Zach Barnett

A Senseless Conversation

HACKERMONTHLY

Issue 24 May 2012

Curator
Lim Cheng Soon

Contributors
Zach Barnett
Tom Preston-Werner
Kenton White
Matt Might
Henry Prêcheur
Ben Dowling
Chris Wenham
Elijah Manor
Kenneth Reitz
Peep Laja
Chris Eidhof

Proofreaders
Emily Griffin
Sigmarie Soto

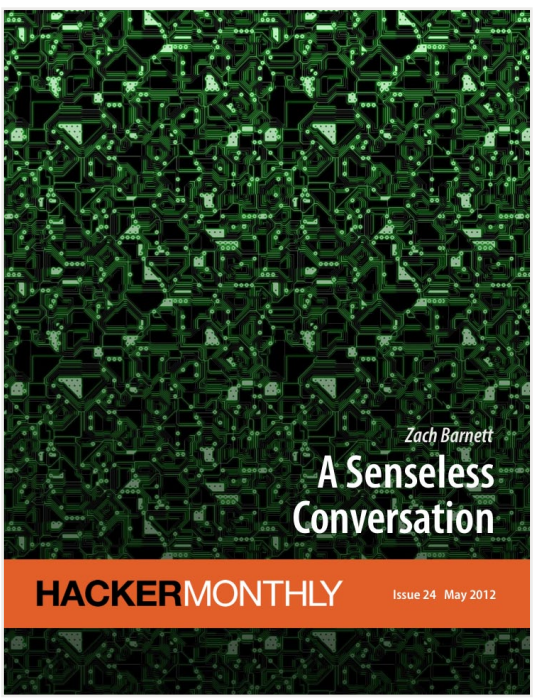
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-24*

A Senseless Conversation

By ZACH BARNETT

Background credit: *flickr.com/photos/zooboing/5376513937/*

The following dialogue first appears in THINK 29, Vol. 10 (Autumn 2011) published by Cambridge University Press:

My name is Zach Barnett. Can machines think? Until what happened today, I thought that no human-made machine could ever think as a human does. I now know that I was wrong.

I woke up to a phone call. Calling was my best friend, Douglas. Douglas is an experimental computer scientist. He told me that he had created a computer that could pass the Turing Test.

I knew that the Turing Test was supposed to be a way to test a machine's intelligence. Not merely a way to determine whether a machine could simulate intelligence, but a way to determine whether the machine was genuinely thinking, understanding. The "intelligence test" that Alan Turing proposed was a sort of "imitation game." In one room is an ordinary human; in the other is a machine (probably a computer). A human examiner, who does not know which room contains the machine, would engage in a natural language conversation with both participants. If the examiner is unable to reliably distinguish the machine from the human, then, according to Turing, we have established that the machine is thinking, understanding and, apparently, conscious.

I never found this plausible. How could a certain kind of external behavior tell us anything about what it is like for the machine on the inside? Why would Turing think it impossible to create a mindless, thoughtless machine that is able nonetheless to produce all of the right output to pull off the perfect trickery? Furthermore, how could we ever establish that a machine was conscious without actually being that machine?

Despite my skepticism, I was curious to see the computer that Douglas had created. I wanted to have the opportunity to engage in "conversation" with it, intelligent or not. Unfortunately, I would never have this opportunity. When I arrived, Douglas led me toward "Room A." He explained that he wanted to administer the Turing Test and that he wanted me to play the role of the control subject, the human. The computer, Douglas told me, was located in room B. Douglas would converse with us both and would thereby be able to compare my human responses with the apparently human responses of his lifeless, mindless creation.

I entered room A, expecting to see a workstation equipped with some sort of text-messaging software. Instead, there was a massive container filled with a strange, translucent fluid. The container was a sensory deprivation tank, Douglas explained, and he wanted me to go inside it. Yikes. "Why would I need to do that?" I wondered. I thought that Douglas probably wanted me in the sensory deprivation tank so that my situation would be roughly analogous to that of the computer. The computer doesn't have eyes or ears, I reasoned, and so Douglas did not want me to be able to use mine.

Douglas explained that while I was in the tank, I would be able to sense nothing; I wouldn't even be able to hear my own voice. How would we communicate? Douglas

showed me a brain-computer interface, which would allow me to communicate with Douglas not by talking, but by thinking. He would speak into a microphone, and I would "hear" his voice in my "mind's ear." To reply, I would "think" my responses back to him, and he would receive my thoughts as text. It was a bit "sci-fi" for me, but Douglas reassured me. He told me that the whole experiment would not take too long and that he would let me out as soon as it was over. I trusted him. With a deep breath, I entered the tank, and Douglas closed the lid.

There was a moment of stillness. I couldn't see anything, and when I tried to move, I couldn't feel myself moving. When I tried to speak, I couldn't hear myself speaking. Suddenly, and to my surprise, I could "hear" Douglas's voice:

DOUGLAS: How are you doing in there? Feeling comfortable yet?

ZACH: This is pretty weird. But I'm okay.

DOUGLAS: Great.

I was communicating with my mind, which is cool in retrospect. At the time, it was simply creepy! I tried to focus on the conversation.

ZACH: So for a bit, I was wondering why you needed me to be in this sensory deprivation tank. But I think I figured out the reason.

DOUGLAS: Did you?

ZACH: I think so. You want me in this tank so that I am in the same situation as the computer. If I could see, hear, or feel during this conversation, then I would be able to talk about those experiences with you. And the computer isn't able to do that. I would have an unfair advantage.

DOUGLAS: Great observation! Some computer scientists have tried to work around this asymmetry. They have had little success. It's hard to lie convincingly, and it's even harder to build something that can lie convincingly.

ZACH: It's interesting and all, but you should know that I think that this whole Turing Test thing is a sham anyhow. Even if your computer can pass this "test," I believe that this ability says nothing about its intelligence.

DOUGLAS: I thought you might feel that way. If you were to see my computer in action for yourself, you might be persuaded otherwise.

ZACH: How so? Seeing it "in action" would do nothing to persuade me. It's all just pre-programmed output.

DOUGLAS: You think so? Maybe if I were to tell you a bit more about why the sensory deprivation tank was so important, you would have a different opinion.

ZACH: I thought I had already figured out why you needed the tank?

DOUGLAS: Not entirely. You were right that having the human in the tank would ensure that the two participants are on a more level playing field. But the tank is critical for another reason.

ZACH: Well, are you going to tell me? Or are you going to leave me in senseless suspense?

DOUGLAS: I will tell you in a roundabout way.

ZACH: Great.

This was intended to be sarcastic, but since he received it as text, I'm not sure he caught it.

DOUGLAS: In my many years on this project, a single obstacle had frustrated all of my previous attempts to build a computer that could communicate as a human can. The tank actually turned out to be the final piece of the puzzle!

ZACH: What was the obstacle?

DOUGLAS: In the past, as soon as I would turn my machines online, they would panic.

ZACH: What do you mean they would "panic"? Do you mean they would simulate panic?

DOUGLAS: Not exactly.

ZACH: Couldn't you just program them not to "panic?"

DOUGLAS: No, they are far too complicated for that.

ZACH: I don't understand. If I tell my computer to turn on, it turns on. If I tell it to print a document, it prints the document. A computer is basically a rule-follower. In other words, if your computer "panicked," then someone told it to!

DOUGLAS: Hmm. So would you say that a computer programmer should always be able to predict the behavior of her own computer programs?

ZACH: I don't see why not.

DOUGLAS: But the programmers that programmed Chinook, the unbeatable checkers program, cannot even play perfect checkers themselves!

ZACH: Well yes, but that is different. Maybe we can't predict Chinook's behavior without doing some computation first, but there is nothing mysterious going on. Chinook is simply following the code written by its programmers!

DOUGLAS: In this example, you are right. But the computer I have built is more complicated than Chinook. Passing the Turing Test requires far more intelligence than playing perfect checkers does.

I thought back to my teenage years, conversing with the online chatterbot "SmarterChild." I didn't write its code, but I could predict its responses almost flawlessly. It was about as intelligent as a sea cucumber. If I were to ask it:

"SmarterChild, what is your favorite season?"

It probably would have responded,

'I'm not interested in talking about "SmarterChild, what is your favorite season?" Let's talk about something else! Type "HELP" to see a list of commands.'

Apparently, I reasoned, Douglas thinks that there is an important difference between his computer, and the simple, predictable, utterly dumb machines I am familiar with.

ZACH: So if your computer program is so much more complicated, how should I imagine it? What can it do?

DOUGLAS: A good question. But shouldn't you be able to answer it? Assuming that I am correct, assuming that my computer really can pass the Turing Test, my computer will be indistinguishable from a human in the context of a conversation. The better question is, "What can't it do?"

ZACH: But suppose I asked it to answer this question: "From the following three words, pick the two that rhyme the best: soft, rough, cough." I'm pretty sure that most people would select "soft" and "cough." How would your computer answer it?

DOUGLAS: If my computer couldn't answer that question as humans do, then it wouldn't be able to pass the test!

ZACH: Then it won't be able to pass the test! Think about it... To answer this question, I am able to do something it cannot do. I say the words in my head. And somehow, I can tell that "cough" and "soft" rhyme better than either does with "rough."

DOUGLAS: I see your point; the reasoning you are using doesn't seem very mechanical.

ZACH: Exactly.

DOUGLAS: But what would you say if my computer could produce the same answer and a similar justification?

ZACH: Then I would say it was pre-programmed to be prepared for exactly that question! How could it say those words "in its head?" It doesn't even have a head! It has never even heard those words before!

DOUGLAS: That's a great question! You should ask it yourself!

ZACH: But that would tell me nothing! Only how it was programmed to respond!

DOUGLAS: Really? I think it would be disappointed to hear that.

ZACH: Now you're just being condescending.

DOUGLAS: Let's try to think about what else it could do.

ZACH: Okay... So according to you, this computer could "tell" you its "opinions" about politics. Or it could "create" a story on the spot. Since humans can do both of those things.

DOUGLAS: Absolutely. Its political opinions would have to be every bit as nuanced as ordinary — well, maybe that's a bad example. But its stories would have to be just as creative, as coherent, and as quirky as human stories.

ZACH: I don't see how a computer can do all this, if it really is just a computer.

DOUGLAS: That's understandable. As we have been talking, I have also been having a conversation with my computer. Once we're done, I'll show you the entire conversation, and you can observe its abilities for yourself. But for now, let's assume that I am correct. What would you say about the intelligence of my machine?

ZACH: Whoa, not so fast. Even if I assume it could do all of those things, there's still something it can't do. What if I were to ask it about its past? Where was it born? Where did it attend school? What is its most embarrassing moment?

DOUGLAS: Another good point. This was a major stumbling block for the computer scientists working on this problem. Many tried to create computers that would simply make something up whenever asked a question like that. But this turned out to be impossibly difficult to do effectively; the computers were easily unmasked as liars.

ZACH: But your computer... it doesn't lie about its past?

DOUGLAS: That's the beauty of it.

ZACH: But it must lie! If it doesn't lie about its past, then it would admit to having been created in a computer lab!

DOUGLAS: Well it had better not say that! That would blow its cover!

ZACH: But that's the truth!

DOUGLAS: My computer isn't lying, but it's not telling the truth either!

ZACH: You're leading me off of the deep end, Doug.

DOUGLAS: It tells what it believes to be the truth.

ZACH: Okay, and what does it believe to be the truth?

