12 Simple Rules for Learning to Code

Dr. Angela Yu
Head of iOS Course Development at London App Brewery
www.londonappbrewery.com

www.londonappbrewery.com

Contents

1.Trick Your Brain with the 20min Rule

Learning to code is a bit like going to the gym. Even if you max out and spent a whole weekend at the gym, you will not see a visible difference in your body. The more regularly you learn to code, the more likely it is that you'll start seeing your ripped coding muscles. (The irony is not lost on me).

But the problem is where do you find the time? Between working your full-time job, seeing family and friends and life admin, when are you supposed to sit down and practice this "daily coding"?

While I was working as a doctor, I spent about 12 hours at the hospital, 1 hour commuting and approximately 2 hours on general life-sustaining stuff, such as eating. So that left me with only 9 hours remaining in my day. Theoretically, 2 hours could be allocated to coding practice and 7 hours on sleep. But there is nothing more difficult than trying to convince your work-saturated brain to sit down and learn, when you could be watching Game of Thrones with a tub of ice-cream.

But then I found a trick.

As humans, we have a lot of inertia. This can be bad for us - I'm looking at you, "24" boxset. However we can also turn it to

our advantage. I found that once I got started coding and making things, I got so absorbed into the project, that I no longer cared about TV, food or sleep. There were quite a few weekends when I coded until sunrise.

So how do we take advantage of this inertia? First, you must understand that task-switching is very difficult. It requires a lot of motivation. If as soon as you get home, you slump on the sofa and switch on the TV, you've already lost that evening. This is because the amount of motivation required to switch-task and do something not driven by evolution like eating or sleeping is a Herculean task.

This is why the moment you enter the door and change to new environment is the most crucial moment. If at this moment, you tell yourself that you are just going to do 20 minutes of coding practice, you will most likely succeed and use your own inertia to end up learning for an hour or more. No brain will perceive a 20 minute task as a lot of effort and you end up tricking your brain to take advantage of your evening.

The next step is to develop a habit. Research suggests that in order to develop a new habit, you have to carry out the task daily for a month. I've used this next trick for loads of different things, from exercising to coding, it invariably works like a charm. To preface this trick, I want you to imagine a wall with

five paintings hanging on it, four of which are perfectly aligned, perfectly horizontal, but one is crooked. Now really imagine it, is there a part of you that wants to fix it?

Now let’s imagine a monthly calendar with boxes representing individual days, if you nurtured that new habit on a particular day then you make a line through that day. If you continued your streak the next day then you extend that line and so on and so forth. There is something about not breaking a continuous line that motivates most people to continue to develop a habit. As strange as it sounds, there’s many times when I would have given up, but compelled to continue because of a long, continuous line.

Try it out with the liner calendar here.

2. Code for a Purpose. Have a project.

When I first started learning how to code, there were countless times when I picked it up then gave up, again and again. This is a common story amongst self-taught adult coders. Looking back, after teaching so many students, I finally realise what's going on. A lot of beginners start learning to code by picking an arbitrary language and follow along with a bunch of tutorials. Copying code, line by line, sometime writing code to work out the fibonacci sequence, other times to find all the even numbers. But you know what? I can find the fibonacci sequence a lot faster by Googling for it and picking out even numbers is really not all that interesting.

Here's the truth. If you are learning to code for the sake of learning to code, it'll be pretty difficult for you to get good at it. Skills that require a lot of time to hone will depend on internal motivation. Something from within that makes you forget to eat and sleep. I can honestly say that coding on my own projects is one of the most enjoyable things I do. It combines logical thinking with creativity and at the end, you will have made something. In most cases, something that the world has never seen. Something that could make you life easier or more

enjoyable. Something that could make loads of people's lives easier and more enjoyable. It’s like making a crazy-ass custom motorbike in your garage, without needing the garage or spending a cent on the components.

This is what motivates most people. The creating part, the making part. So I urge you start learning to code by following a tutorial that makes something, anything. Of course it’s unlikely that at the beginning you’ll be able to code up Clash of Clans or League of Legends. But you’ll be able to make something interesting. It could be a dice game or a flash-card app. But as long at the end of the tutorial you’ll have made something you can use and play with, then you’ll be motivated to code to the end.

