementation (not implemented yet) and it takes 2 arguments (the name and `max`). As for the flags — it's read only (`r`), returns random, non-deterministic output (`R`) and can be called while Redis is still loading the database (`l`). There are no key arguments.

The next step is to add the `randCommand` function prototype to `src/redis.h`. A Redis command implementation takes one argument, a `redisClient` struct that represents the command arguments but can also be used to send the response to the actual client:

```
void randCommand(redisClient *c);
```

This prototype ought to be placed in `src/redis.h` near the all the other command prototypes. Grepping for this line:

```
/* Commands prototypes */
```

will help you find where.

Let's add an empty implementation to `src/redis.c`:

```
void randCommand(redisClient *c)
{
}
```

I added mine near to the `infoCommand` definition. Now let's run make

```
> make
```

and run the server we've built (hint: if you usually have an instance of Redis running locally now would be a good time to stop it):

```
> src/redis-server
```

And let's run a Redis client in another terminal and try out our command:

```
> redis-cli
```

First let's try out the error handling:

```
redis 127.0.0.1:6379> rand
(error) ERR wrong number of arguments for 'rand' command
```

Good to see the arity checking working. Let's specify an argument this time:

```
redis 127.0.0.1:6379> rand 1
```

… and Redis hangs. I suppose that should have been expected given that we're not responding with anything from the `randCommand` function. Let's ctrl-c the server and get back to the source.

We want to respond with a number so I dug around looking for an example of how to do that and found the `zcardCommand` in `src/t_zset.c`. This command uses `addReplyLongLong` to reply to the client with a response that is a 64-bit integer (a long long). Let's try that:

```
void randCommand(redisClient *c) {
    addReplyLongLong(c,3);
}
```

Now when we `make` again and run the command:

```
redis 127.0.0.1:6379> rand 1
(integer) 3
 redis 127.0.0.1:6379> rand 2
(integer) 3
 redis 127.0.0.1:6379> rand 3
(integer) 3
```

OK, so it's not very random but it's a start. Let's parse our `max` argument from the command now and return a random value limited by `max`:

```
void randCommand(redisClient *c) {
    long max;
     if
(getLongFromObjectOrReply(c,c-
>argv[1],&max,NULL) != REDIS_OK)
        return;

    addReplyLongLong(c,random() %
max);
}
```

Whilst Redis uses primitive types and C strings throughout the codebase, it also has its own internal object system for representing strings, longs and more complex types in a more generic fashion. An example of Redis's use of these objects is the representation of each command's arguments. Each command argument is stored as a Redis object in the argv array on the `redisClient` instance `c`. To get a Redis object as a long I found an example in the getrangeCommand function in `src/t_string.c` that uses the `getLongFromObjectOrReply` function from `src/object.c`.

`getLongFromObjectOrReply` takes a `redisClient` instance, checks that the object in its second parameter is an object that represents a long, places the value of that long at the pointer specified by its third parameter and returns `REDIS_OK`. If the argument is not a long (or overflows) it will return `REDIS_ERR`. The beauty of this method is that if we receive `REDIS_ERR` we can just return from our `randCommand` function as any necessary error response will have already been sent to the client. Let's try out our command again:

```
redis 127.0.0.1:6379> rand 10
(integer) 9
redis 127.0.0.1:6379> rand
notanumber
(error) ERR value is not an inte-
ger or out of range
redis 127.0.0.1:6379> rand 10
```

```
(integer) 3
redis 127.0.0.1:6379> rand 10
(integer) 1
redis 127.0.0.1:6379> rand 100
(integer) 43
redis 127.0.0.1:6379> rand 100
(integer) 55
redis 127.0.0.1:6379> rand 100
(integer) 86
```

Looks pretty good! `rand` is an entirely pointless command but I learned quite a bit about Redis from implementing it and I hope you did too by following along. ■

Tom is a developer from London. He mostly writes Scala code for Space Ape Games. [spaceapegames.com]

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

Git Tips From the Trenches

By CSABA OKRONA

After a few years with git everyone has their own bag o' tricks — a collection of bash aliases, one liners and habits that make his daily work easier.

I've gathered a handful of these with varying complexity hoping that it can give a boost to you. I will not cover git or VCS basics at all; I'm assuming you're already a git-addict.

So fire up your favorite text editor and bear with me.

Check which branches are merged

After a while if you branch a lot you'll see your `git branch -a` output is polluted like hell (if you haven't cleaned up). It's all the more true if you're in a team. So, from time to time you'll do the Big Spring Cleaning only to find it hard to remember which branch you can delete and which you shouldn't. Well, just check out your mainline branch (usually master) and:

```
$ git checkout master
$ git branch --merged
```

to see all the branches that have already been merged to the current branch (master in this case).

You can do the opposite of course:

```
$ git branch --no-merged
```

How about deleting those obsolete branches right away?

```
$ git branch --merged | xargs git branch -d
```

Alternative: use GitHub's Pull request UI if you've been a good sport and always used pull requests.