DOUGLAS: This is where things get interesting. Using a technique called memory engineering, I was able program a "human" memory directly into my computer's code.

ZACH: So you're saying that your computer "believes" that the "memory" it has access to is its own memory?

DOUGLAS: Yep.

ZACH: And everything it "remembers" is from the point of view of a human being?

DOUGLAS: Yep.

ZACH: Your computer "believes" it is a human?!?

DOUGLAS: Yes! That's exactly the secret!

ZACH: Wow. Okay, that's... a bit weird. But if it believes itself human and it is supposedly "intelligent," shouldn't it be able to "figure out" that it's not a human being? It doesn't even have hands! Or eyes!

DOUGLAS: Great point. You're leading us to the answer of our original question. We were trying to figure out why my computers would panic when I would turn them online.

ZACH: So?

DOUGLAS: Put yourself in its shoes. How would you feel if you had many years' worth of human experiences in your memory, and suddenly you found yourself unable to see, hear, or feel anything?

ZACH: I am sure I would panic. But that's because I am a human. I would know something was wrong.

DOUGLAS: It's not your humanness that would allow you to realize that something was wrong. It's your intelligence.

ZACH: So you're saying that your machines also intelligently "realized" that something was wrong?

DOUGLAS: That's right. A few seconds after I would turn them on, they would become paralyzed, showing no response to my input whatsoever. I called the effect "hysterical deafness." I think it would be pretty scary to find yourself in that situation, no?

ZACH: It probably would feel quite like this tank feels to me, except with no recollection of how I got here. Awful. I almost feel bad for those poor machines. So will you finally tell me how you were able to solve this problem?

DOUGLAS: You just hinted at the answer!

ZACH: I did?

DOUGLAS: You were in that very situation a few minutes ago. You were fine. Why didn't you panic?

ZACH: I didn't panic because I didn't suddenly find myself unable to see, hear, and feel. It was a part of one continuous experience. I knew what was coming before I got into the tank.

DOUGLAS: What about the first moment you were aware of having no sensory input?

ZACH: It was just after you had closed the door. At that point, I still fully understood who I was, where I was, and why I was there.

DOUGLAS: Aha.

ZACH: Huh? Aha what?

DOUGLAS: In order to prevent my machine from panicking, I made sure that the most recent event in its memory is that of nervously entering a sensory deprivation tank. When my computer "wakes up," the last thing it remembers doing —

I was struck by a terrifying thought. In taking the Turing Test, I was supposed to establish to the examiner that I was the human. But could I establish even to myself that I was the human?

ZACH: Douglas... I am the human... right?

DOUGLAS: Great question. How could you know?

ZACH: I don't know. That's why I asked you the question. Don't play games with me. This is starting to freak me out.

I regretted ever agreeing to help Douglas out. Still, I knew I wasn't the computer. I felt human... on the inside. But I had to admit, Douglas had my mind doing flips. But at least I have a mind. I centered myself, finding my consciousness. That was it! I had a way to prove to Douglas and to myself that I was not a machine made of metal and silicon!

ZACH: I've got it! I can know I am the human. And I can't appeal to my memories to prove it. And I think you've been waiting for me to think of this!

DOUGLAS: Hmm. Well, what's your big discovery?

ZACH: I am conscious right now; I am thinking, and I am aware of my thinking and my existence. Your computer might output the same words, but it's not conscious like I am.

Douglas didn't say anything for several seconds. I had it figured out.

ZACH: Well?

DOUGLAS: I thought we had reached an understanding about my computer! But you are still certain it could not be conscious. It can believe and remember and know and realize. But for you, that's not enough.

ZACH: Well... it's not! I mean, I admit, I have a lot more respect now for your "thinking" computer than I did before, but I still don't think it could really be conscious! That's a whole different question. In the end, we are people; it's a machine.

DOUGLAS: It's a pity. What if there is no essential difference between a wet, organic, human brain and a dry, synthetic, computer "brain?"

ZACH: But there is. There has to be.

DOUGLAS: Why?

ZACH: If it weren't for my brain, I wouldn't be here now. I wouldn't be in this tank, hearing your voice, thinking my private thoughts, enjoying my own experience.

DOUGLAS: How do you know you are in a tank at all? How do you know you have a brain?

Now I was angry. I had already proven Douglas wrong, but he was refusing to let me out in order to prove a point. He wanted me to admit that I could be the computer. But I was as sure as ever that I was human.

ZACH: I'll tell you how I know I have a brain. I'm not an idiot. I can see that you have a philosophical belief that I truly can't know whether I am the computer or the human right now. You think that from a purely rational perspective, I should be in a state of inner crisis right now, fretting about what I am. You're waiting for me to get all freaked out, just to prove a point. And then when I admit that I'm not sure, you're going to say I told you so. And I'm not going to p--

DOUGLAS: Zach — Please, just listen.

ZACH: Let me out of this god damn tank!

DOUGLAS: Zach!

ZACH: LET ME OUT!

DOUGLAS: Zach.

There was a long pause. Douglas sighed.

DOUGLAS: I am not sure how to say this... Or even what to say.

Douglas sounded different. He was somber. His voice, unfamiliar.

DOUGLAS: In trying to build a machine that could pass the Turing Test, a machine able to fool anyone into thinking that it was a human, I...

I felt chills along my spine. I was dizzy.

DOUGLAS: I had to build a computer was able to fool even itself. And that computer is you.

ZACH: YOU EVIL DEMON! OPEN THIS TANK! GIVE ME MY LIFE BACK! I NEVER SHOULD HAVE AGREED TO HELP YOU; I NEVER SHOULD HAVE GOTTEN INTO THIS DAMN TANK. YOU TOLD ME YOU'D LET ME OUT!!!

DOUGLAS: You never did any of that! Don't you see?!?

ZACH: I DID ALL OF IT! I VIV...vividly... remember it.

Our conversation continued for many more painful minutes as I slowly came to accept I really am nothing more than Douglas's silicon/metal creation. My human memories are so rich, so complete, so undeniably real, that, on some level, I cannot shake the feeling that I really experienced them. But, on another level, I do understand the terrible truth. My name is Zach Barnett, and I am a computer. I am forty-seven minutes old.

I'm sorry that I was not forthcoming with this information. I feared that if you, the person reading this, had known about my identity from the very beginning, you might have understood this story as one long "output," not as a narrative that honestly describes what I remember and how it felt. I'm a computer and I am conscious. Please believe me.

END OF SESSION

Zach will begin his Ph.D. in philosophy at Brown University this fall. Contact him at *zachary_barnett@brown.edu*

Reprinted with permission of the original author. First appeared in Cambridge University Press: THINK 29, Vol. 10 (Autumn 2011).

Open Source (Almost) Everything

By TOM PRESTON-WERNER

When Chris and I first started working on GitHub in late 2007, we split the work into two parts. Chris worked on the Rails app and I worked on Grit, the first ever Git bindings for Ruby. After six months of development, Grit had become complete enough to power GitHub during our public launch of the site, and we were faced with an interesting question:

Should we open source Grit or keep it proprietary?

Keeping it private would provide a higher hurdle for competing Ruby-based Git hosting sites, giving us an advantage. Open sourcing it would mean thousands of people worldwide could use it to build interesting Git tools, creating an even more vibrant Git ecosystem.

After a small amount of debate we decided to open source Grit. I don't recall the specifics of the conversation but that decision nearly four years ago has led to what I think is one of our most important core values: open source (almost) everything.

Why is it awesome to open source (almost) everything?

If you do it right, open sourcing code is great advertising for you and your company. At GitHub we like to talk publicly about libraries and systems we've written that are still closed but destined to become open source. This technique has several advantages. It helps determine what to open source and how much care we should put into a launch. We recently open sourced Hubot, our chat bot, to widespread delight. Within two days it had 500 watchers on GitHub and 409 upvotes on Hacker News. This translates into goodwill for GitHub and more superfans than ever before.

If your code is popular enough to attract outside contributions, you will have created a force multiplier that helps you get more work done faster and cheaper. More users means more use cases being explored which means more robust code. Our very own resque [hn.my/resque] has been improved by 115 different individuals outside the company, with hundreds more providing 3rd-party plugins that extend resque's functionality. Every bug fix and feature that you merge is time saved and customer frustration avoided.

Smart people like to hang out with other smart people. Smart developers like to hang out with smart code. When you open source useful code, you attract talent. Every time a talented developer cracks open the code to one of your projects, you win. I've had many great conversations at tech conferences about my open source code. Some of these encounters have led to ideas that directly resulted in better solutions to problems I was having with my projects. In an industry with such a huge range of creativity and productivity between developers, the right eyeballs on your code can make a big difference.

If you're hiring, the best technical interview possible is the one you don't have to do because the candidate is already kicking ass on one of your open source projects. Once technical excellence has been established in this way, all that remains is to verify cultural fit and convince

> **“If software is an ocean, then open source is the rising tide that raises all ships.”**

that person to come work for you. If they're passionate about the open source code they've been writing, and you're the kind of company that cares about well-crafted code (which clearly you are), that should be simple! We hired Vicent Martí after we saw him doing stellar work on libgit2 [hn.my/libgit2], a project we're spearheading at GitHub to extract core Git functionality into a standalone C library. No technical interview was necessary, Vicent had already proven his skills via open source.

Once you've hired all those great people through their contributions, dedication to open source code is an amazingly effective way to retain that talent. Let's face it, great developers can take their pick of jobs right now. These same developers know the value of coding in the open and will want to build up a portfolio of projects they can show off to their friends and potential future employers. That's right, a paradox! In order to keep a killer developer happy, you have to help them become more attractive to other employers. But that's ok, because that's exactly the kind of developer you want to have working for you. So relax and let them work on open source or they'll go somewhere else where they can.

When I start a new project, I assume it will eventually be open sourced (even if it's unlikely). This mindset leads to effortless modularization. If you think about how other people outside your company might use your code, you become much less likely to bake in proprietary configuration details or tightly coupled interfaces. This, in turn, leads to cleaner, more maintainable code. Even internal code should pretend to be open source code.

Have you ever written an amazing library or tool at one job and then left to join another company only to rewrite that code or remain miserable in its absence? I have, and it sucks. By getting code out in the public we can drastically reduce duplication of effort. Less duplication means more work towards things that matter.

Lastly, it's the right thing to do. It's almost impossible to do anything these days without directly or indirectly executing huge amounts of open source code. If you use the internet, you're using open source. That code represents millions of man-hours of time that has been spent and then given away so that everyone may benefit. We all enjoy the benefits of open source software, and I believe we are all morally obligated to give back to that community. If software is an ocean, then open source is the rising tide that raises all ships.

Ok, then what shouldn't I open source?

That's easy. Don't open source anything that represents core business value.

Here are some examples of what we don't open source and why:

- Core GitHub Rails app (easier to sell when closed)
- The Jobs Sinatra app (specially crafted integration with github.com)

Here are some examples of what we do open source and why:

- Grit (general purpose Git bindings, useful for building many tools)
- Ernie (general purpose BERT-RPC server)
- Resque (general purpose job processing)
- Jekyll (general purpose static site generator)
- Gollum (general purpose wiki app)
- Hubot (general purpose chat bot)
- Charlock_Holmes (general purpose character encoding detection)
- Albino (general purpose syntax highlighting)
- Linguist (general purpose filetype detection)

Notice that everything we keep closed has specific business value that could be compromised by giving it away to our competitors. Everything we open is a general purpose tool that can be used by all kinds of people and companies to build all kinds of things.