After all our courses, we always tell our students to think up of a simple app that they want to make. Something that uses the skills that they’ve learnt during the course but will also stretch them a little because they have to find out how to include some new functionality.

We had a student who went on to make an app that wakes them up a minute earlier everyday to ease the transition to an earlier waking time. There’s a student who made a custom slideshow app as a mother’s day present. Someone else made

an app that is a timer for making perfect steaks based on its weight and thickness.

There’s no limits on your imagination. It will be difficult when you start working on your own app because there’s no step by step instructions, but it will also bring about the biggest improvement in your coding ability.

3. There is no “perfect language to learn”

Whenever I do large talks, there will always be one person who asks me “which programming language should I start with”? There is this common perception that somewhere out there lies a *perfect* language for beginner programmers. Some argue it’s Python, some say it’s Swift. But I say they’re all wrong. A programming language is simply a tool. It is no different from any other tool in your hardware box. If you want to hammer a nail, you should be using a hammer. If you want to fix your water pipes, you’ll probably need a spanner. Yes, it’s possible to hammer in a nail using the side of the spanner and the same programming language can be used to solve different types of problems. The carpenter will tell you that his favourite tool is a hammer and the plumber will say it’s the spanner, but it won’t make them the “best tool to fix things”.

A web developer will tell you that javascript is the best language to learn for a beginner. An statistician will advise you that you'll be best served with the R programming language. But at the end of the day, all that matters is what you are trying to do with your tool. If you want to make iOS apps, then learn Swift. If you want to make Android apps, then learn Java. The core programming concepts, loops, conditionals, functions, they're all the same. The difference is mostly syntactical. In English, we have *werewolves*, in German you have Werwölfe. It's still the same shirt-ripping mammal that comes out during a full moon, it's just spelt differently.

Printing to the console in Swift:

print("Hello Werewolves")

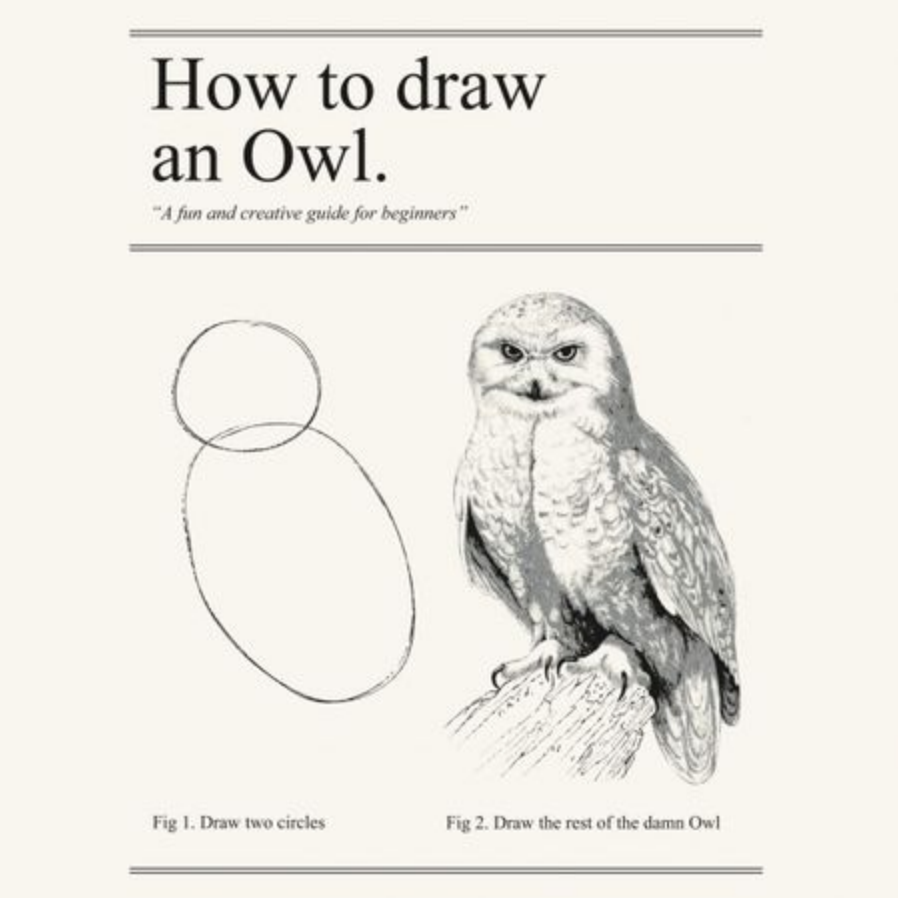
Printing the the console in Java:

println("Hello Werwölfe")

So, decide on the task that you are trying to accomplish, then pick the best tool for that task.