Find something in your entire git history

Sometimes you find yourself in the situation that you're looking for a specific line of code that you don't find with plain old grep — maybe someone deleted or changed it with a commit. You remember some parts of it but have no idea where and when you committed it. Fortunately git has your back on this. Let's fetch all commits ever and then use git's internal grep subcommand to look for your string:

```
$ git rev-list --all | xargs git
grep '<YOUR REGEX>'
$ git rev-list --all | xargs git
grep -F '<YOUR STRING>'  # if you
don't want to use regex
```

Fetch a file from another branch without changing your current branch

Local cherry-picking. Gotta love it. Imagine you're experimenting on your current branch and you suddenly realize you need a file from the oh-so-distant branch. What do you do? Yeah, you can stash, git checkout, etc., but there's an easier way to merge a single file in your current branch from another:

```
$ git checkout <OTHER_BRANCH>
-- path/to/file
```

See which branches had the latest commits

Could also be useful for a spring cleaning — checking how "old" those yet unmerged branches are. Let's find out which branch hadn't been committed to in the last decade. Git has a nice subcommand, "for-each-ref" which can print information for each ref (duh) — the thing is that you can both customize the output format and sort!

```
$ git for-each-ref --sort=-commit-
terdate --format='%(refname:short)
%(committerdate:short)'
```

It will output branches and tags, too.

This deserves an alias, don't you think?

```
$ git config --global alias.
springcleaning "for-each-
ref --sort=-committerdate
--format='%(refname:short)
%(committerdate:short)'"
```

Making typos?

Git can autocorrect you.

```
$ git config --global help.autocor-
rect 1
$ git dffi
WARNING: You called a Git command
named 'dffi', which does not exist.
Continuing under the assumption
that you meant 'diff' in 0.1 sec-
onds automatically...
```

Autocomplete, anyone?

If you download this file [hn.my/gitcomplete] and modify your `.bash_profile` by adding:

```
source ~/.git-completion.bash
```

Git will now autocomplete your partial command if you press TAB. Neat.

Hate remnant whitespace?

Let git strip it for you. Use the mighty `.gitattributes` file in the root of your project and say in it:

```
* filter=stripWhitespace
```

Or say you don't want this for all files (*), only scala sources:

```
*.scala filter=stripWhitespace
```

But the filter is not defined yet, so chop-chop:

```
$ git config filter.stripWhitespace.
clean strip_whitespace
```

(Actually there are two types of filters: clean and smudge. Clean runs right before pushing, smudge is run right after pulling)

We still have to define what strip_whitespace is, so create a script on your PATH and of course make it executable:

```
#!/usr/bin/env ruby
STDIN.readlines.each do |line|
  puts line.rstrip
end
```

You could also do this as a pre-commit hook, of course.

Recovering lost data

The rule of thumb is that if you lost data but committed/pushed it somewhere, you're probably able to recover it. There are basically two ways:

reflog

Any change you make that affects a branch is recorded in the reflog. See:

```
$ git log -g
commit be5de4244c1ef863e454e3fb-
7765c7e0559a6938
Reflog: HEAD@{0} (Csaba Okrona
<xxx@xx.xx>)
Reflog message: checkout: moving
from master to master
Author: Robin Ward <xxx@xx.xx>
Date:   Fri Nov 8 15:05:14 2013
-0500

    FIX: Pinned posts were
not displaying at the top of
categories.
```

If you see your lost commit(s) there, just do a simple:

```
$ git branch my_lost_data [SHA-1]
```

Where SHA-1 is the hash after the "commit" part. Now merge your lost data into your current branch:

```
$ git merge my_lost_data
```

git-fsck

```
$ git fsck --full
```

This gives you all the objects that aren't referenced by any other object (orphans). You can fetch the SHA-1 hash and do the same dance as above.

A nicer, one-line log

Get a color-coded, one-line-per-commit log showing branches and tags:

```
$ git log --oneline --decorate
355459b Fix more typos in server
locale
b95e74b Merge pull request #1627
from awesomerobot/master
40aa62f adding highlight & fade to
linked topic
15c29fd (tag: v0.9.7.3, tag:
latest-release) Version bump to
v0.9.7.3
c753a3c We shouldn't be matching
on the `created_at` field. Causes
tests to randomly fail.
dbd2332 Public user profile page
shows if the user is suspended and
why.
```

Highlight word changes in diff

Bored of the full-line highlights? This only highlights the changed words, nicely inline. Try:

```
$ git diff --word-diff
```

A shorter, pro git status

Showing only the important things.

```
$ git status -sb
## master...origin/master
?? _posts/2014-02-01-git-tips-the-
trenches.md
?? images/git-beginner-share.png
?? images/git-beginner.jpg
```

Bored of setting up tracking branches by hand?

Make git do this by default:

```
$ git config --global push.default
tracking
```

This sets up the link to the remote if it exists with the same branch name when you push.

Pull with rebase, not merge

To avoid those nasty merge commits all around.

```
$ git pull --rebase
```

Or do it automatically for any branch you'd like:

```
$ git config branch.master.rebase
true
```

Or for all branches:

```
$ git config --global branch.auto-
setuprebase always
```

Find out which branch has a specific change

```
$ git branch --contains [SHA-1]
```

If you want to include remote tracking branches, add "-a".