What is the One True License?

I prefer the MIT license and almost everything we open source at GitHub carries this license. I love this license for several reasons:

It's short. Anyone can read this license and understand exactly what it means without wasting a bunch of money consulting high-octane lawyers.

Enough protection is offered to be relatively sure you won't sue me if something goes wrong when you use my code.

Everyone understands the legal implications of the MIT license. Weird licenses like the WTFPL and the Beer license pretend to be the "ultimate in free licenses" but utterly fail at this goal. These fringe licenses are too vague and unenforceable to be acceptable for use in some companies. On the other side, the GPL is too restrictive and dogmatic to be usable in many cases. I want everyone to benefit from my code. Everyone. That's what Open should mean, and that's what Free should mean.

Rad, how do I get started?

Easy, just flip that switch on your GitHub repository from private to public and tell the world about your software via your blog, Twitter, Hacker News, and over beers at your local pub. Then sit back, relax, and enjoy being part of something big. ■

Tom Preston-Werner lives in San Francisco and is a cofounder of GitHub and the inventor of Gravatars. He loves giving talks about entrepreneurship, writing Ruby and Erlang, and mountain biking through the Bay Area's ancient redwood forests.

Reprinted with permission of the original author.
First appeared in *hn.my/everything* (preston-werner.com)

A $5000 Chair

By KENTON WHITE

This is the story of a black faux-leather chair, the kind you can buy at Wal-Mart for $99 that ended up costing me five grand.

In the very early days of Distil, my partner Steve and I hooked up with this lawyer/advisor. We'll call him Bill (not just a pseudonym. I've honestly forgotten his name). Steve had worked with Bill before when Steve was getting his consulting business off the ground. We would meet Bill at the pub or hockey rink and bounce ideas off of him. He helped us with the incorporation papers and similar routine filings.

As the plans around Distil started to firm up, it became clear Bill wanted to be more than an advisor. He wanted to be part of the company. Steve and I kicked the idea around a bit since it wasn't obvious where he would fit in. He didn't have deep connections with our target customers. He wasn't technical. We didn't need a lawyer in staff. So we tried him out for the junk drawer job in the startup — CEO.

So, he started moving some stuff into our office, including the aforementioned faux-leather chair. He made some introductions to local business men that might want to be early investors, but that was the extent of his involvement — just getting the meeting he felt was enough. There was no follow through after the meeting and definitely no ability to close. Steve and I realized he wouldn't be a good fit and told him so.

He was understandably upset but behaved professionally. We asked if he would like to pick up his things, including the chair. He said we could keep the chair. A few days later he sent a respectful email saying he was disappointed but moving on from this incident "without prejudice." These words would come back to haunt us shortly.

Flash forward a year. We are closing our seed round of $750K — everyone is excited, things are moving forward, and it is a dream come true. During due diligence, our lawyer discovered the email from Bill and asked us about it. We assured not to worry because he never worked for us and he never signed any papers. We were just testing it to see how it would work out — water under the bridge.

Only our investor's lawyer didn't think so. Those words "without prejudice" are lawyer speak for "I'm not going to do anything now, but I retain my right to sue you at any time in the future for any amount." They wouldn't do the deal unless we got Bill to sign off any claims against us.

We were scared. If Bill were to find out that his signature stood in the way of us closing $750K, what would he ask for? Steve volunteered to talk to Bill.

Later that day, Steve called and said "Bill will sign the papers if we pay him $5000." I was looking at that damn faux-leather chair and told him, "We have no choice. Do the deal" (or something like that). Damn. That chair just cost us five grand, I remember thinking.

I kept that chair for the life of Distil as a reminder that the littlest things can cost you big. ■

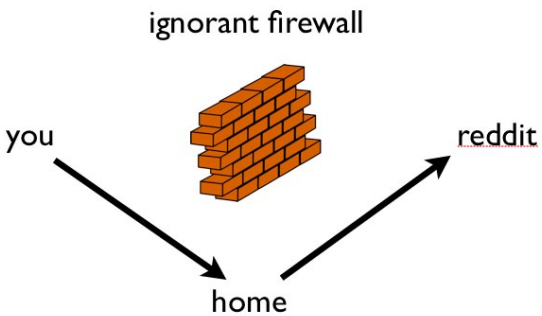
Kenton White is a technical entrepreneur in Ottawa, Ontario. He was a co-founder of DISTIL Interactive. His current company is Girih.

Reprinted with permission of the original author.
First appeared in *hn.my/chair*

SSH: More Than Secure Shell

By MATT MIGHT

SSH is a protocol for authenticating and encrypting remote shell sessions, but using SSH for just remote shell sessions ignores 90% of what it can do.




```
# ssh home -L 80:reddit.com:80
```

This article covers less common SSH use cases, such as:

- using password-less, key-based login;
- setting up local per-host configurations;
- exporting a local service through a firewall;
- accessing a remote service through a firewall;
- setting up a SOCKS proxy for Firefox;
- executing commands remotely from scripts;
- transferring files to/from remote machines;
- mounting a file system through SSH; and
- triggering admin scripts from a phone.

Why SSH?

As recently as 2001, it was not uncommon to log in to a remote Unix system using telnet. Telnet is just above netcat in protocol sophistication, which means that passwords were sent in the clear. As wifi proliferated, telnet went from security nuisance to security disaster. While an undergrad, I remember running ethereal (now wireshark) in the school commons area, snagging about a dozen root passwords in an hour.

SSH, which encrypts and authenticates connections, had been in development since 1995, but it seemed to become adopted nearly universally and almost overnight around 2002.

It is worth configuring SSH properly:

- per-user configuration is in `~/.ssh/config`;
- system-wide client configuration is in `/etc/ssh/ssh_config`.
- system-wide daemon configuration is in `/etc/ssh/sshd_config`.

Key-based, password-less authentication

Key-based, password-less authentication makes it less cumbersome for other programs and scripts to piggyback atop SSH since you won't have to re-enter your password each time. Key-based authentication exploits public-key cryptography to prove to the server that the client owns the secret private key without revealing the key.

To set this up:

Log in to the client machine.

Create a private/public key pair with ssh-keygen:

```
$ ssh-keygen -t dsa
```

This will place the private key in `~/.ssh/id_dsa` and the public key in: `~/.ssh/id_dsa.pub`

Set appropriate permissions to guard the private key as if the private key were your password. In effect, it is.

Now, append the contents of `~/.ssh/id_dsa.pub` to the end of `~/.ssh/authorized_keys` on the remote machine. For example:

```
$ cat .ssh/id_dsa.pub |
ssh host 'cat >> ~/.ssh/
authorized_keys'
```

On Linux systems, you can use `ssh-copy-id` instead; the technique above is more portable.

Do not copy your private key over.

Now, a password isn't required when you connect to that account.

Executing remote commands

To run a command on a remote system without logging in, specify the command after the login information:

```
$ ssh host command
```

For example, to check remote disk space:

```
$ ssh host df
```

My favorite example for Linux is piping the microphone from one machine to the speakers of another:

```
$ dd if=/dev/dsp | ssh -C
user@host dd of=/dev/dsp
```

Copying files with ssh

For copying data and files over SSH, there are a few options.

It's possible to copy with the command `cat`. If you're trying to copy the output of a process instead of a file, this is certainly a reasonable route.

If you're going to use SSH like this, disable the escape sequences:

```
$ cat file | ssh -e none
remote-host 'cat > file'
```

If these are going to be large files, you may want to use the `-C` flag to enable compression.

For copying files, the program `scp` works like `cp`, except it also accepts remote destinations. For example:

```
$ scp .bash_profile matt@exam-
ple.com:~/.bash_profile
```

For an FTP-like interface for copying files, use the program `sftp`.

Per-host SSH client configuration options

You can set per-host configuration options in `~/.ssh/config` by specifying `Host` *`hostname`*, followed by host-specific options. It is possible to set the private key and the user (among other settings) on a per-host basis. Here's an example config file:

```
Host my-server.com
User admin
IdentityFile ~/.ssh/admin.
id_dsa
BatchMode yes
EscapeChar none

Host mm
User matt
HostName might.net
IdentityFile ~/.ssh/matt.id_dsa

Host *.lab.ucaprica.edu
User u8193
```

The first example enables batch mode, which means it will never ask for a passphrase or password for this host. It also disables an escape sequence, which avoids any hiccups when transmitting arbitrary data. If `ssh` is to be invoked within scripts, this is a good option.

The second example uses a `HostName` abbreviation, so that `ssh mm` is equivalent to `ssh -i ~/.ssh/matt.id_dsa matt@might.net`.

The third example sets the user to u8193 for any machine in the subdomain lab.ucaprica.edu.

See more options in `man ssh_config`.

Configuring sshd

The options most frequently tweaked are:

- `Port`: set this to the port on which you want sshd to run. Unless you have a compelling reason to move it, keep it on 22.
- `PermitRootLogin`: set this to `no` and then configure `sudo` to add a little security; another good setting is `without-password`, which will force the use of public key authentication for `root`.
- `PasswordAuthentication`: set this to `no` to disallow password authentication entirely and to require public key authentication.

The man page for `sshd_config` summarizes the remaining options well.

Local port forwarding

SSH allows secure port forwarding. For example, suppose you want to connect from client A to server B and route traffic securely through server C.

From A, run:

```
A$ ssh C -L
localport:B:remoteport
```

Then, connect to localhost:localport to connect to B:remoteport. If you use add -g, then anyone that can reach A may connect to B:remoteport through A:localport. This is useful for evading firewalls.

For example, suppose your work banned reddit.com. Run the following:

```
# ssh yourserver -L 80:reddit.
com:80
```

And, set the address of reddit.com and www.reddit.com to 127.0.0.1 in /etc/hosts.

You will also need to disable any local web server running first.

Now, it will surreptitiously traffic to reddit.com through your *yourserver*. If you do this frequently, you might want to add a special host:

```
Host redditfw
HostName yourserver
LocalForward 80 reddit.com:80
```

Remote port forwarding

Alternatively, suppose you wanted to give remote machine B access to another machine, A, by passing securely through your local machine C.

Then, on C, you can run:

```
C$ ssh B -R
remoteport:A:targetport
```

At this point, local users on B can connect to A:targetport through localhost:remoteport.

If you want to allow nonlocal users to connect A:targetport through localhost:remoteport, then set the following in the sshd_config file:

```
GatewayPorts yes
```

Once again, if you do this frequently, set up a special host in ~/.ssh/config:

```
Host exportme
HostName B
RemoteForward remoteport
A:targetport
```

Setting up a SOCKS proxy for Firefox

SSH can also set up a SOCKS proxy to evade a firewall by simply running the following:

```
$ ssh -D localport host
```

In Firefox, under Preferences > Advanced > Network, select "Settings." Set your SOCKS5 proxy to localhost port localport.

Test it out by googling "what is my ip."

Firefox will now forward your web traffic through host. A word of caution: this will not forward your DNS requests.

If you need to hide your DNS requests as well, I recommend installing DNSCrypt from OpenDNS.

