HACKERMONTHLY

Issue 40 September 2013

```
{
    join: 'Intensive Online Bootcamp',
    learn: 'Web Development',
    goto: 'http://www.gotealeaf.com'
}
```

Tealeaf Academy
an online school for developers

Learn Ruby on Rails | Level up Skills | Launch Products | Get a Job

MEET MANDRILL

By MailChimp

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

MANDRILL.COM

Curator
Lim Cheng Soon

Contributors
Damian Sowers
Peter Cooper
Vibhu Norby
Zach Hoeken Smith
Stuart Sierra
Ben Howdle
Josh Wills
John Graham-Cumming
Nick Knowlson

Illustrator
Matthew Billington

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

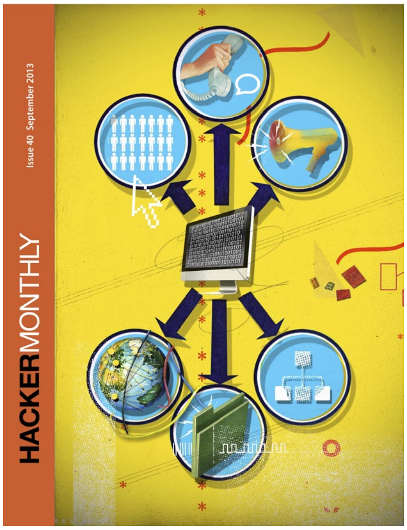
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Cover Illustration: Matthew Billington

Contents

Deimos (Damian's dog)

Mr. Chen

For links to Hacker News dicussions, visit *hackermonthly.com/issue-40*

Sacrificing Everything For My Dog

How I Became A Programmer

By DAMIAN SOWERS

NOBODY WILL BAT an eye if you're making big sacrifices to give your children a better life. This sacrificial behavior is hard wired into our DNA and it's expected of every parent. In fact, evolution depends on this behavior.

However, if you tell somebody you're restructuring your life to make your dog happy, there's a good chance they will laugh and think your future involves a strait-jacket and a big nurse by the name of Ratched.

Until now I never told anybody about my reasons for my big life changes and I feel quite embarrassed as I'm typing this. The truth is, if I didn't love my dog so much, my life would be radically different and I'm sure I would be miserable.

I don't believe in God, but I do believe in karma. Maybe karma is similar to luck, as in "the harder I work, the luckier I get" type of thing. Being good to someone or something seems to create a bunch of collateral happiness.

Here is my story.

It was 2008 and I was headed in a very respectable direction, currently enrolled in a Ph.D. program for chemistry at the University of Colorado at Boulder. I worked incredibly hard to get to this point and had earned myself a number of gold stars along the way, all of which looked really good on paper. For example, I spent some time after my undergraduate studies working on the NASA Genesis mission at the Berkeley Space Sciences Laboratory.

I liked my advisor in Boulder and my situation was ideal. My advisor gave me a two-month paid trip to Prague the summer before graduate school. This trip was arranged so I could do theoretical chemistry work for the Czech Academy of Sciences, but I also used it as an opportunity to drink a lot of good Czech beer.

On the surface I had a great life, but deep down something about my situation was wrong. I was tired of chemistry yet I didn't want to admit this to myself. It's easy to delude yourself when you've invested so heavily into something. After all, everything I had done in the past 10 years was dedicated to this career path.

Enter Deimos, my Golden Retriever/Shar Pei hybrid. A master of Tug o' War and decimator of sticks, Deimos also has the ability to lay on a massive guilt trip whenever I leave him. He puts his head down in that pleading fashion and gives me those eyes that seem to say, "Please dad, take me with you. I just want to spend time with you." Every time I left for graduate school duties I had to suffer through this departure. I died a little on the inside each time.

It's unfair to a dog to make them wait 8 hours each day while we are at work. Dogs only have about a decade of time on this Earth. In dog years, this amounts to waiting 56 hours for us while we are off working every day. That is a waste of a very short, bright life. If you've ever owned a dog, you know just how intelligent they actually are (especially larger breeds). Such intelligence deserves great experiences and grand adventure.

So, while I was unable to admit to myself that I was headed down the wrong career path, I was able to recognize my intense desire to give Deimos a situation where I could spend more time with him. This invisible guiding paw would press on me daily.

Enter Typhoid. Yes, Typhoid. As in the main way your character died when playing The Oregon Trail computer game in the 5th grade. Around month four of graduate school I started getting really high fevers on a regular basis and pain in my lower abdomen. I didn't know what was going on and the doctors didn't either. A couple of doctors tried to tell me I had Crohn's disease. I didn't believe this diagnosis, though.

After an abscess and a lot more pain, I mentioned to my doctor that this could be related to a case of Typhoid I contracted while mountaineering down in Bolivia. I had all sorts of stomach problems after that trip and I theorized that the Typhoid could have caused some scarring in my intestines, which was then acting as nucleation points for infection (maybe also helped along by extreme stress).

The doctor said it was worth a shot to try a prolonged course of strong antibiotics to see if this theory was correct. So I went home for Christmas break with a truckload of drugs. The guiding paw of Deimos would make it so I would never go back.

The antibiotic treatment worked and it wasn't too long before I was completely healthy again. Yet, seeing how happy Deimos was in Tahoe made me think deeply about different career paths which would allow me to work from home so I could spend more time with him.

Enter programming. I decided I wasn't going back to graduate school. I would disappoint a lot of people who had invested in me (especially my advisor), but that bad feeling is so small in comparison to living the wrong life. Some people stay on the wrong path their entire lives just to avoid this disappointment and/or out of a sense of obligation. I didn't want this to be me. Fuck obligation.

I had a plan. I was going to teach myself programming and change my life. I had a very superficial introduction to Fortran in the past with computational chemistry, but that was the extent of my programming knowledge. So I decided I would need to hunker down for many months and learn how to make a living with code.

Fortunately I have amazing parents and they let me live in their house in Tahoe (they retired and moved away to Navarro, near Mendocino). As for my other living expenses, I still had the remainder of my student loan and I decided I was going to gamble and put the rest of my life on a credit card while I went through this self-taught re-education.

Many people believe that credit cards and debt are pure evil and stupid. I see credit cards as being the saving grace which prevented another great depression during the recent economic collapse. Having access to rainy day/investment money is the single greatest invention of our economic system.

As long as you use the money properly, and investing in yourself is the best possible way to use money, credit cards can give you a new life. While I was living on the card I didn't spend my time playing World of Warcraft or watching TV. I made sure I dedicated 8-10 hours a day to learning the wonderful craft of programming.

A little while later I started earning money with my first business, Tallac Interactive. This was just a front-end local web design business and I didn't spend much time interacting with server side languages, other than the occasional dive into the Wordpress back-end. But it was enough to get noticed by the CEO and President of Fretlight Guitar, which led to my first high-paying job as a programmer.

Fast forward to now and I've founded Mycelial, Briarpatch, and AppRaptor. I've taught myself PHP, MySQL, Ruby, Ruby on Rails, Javascript and many more amazing languages and technologies. I can't get enough of programming. I am paid well to do it from home and I even spend all of my free time on programming side projects. Building stuff with code is the most satisfying thing in the world for me.

And yet, I wouldn't be a programmer today if it weren't for my intense desire to find a lifestyle which would make Deimos happy. In other words, I wouldn't be happy today if I didn't sacrifice my previous life for my dog. I'm sure I would still be in graduate school, poor and miserable and covered in organic toxins from the lab.

I'm not trying to knock chemistry. I have tremendous respect for anyone who dedicates their life to chemistry. Chemists make our lives better and often shorten their own in the process. It's a harsh bargain and an extremely noble one. I still have a massive scar from a Nitric acid spill on my wrist which reminds me every day of the sacrifices chemists make for the rest of us.

The guiding paw of Deimos also had another great effect on my life. As a result of not going back to graduate school and staying in Tahoe, I reconnected with Lisa, who is absolutely amazing, and we've been together for four years now.

I don't think the karma of this situation is mystical or spiritual or anything of that nature. Providing my furry little dependent with a life of pure happiness and grand adventure merely showed me the way to a lifestyle with the perfect balance of nature and intellectual stimulation, and I encourage others to find a similar balance.

There are probably a lot of people out there who balk at the idea of personal sacrifice for a dog. It's easy to spot these people, however. They are cat people. ■

Damian Sowers is a Rails and JavaScript programmer living in South Lake Tahoe, California. His work can be found at *appraptor.com*

MAGERIE D
PELLERIN & C

How to Spread The Word About Your Code

By PETER COOPER

YOU SPENT AN entire weekend building a library, jQuery plugin, build tool, or other great piece of code you wanted to share far and wide, but after some tweets and a failed attempt to make the front page of Hacker News, your creation languished, unloved, in a GitHub repo. A common situation for many developers nowadays, but one you can avoid.

As the editor of several programming newsletters, I frequently get two types of e-mails from developers. Those reaching out to ask if I can mention their projects, and those expressing surprise and excitement that their work has been featured. If you're a developer doing good work but feel more like you'd be in that second group, the three steps in this article are for you.

Before we get started, there's a stumbling block we need to kick away. Terms like "marketing" and "advertising" are dirty words for many developers and it's not uncommon for developers to be reluctant to do much promotion. "Build it and they will come" used to work when exciting open source projects were few and far between, but now everyone seems to be working on something and making noise about it. Few successes come through pure luck, but rather because developers are actively promoting their work or, at least, making it discoverable. It's time to join them!

1 Get your project ready

Before you can promote your project, you need to make it attractive to potential users and evangelists (including general well-wishers, the media, and other developers).

A good name

Ensure your project has a palatable name. It doesn't need to be clever or even descriptive, but it's worth avoiding innuendos that may present a problem down the line. For example, the popular Testacular and Authgasm projects, are now named Karma and Authlogic respectively after users raised a fuss.

You should perform a search for the name you choose to be sure you're not clashing with anything else that's popular or trademarked (did you know Firefox was called Phoenix and Firebird prior to Firefox?). The US Patent and Trademark Office has an online trademark search facility. [tess2.uspto.gov]

A benefit of having a relatively unique or uncommon name is so you can search for it over time (or even set up a Google Alerts notification for the name) and find mentions of your project without many irrelevant results popping up. If you want to have something descriptive but unique, consider joining two words together. For example, when I created a Ruby library to do natural language detection, I called it WhatLanguage and it's easy to search for.