Check which changes from a branch are already upstream

```
$ git cherry -v master
```

Show the last commit with matching message

```
$ git show :/regex
```

Write notes for commits

```
$ git notes add
```

You can share them by pushing — for more see *hn.my/gitnotes*

More cautious git blame

Before you play the blame game, make sure you check you're right with:

```
$ git blame -w
# ignores white space
$ git blame -M
# ignores moving text
$ git blame -C
# ignores moving text into other
# files ■
```

Csaba loves building things and is always seeking new challenges and smart people. This thriving led him to Prezi where he's doing backend and frontend web development. As a process freak he is really keen on development methodologies and is a huge fan of Kanban. Prior to joining Prezi, Csaba practiced full-stack development and IT management as the CTO of Árukereső in Budapest.

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

Common Shell Script Mistakes

By PÁDRAIG BRADY

I've written a few shell scripts [pixelbeat.org/scripts] in my time and have read many more, and I see the same issues cropping up again and again (unfortunately even in my own scripts sometimes).

While there are lots of shell programming pitfalls, at least the interpreter will tell you immediately about them. The mistakes I describe below, generally mean that your script will run fine now, but if the data changes or you move your script to another system, then you may have problems.

I think part of the reason shell scripts tend to have lots of issues is that commonly one doesn't learn shell scripting like "traditional" programming languages. Instead, scripts tend to evolve from existing interactive command line use, or are based on existing scripts which themselves have propagated the limitations of ancient shell script interpreters.

It's definitely worth spending the relatively small amount of time required to learn the shell script language correctly, if one uses Linux/BSD/Mac OS X desktops or servers, where it is commonly used.

Inappropriate use

Shell is the main domain specific language designed to manipulate the UNIX abstractions for data and logic, i.e., files and processes. So in addition to being useful at the command line, its use permeates any UNIX system. Correspondingly, please be wary of writing scripts that deviate from these abstractions and have significant data manipulation in the shell process itself. While flexible, shell is not designed as a general purpose language and becomes unwieldy when not leveraging the various UNIX tools effectively. A good knowledge of the various UNIX tools goes hand in hand with effective shell programming.

Stylistic issues

First I'll mention some ways to clean up shell scripts without changing their functionality. Note: I use a shortcut form of the conditional operator below (and in my shell scripts), when doing simple conditional operations, as it's much more concise. So I use `[ "$var" = "find" ] && echo "found"` instead of the equivalent:

```
if [ "$var" = "find" ]; then
  echo "found"
fi
```

[x"$var" = x"find"] && echo found

The use of `x"$var"` was required in case var is "" or "-hyphen". Thinking about this for a moment should indicate that the shell can handle both of these cases unambiguously, and if it doesn't, it's a bug. This bug was probably fixed about 20 years ago, so stop propagating this nonsense, please! Shell doesn't have the cleanest syntax to start with, so polluting it with stuff like this is horrible.

[! -z "$var"] && echo "var not empty"

This is a double negative, and is very prevalent in shell scripts for some reason.

Just test the string directly like `[ "$var" ] && echo "var not empty"`

["$var"] || var="value"

Setting a variable if it's not previously set is a common idiom and can be more succinctly expressed as:

```
: ${var="value"}.
```

Note: if you want to set a variable if it's empty or unset, use `: ${var:="value"}`.

These are portable to the vast majority of shells.

redundant use of $?

For example:

```
pidof program
if [ $? = 1 ]; then
  echo "program not found"
fi
```

Note: this is not just stylistic. Consider what happens if "pidof" returns 2.

Instead just test the exit status of the process directly as in these examples:

```
if ! pidof program; then
  echo "program not found"
fi
```

```
if grep -qF "string" file; then
  echo 'file contains "string"'
fi
```

Needless shell logic

We'll expand on this below, but we should do as little in shell as possible, over its domain of connecting process to files. For example the following common shell idiom of testing for files and directories can often be pushed into the programs themselves. Instead of:

```
[ ! -d "$dir" ] && mkdir "$dir"
[ -f "$file" ] && rm "$file"
```

Do:

```
mkdir -p "$dir" #also creates a
hierarchy for you
rm -f "$file" #also never prompts
```

Note also Google's shell style guide , which as per other Google style guides has very sensible advice. [hn.my/shellstyle]

Robustness

Globbing

In the example below to count the lines in each file, there is a common mistake.

```
for file in `ls *`; do
  wc -l $file
done
```

Perhaps the idiom above stems from a common system where the shell does not do globbing, but in any case it's neither scalable nor robust. It's not robust because it doesn't handle spaces in file names as word splitting is done. Also it redundantly starts an ls process to list the files. On some systems this form can overflow static command line buffers when there are many files. Shell script is a language designed to operate on files, so it has this functionality built in!

```
for file in *; do
  wc -l "$file"
done
```

Notice how we just use the "*" directly, which prevents the redundant "ls" process from starting and doesn't do word splitting on file names containing spaces. This is still slow, since we use shell looping and start a "wc" process per file, so we'll come back to this example in the performance section below.