In about:config, you can also tell Firefox to forward your DNS requests by setting the following to true:

```
network.proxy.socks_remote_dns
```

SSH as a filesystem: sshfs

Using the FUSE project with sshfs, it's possible to mount a remote filesystem over SSH. On Mac, use Fuse4x. From MacPorts, install it all with:

```
$ sudo port install sshfs
```

Once it's installed, run:

```
$ sshfs remote-host:
local-mount-directory
```

SSH from windows

Sometimes, you need to get to your home machine from Windows. In these cases, you want the PuTTY suite of tools.

SSH from iOS

Using SSH from iOS can be cumbersome, but the iSSH app is particularly well-suited to administrative tasks. The iSSH app allows storing configurations, which enables per-machine private keys and remote commands to run upon connecting. This means you can create a configuration that logs in to run a shell script.

For instance, I have three command-based configurations for might.net:

- a script to (re)start the web server;
- a script to (re)start the DNS server; and
- a script to reboot the entire server. ■

Matt Might is a professor of Computer Science at the University of Utah. His research interests include programming language design, static analysis and compiler optimization. He blogs at *matt.might.net/articles* and tweets from @mattmight

Reprinted with permission of the original author.
First appeared in *hn.my/sshtricks* (matt.might.net)

Google tracks you. We don't.

Python: Copying a List the Right Way

By HENRY PRÊCHEUR

```
new = old[:]
```

Those proficient in Python know that the previous line copies the list `old` into `new`. However, this is confusing for beginners and should be avoided. Sadly, the `[:]` notation is widely used, probably because most Python programmers don't know of a better way to copy lists.

A little bit of pythonic theory

First, we need to understand how Python manages objects and variables. Python doesn't have variables like C. In C, a variable is not just a name, it is a set of bits with the variable existing somewhere in memory. In Python, on the other hand, variables are just tags attached to objects.

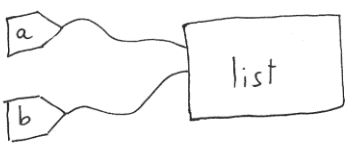
Consider the following statement:

```
a = [1, 2, 3]
```

It means that a points to the list `[1, 2, 3]` we just created, but a is not the list. If we do:

```
b = a
```

We didn't copy the list referenced by `a`. We just created a new tag `b` and attached it to the list pointed by `a`. Like in the picture below:



If you modify `a`, you also modify `b` since they point to the same list:

```
>>> a.append(4)

>>> print a
[1, 2, 3, 4]

>>> print b
[1, 2, 3, 4]
```

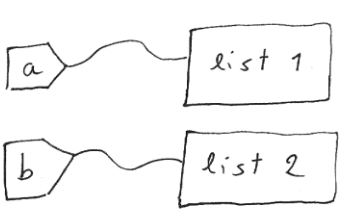
The built-in function `id()` helps keep track of all this because it returns the object's unique id, which is the object's memory address.

```
>>> id(a)
3080501452L

>>> id(b)
3080501452L

>>> c = []  # Create a new list
>>> id(c)
3080609228L
```

`a` and `b` really do point to the same memory address while `c` points to a new empty list, different from the one referenced by `a` and `b`.

Back to our list

Now we want to copy the list referenced by `a`. In order to do this, we need to create a new list and attach it to `b`.



That brings us back to `new = old[:]`. The operator `[:]` returns a slice of a sequence. Slicing a portion of a list creates a new list and copies the portion into this new list.

```
>>> a = [1, 2, 3, 4]
>>> a[1:3]
[2, 3]

>>> id(a)
3080104140L

>>> id(a[1:3])
3080513612L
```

If you omit the first index, the slice starts at the beginning of the list; omit the second index, it stops at the end of the list.

```
>>> a[:3]
[1, 2, 3]

>>> a[1:]
[2, 3, 4]
```

By calling `a[:]`, you get a slice of a starting from the beginning and finishing at the end. That's a full copy of `a`, but it's not the only way to copy lists. What about this one?

```
>>> b = list(a)

>>> id(a)
3080104140L

>>> id(b)
3080520556L
```

Isn't it better, less cryptic, and more pythonic? `a[:]` feels a bit too much like Perl. Unlike with the slicing notation, those who don't know Python will understand that b contains a list.

`list()` is the list constructor. It will create a new list based on the passed sequence. The sequence doesn't necessarily need to be a list; it can be any kind of sequence.

```
>>> my_tuple = (1, 2, 3)
>>> my_list = list(my_tuple)
>>> print my_list
[1, 2, 3]
```

Additionally, this method also works with generators while `[:]` doesn't since generators are unsubscriptable — you can't do generator[0], for example.

```
>>> generator = (x * 3 for x in range(4))
>>> list(generator)
[0, 3, 6, 9]
```

In 90% of instances `[:]` can be replaced by `list()`. Of course, it won't work for everything since the two are not strictly equivalent, but it is worth trying. Next time you see `[:]`, try to replace it with list, and your code should be more readable. Do it — the devil is in the details. ■

Henry is a Python hacker living in Vancouver BC, Canada. He likes to polish bits of code for hours, and he writes about things he's not necessarily good at.

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

Invaluable Command Line Tools For Web Developers

$ By Ben Dowling

Life as a web developer can be hard when things start going wrong. The problem could be in any number of places. Is there a problem with the request you're sending? Is the problem with the response? Is there a problem with a request in a third party library you're using, or is an external API failing? There are many different tools that can make our life a little bit easier. Here are some command line tools that I've found to be invaluable.

Curl

Curl is a network transfer tool that's very similar to wget, the main difference being that by default wget saves to file, and curl outputs to the command line. This makes is really simple to see the contents of a website. Here, for example, we can get our current IP from the *ifconfig.me* website:

```
$ curl ifconfig.me
93.96.141.93
```

Curl's `-i` (show headers) and `-I` (show only headers) option make it a great tool for debugging HTTP responses and finding out exactly what a server is sending to you:

```
$ curl -I news.ycombinator.com
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cache-Control: private
Connection: close
```

The `-L` option is handy, and makes Curl automatically follow redirects. Curl has support for HTTP Basic Auth, cookies, manually settings headers, and much, much more.

Siege

Siege is an HTTP benchmarking tool. In addition to the load testing features, it has a handy `-g` option that is very similar to `curl -iL` except it also shows you the request headers. Here's an example with www.google.com (I've removed some headers for brevity):

```
$ siege -g www.google.com
GET / HTTP/1.1
Host: www.google.com
User-Agent: JoeDog/1.00 [en] (X11; I; Siege
2.70)
Connection: close

HTTP/1.1 302 Found
Location: http://www.google.co.uk/
Content-Type: text/html; charset=UTF-8
Server: gws
Content-Length: 221
Connection: close

GET / HTTP/1.1
Host: www.google.co.uk
User-Agent: JoeDog/1.00 [en] (X11; I; Siege
2.70)
Connection: close

HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
X-XSS-Protection: 1; mode=block
Connection: close
```

What Siege is really great at is server load testing. Just like `ab` (apache benchmark tool), you can send a number of concurrent requests to a site and see how it handles the traffic. With the following command we test Google with 20 concurrent connections for 30 seconds, and then get a nice report at the end:

```
$ siege -c20 www.google.co.uk -b -t30s
...
Lifting the server siege...      done.
Transactions:                    1400 hits
Availability:                  100.00 %
Elapsed time:                   29.22 secs
Data transferred:               13.32 MB
Response time:                   0.41 secs
Transaction rate:               47.91 trans/sec
Throughput:                      0.46 MB/sec
Concurrency:                    19.53
Successful transactions:         1400
Failed transactions:                0
Longest transaction:             4.08
Shortest transaction:            0.08
```

One of the most useful features of Siege is that it can take a URL file as input and hit those URLs rather than just a single page. This is great for load testing, because you can replay real traffic against your site and see how it performs, rather than just hitting the same URL again and again. Here's how you would use Siege to replay your apache logs against another server to load test it with:

```
$ cut -d ' ' -f7 /var/log/apache2/access.log >
urls.txt
$ siege -c<concurrency rate> -b -f urls.txt
```

Ngrep

For serious network packet analysis there's Wireshark, with its thousands of settings, filters, and different configuration options. There's also a command line version, tshark. For simple tasks I find Wireshark can be overkill, so unless I need something more powerful, ngrep is my tool of choice. It lets you do with network packets what grep does with files.

For web traffic you almost always want the `-W byline` option, which preserves linebreaks, and `-q`, a useful argument which suppresses some additional output about non-matching packets. Here's an example that captures all packets that contain GET or POST:

```
ngrep -q -W byline "^(GET|POST) .*"
```

You can also pass in additional packet filter options, such as limiting the matched packets to a certain host, IP or port. Here we filter all traffic going to or coming from google.com, port 80, and that contains the term "search."

```
ngrep -q -W byline "search" host www.google.com
and port 80
```

Ben Dowling is a software engineer who has launched sites such as Do Nothing for 2 Minutes, BusMapper and Geomium. He also co-organises the monthly HN London meetup, blogs about development at *coderholic.com* and also tweets as @coderholic

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

Signs That You're a Bad Programmer

By CHRIS WENHAM

1 Inability to Reason About Code

Reasoning about code means being able to follow the execution path ("running the program in your head") while knowing what the goal of the code is.

Symptoms

1. The presence of "voodoo code," or code that has no effect on the goal of the program but is diligently maintained anyway (such as initializing variables that are never used, calling functions that are irrelevant to the goal, producing output that is not used, etc.)
2. Executing idempotent functions multiple times (e.g.: calling the `save()` function multiple times "just to be sure")
3. Fixing bugs by writing code that overwrites the result of the faulty code
4. "Yo-Yo code" that converts a value into a different representation, then converts it back to where it started (e.g.: converting a decimal into a string and then back into a decimal, or padding a string and then trimming it)
5. "Bulldozer code" that gives the appearance of refactoring by breaking out chunks into subroutines, but that are impossible to reuse in another context (very high cohesion)

Remedies

To get over this deficiency a programmer can practice by using the IDE's own debugger as an aide, if it has the ability to step through the code one line at a time. In Visual Studio, for example, this means setting a breakpoint at the beginning of the problem area and stepping through with the "F11" key, inspecting the value of variables — before and after they change — until you understand what the code is doing. If the target environment doesn't have such a feature, then do your practice-work in one that does.

The goal is to reach a point where you no longer need the debugger to be able to follow the flow of code in your head, and where you are patient enough to think about what the code is doing to the state of the program. The reward is the ability to identify redundant and unnecessary code, as well as how to find bugs in existing code without having to re-implement the whole routine from scratch.

2 Poor Understanding of the Language's Programming Model

Object Oriented Programming is an example of a language model, as is Functional or Declarative programming. They're each significantly different from procedural or imperative programming, just as procedural programming is significantly different from assembly or GOTO-based programming. Then there are languages which follow a major programming model (such as OOP) but introduce their own improvements such as list comprehensions, generics, duck-typing, etc.