An official homepage/project URL

The term "homepage" is a bit outdated but you ideally need a single "home" URL that you can promote and point people to in relation to your project. You don't need to splash out on a fancy template or even a domain name, but your project needs a focal point. That could be an entire site with its own domain, such as those for Yeoman [yeoman.io] or HTML5 Boilerplate [html5boilerplate.com], a simple single page on an existing domain, such as that for RoughDraft.js, [ndreckshage.github.io/roughdraft.js] or even a regular GitHub repo, such as for vague.js. [github.com/GianlucaGuarini/vague.js]

If you have the freedom to do so, make sure your site looks good on the major browsers, hook up some analytics to your page and ensure the <title> tag is well written. Use a title like "MyProject — A JavaScript Library to X, Y and Z" instead of just "MyProject — About" or a blank title. With social bookmarking, this matters as you can't guarantee your evangelists will write a good title of their own.

If you're not a Web designer, don't have the time to spend making a complete design, but still want a complete site rather than just a GitHub repo and README, consider using a framework like Bootstrap as it'll provide a clean layout out of the box and you can forget about many cross browser and device issues.

Documentation and copywriting

It's only just a cliché that developers don't like to write documentation, but you need something for potential users to fall back on, and time invested in producing useful documentation up front will pay dividends later.

At a cynically bare minimum, you need to write enough documentation that someone will be confident about sharing your link or promoting your project and not feel like they're sending their own followers into a black hole of misunderstanding. This means your homepage or README needs to cover a few angles. You'll need to:

- **Prominently feature a "[noun] is" paragraph.** An alarming number of project homepages don't explain, in simple terms, what the project is actually for or does. If you've built a JavaScript library that does language detection, say, you have to say so. For example: "LanguageDetect is a JavaScript library for detecting the natural language of text."

An excellent example of this in action is on *libcinder.org* where it states right up front: "Cinder is a community-developed, free and open source library for professional-quality creative coding in C++." Perfect!

- **Write clear titles, subheadings, and support copy.** At a bare minimum, ensure titles, subtitles, and any sort of writing on your homepage are straightforward and clear. Write for the lowest common denominator on your homepage. You can get more advanced elsewhere.
- **Write a beginner's tutorial and link to it from your home page.** Unless everything's simple enough to explain on a single page, quickly write a tutorial that covers basic installation and usage and either include it in your README file or put it on the Web and link to it from your README and/or homepage.
- **State dependencies and requirements clearly.** Does your library only work on a specific version of Node? Is it a browser extension for Firefox? Does your code require PostgreSQL, Redis, or another specific database? Be sure to include a bullet point list of dependencies and requirements for your project to be usable so as not to disappoint potential users.
- **Specify the license for your code.** While you could get away with keeping your licensing information tucked away in a LICENSE file in your GitHub repo, specifying what license your code is released under up front and center will help put many developers at ease. Likewise, if your project is commercial in nature and costs money, don't hide that detail and mislead visitors.

- **If your project is a library or API, feature some example code on the homepage.** Unless your library is particularly complex, let visitors see an example of its usage on the project homepage. If your API is good, this could be a great way to get an "easy sale." I'm not a huge fan of the code example chosen, but the homepage for Ruby [ruby-lang.org] shows off this technique.

Extra materials

A blog post is a great way to introduce a project that might need more background or have more of a story than it's practical to tell on a homepage or within documentation. If there's any sort of story behind your project, a blog post is a great way to tell it. Be sure to link to the post from your project's homepage and consider promoting the blog post separately to relevant sites within your niche.

If you have the ability, recording a screencast or other sort of video can help. Could you put together a simple 5 minute screencast of how to install and use your library? Or have you built a game that could be demonstrated in a few minutes of gameplay? Record a simple video, put it on YouTube, and embed it on your homepage. Your accent doesn't have to be as crisp as a newsreader's, and you don't even have to appear within the video. All that matters is you get to the point quickly and your audio is tolerable (not muffled, clipping, or drowned in background music).

As the editor of several programming newsletters, I look at thousands of projects each year, and it's still uncommon to see simple screencasts, yet they certainly help a project stand out and, as a consequence, make it more likely for me to talk about it. You can see a perfect example on Punch's homepage. The early popularity of Ruby on Rails also depended upon a popular "build a blog engine in 15 minutes" video, back when the concept of using video to promote an open source project was very novel.

If you're sticking to the straight up, GitHub README approach (and it's certainly not a bad idea for a simple library), a bonus tip is to create a tiny screencast of your code in action and convert it to an animated GIF for inclusion in your README. Richard Schneeman outlines this technique in "Use GIFs in your Pull Request for Good, not Evil." [hn.my/gifs] The result is striking and could help your README stand out.

For further ideas on how to make your project stand out before you begin promoting it, check out the great "How to Make Your Open Source Project Really Awesome" by Michael Klishin. [hn.my/osawesome] It digs into more detail about versioning, announcements, having a changelog and writing good documentation.

2 Get the word out

You've polished your project, got a URL to promote, and you're ready to get the news out.

A word of caution, however. Don't use every technique on day one. You could overload your site with traffic or, worse, be subjected to a barrage of online criticism if your work or site is broken. With something like a library or tool, a gentler approach will work well and building up small streams of visitors and users over time will give you a much better time.

Social networking

Your own social networking profiles are always a good place to start if you have them. You'll get more immediate feedback from people who actually know you and if your project is particularly interesting, it could go viral even from a single mention.

A great example of a simple project going viral was YouTube Instant by Feross Aboukhadijeh. Feross built YouTube Instant quickly, mentioned it on Facebook before going to bed, and woke up to a flood of traffic and press mentions.

If you like to experiment and have several bucks going spare, you could also consider paying for a promoted post on Facebook. This will give your post more visibility in your news feed, but is best reserved for if your Facebook friends are mostly developers or people likely to be interested in your project. If not, and you'd still like to spend some money, consider an ad on Reddit or a relevant programming blog instead.

Influencers, bloggers, and niche media

Whether you're working on a JavaScript library, Firefox extension, backend app in Rails, or a theme for Bootstrap, your code will fit into one or more niches, and every technical niche has a variety of "influencers," people and publications who are popular and well known for the topic at hand.

Getting a tweet, retweet, or even an entire blog post from an influencer could have a significant impact on your project, as could being invited to blog elsewhere (Mozilla Hacks, for example!). If Brendan Eich tweeted about your JavaScript library, Lea Verou wrote a blog post about a CSS trick you discovered, or Paul Irish mentioned a Web development tool you built in a talk, you would attract a lot of interest quickly. It is key, however, to realize there are many great influencers in every space, and you'll achieve nothing by hounding any one person, so be prepared to move on.

Spend some time working out who the influencers and key publications are in your niche. For Twitter, Followerwonk [followerwonk.com] is a handy tool that searches Twitter biographies for certain words. If you search for "JavaScript" the first page includes several users who would be useful to reach out to if you had a particularly interesting JavaScript-related release to promote. Reaching out on Twitter can be as simple as a single tweet and many busy folks prefer Twitter as it takes less time to reply than an e-mail. A single tweet from *@smashingmag* could drive thousands of visitors your way, so consider tweeting them, and other similar accounts, when you have something relevant.

I'd also advise looking for blogs and e-mail newsletters in your niche. Start with something as simple as Googling for "JavaScript blog", "JavaScript newsletter","css blog" or whatever's relevant to your project. Most bloggers or e-mail newsletter publishers will not be offended by you sending them a quick note (emphasis on quick) letting them know about your work. Indeed, some weeks there can be a shortage of interesting things to write about, and you might be doing them a huge favor.

If you choose to e-mail people (and your project will probably be more substantial than a few hours' work to justify this), take care not to make demands or to even expect a reply. Many bloggers and influential people have overflowing inboxes and struggle to reply to everything they receive. Make your e-mail as easy to process as possible by including a single URL (to your now superb homepage or README) and include your "[noun] is" paragraph. Don't take a non-response as an insult but keep moving on to the next most relevant person. You might even consider taking a "Here's my project that does X, Y and Z. No reply needed, I just thought you might like it" approach. Softly, softly works here, as long as you get to the point quickly.

"How To Get Attention From Internet Celebrities" by Jason Cohen [hn.my/emailbrain] and "How to Write the Perfect Outreach Email" by Gregory Ciotti [hn.my/perfectemail] go into more detail about e-mail etiquette when promoting your work to influencers. While you might not need to contact any "celebrities" in your niche, the principles of keeping it short, including a call to action, and ensuring your work is appropriate to the person are really true for anyone you're sending unsolicited messages to.

Podcasters are an often forgotten source of promotion opportunities, too. While some podcasts don't cover news or new releases at all, many do, and being on the radar of their hosts could help you get a mention on a show. Smashing Magazine has put together a list of tech podcasts [hn.my/podcasts] covering the areas of design, user experience, and Web development in general. Again, keep your e-mails short and sweet with no sense of expectation to get the best results.

User-curated social news sites

As well as reaching influencers and niche media, sometimes reaching the public "firehose" of news can work, too. There are few better examples of these in the modern world of development than Hacker News or Reddit.

Hacker News in particular is notoriously hard to reach the front page on and "gaming" it by getting other people to vote up your post can backfire. (Indeed, it will backfire if you link people to your post on Hacker News and encourage them to upvote. They have ways of detecting this behavior. Get people to manually find your post instead.) If you do reach the front page of Hacker News, of course, you can certainly expect an audience of many thousands of developers to be exposed to your work, so be sure to try.

With Reddit, the key isn't to dive straight into a huge sub-Reddit like /r/programming but to look for sub-Reddits more directly related to your project. For a JavaScript library, I'd post to /r/JavaScript or possibly /r/webdev. Reddit ads can also perform well if you're OK with spending some money, and these can be targeted to specific sub-Reddits, too.

There are many similar sites that are less well-known but which are respected in their niches and can drive a lot of interested visitors, including Designer News (mobile and Web design) [news.layervault.com], DZone (general developer stuff) [dzone.com], EchoJS (JavaScript) [echojs.com], RubyFlow (Ruby and Rails) [rubyflow.com], and *Lobste.rs* (general hacker and developer stuff). Finding the right site like this and taking time to make an on-topic, well-written post will help a lot.