4. Understand what you’re writing

I have an issue with the way that most programming tutorials are written. There are far too many tutorials where you see the “this is how you draw an owl” phenomenon.



It’s almost as if the programmer had good intentions and started off by showing you how to do everything, step-by-step. But then, at some point he realises that he has embarked on a sisyphean task and gives up. I’ve seen tutorials where the author starts off with an excruciating level of detail then mid-way reverts to “then you simply set up a cloud database”. Bearing in mind that this is a tutorial aimed at beginners!

This leads to a number of problems. The most common problem is a student who just copies the code in the tutorial and has no clue what any of it does. Why did he add that extra

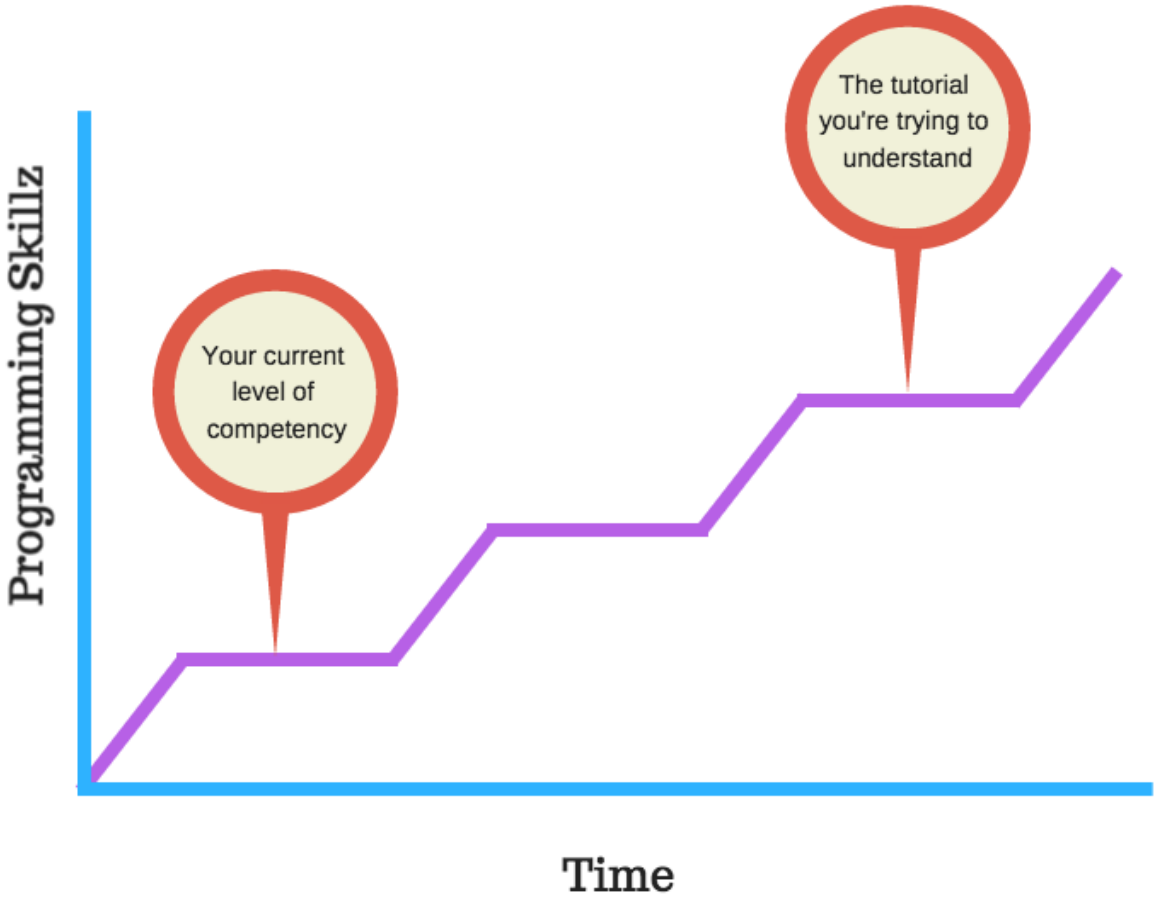
line after parsing the JSON? Why is he making this dictionary differently from the last one?