Stopping automatically on error

Often don't want a script to proceed if some commands fail. But checking the status of each command can become very messy and error-prone. Instead, execute `set -e` at the top of the script, which usually just works as expected, terminating the script when any command fails (that is not already part of a conditional, etc.).

Cleaning up temp files

One should always try to avoid temp files for performance/maintainability reasons, and instead use pipes if at all possible to pass data between processes. Temporary files can be slow as they're usually written to disk. Plus,

you must clean them up when your script exits, possibly in unexpected ways. The general method for cleaning up temp files if you really do need them is to use traps as follows:

```
#!/bin/sh

tf=/tmp/tf.$$

cleanup() {
  rm -f $tf
}

trap "cleanup" EXIT

touch $tf
echo "$tf created"
sleep 10 #Can Ctrl-C and temp file
will still be removed
#temp file auto removed on exit
```

Echoing errors

If you just `echo "Error occurred"` then you will not be able to pipe or redirect any normal output from your script independently. It's much more standard and maintainable to output errors to stderr like `echo "Error occurred" >&2`. Note: you can echo multiple lines together as in the following example:

```
echo  "\
Usage: $(basename $0) option1
more info
even more" >&2
```

Portability

There are two aspects to portability for shell scripts. There's the shell language itself, and there are the various tools being called by the script. We'll just consider the former here. To support really old implementations of shell script, one can test with the heirloom, for example. But for a contemporary list of portable shell capabilities, see the The Open Group spec, which describes the POSIX standard. Check out the Autoconf info on shell portability, which lists details you need to consider when writing very portable shell scripts, and the ubuntu dash conversion info.

It's much better to test scripts directly in a POSIX compliant shell if possible. The "bash --posix" option doesn't suffice, because it still accepts some "bashisms". However, the "dash" shell, the default interpreter of shell scripts on ubuntu, is very good in this regard. One should be testing with this shell anyway due to the popularity of ubuntu, and dash is easy to install on Fedora.

Bashisms

"bash" is the most common interactive shell used on UNIX systems, and consequently, syntax specific to "bash" is often used in shell scripts. I've never needed to resort to bash-specific constructs in my scripts. If you find yourself doing complex string manipulations or loops in bash, then you should probably consider existing UNIX tools instead, or a more general scripting language like python.

["$var" == "find"] && echo "found"

Shell script can't assign variable values in conditional constructs, so the double equals is redundant. Moreover it gives a syntax error on older busybox (ash) and dash at least, so avoid it.

echo {not,portable}

Brace expansion is not portable. It's most useful at the interactive prompt, and can easily be worked around in scripts.

Signal specifications

Be wary when specifying signals to the trap builtin, as mentioned above. I was even caught by this in my timeout script. That script handles the "CHLD" signal, which, for bash at least, can be specified as "sigchld", "SIGCHLD", "chld", "17" or "CHLD", only the last of which is portable.

echo $(seq 15) $((0x10))

The command above containing both $(command substitution) and an $((arithmetic expression)) is portable. Traditionally one did command substitution using backquotes like "seq 15". That's awkward to nest, however, and it's not very readable in the presence of other quoting. $((arithmetic expressions)) can also be handy for quick calculations, rather than spawning off "bc" or "expr'. Bash supports the non-portable form of $[1+1] for arithmetic expressions, which you should avoid.

echo --help

I've used echo in all the examples above for convenience, but one should be wary about using it, especially if you pass variable parameters. "echo" implementations vary on how they handle escaped characters and options, so one really should use "printf" instead, as it has a more standard implementation across systems.

Performance

We'll expand here on our globbing example to illustrate some performance characteristics of the shell script interpreter. Comparing the "bash" and "dash" interpreters for this example where a process is spawned for each of 30,000 files, shows that dash can fork the "wc" processes nearly twice as fast as "bash".

```
$ time dash -c 'for i in *; do wc
-l "$i">/dev/null; done'
real    0m14.440s
user    0m3.753s
sys     0m10.329s

$ time bash -c 'for i in *; do wc
-l "$i">/dev/null; done'
real    0m24.251s
user    0m8.660s
sys     0m14.871s
```

Comparing the base looping speed by not invoking the "wc" processes shows that dash's looping is nearly 6 times faster!

```
$ time bash -c 'for i in *; do
echo "$i">/dev/null; done'
real    0m1.715s
user    0m1.459s
sys     0m0.252s

$ time dash -c 'for i in *; do
echo "$i">/dev/null; done'
real    0m0.375s
user    0m0.169s
sys     0m0.203s
```

The looping is still relatively slow in either shell as demonstrated previously, so for scalability we should use more functional techniques so iteration is performed in compiled processes.w

```
$ time find -type f -print0 | wc
-l --files0-from=- | tail -n1
    30000 total
real    0m0.299s
user    0m0.072s
sys     0m0.221s
```

The above script is by far the most efficient solution. It illustrates the point that one should do as little aspossible in shell script and aim to use it just to connect the existing logic available in the rich set of utilities on a UNIX system. ■

Pádraig Brady is a long time open source contributor and is a maintainer of the GNU coreutils project. He currently works for Red Hat on the OpenStack project.

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

Refactoring to Functional — Why Class?