3 Maintain momentum

You've built up some interest, your GitHub stars, Reddit votes, and pageviews are all rocketing up, but now you want to capitalize on the attention and maintain some momentum.

User support

Whether you've built an open source project or a cool tool, you're going to end up with users or fellow developers who want to provide feedback, get help, or point out issues with your work. On GitHub, the common way to do this is through the built-in "issues" tracker, but you might also find people start to e-mail you, too.

Be sure to define a policy, whatever it is. Users won't feel good about opening issues on your GitHub repo if there are already many unresolved issues there, and your project could stagnate. Ensure you respond to your audience or at least make your policy clear within your README or on your site. If you don't want issues raised or code contributions, make this clear up front.

Extending your reach

For many projects, create a dedicated Twitter account, blog, Facebook page, or Google+ page in advance is overkill, but if your project starts to take off, consider these things. They'll provide an extra way not only for users to remain in touch with your project, but also a way for them to help promote it by retweeting things you post or by directing potential new users your way.

You can also extend your reach in person by going to user groups and conferences and, if you're really lucky, you can speak about your work, too. This is a great way to get new users, as people are much more likely to look into your work if they've met you in person.

Avoid being defensive

If your project does well on sites like Hacker News or Reddit, you'll be tempted to read all of the comments your peers leave, but be careful. Comments about your work will, naturally, seem magnified in their intensity and critical comments that might not actually be mean spirited may seem as if they are to you.

It's hard, but the best policy is to not let any overtly mean comments get to you, duly correct any observations that are wrong, and to thank anyone who goes out of their way to compliment your work. Even if you're in the right, with the lack of body language and verbal cues, being too defensive can look bad online and result in the post becoming a lightning rod for drama. Engage as best you can, but if it feels wrong to reply to something, listen to your gut.

Be careful if you go into a new community to promote your work and get negative feedback. Most communities have rules or expectations and merely entering a community to promote your work is frequently considered a faux pas. Be sensitive to people's environments and try to abide by a community's rules at all times.

The long term

If your project does particularly well, you could be presented with the opportunity of turning it into a business in its own right. Many simple open source projects, often started by a single developer, have turned into long term work or even entire companies for their creators.

Back in 2010, Mitchell Hashimoto released Vagrant, a Ruby-based tool for building a deploying VirtualBox-based virtualized development environments. In late 2012, Mitchell launched Hashicorp, a company providing Vagrant consulting services to enterprise customers. An even higher profile example is Puppet Labs, a company built around the Puppet open-source configuration management tool and which has taken total funding of $45.5 million so far.

If your project becomes respected and heavily used within its field, you might also be approached to write a book or article about it or even speak at a conference. This is a good sign that your project has "made it" to some extent as publishers and event organizers are in the business of working out what it makes business sense to present.

Putting it all together: A checklist

This has only been a basic introduction to promoting your work and with practice you'll come up with tons of tips of your own. Based on all of the ideas above, here's a basic checklist to run through next time you release a new project and want to get some added exposure:

- Focus most of your efforts on your project's homepage or README.
- Check your project's name so it doesn't clash with anything else and is unique enough to find references to your work later.
- Promote your work to your closest social group first to unbury any problems with your work.
- Record a screencast or write a blog post about your project if some extra background would be useful for others.
- Work out a perfect *"[project name] is"* sentence to describe what your project is or does.
- Use your *"[project name] is"* sentence to give your page a descriptive title.
- Find influential people, blogs, podcasts, and e-mail newsletters in your niche and send them a short, pleasant note.
- Post to social news and bookmarking sites. Ensure your title is descriptive.
- Use your *"[project name] is"* sentence in e-mails and contacts with influencers.
- Take a positive, "look on the good side" approach to responding to comments about your work.

Good luck! ■

Peter is the chief publisher at Cooper Press, programmer, editor of Ruby & HTML5 Weekly and founding chair of O'Reilly's Fluent conference.

Don’t Launch Your Product

By VIBHU NORBY

Photo: *flickr.com/photos/jurvetson/6252455266*

It was Monday, April 9, 2012 and we were at a team dinner. We were launching Everyme the next day at 10:00am. Launch was a Tuesday, of course. Consumers use their phones and the web more on Tuesdays than any other day. We had everything set: the TechCrunch article, the AllThingsD piece, and a handful of interviews with top tech blogs. We had 25,000+ people that had signed up to be notified about our launch. We designed and shipped a special page with a countdown three weeks earlier on our homepage. It seemed like the perfect time for our iPhone-only social network for groups: Instagram had been purchased the same day as our team dinner for a billion dollars and Everyme was, in our minds, the Next Big Thing™.

Our plan was simple. Launch the app and generate enough buzz for 25-50,000 downloads, or what we guessed was enough to propel us into the top apps in the Social Networking category in the App Store. Once we got there, we would start generating “organic” downloads from people checking out the top free social apps. A month later we’d roll out an Android app and web, and we would be proclaimed king of the messaging space. Mark Zuckerberg would invite us to Fuki Sushi for vegetable tempura rolls, and we would laugh about how we crushed all of our competitors as he handed us a billion-dollar check addressed to Everyme, Inc.

So that Monday night, we were on top of the world and there was no way we could lose. We were probably in the top percentile of all startups already, having checked everything off of our startup bucket list: Y Combinator. CHECK. Raise a big seed round. CHECK. TechCrunch. CHECK. When I went to sleep that night, my body buzzed with excitement.

Tuesday morning rolled around, and everything went live. Mandatory tweets and Facebook posts went out, congratulatory emails from investors filled our inboxes, and my second monitor looked like NASA mission control, full of custom stats dashboards and Twitter searches.

Just hours later, by Tuesday afternoon, we already knew that our plans and the reality were far apart. Sign-ups were coming in, but at a pace that would never reach 25,000 the first day. TechCrunch was sending hundreds of visitors, not thousands. Our Twitter searches were full of users that didn't get it. And there was no Zuckerberg dinner invitation in our inbox. We peaked at rank 35 in the Social Networking category and ended up with 11,000 downloads and 6,000 sign-ups for our first day. Not exactly the day we expected.

It didn't get any better the rest of the week either. We had fewer sign-ups on Wednesday than on Tuesday: 2,000. And fewer sign-ups on Thursday than on Wednesday. And so on. To top it off, all of our team members had access to the stats dashboards. You could see the psychological effects of dropping numbers significantly impacting productivity and morale. It felt like we had bet it all on red and the ball stopped on black.

The fact is that when you create the big launch event, you will always see the subsequent big drop-off. Your market is not TechCrunch readers and Mark Zuckerberg does not want to eat vegetable tempura rolls with you. If you plan for massive scale out of the gate, you will face disappointment and a morale drain that can kill your company. And unlike a lot of other problems that you face in the startup world, learning this lesson the hard way can cost you your startup right at the outset. Here are a couple reasons why focusing on a big launch is the wrong strategy:

- **"Launching" screws with your metrics — and you need clean metrics to evaluate and iterate on your business.** If you see 6,000 signups on day one and 2,000 on day two, you can be misled about the strength of your vision. It clouds your ability to single out the passionate users and understand their usage patterns.
- **You're probably not going to find product/market fit right out of the gate.** So whatever press or marketing you have planned will fall on uninterested eyes. Again, this will mislead you. You'll spend less money and waste less time by locating your

interested market first and then pursuing marketing channels to reach them when ready. It sounds obvious, but it isn't. When you have a consumer app, at first everyone seems like part of your target audience even though they aren't. Likewise with enterprise, not all businesses are candidates for your software.

- **As mentioned earlier, the bigger your launch, the quicker you will enter the famous "trough of sorrow."** No human can easily withstand the emotional rollercoaster of startup metrics. Such baggage can lose you co-founders, employees, and your capital. And you will lose faith in yourself in the process.
- **You'll be penalized when raising your next round.** Neither the bell-curve nor the downward slope is an attractive graph to show investors. You can demonstrate growth by finding one passionate user, and then ten, and then 100 instead of taking in 6,000 sign-ups to find 111 passionate ones. Some savvy investors will ignore your charts and focus on you — fine — but you have to be a champion. You can't afford to think negative thoughts about your business when talking to an investor.

Having been through multiple launches, seen companies launch at big conferences, and talked with many startups that have experienced the same effect, what I recommend — and what we're doing at Origami — is not launching at all. Take the word launch out of your vocabulary — it's a sign that you are gambling on your app and not building a long-term, sustainable company. Instead, put your sign-up page up or your app out because you need more feedback on your idea. Find an audience of passionate users, even if small, and reach out to their community through appropriate means. Try SEM and Facebook ads to find a target market. Experiment with business models and onboarding flows. Let the press come to you because they love what you've made.

You wouldn't know it by its plain homepage, but our new product has a thriving community of families in the hundreds already. We've been "testing it" for months. One of these days we might put out a homepage where families can sign-up — but you won't hear about it from the press. You'll hear about it from a passionate fan. ■

Vibhu the founder of *Origami.com* (YC S11), an online home for families. Previously, he was a lead engineer at Myspace and Threadbox.

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

Visual Website Optimizer

A B Advanced A/B Testing for SaaS apps

server density

See how Server Density **increased Revenue by 114%** with Visual Website Optimizer at http://vwo.me/serverdensity

VisualWebsiteOptimizer.com

Shenzhen Maker: Mr. Chen

By ZACH HOEKEN SMITH

I've been living in Shenzhen for almost 2 years now, and I'm continually amazed by this city. The people here are creative, it has the best resources for building things you can find anywhere in the world, an amazing climate, and friendly people everywhere. This is the story of one particular Maker I've met in Shenzhen, Mr. Chen.

In my ongoing obsession with digital fabrication and small volume manufacturing, I stumbled upon the Chinese SMT Pick and Place scene. It started with the TM-240A that I found on Taobao, and through that I discovered-*diysmt.com* and *oursmt.com*. It turns out there are a bunch of people building and using low-cost pick and place machines for actual production of real products. I had to find out more.

I used my super-crappy Chinese skills and posted in the diysmt forum to see if anyone was local to Shenzhen and could show me their machine. I got a couple responses, and Mr. Chen agreed to meet me and show me his operation. Always down for an adventure, I agreed and got his address. My assistant/translator and I hopped in a taxi and away we went.