It’s very easy to get knees deep in one of these types of tutorials because it promises to teach you how to build “Flappy Bird” or “Candy Crush”. But two thirds of the way in, none of the things you’re typing makes sense and you start seeing red all over the screen. Bugs. Loads of them. Why? No idea. Nothing runs. The last 3 hours were spent copying code and you learnt nothing other than coding sucks.

Don’t get into this trap. If you see a tutorial that has jumps from beginner to advanced after line 3 or uses the word “simply” too liberally or doesn’t explain any of their code, then stop. Leave that tutorial.

There’s plenty of fish in the sea.

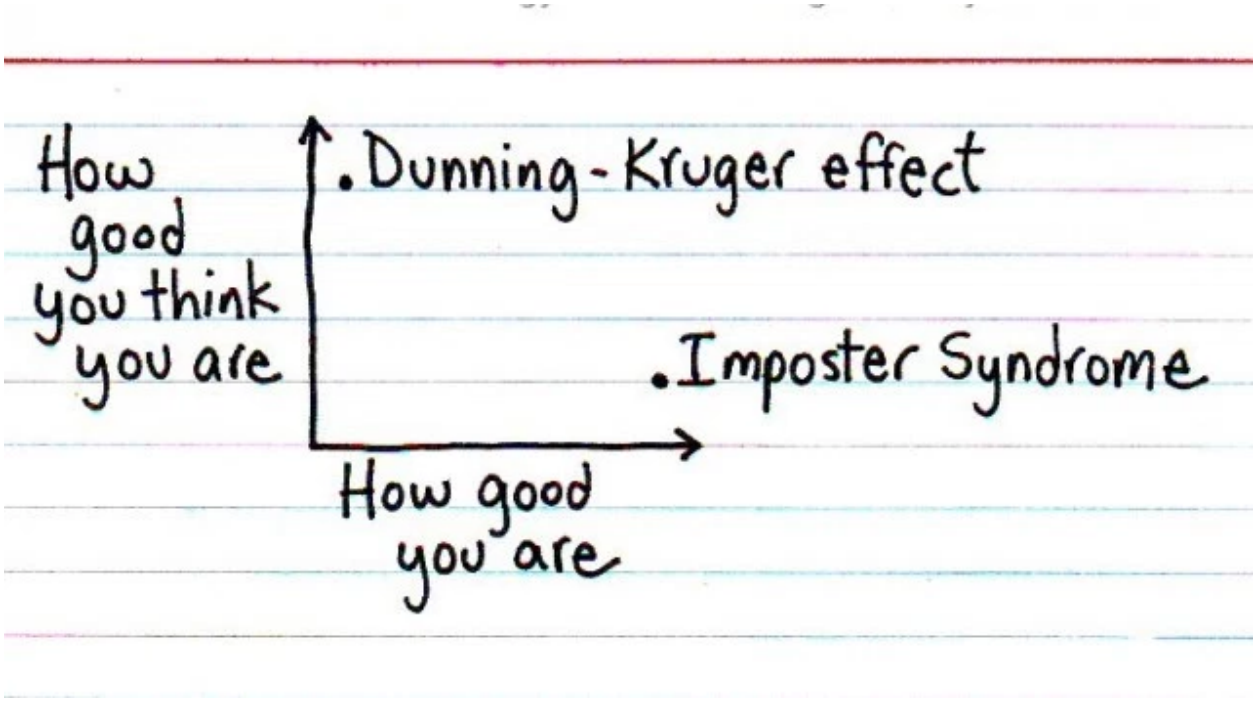
Other times, the author does try to explain what they’re doing. But you still don’t understand a thing that they’re saying, then you’re in an advanced tutorial that won’t improve your programming. It can be tempting to build grand things, especially when the blog is promising that anyone can make it. But if you can’t work out what’s going on, you’re be better served by building a better foundation.