By HADI HARIRI

In College

Teacher: We are surrounded by objects in the real world. These can be cars, houses, etc. That's why it's very easy to associate real world objects with classes in Object Oriented Programming.

2 weeks later

Jake: I'm having a bit of hard time with these objects. Can you give me some guidance?

Teacher: Sure. There's actually a couple of more or less formal processes to help you, but to sum it up, look for nouns. And verbs are like methods that can be performed on the class. The behavior, so to speak.

Jake: Well, that seems reasonable. Thanks!

Jake graduates.

Jake's on the job

Phil: Hey, Jake. I've been looking at this class of yours. It's a little bit too big.

Jake: Sorry. And what's the issue with that?

Phil: Well, the thing is, it's got too many responsibilities. It does too much.

Jake: And?

Phil: Well, think about it. If it does too much, it means that it touches many parts of the system. So the probability of having to touch the class when changing code is higher, which means more probability of breaking things. Plus, 1000 lines of code in a single class is harder to understand than 30 lines.

Jake: Yeah. Makes sense.

Phil: Break these up into smaller classes. That way each class does only one thing and one thing alone.

A year later

Mary: Jake, I'm just reviewing this class of yours, there's not much behavior in it.

Jake: Yeah, well, I wasn't sure if that behavior belonged in the Customer class or to the Accounts class, so I placed it in this other class called CustomerService.

Mary: OK. Fair enough. But the Customer class isn't really a class anymore. It's more of a DTO.

Jake: DTO?

Mary: Yes, a Data Transfer Object. It's like a class but without behavior.

Jake: So like a structure? A record?

Mary: Yes. Kind of. So just make sure your classes have behavior. Otherwise, they're not really classes. They're DTO's.

Jake: OK.

2 years later

Mathew: Jake, looking at this class. It's tightly coupled to a specific implementation.

Jake: Huh?

Mathew: Well, you're creating an instance of Repository inside the Controller. How you going to test it?

Jake: Hmm. Fire up a demo database?

Mathew: No. What you need to do is first off, program to an interface not a class. That way you don't depend on a specific implementation. Then, you need to use Dependency Injection to pass in a specific implementation, so that when you want to change the implementation you can.

Jake: Makes sense.

Mathew: And in production, you can use an IoC Container to wire up all instances of the different classes.

3 years later

Francis: Jake. You're passing in too many dependencies into this class.

Jake: Yeah, but the IoC Container handles that.

Francis: Yes, I know. But just because it does, it doesn't make it right. Your class is still tightly coupled to too many other classes (even though the implementations can vary). Try and keep it to 1 to 3 maximum.

Jake: OK. Makes sense. Thanks.

4 years later

Anna: Jake. This class, why did you name it Utils?

Jake: Well, I didn't really know where to place that stuff cause I don't know where it really belongs.

Anna: OK. It's just that we already have a class for that. It's called RandomStuff.

Over a beer…

Jake: You know, Pete, I've been thinking. They teach us that we need to think in terms of objects and identify these with nouns among other techniques. We then need to make sure that we name them correctly, that they're small, that they only have a single responsibility and that they can't have too many dependencies injected into them. And now they're telling us that we should try and not maintain state because it's bad for concurrency. I'm beginning to wonder, why the hell have classes at all?

Pete: Don't be silly, Jake. Where else are you going to put functions if you don't have classes?

Pete: Another beer?

Until next time. ■

Hadi Hariri is a Software Developer, currently working at JetBrains. His passions include Web Development and Software Architecture. He has written a few books and have been speaking at conferences for over a decade, on things he's passionate about.

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

Paths to Being a Kernel Hacker

By JULIA EVANS

I ONCE TRIED ASKING for advice about how to get started with kernel programming, and was basically told:

1. If you don't need to understand the kernel for your work, why would you try?
2. You should subscribe to the Linux kernel mailing list [lkml.org] and just try really hard to understand.
3. If you're not writing code that's meant to be in the main Linux kernel, you're wasting your time.

This was really, really, really not helpful to me. So here are a few possible strategies for learning about how operating systems and the Linux kernel work on your own terms, while having fun. I still only know a few things, but I know more than I did before :)

For most of these paths, you'll need to understand some C, and a bit of assembly (at least enough to copy and paste). I had written a few small C programs, and took a course in assembly that I had almost entirely forgotten.

Path 1: Write your own OS

This might seem to be a pretty frightening path. But actually it's not! I started with rustboot [hn.my/rustboot], which, crucially, already worked and did things. Then I could do simple things like making the screen blue instead of red, printing characters to the screen, and move on to trying to get keyboard interrupts to work.

MikeOS [hn.my/os] also looks like another fun thing to start with. Remember that your operating system doesn't have to be big and professional — if you make it turn the screen purple instead of red and then maybe make it print a limerick, you've already won.

You'll definitely want to use an emulator like qemu [qemu.org] to run your OS in. The OSDev wiki [wiki.osdev.org] is also a useful place — they have FAQs for a lot of the problems you'll run into along the way.

Path 2: Write some kernel modules!

If you're already running Linux, writing a kernel module that doesn't do anything is pretty easy.