We arrived in a neighborhood on the outskirts of Shenzhen — the type with small alleys separating dozens of dusty apartments with stray dogs running around and open-air grocery stores selling meat on hooks. If you've ever been to China and ventured off the beaten track, you'll know exactly what it's like.

Entering Mr. Chen's place, you feel like you're stepping into a whole new world. His apartment was immaculate, but signs of making were there if you knew what to look for. Tucked away in one corner was the pick and place machine. Next to it was a coffee table with boards ready to be populated, surrounded by tea cups.

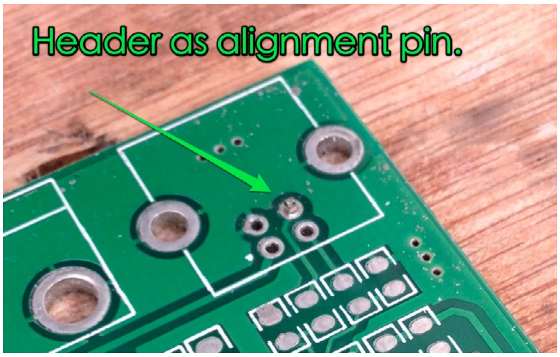
After a round of tea, he showed me the machine in operation. This $4000 pick and place machine was awesome to see. He had about 16 feeders and was populating entire boards in a single go. Between snapping pics and taking video, I asked him about what he does with it and why he needs gear like this.

It turns out, Mr. Chen was more interesting than his machine! You see, he's managed to carve out a nice little niche for himself by designing and manufacturing his own electronics and then selling them at the infamous Huaqiangbei electronics market. He started about 7 years ago and has been building and selling various things that whole time. Today he was making AVR ICE programmers, but tomorrow he might build controllers for the fans for his brother's small DC fan factory.

As we got to talking about making and DIY culture, I began to get a sense that this down-to-earth guy was someone who really understands the so-called Maker culture. He was very business savvy, and even had a slogan: 花小钱,赚大钱 which roughly means, "spend less and earn more." What he was describing was a lean operation where he had digital fabrication tools that allowed him to retool and switch around really quickly and efficiently. His house was doubling as his production floor, so he had very little overhead. He also understood that he needed to find niche markets in order to remain competitive.

His setup was slick and efficient: order pcbs and stencils from a fab, apply solder paste using a clever fixture, use the pick and place machine to get the parts on the board, reflow everything in his smt oven, and then hand-solder the connectors. The solder paste fixture itself was rather brilliant.

The stencil was attached to a hinged lid. He took a sacrificial pcb, hand aligned it with the stencil, and then glued it in place. He then took 2 header pins and nailed them into a connector hole until just a small nub was sticking out. These pins then became the alignment pins for the pcb to apply solder to. Brilliant, cheap, and effective.



I complimented him on his self-reliance and was surprised by yet another twist that would be enough to turn any urban farm-loving hipster green with jealousy. In addition to running his own electronics manufacturing operation, Mr. Chen was growing organic vegetables, and raising chickens and pigeons on the roof of his apartment! This guy was the picture of self-reliance, and he had a relaxed attitude that told me immediately that he had carved out a cozy existence in his life with his wife, son, pigeons, and electronics. Watching the flock of pigeons flying freely through the sky on a sunny winter afternoon, it was easy to see why.

All in all, it was a lovely afternoon, and I feel like I've come closer to understanding the impenetrable culture of Shenzhen makers. To all the Mr. Chen's of the world out there, and anyone else who pursues the goal of self-employment through making, I salute you! ■

Zach Hoeken Smith is the co-founder of MakerBot Industries and built the object sharing website *Thingiverse.com*, as well as the web-based digital manufacturing hub *BotQueue.com*. Lately he is living in hardware paradise also known as Shenzhen, China.

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

The Easiest & Quickest Affiliate System

Recruit, track, and promote your business

AFFILIATE.IO

The fast and easy way to accept affiliates into your online business

Visit affiliate.io/hacker for discount

My Clojure Workflow, Reloaded

By STUART SIERRA

One of the great pleasures of working with a dynamic language is being able to build a system while simultaneously interacting with it. To make this possible, first you need the ability to redefine parts of the program while it is running: Clojure provides this capability admirably. However, some aspects of Clojure's runtime are not quite as late-binding as one might wish for interactive development. For example, the effect of a changed macro definition will not be seen until code which uses the macro has been recompiled. Changes to methods of a `defrecord` or `deftype` will not have any effect on existing instances of that type.

The facilities that Clojure provides for loading code from files are not sufficient to deal with these issues. I wrote the second version of `tools.namespace` to make a "smarter" `require` that recognizes dependencies between namespaces and reloads them appropriately.

But `tools.namespace` is only part of the story. To really get the benefit of interactive development, I want to ensure that the version of the application I am currently interacting with is congruent with the source files I'm editing. That means not only that the application must be running the most up-to-date version of the code, but also that any state in the application was produced by that same code. It is dangerously easy, when changing and reloading code at the REPL, to get an application into a state which could not have been reached by the code it is currently running.

Therefore, after every significant code change, I want to restart the application from scratch. But I don't want to restart the JVM and reload all my Clojure code in order to do it: that takes too long and is too disruptive to my workflow. Instead, I want to design my application in such a way that I can quickly shut it down, discard any transient state it might have built up, start it again, and return to a similar state. And when I say quickly, I mean that the whole process should take less than a second.

To achieve this goal, I make the application itself into a transient object. Instead of the application being a singleton tied to a JVM process, I write code to construct instances of my application, possibly many of them within one JVM. Each time I make a change, I discard the old instance and construct a new one. The technique is similar to dealing with virtual machines in a cloud environment: rather than try to transition a VM from an old state to a new state, we simply discard the old one and spin up a new one.

Designing applications this way requires discipline. First and foremost, all states must be local. Any global state, anywhere, breaks the whole model. Second, all resources acquired by the application instance must be carefully managed so that they can be released when the instance is destroyed.

Enough talk. Here's how it works.

The System Constructor

In some "main" namespace, I provide a constructor function for the application. I usually call it `system` because it represents the whole system I am working on.

```
;; In src/com/example/my_project/
system.clj

(ns com.example.my-project.system)

(defn system
  "Returns a new instance of the
whole application."
  []
  ...)
```

The system constructor can optionally take parameters which specify its configuration.

Creating a system is not the same as starting it and should not have side effects. Usually the system constructor will create instances of other components it depends on and return a data structure such as a map or `defrecord` which contains them. My system instance might look something like this:

```
{:db
{:uri "datomic:mem://dev"}
 :scheduler #<ScheduledThreadPoo-
lExecutor...>
 :cache     #<Atom {}>
 :handler   #<Fn ...>
 :server    #<Jetty ...>}
```

Sometimes I have different versions of the constructor that produce different systems for interactive development, testing, and production.

Notice that some things which are "global" from the point of view of the application, such as my web server and scheduled thread pool, become "local" instances in this data structure. Any function which needs one of these components has to take it as a parameter. This isn't as burdensome as it might seem: each function gets, at most, one extra argument providing the "context" in which it operates. That context could be the entire system object, but more often will be some subset. With judicious use of lexical closures, the extra arguments disappear from most code. In addition to enabling more interactive development, this approach makes testing easier.

Start and Stop

Next, I have functions to start and stop the system. Ideally, these behave like real functions, in that they return a new value representing the "started" or "stopped" system, but they also have to perform side effects along the way, such as opening a connection to a database or starting a web server.

```
;; In src/com/example/my_project/
system.clj

(defn start
  "Performs side effects to
initialize the system, acquire
resources, and start it running.
Returns an updated instance of the
system."
  [system]
  ...)
(defn stop
  "Performs side effects to shut
down the system and release its
resources. Returns an updated
instance of the system."
  [system]
  ...)
```

These functions can call similar start/stop functions on sub-systems in turn. In the past, I've talked about a "Lifecycle" protocol containing start and stop methods. It's not necessary, but is sometimes useful to ensure that all components of the system can be started and stopped in a consistent way.

There's usually a bit of trial-and-error while I get the start/stop functions working correctly. If something in start/stop throws an exception, I could easily end up in a state where a sub-system has acquired a resource — such as a socket connection — but I do not have any handle on that sub-system with which to shut it down and release the resource. In that situation, there's nothing for it but to restart the JVM.

Dev Profile and user.clj

You probably know that the Clojure REPL starts by default in the user namespace. In addition, if there is a file named `user.clj` at the root of the Java classpath, Clojure will load that file automatically when it starts.

You probably don't want `user.clj` to be loaded in a deployed production app or library release, but by using Leiningen 2 profiles we can ensure that it is only loaded during development.

In my Leiningen `project.clj` file, I create a `:dev profile` with an extra `:source-paths` directory, plus whatever dependencies I want to use during development. `tools.namespace` has to be there, and I frequently add testing/development tools such as java.classpath or Criterium.

```
;; In project.clj:
(defproject com.example/my-project
"0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clo-
jure "1.5.1"]]
  :profiles {:dev {:source-paths
["dev"]
                :dependencies
[[org.clojure/tools.namespace
"0.2.3"]

[org.clojure/java.classpath
"0.2.0"]]}})
```

Leiningen will automatically merge the `:dev` profile into the project configuration for the `repl`, `test`, and `run` tasks, but not `jar` or `uberjar`. That means if I deploy my application (or release a library) as a JAR file, the files in `dev` will be excluded.

I create a `user.clj` file in the dev directory which defines a normal namespace called `user` and refers to a bunch of symbols I commonly use during development, as well as the symbols to construct, start, and stop the system.

```
;; In dev/user.clj
(ns user
  (:require
  [clojure.java.io :as io]
  [clojure.string :as str]
  [clojure.pprint :refer (pprint)]
  [clojure.repl :refer :all]
  [clojure.test :as test]
  [clojure.tools.namespace.repl
:refer (refresh refresh-all)]
  [com.example.my-project.system
:as system]))
```

Also in `user.clj`, I have a few things that I will only use during development, starting with a global Var to hold the system itself:

```
;; In dev/user.clj
(def system nil)
```

Now wait a minute, you might say, isn't that the global state you told us to avoid? It would be, if it were part of the application. Instead it's a container in which I can put the current instance of the application. I'm only going to use it for interactive development.