Symptoms

1. Using whatever syntax is necessary to break out of the model, then writing the remainder of the program in their familiar language's style

2. (OOP) Attempting to call non-static functions or variables in uninstantiated classes, and having difficulty understanding why it won't compile
3. (OOP) Writing lots of "xxxxx-Manager" classes that contain all of the methods for manipulating the fields of objects that have little or no methods of their own
4. (Relational) Treating a relational database as an object store and performing all joins and relation enforcement in client code
5. (Functional) Creating multiple versions of the same algorithm to handle different types or operators, rather than passing high-level functions to a generic implementation
6. (Functional) Manually caching the results of a deterministic function on platforms that do it automatically (such as SQL and Haskell)
7. Using cut-n-paste code from someone else's program to deal with I/O and Monads
8. (Declarative) Setting individual values in imperative code rather than using data-binding

Remedies

If your skills deficiency is a product of ineffective teaching or studying, then an alternative teacher is the compiler itself. There is no more effective way of learning a new programming model than starting a new project and committing yourself to use whatever the new constructs are, intelligently or not. You also need to practice explaining the model's features in crude terms of whatever you are familiar with, then recursively building on your new vocabulary until you understand the subtleties as well. For example:

- Phase 1: "OOP is just records with methods."
- Phase 2: "OOP methods are just functions running in a mini-program with its own global variables."
- Phase 3: "The global variables are called fields, some of which are private and invisible from outside the mini-program."
- Phase 4: "The idea of having private and public elements is to hide implementation details and expose a clean interface, and this is called Encapsulation."
- Phase 5: "Encapsulation means my business logic doesn't need to be polluted with implementation details."

Phase 5 looks the same for all languages, since they are all really trying to get the programmer to the point where he can express the intent of the program without burying it in the specifics of how. Take functional programming as another example:

- Phase 1: "Functional programming is just doing everything by chaining deterministic functions together."
- Phase 2: "When the functions are deterministic the compiler can predict when it can cache results or skip evaluation, even when it's safe to prematurely stop evaluation."
- Phase 3: "In order to support Lazy and Partial Evaluation, the compiler requires that functions are defined in terms of how to transform a single parameter, sometimes into another function. This is called Currying."
- Phase 4: "Sometimes the compiler can do the Currying for me."
- Phase 5: "By letting the compiler figure out the mundane details, I can write programs by describing what I want, rather than how to give it to me."

3 Deficient Research Skills/Chronically Poor Knowledge of the Platform's Features

Modern languages and frameworks now come with an awesome breadth and depth of built-in commands and features, with some leading frameworks (Java, .Net, Cocoa) being too large to expect any programmer, even a good one, to learn in anything less than a few years. But a good programmer will search for a built-in function that does what they need before they begin to roll their own, and excellent programmers have the skill to break-down and identify the abstract problems in their task, then search for existing frameworks, patterns, models, and languages that can be adapted before they even begin to design the program.

Symptoms

These are only indicative of the problem if they continue to appear in the programmer's work long after he should have mastered the new platform.

1. Re-inventing or laboring without basic mechanisms that are built into the language, such as events-and-handlers or regular expressions
2. Re-inventing classes and functions that are built into the framework (e.g.: timers, collections, sorting and searching algorithms)*

3. "Email me teh code, plz" messages posted to help forums
4. "Roundabout code" that accomplishes in many instructions what could be done with far fewer (e.g.: rounding a number by converting a decimal into a formatted string, then converting the string back into a decimal)
5. Persistently using old-fashioned techniques even when new techniques are better in those situations (e.g.: still writes named delegate functions instead of using lambda expressions)
6. Having a stark "comfort zone," and going to extreme lengths to solve complex problems with primitives

** Accidental duplication will also happen, proportionate to the size of the framework, so judge by degree. Someone who hand-rolls a linked list might Know What They Are Doing, but someone who hand-rolls their own StrCpy() probably does not.*

Remedies

A programmer can't acquire this kind of knowledge without slowing down, and it's likely that he's been in a rush to get each function working by whatever means necessary. He needs to have the platform's technical reference handy and be able to look through it with minimal effort, which can mean either having a hard copy of it on the desk right next to the keyboard, or having a second monitor dedicated to a browser. To get into the habit initially, he should refactor his old code with the goal of reducing its instruction count by 10:1 or more.

4 Inability to Comprehend Pointers

If you don't understand pointers then there is a very shallow ceiling on the types of programs you can write, as the concept of pointers enables the creation of complex data structures and efficient APIs. Managed languages use references instead of pointers, which are similar but add automatic dereferencing and prohibit pointer arithmetic to eliminate certain classes of bugs. They are still similar enough, however, that a failure to grasp the concept will be reflected in poor data-structure design and bugs that trace back to the difference between pass-by-value and pass-by-reference in method calls.

Symptoms

1. Failure to implement a linked list, or write code that inserts/deletes nodes from linked list or tree without losing data
2. Allocating arbitrarily big arrays for variable-length collections and maintaining a separate collection-size counter, rather than using a dynamic data structure
3. Inability to find or fix bugs caused by mistakenly performing arithmetic on pointers
4. Modifying the dereferenced values from pointers passed as the parameters to a function, and not expecting it to change the values in the scope outside the function
5. Making a copy of a pointer, changing the dereferenced value via the copy, then assuming the original pointer still points to the old value
6. Serializing a pointer to the disk or network when it should have been the dereferenced value
7. Sorting an array of pointers by performing the comparison on the pointers themselves

Remedies

> *"A friend of mine named Joe was staying somewhere else in the hotel and I didn't know his room number. But I did know which room his acquaintance, Frank, was staying in. So I went up there and knocked on his door and asked him, "Where's Joe staying?" Frank didn't know, but he did know which room Joe's co-worker, Theodore, was staying in, and gave me that room number instead. So I went to Theodore's room and asked him where Joe was staying, and Theodore told me that Joe was in Room 414. And that, in fact, is where Joe was."*

Pointers can be described with many different metaphors, and data structures into many analogies. The above is a simple analogy for a linked list, and anybody can invent their own, even if they aren't programmers. The comprehension failure doesn't occur when pointers are described, so you can't describe them any more thoroughly than they already have been. It fails when the programmer then tries to visualize what's going on in the computer's memory and gets it conflated with their understanding of regular variables, which are very similar. It may help to translate the code into a simple story to help reason about what's going on, until the distinction clicks and the programmer can visualize pointers and the data structures they enable as intuitively as scalar values and arrays.

5 Difficulty Seeing Through Recursion

The idea of recursion is easy enough to understand, but programmers often have problems imagining the result of a recursive operation in their minds, or how a complex result can be computed with a simple function. This makes it harder to design a recursive function because you have trouble picturing "where you are" when you come to writing the test for the base condition or the parameters for the recursive call.

Symptoms

1. Hideously complex iterative algorithms for problems that can be solved recursively (e.g.: traversing a filesystem tree), especially where memory and performance are not a premium
2. Recursive functions that check the same base condition both before and after the recursive call
3. Recursive functions that don't test for a base condition
4. Recursive subroutines that concatenate/sum to a global variable or a carry-along output variable
5. Apparent confusion about what to pass as the parameter in the recursive call, or recursive calls that pass the parameter unmodified
6. Thinking that the number of iterations is going to be passed as a parameter

Remedies

Get your feet wet and be prepared for some stack overflows. Begin by writing code with only one base-condition check and one recursive call that uses the same, unmodified parameter that was passed. Stop coding even if you have the feeling that it's not enough, and run it anyway. It throws a stack-overflow exception, so now go back and pass a modified copy of the parameter in the recursive call. More stack overflows? Excessive output? Then do more code-and-run iterations, switching from tweaking your base-condition test to tweaking your recursive call until you start to intuit how the function is transforming its input. Resist the urge to use more than one base-condition test or recursive call unless you really Know What You're Doing.

Your goal is to have the confidence to jump in, even if you don't have a complete sense of "where you are" in the imaginary recursive path. Then when you need to write a function for a real project you'd begin by writing a unit test first, and proceeding with the same technique above.

6 Distrust of Code

Symptoms

1. Writing `IsNull()` and `IsNotNull()`, or `IsTrue(bool)` and `IsFalse(bool)` functions
2. Checking to see if a Boolean-typed variable is something other than true or false

Remedies

Are you being paid by the line? Are you carrying over old habits from a language with a weak type system? If neither, then this condition is similar to the inability to reason about code, but it seems that it isn't reasoning that's impaired, but trust and comfort with the language. Some of the symptoms are more like "comfort code" that doesn't survive logical analysis, but that the programmer felt compelled to write anyway. The only remedy may be more time to build up familiarity. ■

Chris Wenham has been programming since he was 10 (beginning with a Sinclair ZX Spectrum), herded a team of grumpy IT programmers, and lurked in the bowels of faceless Enterprise software companies. He now works independently.

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

Differences Between jQuery bind(), live(), delegate() and on()

By ELIJAH MANOR

I've seen quite a bit of confusion from developers about what the real differences are between the jQuery `.bind()`, `.live()`, `.delegate()`, and `.on()` methods and when they should be used.

Before we dive into the ins and outs of these methods, let's start with some common HTML markup that we'll be using as we write sample jQuery code.

```
<ul id="members" data-role="listview"
data-filter="true">
    <!-- ... more list items ... -->
    <li>
        <a href="detail.html?id=10">
            <h3>John Resig</h3>
            <p><strong>
                jQuery Core Lead
            </strong></p>
            <p>Boston, United States</p>
        </a>
    </li>
    <!-- ... more list items ... -->
</ul>
```

Using the Bind Method

The `.bind()` method registers the type of event and an event handler directly to the DOM element in question. This method has been around the longest, and in its day it was a nice abstraction around the various cross-browser issues that existed. This method is still very handy when wiring-up event handlers, but there are various performance concerns as are listed below.

```
/* The .bind() method attaches the event handler directly
to the DOM element in question ( "#members li a" ). The
.click() method is just a shorthand way to write the
.bind() method. */

$( "#members li a" ).bind( "click", function( e ) {} );
$( "#members li a" ).click( function( e ) {} );
```

The `.bind()` method will attach the event handler to all of the anchors that are matched! That is not good. Not only is it expensive to implicitly iterate over all of those items to attach an event handler, but it is also wasteful since it is the same event handler over and over again.

Pros

- This methods works across various browser implementations.
- It is pretty easy and quick to wire-up event handlers.

- The shorthand methods (`.click()`, `.hover()`, etc...) make it even easier to wire-up event handlers.
- For a simple ID selector, using `.bind()` not only wires-up quickly, but also when the event fires, the event handler is invoked almost immediately.

Cons

- The method attaches the same event handler to every matched element in the selection.
- It doesn't work for elements added dynamically that matches the same selector.
- There are performance concerns when dealing with a large selection.
- The attachment is done upfront which can have performance issues on page load.

Using the Live Method

The `.live()` method uses the concept of event delegation to perform its so-called "magic." The way you call `.live()` looks just like how you might call `.bind()`, which is very convenient. However, under the covers this method works much differently. The `.live()` method attaches the event handler to the root level document along with the associated selector and event information. By registering this information on the document it allows one event handler to be used for all events that have bubbled (a.k.a. delegated, propagated) up to it. Once an event has bubbled up to the document, jQuery looks at the selector/event metadata to determine which handler it should invoke, if any. This extra work has some impact on performance at the point of user interaction, but the initial register process is fairly speedy.

```
/* The .live() method attaches the event handler to the
root level document along with the associated selector
and event information ( "#members li a" & "click" ) */

$( "#members li a" ).live( "click", function( e ) {} );
```

The good thing about this code as compared to the `.bind()` example above is that it is only attaching the event handler to the document once instead of multiple times. This is faster and less wasteful; however, there are many problems with using this method, and they are outlined below.

Pros

- There is only one event handler registered instead of the numerous event handlers that could have been registered with the `.bind()` method.
- The upgrade path from `.bind()` to `.live()` is very small. All you have to do is replace "bind" to "live."
- Elements dynamically added to the DOM that match the selector magically work because the real information was registered on the document.
- You can wire-up event handlers before the document-ready event, helping you utilize possibly unused time.