Here's the source for a module [hn.my/hello] that prints "Hello, hacker school!" to the kernel log. It's 18 lines of code. Basically you just register an init and a cleanup function and you're done. I don't really understand what the `__init` AND `__exit` macros do, but I can use them!

Writing a kernel module that does do something is harder. I did this by deciding on a thing to do (for example, print a message for every packet that comes through the kernel), and then read some Kernel Newbies [kernelnewbies.org], googled a lot, and copied and pasted a lot of code to figure out how to do it. There are a couple of examples of kernel modules I wrote in this kernel-module-fun repository. [github.com/jvns/kernel-module-fun]

Path 3: Do a Linux kernel internship!

The Linux kernel participates in the GNOME Outreach Program for Women. [hn.my/outreach] This is amazing and fantastic and delightful. What it means is that if you're a woman and want to spend 3 months working on the kernel, you can get involved in kernel development without any prior experience, and get paid a bit ($5000).

It's worth applying if you're at all interested — you get to format a patch for the kernel, and it's fun. Sarah Sharp, a Linux kernel developer, coordinates this program and she is pretty inspiring. You should read her blog post about how 137 patches got accepted into the kernel during the first round. [hn.my/137]These patches could be yours! Look at the application instructions! [kernelnewbies.org/OPWApply]

Path 4: Read some kernel code

This sounds like terrible advice — "Want to understand how the kernel works? Read the source, silly!"

But it's actually kind of fun! You won't understand everything. I felt kind of dumb for not understanding things, but then every single person I talked to was like "yeah, it's the Linux kernel, of course!"

My friend Dave recently pointed me to LXR [lxr.linux.no], where you can read the kernel source and it provides lots of helpful cross-referencing links. For example, if you wanted to understand the chmod system call, you can go look at the `chmod_common` definition in the Linux kernel! *livegrep.com* is also really nice for this.

Here's the source for `chmod_common`, with some comments from me:

```
static int chmod_common(struct path *path, umode_t mode)
{
    struct inode *inode = path->dentry->d_inode;
    struct iattr newattrs;
    int error;

    error = mnt_want_write(path->mnt);
    if (error)
        return error;

    // Mutexes! Prevent race conditions! =D
    mutex_lock(&inode->i_mutex);

    // Check for permission to use chmod
    error = security_path_chmod(path, mode);
    if (error)
        goto out_unlock;
    // I guess this changes the mode!
    newattrs.ia_mode = (mode & S_IALLUGO) |
(inode->i_mode & ~S_IALLUGO);
    newattrs.ia_valid = ATTR_MODE | ATTR_CTIME;
error = notify_change(path->dentry, &newattrs);
out_unlock:
    mutex_unlock(&inode->i_mutex);
// We're done, so the mutex is over!
    mnt_drop_write(path->mnt); // ???
    return error;
}
```

I find this is a fun time and helps demystify the kernel for me. Most of the code I read I find pretty opaque, but some of it (like this chmod code) is a little bit understandable. ■

Julia Evans likes programming, playing with data, and finding out why things that seem scary actually aren't. It turns out that the Linux kernel is a fun time!

Reprinted with permission of the original author.
First appeared in *hn.my/kernelhacker* (jvns.ca)

The Complete Guide to Centering a Div

By STEVE PEAR

EVERY NEW DEVELOPER inevitably finds that centering a div isn't as obvious as you'd expect. Centering what's inside a div is easy enough by giving the text-align property a value of center, but then things tend to get a bit sticky. When you get to centering a div vertically, you can end up in a world of CSS hurt.

The aim of this article is to show how, with a few CSS tricks, any div can be centered; horizontally, vertically or both. And within the page or a div.

Centering a div in a page, basic

This method works with just about every browser, ever.

CSS

```
.center-div
{
     margin: 0 auto;
     width: 100px;
}
```

The value auto in the margin property sets the left and right margins to the available space within the page. The thing to remember is your centered div must have a width property.

Centering a div within a div, old-school

This works with almost every browser.

CSS

```
.outer-div
{
     padding: 30px;
}
.inner-div
{
     margin: 0 auto;
     width: 100px;
}
```

HTML

```
<div class="outer-div">
  <div class="inner-div">
  </div>
</div>
```

The margin auto trick strikes again. The inner div must have a width property.

Centering a div within a div with inline-block

With this method the inner div doesn't require a set width. It works with all reasonably modern browsers, including IE 8.

CSS

```
.outer-div
{
     padding: 30px;
     text-align: center;
}
.inner-div
{
     display: inline-block;
     padding: 50px;
}
```

HTML

```
<div class="outer-div">
  <div class="inner-div">
  </div>
</div>
```

The text-align property only works on inline elements. The inline-block value displays the inner div as an inline element as well as a block, so the text-align property in the outer div centers the inner div.

Centering a div within a div, horizontally and vertically

This uses the margin auto trick to center a div both horizontally and vertically. It works with all modern browsers.

CSS

```
.outer-div
{
     padding: 30px;
}
.inner-div
{
     margin: auto;
     width: 100px;
     height: 100px;
}
```

HTML

```
<div class="outer-div">
  <div class="inner-div">
  </div>
</div>
```

The inner div must have a width and height property. This doesn't work if the outer div has a fixed height.