The `system` Var is manipulated by the following functions:

```
;; In dev/user.clj

(defn init
  "Constructs the current
development system."
  []
  (alter-var-root #'system
    (constantly (system/system))))

(defn start
  "Starts the current development
system."
  []
  (alter-var-root #'system system/
start))

(defn stop
  "Shuts down and destroys the
current development system."
  []
  (alter-var-root #'system
    (fn [s] (when s (system/
stop s)))))

(defn go
  "Initializes the current
development system and starts it
running."
  []
  (init)
  (start))
```

The exact division of these functions isn't important. Sometimes I omit `init` and `start` and just have `go`. The important thing is to have one function that creates and starts the system, and another function that tears it down.

Finally, the heart of my workflow: the `reset` function. This is one function which I can call at the REPL to 1) stop the current application instance; 2) reload any source files that have changed; and 3) create and start a new application instance.

```
;; In dev/user.clj

(defn reset []
  (stop)
  (refresh :after 'user/go))
```

The real work of reloading files is handled by the `clojure.tools.namespace.repl/refresh` function. It takes my go function as an argument, but go has to be passed as a namespace-qualified symbol so that it can be resolved after the `user` namespace has been reloaded.

Workflow

I do all my Clojure development in Emacs using nREPL.el, but nothing about this workflow is Emacs-specific. It should work with any environment that provides a REPL, as long as it doesn't try to do any code-reloading of its own. (For example, the reload-on-every-request functionality of ring-devel is incompatible with `tools.namespace`.) The fact that I use Emacs as my REPL is one reason I use `user.clj` instead of `:repl-options` in Leiningen's `project.clj`: those options have no effect on remote nREPL sessions.

The first thing I do when I start work is launch an nREPL session and call `reset`. Now my application is running and I can start working on it. Every time I make a change, I save the file and call `reset` at the REPL. (I have an Elisp helper function that I can bind to a keystroke.) Presto! My application is running again in a clean state with all the new code.

Rather than switching the REPL among several namespaces, I generally stay in user, where I have all my development tools like `clojure.pprint` and `clojure.repl`. I use the REPL itself for examining the application's state and testing individual functions. I frequently define little helper functions to examine the state of the application as I work on it, all of which are accessible by navigating the `system` object.

Anything I want to hang on to, such as a snippet of test data, I define in the `user.clj` file, because `tools.namespace` will destroy any Vars I created with `def` at the REPL.

Snags

This process isn't perfect by any means. One of the more irritating aspects is that any syntactic errors in a source file prevent all the code from being loaded, including `user.clj`. If a file fails to compile during the `tools.namespace` reloading process, any namespaces which depend on it no longer exist. So the `reset` function isn't available to call, nor are any of my aliases or referred symbols in the `user` namespace.

As a work-around, in tools.namespace 0.2.3 I added a feature to recover aliases and referred symbols in the current REPL namespace after a failed reload. This isn't perfect: the `reset` function still doesn't exist. But at least I can call the `refresh` function from `tools.namespace` without typing out its fully-qualified name `clojure.tools.namespace.repl/refresh`. Once I have successfully reloaded all the source files with `refresh`, I can call `reset` again to start the app.

A slightly worse problem occurs when starting a new REPL process: if there are any compilation errors in something loaded by `user.clj`, then the REPL will not start at all. I try to avoid this by starting the REPL from a known working commit, then only changing code after it's running. I also try not to commit any code which does not compile, but sometimes it happens.

Occasionally I do get my application into a state that I cannot recover from. Usually this happens when something in a `start` or `stop` function throws an exception. At that point, some component of the application may be in a broken state, but I don't have a reference to it that I can use to shut it down. If that component acquired external resources which I need to release before restarting it, e.g. socket connections, then there's basically nothing I can do but restart the JVM. Fortunately, these situations usually only occur while I'm writing the start/stop functions themselves, so after a few development cycles to get them working I don't have to worry about it.

Entry Points

The central thrust of this approach is to design your application so that you can construct multiple instances of it within a single JVM process. That's ideal for development, but what about production?

If you control the entry point to your application process, it's easy. Just write a `-main` function that creates a single instance of your application and starts it.

But often we deploy apps to environments where we do not control the `-main` function. For example, Ringweb apps deployed to a Servlet container have no `-main`. Furthermore, they expect a static reference to a Var which contains the root web handler function. If that handler is meant to be a closure over some contextual state, there's no place to construct it.

There are a couple of ways to work around this. One is to have a separate namespace in a "production" profile that constructs a single instance of the application and assigns it to a global Var. Alternatively, if the framework provides an "initialization" hook (as lein-ring does), you can use that to create the application instance and store it in a global Var. The root web handler function, created exclusively for production deployment, can pass the system object to functions that need it.

Epilogue

I'm continually tweaking this process, looking for improvements, but overall I'm pretty happy with it. It has enabled me to work rapidly on some fairly large applications. Best of all, it's agnostic with regard to development tools. You can adapt this workflow to any build tool that can substitute different CLASSPATHs for different circumstances.

Some of my Relevance coworkers like this approach; others find it too constraining. The Pedestal team uses pieces of this technique, such as the `:dev` profile, but without `tools.namespace`. They were annoyed that compiler errors prevented them from starting a new REPL, so they came up with a variation that uses a function in `user.clj` to load another file called `dev.clj`.

Stuart Sierra is a developer at Relevance, Inc., contributor to Clojure, and the co-author of Practical Clojure (Apress 2010) and ClojureScript: Up and Running (O'Reilly 2012).

Too Scared To Write A Line Of Code

By BEN HOWDLE

Kill your output with premature optimisation

"Design Patterns," "Code architecture," "Scalability," "OOP," "Maintainability," "The code you write now, is the legacy code of the future," "Be kind to your future self" & "Code smells."

Just like Bruce Almighty trying to block out the voices in his head, my pangs of guilt and angst come from the paradigms above; like an unwavering, continuous stream of distraction overwhelming my thinking as I'm trying to write one single line of code. One single line of code. That's it. Nothing special. No one's going to live or die if it's not the most optimised, architected and scalable line in the world.

We are under a constant barrage of posts, tutorials and articles about these paradigms. I often feel guilty if these paradigms aren't at the forefront of my mind whilst developing.

It often kills my output.

I'm trying to adopt a new workflow where I won't try and solve a problem until it becomes a problem; until I see it in the "wild." Just like Adii Pienaar wrote , why do we worry about scalability on day 1? [hn.my/day1]

This is precisely the approach I'm trying to apply to my development. However, it's the same with any new approach — I'm not blindly following it. It doesn't give you an excuse to write shit code, but forces you to complete a task more quickly and get that feature out there.

Your users care about precisely two things, "Does it work?" and "Is it fast?" (I'm talking specifically development here; they obviously care about design and all that jazz.)

So, I'm trying to stick to the following mantra:

Build it, release it, analyse it and only then decide if it needs optimising. ■

London-based JavaScript developer Ben Howdle got into development aged 19. He started learning to code in the evenings until he managed to get his first paid client. This year, Howdle has been building the next generation of KashFlow using BackboneJS and co-hosting Upfront Podcast.

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

Reservoir Sampling

Algorithms Every Data Scientist Should Know

By JOSH WILLS

DATA SCIENTISTS, THAT peculiar mix of software engineer and statistician, are notoriously difficult to interview. One approach that I've used over the years is to pose a problem that requires some mixture of algorithm design and probability theory in order to come up with an answer. Here's an example of this type of question that has been popular in Silicon Valley for a number of years:

> *Say you have a stream of items of large and unknown length that we can only iterate over once. Create an algorithm that randomly chooses an item from this stream such that each item is equally likely to be selected.*

The first thing to do when you find yourself confronted with such a question is to stay calm. The data scientist who is interviewing you isn't trying to trick you by asking you to do something that is impossible. In fact, this data scientist is desperate to hire you. She is buried under a pile of analysis requests, her ETL pipeline is broken, and her machine learning model is failing to converge. Her only hope is to hire smart people such as yourself to come in and help. She wants you to succeed.

The second thing to do is to think deeply about the question. Assume that you are talking to a good person who has read Daniel Tunkelang's excellent advice about interviewing data scientists [hn.my/dtunkelang]. This means that this interview question probably originated in a real problem that this data scientist has encountered in her work. Therefore, a simple answer like, "I would put all of the items in a list and then select one at random once the stream ended," would be a bad thing for you to say, because it would mean that you didn't think deeply about what would happen if there were more items in the stream than would fit in memory (or even on disk!) on a single computer.

The third thing to do is to create a simple example problem that allows you to work through what should happen for several concrete instances of the problem. The vast majority of humans do a much better job of solving problems when they work with concrete examples instead of abstractions, so making the problem concrete can go a long way toward helping you find a solution.

A Primer on Reservoir Sampling

For this problem, the simplest concrete example would be a stream that only contained a single item. In this case, our algorithm should return this single element with probability 1. Now let's try a slightly harder problem, a stream with exactly two elements. We know that we have to hold on to the first element we see from this stream, because we don't know if we're in the case that the stream only has one element. When the second element comes along, we know that we want to return one of the two elements, each with probability 1/2. So let's generate a random number R between 0 and 1, and return the first element if R is less than 0.5 and return the second element if R is greater than 0.5.

Now let's try to generalize this approach to a stream with three elements. After we've seen the second element in the stream, we're now holding on to either the first element or the second element, each with probability 1/2. When the third element arrives, what should we do? Well, if we know that there are only three elements in the stream, we need to return this third element with probability 1/3, which means that we'll return the other element we're holding with probability 1 — 1/3 = 2/3. That means that the probability of returning each element in the stream is as follows:

1. First Element: (1/2) * (2/3) = 1/3
2. Second Element: (1/2) * (2/3) = 1/3
3. Third Element: 1/3

By considering the stream of three elements, we see how to generalize this algorithm to any N: at every step N, keep the next element in the stream with probability 1/N. This means that we have an (N-1)/N probability of keeping the element we are currently holding on to, which means that we keep it with probability (1/(N-1)) * (N-1)/N = 1/N.