The key to learning to code is all about ramping. You want to be stretched over and over again and for knowledge to be build on previous knowledge. If that ramp is too steep, you'll get lost. If that ramp is too shallow, you'll get bored. The right gradient is different for everyone. That's why our video/in-person tutorials are always accompanied with a printed course book. This way, you can speed along if you're comfortable with the concepts and slow down to absorb if it's something unfamiliar.

5. It’s Ok to Not Know

Software engineers are purportedly the profession that is most likely to suffer from imposter syndrome. The psychological phenomenon where people feel like frauds and far underestimate their own skills and abilities.



Programmers tend to be self-critical and constantly feel that everyone else is better at programming than they. If you’ve ever felt this way, you’re not alone, as studies show that over 70 percent of people have imposter syndrome.

I recently saw a post on the Q&A site Quora where somebody asked "Would I get fired at Google (or another big tech firm if I got caught using StackOverflow as a reference?"

He got a bunch of really great answers from engineers working at Google, Amazon and silicon valley. Anybody who has worked as a software engineer at a tech company will tell you that not looking at references is frowned upon. In fact I challenge you find a single Google programmer who has not used Stack Overflow. (If you're not familiar, StackOverflow is a collaborative Q&A site for programmers).

A lot of new programmers are afraid that by checking references and asking people for help it will out them as a fraud who doesn't know programming. Nobody can hold all the relevant information in their head. For example, this is the name of an iOS method:

```
- (id)initWithBitmapDataPlanes:(unsigned char **)planes pixelsWide:(NSInteger)width
pixelsHigh:(NSInteger)height bitsPerSample:(NSInteger)bps
samplesPerPixel:(NSInteger)spp hasAlpha:(BOOL)alpha isPlanar:(BOOL)isPlanar
colorSpaceName:(NSString *)colorSpaceName
bitmapFormat:(NSBitmapFormat)bitmapFormat bytesPerRow:(NSInteger)rowBytes
bitsPerPixel:(NSInteger)pixelBits;
```

It's almost 400 characters!

In iOS programming, there's over 800 classes, 9000 methods and growing. No one will expect you to be able to remember all

of them. This is the precise reason why we are programmers, we can get the computer to do the boring stuff for us. For example, the code for recording sound is only a short search away, why would you need to memorise it?

The skill that most employers look for when recruiting is the ability to think. Knowledge is valued in a world where information is hard to come by. In the 1900s, only the well educated had access to good books and good teachers. Now, everyone has that at the tap of a mouse. Information is losing value, the ability to think is growing in value. So don’t be afraid to search, to ask or to find resources. The best programmers do it.

The skill is in asking good questions and understanding the answer. There is no point copy and pasting code from a StackOverflow answer if you have no clue how it works. Because StackOverflow works on a reputation system, it’s in their interest to be as clear as possible in their answer in order to be marked as correct and collect upvotes.

In most cases it doesn’t make sense to start searching StackOverflow whenever you get stuck. The first option should always be trying to figure it out yourself. So it doesn’t run, but

before I typed the last 3 lines the app ran ok, so what's in those 3 lines that broke my app?

If you really can't figure it out, start with Google. Search for your query or if you have a bug enter the error codes and the error message. Chances are that as a beginner, your programming woes will be very common and somebody might have even taken the time to write a clear and concise tutorial to help you understand your bug. As you grow more skilled in programming, the problems you'll encounter get more and more obscure, but hopefully if you followed the other 11 rules, you will also be a more capable programmer and figure it out yourself or know exactly where to get help.

The other reason why you should start with Google is because StackOverflow's search algorithm organises questions and answers by recency and not popularity. A lot of the problems you will encounter while starting out will have been asked and answered years ago but still massively popular.

So ask wisely and you will reap the benefits from the community. One day when you yourself become a code ninja, you'll be giving back to that same community and helping the next generation of programmers.

6. Be a Copycat

In the beginning of my coding journey, there were loads of books. I bought books on C++, C#, Java and loads more. You name it, I had it. But they didn’t do very much other than make me confused.

I read. I highlighted. I forgot. I fell asleep.

Books are good as references. If you want to dive deep into delegates and protocols, read the book chapter. But if you want to start learning, make something.