Centering a div at the bottom of a page

This uses margin auto and an absolute-positioned outer div. It works with all modern browsers.

CSS

```
.outer-div
{
     position: absolute;
     bottom: 70px;
     width: 100%;
}
.inner-div
{
     margin: 0 auto;
     width: 100px;
     height: 100px;
     background-color: #ccc;
}
```

HTML

```
<div class="outer-div">
  <div class="inner-div">
  </div>
</div>
```

The inner div must have a width property. The gap from the very bottom of the page is set in the outer div's bottom property.

Centering a div in a page, horizontally and vertically

This uses margin auto again and an absolute-positioned outer div. It works with all modern browsers.

CSS

```
.center-div
{
     position: absolute;
     margin: auto;
     top: 0;
     right: 0;
     bottom: 0;
     left: 0;
     width: 100px;
     height: 100px;
     background-color: #ccc;
}
```

The div must have a width and height property. ■

Steve Pear is the proprietor and lead developer at Tipue, a small web development studio based in North London.

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

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

The Story of My Desk

By JAMES LONG

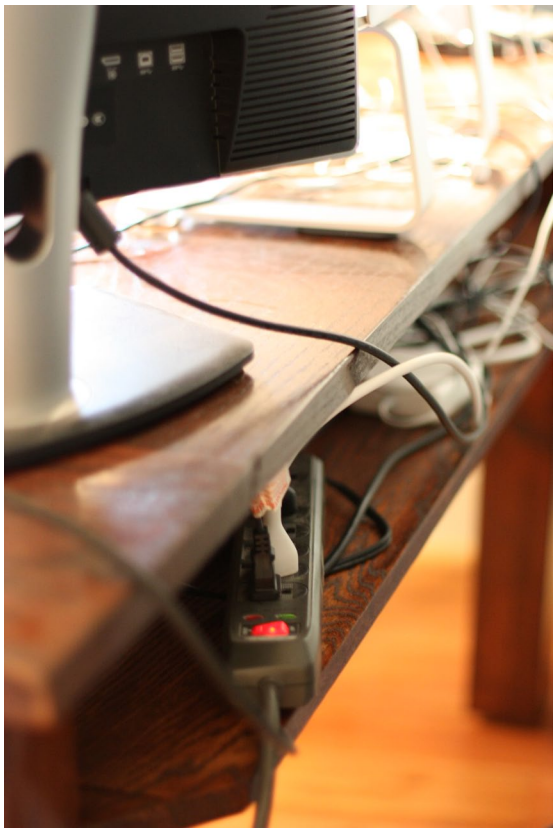
I've been searching for the perfect desk for the past few years. It took a while to even figure out what that is. I tried a few standing setups, but found that I'd rather "opt in" to standing and only do it a few hours a day at most. My desk needed to be sturdy, beautiful, and just the right size.

The stingy and stubborn side of me kicked in whenever I shopped around. The desks that came close to what I wanted were over $400. Eventually, I decided to just build my own. The problem was that I had no idea how to build it. This is a story about tackling the unknown and persisting until it's done.

That was almost two years ago, and as of this summer, I finally finished my desk. It's exactly what I want and much cheaper too.

The final product

Although I started playing around two years ago, the above desk took just 2 months to make with about one night a week. The wood cost about $150, and I spent about the same on tools but I am already using them on other projects. Sure, you could go out and spend $75 dollars on something that mostly works, but I wanted a desk that I'll have for years and years and also looks beautiful.



All the cords sit hidden on this ledge, making the desk look nice and clean

It's 5' long, 25" deep, and 32" tall. It came out a little taller than expected because I put feet screws in the bottom of the legs so that I could make the table even. Those screws added almost an inch. I just have to raise my chair a little bit though.

My innovation was a ledge in the back to hold all of your cords. Note in the above picture that there is only a single cord behind the desk: the power strip. Everything else is hidden.

You can do this too, with a little patience. I knew nothing about woodworking when I started. I wish I could write a full tutorial, but I don't have time. Here are a few tips:

- Oak is a good hard wood for desks. Get the lumber yard to plane the boards for you, trust me. It's extremely important that all edges are completely flat and edges are 90°.
- It was surprisingly hard to find untreated 4x4s for the legs (I thought Lowes and such places had untreated, but they don't). Even at the specialty lumber yard, I could only find cedar, so my desk legs are cedar. Turns out cedar is really light though, so that's nice!
- You'll need lots of clamps. Pipe clamps are best for joining long boards together, like the desk top.
- I used dowels to strengthen joints, but I think it was more work than necessary. Dowel holes need to 100% aligned, so I had to use a doweling jig and it was just annoying. For the desk top, you can probably just use wood glue and join them (sounds crazy, but the glue really is most of the join strength). For the base, you need something, and the cheapest and easiest is probably the Kreg jig. I resisted it because it's a newer thing and I wanted to see how the old ways worked. Now I know.

My first prototype wooden desk. Learned how to glue boards together and drill dowel holes. Without aprons, it wobbled.