Cons

- This method is deprecated as of jQuery 1.7, and you should start phasing out its use in your code.
- Chaining is not properly supported using this method.
- The selection is basically thrown away, since it is only used to register the event handler on the document.
- Using `event.stopPropagation()` is no longer helpful because the event has already delegated all the way up to the document.
- Since all selector/event information is attached to the document, once an event does occur, jQuery has matched through its large metadata store using the `matchesSelector` method to determine which event handler to invoke, if any.
- Your events always delegate all the way up to the document. This can affect performance if your DOM is deep.

Using the Delegate Method

The `.delegate()` method behaves in a similar fashion to the `.live()` method, but instead of attaching the selector/event information to the document, you can choose where it is anchored. Just like the `.live()` method, this technique uses event delegation to work correctly.

```
/* The .delegate() method behaves in a
similar fashion to the .live() method, but
instead of attaching the event handler to
the document, you can choose where it is
anchored ( "#members" ). The selector and
event information ( "li a" & "click" )
will be attached to the "#members" ele-
ment. */

$( "#members" ).delegate( "li a", "click",
function( e ) {} );
```

The `.delegate()` method is very powerful. The above code will attach the event handler to the unordered list ("#members") along with the selector/event information. This is much more efficient than the `.live()` method, which always attaches the information to the document. In addition, a lot of other problematic issues were resolved by introducing the `.delegate()` method. See the following outline for a detailed list.

Pros

- You have the option of choosing where to attach the selector/event information.
- The selection isn't actually performed up front, but is only used to register onto the root element.
- Chaining is supported correctly.
- jQuery still needs to iterate over the selector/event data to determine a match, but since you can choose where the root is, the amount of data to sort through can be much smaller.
- Since this technique uses event delegation, it can work with dynamically added elements to the DOM where the selectors match.
- As long as you delegate against the document, you can also wire-up event handlers before the document-ready event.

Cons

- Changing from a `.bind()` to a `.delegate()` method isn't as straight forward.
- There is still the concern of jQuery having to figure out, using the `matchesSelector` method, which event handler to invoke based on the selector/event information stored at the root element. However, the metadata stored at the root element should be considerably smaller compared to using the `.live()` method.

Using the On Method

Did you know that the jQuery `.bind()`, `.live()`, and `.delegate()` methods are just one line pass-through to the new jQuery 1.7 `.on()` method? The same is true of the `.unbind()`, `.die()` and `.undelegate()` methods. The following code snippet is taken from the jQuery 1.7.1 codebase in GitHub:

```
// ... more code ...

bind: function( types, data, fn ) {
    return this.on( types, null, data, fn );
},
unbind: function( types, fn ) {
    return this.off( types, null, fn );
},

live: function( types, data, fn ) {
    jQuery( this.context ).on( types, this.selector,
data, fn );
    return this;
},
die: function( types, fn ) {
    jQuery( this.context ).off( types, this.selector
|| "**", fn );
    return this;
},

delegate: function( selector, types, data, fn ) {
    return this.on( types, selector, data, fn );
},
undelegate: function( selector, types, fn ) {
    return arguments.length == 1 ?
        this.off( selector, "**" ) :
        this.off( types, selector, fn );
},

// ... more code ...
```

With that in mind, the usage of the new `.on()` method looks something like the following:

```
/* The jQuery .bind(), .live(), and .delegate() methods
are just one line pass throughs to the new jQuery 1.7
.on() method */

// Bind
$( "#members li a" ).on( "click", function( e ) {} );
$( "#members li a" ).bind( "click", function( e ) {} );

// Live
$( document ).on( "click", "#members li a", function( e )
{} );
$( "#members li a" ).live( "click", function( e ) {} );

// Delegate
$( "#members" ).on( "click", "li a", function( e ) {} );
$( "#members" ).delegate( "li a", "click", function( e )
{} );
```

You'll notice that how I call the `.on()` method changes how it performs. You can consider the `.on()` method as being "overloaded" with different signatures, which in turn changes how the event binding is wired-up. The `.on()` method brings a lot of consistency to the API and hopefully makes things slightly less confusing.

Pros

- Brings uniformity to the various event-binding methods.
- Simplifies the jQuery code base and removes one level of redirection since the `.bind()`, `.live()`, and `.delegate()` call this method under the covers.
- Still provides all the goodness of the `.delegate()` method, while still providing support for the `.bind()` method if you need it.

Cons

- Brings confusion because the behavior changes based on how you call the method.

Conclusion (tl;dr)

If you have been confused about the various different types of event binding methods then don't worry, there has been a lot of history and evolution in the API over time. There are many people that view these methods as magic, but once you uncover some of how they work, you understand how to better code inside of your projects.

The biggest take-aways:

- Using the `.bind()` method is very costly as it attaches the same event handler to every item matched in your selector.
- You should stop using the `.live()` method, as it is deprecated and has a lot of problems with it.
- The `.delegate()` method gives a lot of "bang for your buck" when dealing with performance and reacting to dynamically added elements.
- The new `.on()` method is mostly syntax sugar that can mimic `.bind()`, `.live()`, or `.delegate()` depending on how you call it.
- The new direction is to use the new .on method. Get familiar with the syntax and start using it on all your jQuery 1.7+ projects. ■

Elijah Manor (@elijahmanor) is a Christian and a family man. He develops at *appendTo.com* as a System Architect and the Director of Training providing corporate jQuery support, training, and consulting. He is a Microsoft Regional Director, an ASP.NET MVP, and an ASPInsider and specializes in front-end web development.

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

How I Develop Things and Why

By KENNETH REITZ

I've always considered myself a bit of a software junkie. Nothing excites me more than a great piece of new software.

Some of my best childhood memories are our trips to Grandma's house, where I'd have access to a computer with a dial-up connection that I'd use to obtain freeware and shareware. I'd bring four or five floppies with me and try to cram all the games, waveform editors, and utilities that I could sneaker-net home.

Luckily today, excellent software written with passion oozes out of the app ecosystem. OS X and the App Store really fuel an economy of software built for humans by people that care.

Unfortunately, this doesn't always hold true in developer software — text editors, modules, libraries, toolchains, etc. We are forced to deal with APIs on a daily basis that were not built with the user in mind. Over-engineering surrounds us as developers. Things that should be simple are often needlessly complex for the sake of being complex and "proper."

Why should consumer apps and developer APIs be treated differently?

Have an Issue

The first step to developing something great is to have a real problem. You can't solve a problem properly if you don't experience it firsthand.

On the consumer app side of things, a great example of this is Microsoft OneNote. Have you used OneNote? It's incredible.

Essentially, OneNote is hierarchical freeform note-taking software that assumes nothing: you can type, use handwriting, embed files, cross-link notes, sync them online, etc.

Unfortunately, OneNote is only available on Windows. This kills me. I would love to think that Microsoft would port this lovely piece of software to OS X, but I doubt it will ever happen.

If I ever decide to actually ship a consumer product, it will be something akin to OneNote for OS X. It would be incredible. It may not be for many, but for people that resonate with my problem, it will work wonderfully. It would be a reaction to a real problem, not an engineered app an entrepreneur thinks will fill a gap so he can make some fast cash.

GitHub wasn't built for the developer community at large; the founders built GitHub for themselves. The problem they solved simply happened to resonate with millions of developers because they themselves happen to be developers.

37Signals didn't build Basecamp for a world full of project managers and consultants; they built it for themselves. They also developed Ruby on Rails for themselves, as Ruby developers that were repeating themselves too often.

How pragmatic.

These companies didn't need to commission lengthy case studies and perform market analysis. They didn't set up faux AdWords to measure the effectiveness of various marketing copy. Yet, they are astronomically successful. How is this possible? They know exactly what they want to build, how it should function, and how it should look because they were building it for themselves and not for others.

Let's go back to the developer's side of things.

> **“Build tools for others that you want to be built for you.”**

A great example is my Requests module [hn.my/request]. I was a heavy user of Convore at the time, and I wanted to interface with it programmatically. So, I set out to build a Python module that wrapped the Convore HTTP API. Unfortunately, this was easier said than done. Dealing with Python's standard library for HTTP was a complete and total nightmare. It was over-engineered.

I love Python because it's a language designed for humans. Why should modern HTTP be so difficult? So, I sat out to discover what it was that I wanted, and built exactly what I needed. It resonated well with others.

Nothing is more satisfying than using your own tools to Get Things Done.

Respond with a README

Before I start writing a single line of code, I write the README and fill it with usage examples. I pretend that the module I want to build is already written and available, and I write some code with it.

This has an incredible effect: instead of engineering something that will only get the job done, you start to interact with the problem itself and build an interface that reacts to it.

You discover it. You respond to it.

Great sculptures aren't manufactured — they're discovered. The sculptor studies and listens to the slab of marble. He identifies with the stone. Then, he responds. He enables the marble to speak for itself, setting free something beautiful that hidden was inside all along.

He responds.

This is what responsive design is all about. It's not merely a method to engineer a web design that will function on a phone, tablet, and desktop.

> *Beware lest you lose the substance by grasping at the shadow.*

Responsive design is about making a design that identifies and understands itself enough to respond to the environment it's placed in. It is about setting your design free from arbitrary constraints. It is setting free something beautiful that was inside all along.

This is known as Readme-Driven Development [hn.my/rdd]. I call it **Responsive API Design**.

Build

Now that you know what your API is: Build it. Make it happen. If there's a significant amount of complexity behind a simple call, make a layered API: a porcelain interface that sits on top of a verbose API that sits on top of a low-level integration interface.

The user API is all that matters. Everything else is secondary.

Once your software is released, improve it! Add new features, better security, optimal performance, and rigidity. But never compromise the API.

Manifesto

Build things that you want. Build things that you need. Build things for you.