This general technique is called reservoir sampling, and it is useful in a number of applications that require us to analyze very large data sets. You can find an excellent overview of a set of algorithms for performing reservoir sampling in this blog post [hn.my/gregable] by Greg Grothaus. I'd like to focus on two of those algorithms in particular, and talk about how they are used in Cloudera ML [hn.my/ml], our open-source collection of data preparation and machine learning algorithms for Hadoop.

Applied Reservoir Sampling in Cloudera ML

The first of the algorithms Greg describes is a distributed reservoir sampling algorithm. You'll note that in order for the algorithm we described above to work, all of the elements in the stream must be read sequentially. To create a distributed reservoir sample of size K, we use a MapReduce analogue of the `ORDER BY RAND()` trick/anti-pattern from SQL: for each element in the set, we generate a random number R between 0 and 1, and keep the K elements that have the largest values of R. This trick is especially useful when we want to create stratified samples from a large dataset. Each stratum is a specific combination of categorical variables that is important for an analysis, such as gender, age, or geographical location. If there is significant skew in our input data set, it's possible that a naive random sampling of observations will underrepresent certain strata in the dataset. Cloudera ML has a sample command that can be used to create stratified samples for text files and Hive tables (via the HCatalog interface to the Hive Metastore) such that N records will be selected for every combination of the categorical variables that define the strata.

The second algorithm is even more interesting: a weighted distributed reservoir sample, where every item in the set has an associated weight, and we want to sample such that the probability that an item is selected is proportional to its weight. It wasn't even clear whether or not this was even possible until Pavlos Efraimidis and Paul Spirakis figured out a way to do it and published it in the 2005 paper "Weighted Random Sampling with a Reservoir." The solution is as simple as it is elegant, and it is based on the same idea as the distributed reservoir sampling algorithm described above. For each item in the stream, we compute a score as follows: first, generate a random number R between 0 and 1, and then take the nth root of R, where n is the weight of the current item. Return the K items with the highest score as the sample. Items with higher weights will tend to have scores that are closer to 1, and are thus more likely to be picked than items with smaller weights.

In Cloudera ML, we use the weighted reservoir sampling algorithm in order to cut down on the number of passes over the input data that the scalable k-means++ algorithm needs to perform. The ksketch command runs the k-means++ initialization procedure, performing a small number of iterations over the input data set to select points that form a representative sample (or sketch) of the overall data set. For each iteration, the probability that a given point should be added to the sketch is proportional to its distance from the closest point in the current sketch. By using the weighted reservoir sampling algorithm, we can select the points to add to the next sketch in a single pass over the input data, instead of one pass to compute the overall cost of the clustering and a second pass to select the points based on those cost calculations. ■

Josh Wills is Cloudera's Senior Director of Data Science, working with customers and engineers to develop Hadoop-based solutions across a wide-range of industries. He is the founder and VP of the Apache Crunch project for creating optimized MapReduce pipelines in Java and lead developer of Cloudera ML, a set of open-source libraries and command-line tools for building machine learning models on Hadoop.

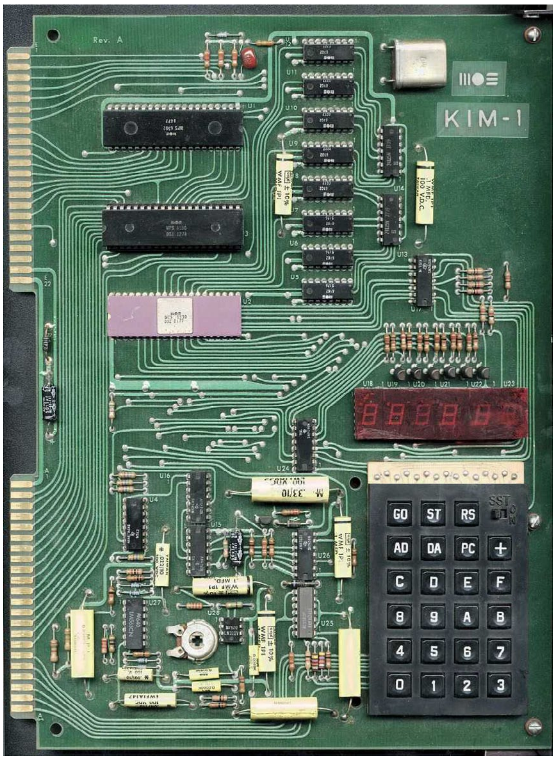
How I Coded In 1985

By JOHN GRAHAM-CUMMING

BACK IN 1985 I worked on the computerization of a machine designed to stick labels on bottles. The company that made the machines was using electromechanical controls to spool labels off a reel and onto products (such as bottles of shampoo) passing by on a conveyor. The entire thing needed to work with mm accuracy because consumers don't like labels that aren't perfectly aligned.

Unfortunately, electromechanical controls aren't as flexible as computer controls, so the company contracted a local technical college (where I was studying electronics) to prototype computer control using a KIM-1. Another student had put together the machine with a conveyor, a mechanism for delivering the labels, control of stepper motors, and infrared sensors for detecting labels and products.

My job was to write the software in 6502 assembly. Unfortunately, there wasn't an assembler and the KIM-1 just had a hex keypad and small display.



So, it meant writing the code by hand, hand assembling, and typing it in. The code looked like this:

Computer Operated Labelling Machine - Control Program 1

Main Loop

```
               Ø2ØØ   2Ø 4Ø Ø3   JSR set_up               ; Initialise ports
start:         Ø2Ø3   A9 5E      LDA #&5E                 ; Show mode
               Ø2Ø5   2Ø 8Ø Ø3   JSR display              ; as SE (Select)
bad_key:       Ø2Ø8   2Ø 94 Ø3   JSR get_key              ; Wait for A or B to be pressed
               Ø2ØB   C9 ØA      CMP #&A                  ; A = auto calibrate
               Ø2ØD   FØ Ø6      BEQ auto_calibrate
               Ø2ØF   C9 ØB      CMP #&B                  ; B = begin using data in memory
               Ø211   DØ F5      BNE bad_key
               Ø213   FØ 3D      BEQ go1

auto_calibrate: Ø215  A9 AC      LDA #&AC                 ; Show mode as
               Ø217   2Ø 8Ø Ø3   JSR display              ; AC (Auto calibrate)

               Ø21A   2Ø 8Ø Ø2   JSR find_first_label     ; Find the first label
               Ø21D   2Ø BC Ø2   JSR measure_length       ; Measure its length

               Ø22Ø   A9 FD      LDA #&FD                 ; Show mode as
               Ø222   2Ø 8Ø Ø3   JSR display              ; FD (Find wind on Distance)
               Ø225   2Ø 9F Ø3   JSR wait_for_B           ; Wait until B is pressed

               Ø228   2Ø DE Ø2   JSR measure_wind_on      ; Measure the wind on distance

               Ø22B   A9 ØD      LDA #&ØD                 ; Show mode as
               Ø22D   2Ø 8Ø Ø3   JSR display              ; OD (find Overhang Distance)
               Ø23Ø
               ~~Ø23Ø~~   EA EA EA
                      ~~2Ø 9F Ø3~~   ~~JSR wait_for_B~~  3*NOP     ~~; Wait until B is pressed~~

               Ø233   2Ø FC Ø2   JSR measure_overhang     ; Measure the overhang distance

go:
               Ø236   A9 A5      LDA #&A5                 ; Show mode as
               Ø238   2Ø 8Ø Ø3   JSR display              ; AS (All Set)

loop1:
               Ø23B   2Ø 6A 1F   JSR get_keys             ; Check to see if
               Ø23E   C9 ØE      CMP #&E                  ; E = Finish is pressed
               Ø24Ø   FØ 19      BEQ halt                 ; If so then halt

               Ø242   2Ø 1E Ø3   JSR wait_for_product     ; Wait until a product appears
               Ø245   2Ø 49 Ø3   JSR wind_off             ; Wind off one label
               Ø248   BØ F1      B~~EQ~~NE loop1              ; Carry on if labels still available

               Ø24A   A9 E5      LDA #&E5                 ; No labels so show mode as
               Ø24C   2Ø 8Ø Ø3   JSR display              ; ES (Empty Spool)
               Ø24F   2Ø 9F Ø3   JSR wait_for_B           ; Wait for B to be pressed to signal
```

Major subroutines

```
find-first-label:  Ø2BØ  2Ø  CØ  Ø3    JSR  read-label-detector   ; Wait until label detector
                   Ø2B3  FØ  Ø6        BEQ  got-first-label       ; goes low
                   Ø2B5  2Ø  52  Ø3    JSR  advance-one-step      ; Advancing label if still high
                   Ø2B8  4C  BØ  Ø2    JMP  find-first-label
got-first-label:   Ø2BB  6Ø            RTS

measure-length:    Ø2BC  A9  ØØ        LDA  #Ø                    ; Clear label length store
                   Ø2BE  8D  FB  Ø3    STA  label-length
                   Ø2C1  8D  FC  Ø3    STA  label-length+1
                   Ø2C4  2Ø  C6  Ø3    JSR  read-Hip-Hop          ; Store current Hip Hop
                   Ø2C7  8D  FF  Ø3    STA  temp                  ; setting
measure-loop2:     Ø2CA  2Ø  52  Ø3    JSR  advance-one-step      ; Move forward
                   Ø2CD  EE  FB  Ø3    INC  label-length          ; Increment length
                   Ø2DØ  DØ  Ø3        BNE  measure-loop          ; counter
                   Ø2D2  EE  FC  Ø3    INC  label-length+1
measure-loop:      Ø2D5  2Ø  C6  Ø3    JSR  read-Hip-Hop          ; Read Hip Hop
                   Ø2D8  CD  FF  Ø3    CMP  temp                  ; Compare with state on entry
                   Ø2DB  FØ  ED        BEQ  measure-loop2         ; Carry on advancing if same as before
                   Ø2DD  6Ø            RTS                        ; else length measured

measure-wind-on:   Ø2DE  A9  ØØ        LDA  #Ø                    ; Clear wind on distance store
                   Ø2EØ  8D  F9  Ø3    STA  wind-on
                   Ø2E3  8D  FA  Ø3    STA  wind-on+1
wind-loop2:        Ø2E6  2Ø  52  Ø3    JSR  advance-one-step      ; Move forward
                   Ø2E9  2Ø  59  Ø3    JSR  delay                 ; Wait a while
                   Ø2EC  EE  F9  Ø3    INC  wind-on               ; Increment distance counter
                   Ø2EF  DØ  Ø3        BNE  wind-loop
                   Ø2F1  EE  FA  Ø3    INC  wind-on+1
wind-loop:         Ø2F4  2Ø  6A  1F    JSR  get-keys              ; Check to see if
                   Ø2F7  C9  ØF        CMP  #$F                   ; F = Finish has been pressed
                   Ø2F9  DØ  EB        BNE  wind-loop2            ; Carry on advancing if not
                   Ø2FB  6Ø            RTS

measure-overhang:  Ø2FC  A9  ØØ        LDA  #Ø                    ; Clear overhang length store
                   Ø2FE  8D  FD  Ø3    STA  overhang
                   Ø3Ø1  8D  FE  Ø3    STA  overhang+1
over-loop:         Ø3Ø4  2Ø  94  Ø3    JSR  get-key               ; Wait for F or C to be pressed
                   Ø3Ø7  C9  ØF        CMP  #$F                   ; If F then exit
                   Ø3Ø9  FØ  12        BEQ  over-done
                   Ø3ØB  C9  ØC        CMP  #$C                   ; If C then advance
                   Ø3ØD  DØ  F5        BNE  over-loop
                   Ø3ØF  2Ø  52  Ø3    JSR  advance-one-step      ; labels and
```