But what do you make?

Lacking in ideas? Be a copycat. Make your own notepad, make your own Paint, make your own piano. If you’re into games, make minesweeper, make tetris, make flappy bird. Not only will they be sort-of useful, they’ll be the perfect opportunity for

you to figure out how to do things and get experience in finding help. Something that is brand new to the world like holographic smartphone projections, no one will be able to help you with. By making copycat apps or programs you'll be treading in the path that many have walked before you. This way you maximise the chance that someone will be able to offer you help and advice when you get stuck.

7. Be accountable to someone. Show your work.

The biggest problem with online coding courses is the lack of accountability. No doubt there are loads of great Massive Open Online Courses (MOOCs), such as Coursera, Udacity, Udemy, Skillshare. But what are the consequences of not doing your homework or missing a month's worth of lectures? Nothing. Nobody cares.

Let's face it, internal motivation is not strong in any of us. We can always find a reason why we *deserve* to "Netflix and chill". I can't even count how many free online courses I've signed up to and subsequently not listened to a single lecture or completed a single piece of coursework.

You need accountability and commitment to learn. Think back to your university days, would have bothered to finish that

essay at 3AM if nothing depended on it? Would you have gone to any of the lectures if you didn't care about pass or fail?

This is why we try to introduce accountability in our own courses. We've realised that matching up students with a buddy helps. Somebody else who is a beginner, at the same level as you who sometimes helps you and other times needs your help. Sometimes as people's learning rates diverge or if you're paired up with a lazy bugger, we'll swap it up and give you a new buddy. Because this system is entirely voluntary, there a degree of self selection for people who work well in teams and are motivated by others. Just as you're more likely to go to the gym if you sign up with your partner, you're more likely to learn if you have a coding buddy.

So if you're not on our course then find your own. There's plenty of Facebook groups dedicated to those who are learning to code. There's an entire subreddit (r/learnprogramming) dedicated to this, I'm sure you'll find like-minded people somewhere online or offline.

The next thing I'm going to tell you will be controversial. We believe that people don't value things that don't have a value. This is reason why Coursera is taking down a large number of their free courses. They saw that millions of people were signing up for it but no one was taking any of the classes let

alone complete any of the projects. It was actually detrimental to students' learning to offer a free course. We all have a degree of hoarding tendencies and it's very easy to signup for a bunch of stuff that the future you can suffer through. There's always tomorrow, she says. By moving to a monthly subscription model, a lot of these online course providers saw increased engagement, increased coursework completion and generally people were taking advantage of the teaching materials. So if you are driven by external motivation, try to use a little bit of financial motivation to drive your learning. Think about how much is a life skill worth to you and trial a monthly commitment that doesn't tie you into a yearly contract. See if you're engaging with the course content more with or without the financial commitment. There are plenty of places where you can pay something affordable like $10 a month, the price of a few coffees, to motivate yourself to start a regular learning habit.

The final part of this rule is to try and find ways of getting assessed. Ok, so getting assessed is right up there with death and taxes in terms of how much people enjoy it. But when learning anything, it's always important to get feedback. You will get an objective assessment of your current skill level, instead of feeling like an imposter or brimming with false confidence. Coursera has a system where the students mark

each other's' work. We use Github education to test your code and look for bugs and problems with your code. Then we offer advice on things like structure, coding good practice and hints at how you can solve the bug. If you're on a coding course that doesn't have a system like these then it'll be worth your while to find a code mentor who can review your code and give you feedback. Only what's measured can be improved.

8. Keep Learning

Being a good programmer is a bit like being Madonna. In order to stay relevant, you have to keep re-inventing yourself. Don't run and buy your cone-shaped bras just yet. What I mean is programming will keep evolving.

There's always new trends, new technologies and new languages. Great programmers relish in learning new things, even if it means they become a beginner again.

The world will keep moving, if you stay in one place, you'll eventually be left behind. I know programmers who never learnt anything else apart from Fortran. I know objective-C programmers who can't persuade themselves to make the leap and learn Swift. Even though Apple is telling developers that very soon Objective-C will be phased out. We all know that Apple never makes threats that they carry out, just look at the optical drive (and soon the headphone jack?).