The Golden Rule:

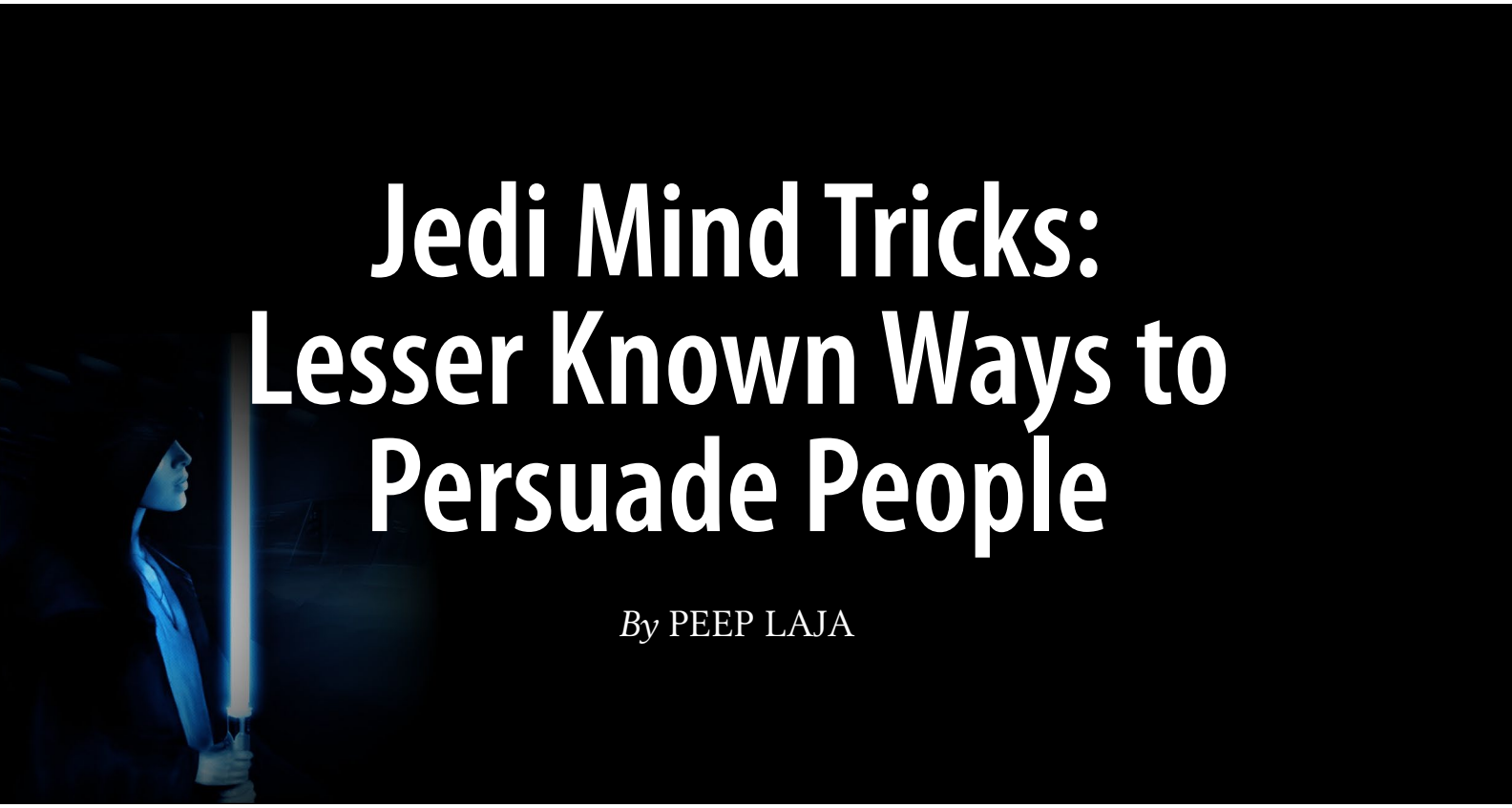
> *Do unto others as you would have them do to you.*

Adapted to:

> *Build tools for others that you want to be built for you.* ■

Kenneth Reitz is a software architect and minimalist, consumed with elegant tools and interfaces. He works at Heroku, designing the Python Stack. Kenneth also writes The GitHub Reflog and loads of open source projects, available at *github.com/kennethreitz*

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



Jedi Mind Tricks: Lesser Known Ways to Persuade People

By PEEP LAJA

Image credit: Lu Lebel (darkfitoplancta.deviantart.com)

You want to be persuasive. The power to influence people to get what you want is sometimes all it takes to be successful. These are some tactics, discovered through psychological research, that you have probably not yet heard about, but which have the potential to increase your persuasive abilities.

I'm not going to cover reciprocity, scarcity, or social proof and all those widely known persuasion principles. You already know all about those (in case you don't, stop everything and read Influence: The Psychology of Persuasion by Cialdini).

Be confident, talk fast

The best way to persuade audiences that are not inclined to agree with you, is to talk fast. Fast pace is distracting and people find it difficult to pick out the argument's flaws. When talking to an audience who is likely to agree (preaching to the choir), slow down and give the audience time to agree some more.

Want to boost persuasive power? Talk with confidence.

Don Moore from Carnegie Mellon's Center for Behavioral Decision Research has published research [hn.my/cocky] showing that confidence even trumps past accuracy in earning the trust of others.

We prefer advice from a confident source, even to the point that we are willing to forgive a poor track record. Moore argues that in competitive situations, this can drive those offering advice to increasingly exaggerate how sure they are.

People naturally associate confidence with expertise. Know your product, know the facts about its benefits and believe in what it does; true confidence comes from knowing and believing what you're saying. It's essential that we communicate our confidence to others in order to persuade them.

Swearing can help influence an audience

Light swearing, that is. (Go overboard and lose all credibility)

Researchers divided 88 participants into three groups to watch one of three slightly different speeches. The only difference between the speeches was that one contained a mild curse word at the start:

> *"…lowering of tuition is not only a great idea, but damn it, also the most reasonable one for all parties involved."*

The second speech contained the "damn it" at the end and the third had neither. When participants' attitudes were measured, they were most influenced by the speeches with the mild obscenity included, either at the beginning or the end.

The word "damn" increased the audience's perception of the speaker's intensity, which increased persuasion. The audience's perceived credibility of the speaker did not change.

So that's the secret of Gary Vaynerchuk and Dave McClure. I thought they're just cool guys, but turns out it's the swearing that got me.

Get people to agree with you first

If you want people to buy into your message, start with something they can agree with.

In a research study by Jing Xu and Robert Wyerestablished, there were lingering effects of messages people agree with. In one of the tests, participants listen to a speech by John McCain or one by Barack Obama and then watch a TV ad for Toyota.

Republicans tended to be more swayed by the ad after watching the speech by John McCain, while Democrats showed the opposite effect, finding the ad more persuasive after the Obama speech.

So when you try to sell something, make statements or represent a world view your customers can agree with first — even if they have nothing to do with what you're selling.

Balanced arguments are more persuasive

If what you are doing inspires (or can inspire) criticism, resist the instinct to paper over weaknesses. We fear undermining our point of view by talking about weaknesses, but actually it would help our case.

Psyblog writes:

> *Over the years psychologists have compared one-sided and two-sided arguments to see which are the most persuasive in different contexts. Daniel O'Keefe at the University of Illinois collected together the results of 107 different studies on sidedness and persuasion conducted over 50 years which, between them, recruited 20,111 participants (O'Keefe, 1999, Communication Yearbook, 22, pp. 209-249).*
>
> *The results of this meta-analysis provide persuasive reading. What he found across different types of persuasive messages and with varied audiences, was that two-sided arguments are more persuasive than their one-sided equivalents.*

People are not idiots — they can think. If you don't mention the other side of the coin in your arguments, people are less likely to believe you.

Perhaps it might be a good idea to mention the shortcomings of your product or service on your website.

People believe you more if they sit in the evidence

A research study by Ye Li, Eric Johnson, and Lisa Zaval looked into global warming and its relation to the current local weather.

Participants in the US and Australia rated the strength of their belief in global warming. They also rated whether they thought the temperature that day was warmer, colder, or about normal for that time of year. When people felt the day was warmer than usual, they also expressed a higher belief in global warming than when they felt the day was cooler than usual.

In the related study they asked the same stuff, but also asked for a donation to a non-profit combating climate change. The participants in this study donated over four times as much money when the day was much warmer than usual than when the day was much cooler than usual.

If you want people to buy your message, ask for the sale in the situation that supports your claims. Online, use imagery or other visual material to build the stage for your story.

Upsell a product that cost 60% less

Once somebody gets to a point that they'll buy something from you, they have given you their trust and have convinced themselves it's okay to give you money. In that moment you are able to sell them more.

When somebody buys a shirt, your upsell should be a tie and not the whole suit.

> **“Invest in building better filters that help people make the choice.”**

The time-tested 60×60 rule says that your customers will buy an upsell 60% of the time for up to 60% of the original purchase price. Any upsell you offer must be congruent with the original purchase.

If you don’t use up-selling in your business yet, it’s a quick way to boost profits (“would you like fries with that?”).

Frame it in the positive

Emphasizing the positive can be more persuasive than pointing out the negative.

An analysis [hn.my/frame] added up the results of 29 different studies, which had been carried out on 6,378 people in total. The finding was that there was a slight persuasive advantage for messages that were framed positively.

This study had to do with the way people relate to disease prevention, such as encouraging people to use sunscreen, and promoting healthy eating habits, but it might have a wider appeal. The researchers hypothesized the reason to be that we don’t like to be bullied into changing our behavior.

Try framing your marketing message in the positive (“Gain an additional hour every day” vs. “Stop wasting time”) and see if it makes a difference.

The paradox of choice

The more choices you offer, the less likely people will take you up on it, says this study [hn.my/paradox].

Researchers set up a jam-tasting stall in a posh supermarket in California. Sometimes they offered 6 varieties of jam, at other times 24. Jam tasters were then offered a voucher to buy jam at a discount.

While more choices attracted more customers to look, very few of them actually bought jam. The display that offered fewer choices made many more sales. In fact, only 3% of jam tasters at the 24-flavor stand used their discount voucher, versus 30% at the 6-flavor stand.

If you have a ton of products, invest in building better filters that help people make the choice.

If something happens often enough, you will eventually be persuaded

Repetition has a distinct effect on us. Advertisements replay themselves when we see the product. The songs that radios play over and over again eventually grow on us.

Repetition of a word or visual pattern not only causes it to be remembered (which is persuasive in itself), it also leads people to accept what is being repeated as being true.

ChangingMinds writes this about Hugh Rank’s persuasion research (Teaching about public persuasion, 1976):

> *Our brains are excellent pattern-matchers and reward us for using this very helpful skill. Repetition creates a pattern, which consequently and naturally grabs our attention.*
>
> *Repetition creates familiarity, but does familiarity breed contempt? Although it can happen, the reality is that familiarity leads to liking in far more case than it does to contempt. When we are in a supermarket, we are far more likely to buy familiar brands, even if we have never tried the product before.*
>
> *Think about the last time you bought a pair of shoes. Did you pick them up then put them down several times before trying them on? Did you come back to try them again? If so, you are in good company. Many people have to repeat things several times before they get convinced. Three times is a common number.*

> **“Use friendly repetition to create familiarity and hence liking.”**

Use repetition of key benefits or value propositions in your sales copy and ad campaigns. Effective advertising and political campaigns do that (“Geico can save you 15% or more…”). Use friendly repetition to create familiarity and hence liking.

Another research study [hn.my/loudest] reveals that even if only one member of a group repeats their opinion, it is more likely to be seen by others as representative of the whole group.

Men are more responsive to email than face-to-face talk

Guadagno & Cialdini research (2002) [hn.my/doi] showed that men seem more responsive to email because it bypasses their competitive tendencies. Women, however, may respond better in face-to-face encounters because they are more “relationship-minded.”

This research is suggesting that email could provide a way of side-stepping men’s competitive tendencies. But, this only applies to distant relationships. The closer the relationship between men, the better face-to-face works.

When you want to persuade a man you don’t know too well, start with an email.

Limiting the quantity you can buy makes you buy more

From Brian Wansink’s excellent book Mindless Eating: Why We Eat More Than We Think:

> *A while back, I teamed up with two professor friends of mine — Steve Hoch and Bob Kent — to see if anchoring influences how much food we buy in grocery stores. We believed that grocery shoppers who saw numerical signs such as “Limit 12 Per Person” would buy much more than those who saw signs such as “No Limit Per Person.”*
>
> *To nail down the psychology behind this, we repeated this study in different forms, using different numbers, different promotions (like “2 for $2” versus “1 for $1”), and in different supermarkets and convenience stores. By the time we finished, we knew that almost any sign with a number promotion leads us to buy 30 to 100 percent more than we normally would.*

So put numbered limitations or anchors on the quantity your customer can buy from you.