It was immediately obvious that computer control was going to be more flexible. The program first did automatic calibration: it measured the length of labels on the spool itself, it measured the distance between labels itself, and it enabled an operator to quickly set up the "overhang" distance (how much of the label is sticking out so the product can catch onto it).

While running, it could automatically detect how fast the conveyor was moving and compensate, and spot when a label was missing from the supply spool (which happened when one peeled off by accident).

Of course, writing code like this is a pain. You first had to write the code (the blue), then turn it into machine code (the red) and work out memory locations for each instruction and relative jumps. At the time I didn't own a calculator capable of doing hex, so I did most of the calculations needed (such as for relative jumps in my head).

But it taught me two things: to get it right the first time and to learn to run code in my own head. The latter has remained important to this day. I continue to run code in my head when debugging, typically I reach for the brain debugger before gdb or similar. On the KIM-1 there were only the most basic debugging functions and I built a few into the program, but most of the debugging was done by staring at the output (on the hex display) and the behavior (of the steppers), and by running the code through in my head.

Here's the full program [hn.my/kim1] for the curious.

P.S. A number of people have pointed out that in 1985 the KIM-1 was far from state-of-the-art and we had lots of niceties like compilers, etc. True. In fact, prior to this I had been programming using BASIC and ZASM (Z80 assembler) under CP/M, but you go to war with the army you have: the technical college had a KIM-1 to spare; it had good I/O and thus made a fine prototyping system for an embedded controller. ■

Dr John Graham-Cumming is a computer programmer and author. He can be found on the web at *jgc.org*

Why Maybe Is Better Than Null

By NICK KNOWLSON

THIS ARTICLE IS divided into two parts: Explanation and FAQ. The explanation shows the reasons why a bunch of people think Maybe is way more useful than `null`. The FAQ is a list of my responses to common arguments I've seen about the shortcomings of Maybe.

I'm going to try to not go overboard with details here — my aim is to make it accessible to as many programmers as possible, not to be as thorough as possible.

Explanation

Motivation

Tony Hoare, the inventor of `null`, has gone on record calling it his "billion-dollar mistake". So what should replace it?

Maybe, at its core, is a construct that allows programmers to move `null` checks into the type system so they can be enforced at compile-time. Instead of forgetting to deal with a `null` check and finding out with an exception at run-time, you forget to deal with a `null` check and find out with an error at compile-time, before anyone else even sees it! And that's not just some `null` checks, that's all of them!

Details

There are two components to an environment free of `null` pointer exceptions:

1. The elimination of `null`. This means that all types (even reference types!) become non-nullable.
2. An alternative representation for the idea of "may contain an empty or invalid value". This is what Maybe is for.

So how does Maybe accomplish this, and how does it achieve all those benefits listed above? It's actually very straightforward.

I'm going to explain this in object-oriented terms, because if you're already familiar with algebraic data types, odds are you already know about Maybe too. Anyway, think of `Maybe<T>` as an interface with a single type parameter that has exactly two implementing classes: `Just<T>` and `Nothing`. The `Just<T>` class wraps a value of some other type and the `Nothing` class doesn't. There are a variety of methods provided by Maybe to extract the value safely, but I'm going to omit these for now, as they're not the point. When you receive an object of type `Maybe<String>` (for example) you now have the type system helping you out, telling you "there might be a String here, but it might be empty". You can't perform operations on the String until you've safely extracted it and made a choice about what to do in the case that it's empty.

By itself (without point #1) this is nice but not fantastic. The benefit really kicks in when you also have non-nullable types. It simplifies the 80% of the cases that don't involve `null` and gives significance and meaning to the times when you do deal with objects wrapped in `Maybe<T>`. It lets you say both "I know that this value will literally never be `null`" and "It is immediately obvious to me that I need to handle the case of an empty value here".

Conclusion

It's not that dealing with any given instance of `null` is particularly hard; it's that it is so easy to miss one. Removing this concern and encoding the information in the type system means programmers have less things to keep track of and simplifies control flow across the entire program. Like with memory management: when you don't have to keep track of it manually, it is just plain easier to write code. More importantly, it is easier to write more robust code. This goes for all programmers, not just the experienced or talented.

And that is something I am firmly in favor of. A product is never the result of a single person's code — everything has dependencies. Improvements to other people's code benefit all of us.

Addendum

There are two more points I'd like to address about Maybe that are separate from actually explaining why it is useful.

First, I've been a bit inaccurate on purpose when just referring to this idea as Maybe. There is an implementation of this idea in Haskell called Maybe, but implementations in different languages have different names.

- ML, Scala, F#, Rust: Option
- Fantom, Kotlin: ? appended to type
- C#: Nullable or ? appended to type

Second, not all languages with Maybe have non-nullable types. This makes Maybe less valuable in those environments (since you lose the very useful "I know this value will never contain `null`" guarantee") and ends up confusing people who are skeptical of Maybe's benefits.

To help clarify this: I agree that in a language where you don't have the guarantee provided by non-nullable types, Maybe just isn't as useful. But it is not useless either and, depending on the environment, may still provide some benefit.

FAQ

Posts like these are tricky. To explain something understandably and (relatively) concisely I can't qualify every statement and address all the holes inline. Here is where I'll address the bits I skipped as well as some common sentiments I've previously seen on this topic.

Maybe isn't the be-all end-all.

I definitely agree. For one thing, I haven't even mentioned Either! This article is for people who aren't even convinced of the benefits of Maybe yet. In order to get my point across effectively I want to avoid overwhelming the reader with information, so I restricted the topics brought up here.

If you want a higher level perspective on this issue, take a look at dmbarbour's view:

There are two mistakes. One mistake is providing a "sum" type (eqv. to Just Object | Nothing) without recognition of the typechecker. The other mistake is joining this sum type at the hip with the idea of references, such that you cannot have one without the other.

These mistakes may, and I suspect should, be resolved independently. Thinking there's just one mistake, and thus just one language feature to solve it, might very well be a third mistake.

Beautifully stated! This is a much more general (and elegant) way to look at it. It's a somewhat harder sentiment to communicate effectively to a lot of people though.

My IDE plugin already does this.

Yes, there are some IDEs and plugins that provide limited `null` reference analysis. The key though is that it is limited. As far as I know (and I've looked) none of them provide the same system-wide elimination of `null` that encoding it in the type-system can guarantee.

And so, you still don't get the same reassurances of "I know this value will literally never be `null`" and "It is immediately obvious that I have to handle the case of an empty value here".

NPEs are the proper response to a missing value you forgot to consider. You should be notified when something goes wrong, not hide it with Maybe.

I've got good news for you — we fundamentally agree in our approach to how errors should be handled! You might have seen some bad examples of Maybe usage, since proper usage would lead to these errors being caught even earlier than a NullPointerException would have.

You can still choose to do the equivalent of `if (null) return;` and some examples will do that, because it makes sense in some contexts. What matters is that Maybe forces you to think about it at the time of writing the code, and to be explicit about it.

Instead of you being notified when things go wrong, Maybe forces you to think things through in the first place and make an explicit choice about what to do (at least as far as possibly empty values are concerned).

And finally, for those of you who really love exceptions, implementations of Maybe usually provide an unsafe retrieval method, so you can replicate the behavior of `null` (run-time exceptions and all) if that is what you choose to do.

The real problem is people not properly reasoning about their functions, that isn't the fault of null.

Sure, that is one way to look at it: it's not `null`'s fault, it is the programmer's fault. If you take this view then `null` is just one of the tools used to represent emptiness and invalid values. But it isn't a very good tool, or at least not as good as it could be.

Maybe is a tool that fills the same gap as `null` but is much more helpful to programmers. It helps directly address the core problem of "people not properly reasoning about their functions" by pointing out mistakes in reasoning earlier. With it you can statically verify that all `null` checks are made, and eliminate an entire class of run-time errors.

I'm not claiming it is a silver bullet, but it is a better tool.

Null is meaningful! What if a value cannot have any meaningful default value?

Then either wait until it has a meaningful value to put in it or wrap it in Maybe and give it a value of `Nothing`. That's what Maybe is for — to provide a type-checkable alternative to `null`!

So you're still testing against null, except that it's called Nothing. What have we gained?

We have gained earlier detection of an entire class of errors! Now if there is a missed check for an empty value you will find out at compile-time rather than run-time. Using Maybe forces you to be explicit about possibly-empty values and deal with the case where they are empty.

The user doesn't see any `null` reference exceptions; they are all fixed before they even get outside the developer's computer.

I think the safe navigation operator in Groovy/Kotlin/Fantom/CoffeeScript is better than Maybe.

I'm going to talk about Kotlin and Fantom separately in the next section because they're special.

In Groovy/CoffeeScript, the safe navigation operator (`?.`) lets you safely call a method or access a field on an object that may be `null`. If the object IS `null` then the method/field just returns `null` as well, instead of an exception being thrown.