Don't be the optical drive. Or rather, don't be the laptop that's still trying to play CDs. If your needs change, learn to use a new tool. Keep learning, stay relevant.

Are you a web developer who always wanted to get into mobile development? Learn objective-orientated programming. Are you a Java programmer who is tempted by iOS development? Pick up Swift. If you already understand the core programming concepts, picking up a few more will be a lot easier than starting from scratch.

"Learn x in y minutes" is a great resource for existing programmers to learn new programming languages. Check out their resources here: learnxinyminutes.com

9. Play Foosball

When you see Hollywood movies about programmers, they're usually sat in front of a laptop, mashing the keyboard like some sort of high-stakes "smash the mole" game.

When you see **real** programmers working. They tend to look like this:

Yep, that's right. No typing. Just staring. A lot of staring.

In a company, people tend to complain that the programmers are always playing foosball or doing something else that

doesn’t look like work. People might not be able to tell, but they are in-fact working.

When you see them enjoying their foosball game, laughing and joking, they’re probably suffering inside. For there’s a bug, there’s always a bug. Or there’s something mysterious about their code that they can’t work out. Maybe it’s working, unexpectedly (programmers don’t like anything unexpected by the way).

Other people might not understand, but in these situations, it’s almost always worth stepping away from your code and give it sometime and distance.

Do you have a bug in your code that you can’t work out? Sleep on it, play foosball, go for a walk. In 9 out of 10 cases, the solution will become apparent. In the remaining 1 out of 10 cases, you’re simply mightily screwed.

This may sound unintuitive, but my advice is always to code less, think more. Once poorly thought-out code is written, you’ll inevitably have to go back and comb through your code, line-by-line, refactoring and deleting things. This is always a painful experience. So remember, the easiest code to get rid of is code that was never written.

10. Get a mentor - Try Pair Programming

When I was learning French, I came across a method that resulted in the greatest leap in my speaking abilities. That was having language exchanges over Skype. I would pair up with a native French speaker who wanted to learn English. We would spend half an hour speaking French and half and hour speaking English. We would both dedicate an hour each week to improving the language that we were trying to learn.

While we were having a conversation in French, he would correct my pronunciation, my grammar and suggest the ways that I could construct my sentences to sound more native.

Pair programming is an agile software development technique that’s based on very similar principles. For example, a learner and a mentor would sit down at the same workstation and work on a problem. The learner is in charge of writing code and the mentor reviews the code line-by-line as they are written.

It can be uncomfortable at first, because it’s a bit embarrassing making mistakes and having them pointed out. But if you have a mentor who is a good teacher then they will offer you decades of wisdom that can lead to massive improvements in your own ability within a few hours.

You get to tap into someone who’s had the time to hone their skills, find efficient ways of doing things and showing you how they program and approach problems.

Good mentors don’t solve your problems, rather they practice the Socratic method of asking good questions that get you thinking. If you ask me how to write a networking call, of course I can simply type it all out and get you to copy it. But that doesn’t help you. Instead, if you show me how you approach the problem and I show you how I approach the problem then you can learn so much more than just following a recipe.

The next time you encounter a different problem, you can apply the same approach and start solving it yourself. In this day and age, information is cheap. A century ago, if I wanted to learn about the causes of disease, I probably had to be an aristocrat, or chop wood and carry water for a master and

become their apprentice. Nowadays I can search Google and get my answer in a few seconds.

So don’t get hung up on information. Learn to think instead. How to approach a problem, how to break down the problem, how to frame the problem. These skills will take you much further than simple memorisation and regurgitation.

Where do you find a mentor? There are programming related Meetups happening in almost every city in the world. Go to www.meetup.com and find one related to a language you’re trying to learn. Attend the meetups, get to know people. Exchange your expertise for their expertise. Maybe someone needs an accountant, maybe someone needs legal advice. Exchange your time for their time. Don’t say to someone, “will you be my mentor?”. No one wants to throw away their free time for some stranger. Instead offer your help in return for their help and you’ll be successful in finding a mentor 95% of the time.

11. Get into the Habit of Chunking

So you have an awesome app idea. But it’s way way way too complicated for your current skill level. What do you do? You join the Chunking express.

Nope, we’re not talking about the art house movie. We’re talking about breaking down your programming problem.