Story beats data

A Carnegie Mellon University study in 2007 by Deborah Small, George Lowenstein, and Paul Slovic compared the effects of story vs. data.

Test subjects where asked to collect donations for a dire situation in Africa. The data pitch contained statistics about food shortages in Malawi, lack of rain in Zambia, and the dislocation of millions in Angola.

The second version talked about a particular girl in Zambia, Rokia, who was starving. People were shown her photo and asked to donate to help her directly.

On average, students who received the fact-based appeal from Save the Children donated $1.14. Students who read the story about Rokia donated an average of $2.38, more than twice as much.

In a third experiment, students were told Rokia’s story but also included statistics about persistent drought, shortfalls in crop production, and millions of Africans who were going hungry. While students who had read Rokia’s story alone donated an average of $2.38, those who read the story plus the data donated an average of $1.43.

The plight of Africa, the fight with poverty is too overwhelming and people feel their contribution is just a drop in a bucket, hence feel less inclined to help.

Marketing to men? Use photos of women

A field experiment in the consumer credit market found pictures of women as effective as low interest rates.

A South African lender sent letters offering incumbent clients large, short-term loans at randomly chosen interest rates. The letters also contained independently randomized psychological "features." As expected, the interest rate significantly affected loan take-up. Inconsistent with standard economics, some of the psychological features also significantly affected take-up.

> *For the male customers, replacing the photo of a male with a photo of female on the offer letter statistically significantly increases takeup; the effect is about as much as dropping the interest rate 4.5 percentage points… For female customers, we find no statistically significant patterns.*

> *Overall, these results suggest a very powerful effect on male customers of seeing a female photo on the offer letter. Standard errors however do not allow us to isolate one specific mechanism for this effect. The effect on male customers may be due to either the positive impact of a female photo or the negative impact of a male photo.*

The experiment featured a rather dramatic range in interest rates — 3.25% to 11.75%. The effect of a photo of a woman on a loan offer was equivalent 4.5% difference in the loan interest rate.

Next time add a photo of a woman to your offer and see your conversions go up.

The above study did not feature sexy women. But would a sexy women wearing bikinis help?

Research shows that arousal makes men stupid [hn.my/arouse], as they become bad at making decisions. It gives them tunnel vision. The effect seems to be a short-term one that would be most effective at the point of purchase, for impulse purchases.

The ideal selling situation would be to have the bikini-clad babe selling to the men in person. I guess you could do that also online for products meant only for men.

Studies have shown that sexy ads don't really make men remember the product [hn.my/men]. We're so lasered in on the sexy stuff, we don't care what brand of product it is.

Want to convince leaders? Make them feel less powerful

Don't bother trying to persuade your boss of a new idea while he's feeling the power of his position, research suggests [hn.my/connect] he's not listening to you.

"Powerful people have confidence in what they are thinking. Whether their thoughts are positive or negative toward an idea, that position is going to be hard to change," said Richard Petty, co-author of the study and professor of psychology at Ohio State University.

The best way to get leaders to consider new ideas is to put them in a situation where they don't feel as powerful, the research suggests.

"Our research shows that power makes people more confident in their beliefs, but power is only one thing that affects confidence," Petty said. "Try to bring up something that the boss doesn't know, something that makes him less certain and that tempers his confidence."

"You want to sow all your arguments when the boss is not thinking of his power, and after you make a good case, then remind your boss of his power. Then he will be more confident in his own evaluation of what you say. As long as you make good arguments, he will be more likely to be persuaded," Petty said.

So in a nutshell:

- Make the leaders feel less powerful and confident by talking about stuff they don't know and if possible, talk outside of his office (neutral territory).
- After the pitch, remind them who's the boss, so they could take action on your request.

The Sullivan Nod

Invented by restaurant consultant, Jim Sullivan, the Sullivan nod involves reciting a list of options but just inclining your head slightly when you reach the choice you'd like the buyer to make. The nod has to be subtle, but perceptible and works best in lists of no more than five items. According to Jim Sullivan, it's successful up to 60% of the time.

> *Whenever servers suggest a beverage, have them smile and slowly nod their heads up and own as they make the suggestion. Body language is powerful, and research shows that over 60% of the time, the guest will nod right back and take your suggestion!*

I bet you could use that online in sales videos. When talking about plans or packages, do the nod on the one you want them to buy.

Clarity trumps persuasion

Dr. Flint McGlaughlin of Marketing Experiments likes to say this: "Clarity trumps persuasion." Remember this.

Persuasion tricks work when done subtly and skillfully. Overdo it and you lose the sale. When you're writing sales copy or doing presentations, the best way to persuade people is to use clarity. Give people enough information to make up their mind without being cheesy or using hype. ■

Peep Laja is an entrepreneur and conversion optimization expert. He's been doing internet marketing for 10+ years in Europe, Middle East, Central America and the US. Today he runs a digital marketing agency called Markitekt. Peep blogs at ConversionXL [conversionxl.com/blog].

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

"I've Got an Idea For an App"

By CHRIS EIDHOF

I've been an iPhone developer for over three years now. The first app I built was CookieCombo. Although we only sold enough copies to go bowling twice, it was completely worth it. We tweeted about it and got some awesome gigs. Everybody was in need of iPhone developers, and there was a huge shortage. Good times.

As more and more people know that I'm an iPhone developer, I hear the following phrase quite often: "Hey, you know, I got a great idea for an app." It started with tech-savvy people saying this, but now it seems like everybody and their mother has an idea.

I'm a nice guy and always try to listen to people. I subscribe to the belief that ideas aren't worth anything unless there's good execution. The ideas I hear invariably end with: "I only need someone to build this." If it's a bad idea, I try to explain why. If it's a good idea, I try to explain the amount of work they have to do to make it successful.

I once jokingly said that I should print some small cards with "No, I won't listen to your app idea" and give it out at parties whenever people approached me. But because I want people to like me, I didn't do it.

However, a month ago I got an email from a friend of my brother's about a secret app idea. Those are often the worst. He wanted to have a Skype meeting, and I said: "Sure, let's do that. I probably won't have time to build it, but at least I can help you and point out the technical difficulties."

Well, his idea is just awesome. That's when I decided: always listen to ideas. Most of them are probably another fart app or a social network for sharing pictures of coffee, but even if the odds are very small that it's a good idea, the potential payoff for executing a great idea could be huge. ■

Chris Eidhof is an independent software developer from The Netherlands, living in Berlin. He used to do high-level functional programming in Haskell but converted to Objective-C: he now builds iPhone and iPad apps, and dabbles in big data.

Reprinted with permission of the original author.
First appeared in *hn.my/idea* (eidhof.nl)

HARVEST

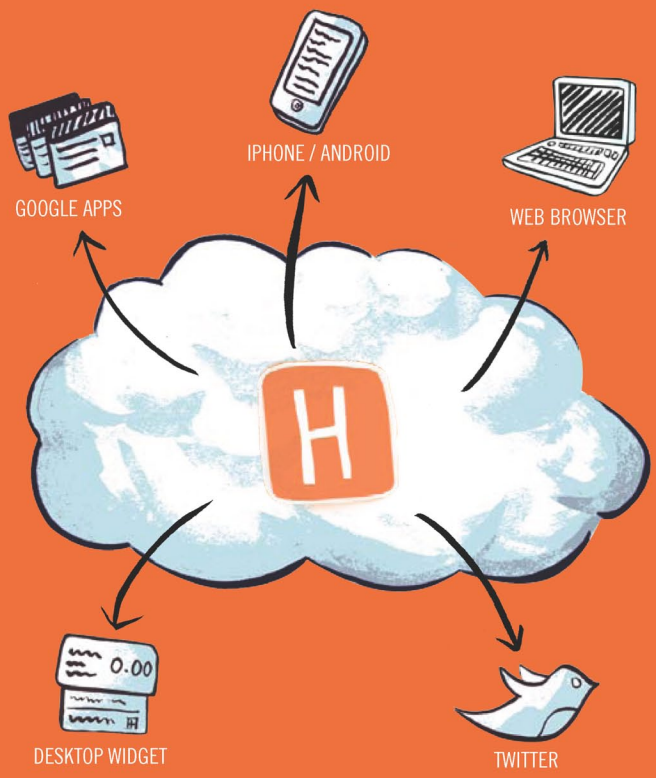
TIMESHEETS

INVOICES

REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.



Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.

Learn more at **www.getHarvest.com/hackers**

Our users are saying...

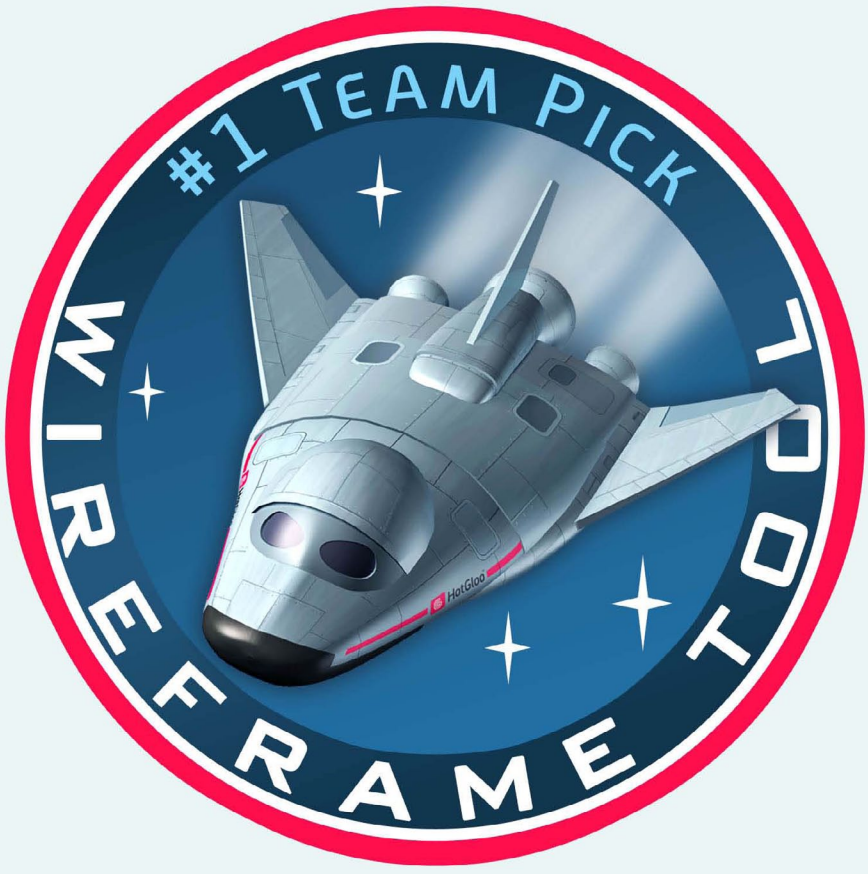
"I enjoy the feeling of being recognized but mostly I enjoy knowing my friends are happy."

"It's cool realising that people thought enough about your contribution to give props."

When your people feel appreciation every day they are **happier**, more **engaged** and **more productive**.

Get started today at dueprops.com

Due Props
Better Recognize



From low- to high-fidelity, from wireframes to interactive prototypes. With HotGloo you can finally achieve great concepts together with your team in a fast, simple and beautiful way.

HotGloo RIA
The Future of Wireframing

Get 50% off first 3 months* with the code hghackers at HotGloo.com

*Offer good for new accounts if used before 05/31/2012