I agree that the safe navigation operator is certainly convenient, but it is solving a different problem. If you compare it directly to Maybe, it's only solving the "retrieve value from possibly empty object" part of Maybe. This is a nice thing to have, but it isn't nearly as interesting as moving a whole class of run-time exceptions to compile-time.

Which is fine, it doesn't have to be as good as Maybe to still be useful. Just don't misrepresent it as being anything more than a convenient syntax for `null` checks.

What about Fantom & Kotlin?

Fantom and Kotlin are different because they are both languages that have non-nullable reference types and have built Maybe in as a language feature. In both languages (Fantom, Kotlin), you can distinguish a reference that may hold `null` by appending a ? to its type (i.e. `String?`). The compiler can then keep track of it as if it were a `Maybe<String>` and is able to prevent you from unsafely accessing its contents. They provide safe navigation and elvis operators to extract the value like Groovy does.

This is probably where opinions will start to differ among people who think Maybe is a good idea.

I personally am thrilled by the steps Fantom and Kotlin have taken and think that they are a great solution to eliminating `null` reference exceptions. They use the fact that they've implemented it as a language feature to provide really convenient and easy to understand syntax. So easy to understand, in fact, that it might not be obvious that it is the same damn thing as Maybe. The only differences are that Fantom and Kotlin have a special syntax for it baked in, and that (in exchange) it is a little bit more limited than Maybe as a library is.

The only downsides to this approach are related to the fact that it is specialized. When you stretch against the limits of Maybe you can't drop in Either instead. You also can't wrap Maybes in another Maybe (`Maybe<Maybe<String>>`), which you might do when you have nested calls that could fail.

I can't speak to how often this ends up being an issue for people working in Fantom/Kotlin and what alternatives the language provides because, frankly, I am pretty unfamiliar with them. If anyone with experience would like to speak up I'd be happy to add their information to this section.

But Option in Scala DOESN'T save you from null!

Yes, in Scala you can still get NullPointerExceptions. Scala doesn't have non-nullable reference types because Martin Odersky (for what were probably good reasons — I'm guessing related to java interop) decided to include `null` in his language. That doesn't invalidate all the other implementations of Maybe and it doesn't mean it can't still be somewhat useful in Scala.

Feel free to point out to people that Scala's implementation of Option still allows for NullPointerExceptions, just don't generalize it to "Maybe and Option aren't useful".

Safety ISN'T guaranteed because of the existence of unsafe extraction methods.

Often implementations of Maybe will include more than just safe extraction methods. Haskell's `fromJust` and Scala's `get` are both retrieval functions that throw runtime errors if the value wrapped in Maybe doesn't exist. Just like how `null` usually works.

So it is possible to shoot yourself in the foot if you want to. The difference is you have to explicitly ask for this behavior — it cannot sneak in by accident.

Whenever I claim Maybe can move `null` reference exceptions to compile-time, it comes with the assumption that you're using the built-in safe extraction methods and that you're not requesting run-time exceptions.

Using Maybe is not worth the overhead.

This is a hard question to answer without getting specific. If this was said about a specific language or kind of application and the person saying it has done their due diligence or has some working code to back it up, then I can't address that here.

If it is a less qualified statement however, I have some counterpoints that I can share.

1 Bugs are expensive, even more so the later on they are caught. It takes a developer time to find and fix bugs — the more bugs, the more time it takes. For each bug, overhead is introduced in the form of finding, tracking, fixing, and testing it. Worse, bugs that make it all the way to production impact your users and can have even more expensive consequences like data corruption. For some applications, small amounts of bug-related downtime could cost thousands (or millions!) of dollars.

This is the whole reason why we have test suites, type systems, static analysis tools, code reviews, even exceptions! We want to catch bugs earlier.

Maybe lets you catch one of the most common bugs, `null` reference exceptions, at compile-time instead of runtime. So if you say "it is not worth the overhead", think about what `null` reference exceptions are costing you first, and make sure you really do know how much it is worth.

Unless… you are one of those lucky few who say that `null` reference exceptions really are just not an issue for you. Maybe your other bug prevention measures combined are good enough and when you tracked your faults you found you don't end up dealing with `null` reference exceptions very much. For you guys, keep in mind that you are probably not in the majority.

2 There might not be as much syntactic overhead as you think. In many cases it actually reduces overhead. Languages that provide Maybe usually provide many convenient ways to extract values which are actually often shorter than the `null` checks you would otherwise be writing. On top of that, code that doesn't deal with possibly-empty values doesn't need to use Maybe (or check for `null`!) at all.

For those of you talking about having to mark too many properties as optional and having to deal with Maybe everywhere, think about it like this: You would have had to deal with the same amount of possibly-empty values either way. The only difference is that now you have the compiler helping you out. If your code is meant to be robust, it will need `null` checks anyway. For the cost of adding a little wrapper around your types you can replace those easy-to-forget `null` checks with their equivalent compiler-checked Maybe extraction methods.

You also get perfect safety and ease of mind when dealing with values that cannot logically be empty.

Enough vague, high-level information. Show me some examples!

I intentionally avoided showing examples in the explanation section to avoid taking attention away from the main points. For a topic like this one, as soon as you show some code it is like sticking a bikeshedding magnet right in the middle of your article. But since this post is aiming to be a definitive reference, it could use at least a few examples.

Another measure I am going to be taking to avoid stirring up unnecessary arguments is comparing like with like. I will show a scenario where `null` checks are used to deal with an empty value in a certain way, then I will show what that example would look like in a language with good support for Maybe.

The languages I picked are:

- **Java**: to represent the traditional ways of handling nulls that most people are hopefully familiar with.
- **Kotlin**: to represent languages with non-nullable references and support for Maybe baked into the language.
- **Haskell**: to represent languages with non-nullable references and support for Maybe as a library.

The scenarios follow:

Dealing with it explicitly

Java

```
public static void retrieveInfoExplicit() {
    String information = new Random().nextInt(2) == 0 ? "a,b,c" :
null;

    if (information == null) {
        System.out.println("No information receved.");
    } else {
        System.out.println(Arrays.toString(parseInfo(information)));
    }
}
```

Kotlin

```
fun retrieveInfoExplicit() : Unit {
    val information = if (Random().nextInt(2) == 0) "a,b,c" else null

    if (information == null)
        println("No information received.")
    else
        safeParseInfo(information) forEach { println(it) }
}
```

Haskell

```
retrieveInfoExplicit :: IO ()
retrieveInfoExplicit = do
    num <- randomRIO (0, 1)
    let information = if (num :: Int) == 0 then (Just "a,b,c") else
Nothing
    case information of
        Nothing -> putStrLn "No information retrieved."
        Just i -> putStrLn $ show $ safeParseInfo i
```

Dealing with it implicitly

Java

```
public void retrieveInfo() {
    String information = Random().nextInt(2) == 0 ? "a,b,c" : null
    parseInfo(information);
}
```

Kotlin

```
fun retrieveInfo() : Unit {
    val information = if (Random().nextInt(2) == 0) "a,b,c" else null
    parseInfo(information)?.forEach { println(it) }
}
```

Haskell

```haskell
retrieveInfo :: IO ()
retrieveInfo = do
    num <- randomRIO (0, 1)
    let information = if (num :: Int) == 0 then (Just "a,b,c") else
Nothing
    putStrLn $ show $ parseInfo information
```

Returning null as well (e.g., guard statements)

Java

```java
public String[] parseInfo(String information) {
    if (information == null) {
        return null;
    }
    return information.split(",");
}
```

Kotlin

```kotlin
// Can choose to imitate java...
fun parseInfo(information : String?) : Array<String>? {
    if (information == null) {
        return null
    }
    return information.split(",")
}

// ...Or take advantage of the safe navigation operator
fun parseInfo(information : String?) : Array<String>? {
    return information?.split(",")
}
```

Haskell

```haskell
-- Can do it with pattern matching...
parseInfo :: Maybe String -> Maybe [String]
parseInfo Nothing = Nothing
parseInfo (Just information) = Just (splitOn "," information)
```

```
-- ...Or with do notation...
parseInfo :: Maybe String -> Maybe [String]
parseInfo information = do i <- information
                           Just (splitOn "," i)

-- ...Or with fmap
parseInfo :: Maybe String -> Maybe [String]
parseInfo information = fmap (splitOn ",") information
```

Giving something a default value

Java

```
String name = (author == null) ? "Anonymous" : author;
```

Kotlin

```
val name = author ?: "Anonymous"
```

Haskell

```
let name = fromMaybe "Anonymous" author
```

Throwing an exception

Java

```
public String[] parseInfo(String information) {
    return information.split(",");
}
```

Kotlin

```
fun parseInfo(information : String?) : Array<String> {
    return information!!.split(",")
}
```

Haskell

```
parseInfo :: Maybe String -> Maybe [String]
parseInfo information = Just (splitOn "," (fromJust information))
```

Not having to check for null

Java

Does not have this option, you always have to deal with `null`.

Kotlin

```
// note: no ? appended to types
fun safeParseInfo(information : String) : Array<String> {
    return information.split(",")
}
```

Haskell

```
-- and these types are not wrapped in Maybe
safeParseInfo :: String -> [String]
safeParseInfo information = splitOn "," information
```

The nice thing about this last example is that if the code changes and this function now needs to be called with a value that might be `null`, the code won't compile until the developer has revisited `safeParseInfo` and explicitly chosen to deal with `null` in one of the ways shown above.

In case you want to run the examples yourself:

- The Java examples import `java.util.Arrays` and `java.util.Random`
- The Kotlin examples import `java.util.Random`
- The Haskell examples import `System.Random`, `Data.List.Split`, and `Data.Maybe`

That's it — that's the end of both the FAQ and this article. Hope you enjoyed it! ■

Nick Knowlson is a software developer from Victoria, British Columbia. He enjoys reading, gaming, tea and (of course) programming. Right now he is focusing on programming languages and language features.

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

TWILIO CON 3

SAN FRANCISCO, CA

SEPTEMBER 17-19, 2013

TwilioCon is 3 days of inspiration and technical sessions with Twilio engineers and the brightest minds in web and mobile technology on the future of communication services.

$199 TWO-DAY CONFERENCE

Use promo code Hacker20 for 20% off the full price

REGISTER TODAY

twilio.com/conference