Let’s say that you’re trying to make a robot that can butter toast. (If anyone is working on one of these I’d happily fund your Kickstarter!) The robot doesn't know anything about toast or butter or knives. Believe it or not, it actually takes pretty sophisticated circuitry in our brains to be able to achieve something as simple as buttering a slice of toast. (This is probably why I can’t seem to do it when I’ve just woken up).

So creating a robot that does all of that autonomously is really complicated and difficult. But as we're good programmers, we can do some chunking and break down the problem.

The robot doesn't really need to know what is toast and what is butter, we're not making skynet here, so let's just stick to the practical things. There's three things we need the robot to do:

1. Pick up and arrange the piece of toast in the ideal buttering position.
2. Pick up a serving of butter.
3. Place butter on toast with decent coverage.

Next, you break each module down even further. In the process, you can think about alternate ways of solving the problem. For example, does the robot need to "spread" the butter? Or can it just melt the butter on to the toast? Does it need to use a knife to pick up the butter? Or can it have some inbuilt knife-arm?

The more that you break down problems and define the issue that you're trying to solve, the easier it is to package your code into bite-sized modules. The simpler the module, the easier it is to tackle.

So the next time that you're trying to make that "cross between Snapchat and Evernote", remember to break down the problem into solvable chunks.

12. Break someone else's code

One of the most important steps to take in order to make the jump from learner coder to fully fledged programmer is understanding how to get help. Everyone needs help. Everyone, including those so-called "God Programmers".

But what you do with the help will determine how fast you progress as a coder. On a site like StackOverflow, it can be very tempting to just copy and paste the code that someone has provided. Your program works exactly as you hoped it would and off you go on your merry programming ways. This exercise didn't teach you anything other than code reliance. Because the next time you encounter the same problem but in

a different situation, that same code snippet that someone provided may not work anymore. Then what do you do? You're stuck.

That's why there's a rule in programming that says "never copy and paste code that you don't understand". So what should you do when you're confronted with a block of code that solves your problem but you have no clue how it works? Break it down.

Step 1 - Copy and paste the code into your program. (yes, yes, I know I just not to do that, patience, patience).

Step 2 - Make sure that your program or application is functioning as expected. I.e. confirm that block of code really did solve your problem.

Step 3 - Delete the copy and pasted block of code line by line.

Step 4 - Each time you delete a line, check to see what's been broken. Does the app still run? What are the error codes? What has deleting that line of code done to your app?

Step 5 - Even if you think you know what a line of code does, delete it anyways. The most important task as a programmer is to always test your assumptions against the outcome. For the

most enjoyable feeling as a programmer is for the real world to validate your assumptions.

Step 6 - Swap some of the lines around. Can the same functionality be achieved with a different order of lines? Why are they written in the order they are written in?

By breaking the solution code, line-by-line, you'll learn and understand what each line does and why it's been written. This is a far better way to use code from other people than just pasting it in and hoping for the best. Once you understand why each of those lines were necessary, the next time you encounter a similar problem, you'll be able to tease out the problem and solve it yourself.

Once you've mastered breaking code from StackOverflow, the next resource to target is GitHub. It's a tool used by programmers for collaboration but it is also one of the largest repositories of open source code.

So how can you use it to become a better programmer? Let's say that you want to make an instagram clone. But unfortunately, you don't know how to do that. So you head over to github.com and search "instagram" or "photo app".

Inevitably, there will be something written in Swift/Objective-C/Java that you can download and take a look at.

Think about the structure of their program. Take a look at all the classes, the constants, the interplay. Make some modifications to the code. Does it still work or have you broken it? Why did you break it? Is there a link that you didn't identify? Ask yourself a bunch of questions, learn through the Socratic method. Tear down the project and understand how it was built.

When you start getting really good at this, the next thing you can try is reverse engineering. Find a small project on GitHub made by a reputable programmer, download the app. Run it and see all of its functionality. Play around with it.

Then build it from scratch and once you're done, compare your code to their code. Are there efficiency gains that you could have made? Are there solutions to things you couldn't figure out? Now you're really getting into the big leagues.

Ok, so that's all folks. What are you waiting for? Learn to code and start developing iOS and Android apps today! Head over to

online.londonappbrewery.com for a free video course to start learning!

www.londonappbrewery.com