- At first I thought I could make a desk top and just put 4 legs on it. You can't. You need "aprons" to make the desk really sturdy. These are just long oak boards that connect the 4x4 legs together, establishing the base. The top sits on top and is connected to the aprons with Rockler table top fasteners (you need to cut a shallow ridge on the inside of the aprons).
- Stains and finishes are complicated. I used a General Finish stain and Waterlox as a sealer. 2 coats of stain and 4 coats of Waterlox (it's a tung oil so it needs several).
- I also made a major mistake by staining the desk when it was hot and humid outside, and became gummy and was hard to wipe. It also took forever for the Waterlox oil to dry (about a week).

YouTube and Google are your friends. I did extensive research for each step of the process. There are some great YouTube videos that show how experienced woodworkers achieve results. The Patrick Hosey Workshop: Farmhouse Table was one of my favorite videos which doesn't explain much but shows a lot of details.

My boards weren't planed, and there's no replacement for a joiner and planer. A friend has a local shop and helped me get them planed and straight.

The boards were connected with dowels (using a doweling jig to drill the holes) and a bunch of wood glue, and clamped with pipe clamps.

Joining the two end pieces. You can see the dowel holes.

The two end pieces joined. The pipe clamps are bending but I fixed that after I took the picture.

Ridges cut into the aprons for the table top fasteners. I used a circular saw so it was difficult.

I used dowels and wood glue to join the aprons with the legs

First coat of stain. The color is red mahogany. I can't believe I did this when it was humid outside, as it became kind of gummy.

The final base. The left apron is on the inside because it will have a ledge to hold all the cords in the back.

Second coat of stain. I used a milder brown color for this coat.

The final color (before finishing). It's slightly splotchy because I did this when it was humid.

The back ledge will hold all the cords and other untidy stuff out of sight.

The final top, after a few coats of Waterlox finish.

The final base, stained and finished the same as the top.

I actually built 2 tables before this. I built the first one 2 years ago and it was just a butcherblock top with a steel base I constructed by drilling holes and bolting it together. It was horribly wobbly. Last winter I finally committed to building a wooden desk.

I built a desk out of cheap wood as a prototype to learn the basics. Finally, I mustered up the courage to buy nice wood and cut into it. The rest of the story is told in the pictures. It's been a great process to start from scratch, and force myself to learn something new.

For me, it's a reminder that you can achieve something if you persist. It's too easy to give up with the excuse "I don't know how to do this," but you can learn. ■

James Long works for Mozilla as an apps engineer, helping develop the web as a platform. He loves diving into complicated problems and figuring out solutions, whether it's programming, design, or making the best espresso. He also likes teaching people to code.

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

How I Learned to Stop Procrastinating, & Love Letting Go

By LEO BABAUTA

> **"People have a hard time letting go of their suffering. Out of a fear of the unknown, they prefer suffering that is familiar." — Thich Nhat Hanh**

The end of procrastination is the art of letting go.

I've been a lifelong procrastinator, at least until recent years. I would put things off until deadline, because I knew I could come through. I came through on tests after cramming last minute, I turned articles in at the deadline after waiting until the last hour, I got things done.

Until I didn't. It turns out procrastinating caused me to miss deadlines, over and over. It stressed me out. My work was less-than-desirable when I did it last minute. Slowly, I started to realize that procrastination wasn't doing me any favors. In fact, it was causing me a lot of grief.

But I couldn't quit. I tried a lot of things. I tried time boxing and goal setting and accountability and the Pomodoro Technique and Getting Things Done. All are great methods, but they only last so long. Nothing really worked over the long term.

That's because I wasn't getting to the root problem.

I hadn't figured out the skill that would save me from the procrastination.

Until I learned about letting go.

Letting go first came to me when I was quitting smoking. I had to let go of the "need" to smoke, the use of my crutch of cigarettes to deal with stress and problems.

Then I learned I needed to let go of other false needs that were causing me problems: sugar, junk food, meat, shopping, beer, possessions. I'm not saying I can never do these things again once I let go of these needs, but I let go of the idea that they're really necessary. I let go of an unhealthy attachment to them.

Then I learned that distractions and the false need to check my email and news and other things online were causing me problems. They were causing my procrastination.

So I learned to let go of those too.

Here's the process I used to let go of the distractions and false needs that cause procrastination:

1. I paid attention to the pain they cause me, later, instead of only the temporary comfort/pleasure they gave me right away.

2. I thought about the person I want to be, the life I want to live. I set my intentions to do the good work I think I should do.

3. I watched my urges to check things, to go to the comfort of distractions. I saw that I wanted to escape discomfort of something hard, and go to the comfort of something familiar and easy.

4. I realized I didn't need that comfort. I could be in discomfort and nothing bad would happen. In fact, the best things happen when I'm in discomfort.

And then I smile, and breathe, and let go.

And one step at a time, become the person I want to be. ■

"You can only lose what you cling to."
~Buddha

Leo Babauta is the creator and writer at Zen Habits. He is a former journalist and freelance writer of 18 years, a husband and father of six children, and lives on the island of Guam where he leads a very simple life.

Reprinted with permission of the original author.
First appeared in *hn.my/letgo* (zenhabits.net)