CODING FOR EVERYONE

A BEGINNER'S GUIDE INTO THE WORLD OF SOFTWARE DEVELOPMENT

FATOS MORINA

Coding for everyone

Fatos Morina

Contents

Introduction to Programming

What is programming?

Before we start with the advice, let's learn what computer programming is.

First of all, remember that a computer is not only a laptop or desktop. It's also your phone, tablet, and any other device that is capable of performing logical and arithmetic operations that we program into it.

This includes your car's navigation system, the ATM where you withdraw or deposit money, the systems that allow you to order food digitally, and so on.

Computer programming is the process of designing and building instructions (a "program") that the computer can execute.

As an example, we can take a calculator that performs arithmetic operations such as addition, subtraction, multiplication, and division.

Instead of doing all this with a pencil on paper – but instead, we have a calculator on our phones. And this calculator is a computer program that helps us find the result quickly.

Other programs that you may use every day include Facebook, WhatsApp, Instagram, YouTube, TikTok, and many others.

Writing a program is sort of like cooking a meal. You may have an idea for a delicious meal, but you need to buy the necessary ingredients beforehand.

Once you have what you need, you prepare that food, put it in the oven, and then wait while it cooks. After that, you serve the food at the table and eat it.

Things work similarly with programming: first, you have an idea and the knowledge to turn that idea into a program (or maybe you learn while you're building the program – just like how you can watch someone cooking food on YouTube and learn how to make that recipe).

Then, you start to put the ingredients of the program into place, piece by piece, until the entire program is ready to be served to users.

Unlike cooking, where dishes are consumed, programs remain intact and reusable even after being used.

This unique aspect means that a single program can serve multiple users simultaneously and over time.

There are costs associated with creating and maintaining software, but once developed, a program like a website can be accessed by numerous people at the same time without diminishing in availability or quality.

So to summarize, programming is the process of turning an idea into a computer program that other people can use. And this is something you can learn how to do yourself.

In the ever-evolving landscape of software engineering, many people are wondering about the impact AI will have.

Should you still learn how to code? (Short answer: yes.) Let's delve into this concern, especially for anyone who's skeptical about the future demand for coding skills.

Should You Still Learn to Code? [Answer: Yes]

We use software all the time – to chat with others, stay informed about the latest news, and even for our alarm clocks that help us wake up on time.

Even the browser or the PDF reader that you use to read documents are pieces of software. These words were even typed in a software program.

Programming is something that's ever-evolving, and many things that we're currently doing manually may get automated in the future.

But software is deeply embedded in our daily lives, powering tools and applications across various professions and personal activities.

The more we integrate into the digital world, the more central software becomes. It's reshaping our work, learning, and social interactions, with its importance only set to increase.

The Future of Building Software

Imagine a world where building software was as simple as having a conversation with a highly advanced AI, like a future ChatGPT. You describe your needs, and the AI creates the entire software for you – no coding required.

So you may wonder: why should I learn coding when ChatGPT can code an entire app for me in seconds? Well, here's something I want you to understand: AI is not here to replace us, but to **help us become more productive**.

Sure, AI can help you save time and energy by performing certain basic, repetitive tasks for you. But grasping the basics of software

development remains crucial, especially for the more complex problems that require your creative, experience-based solutions.

You need to be able to harness AI's power while continuing to learn and adapt. This will help make sure that we're steering the course in our rapidly evolving digital landscape.

As you can probably tell, I want to emphasize the irreplaceable role of software engineers and coders, even in an AI-dominated future. Here are a few reasons why:

1. **Understanding Over Automation**: Coding isn't just about executing tasks – it's about grasping complex logic and structures. Our human ability to analyze and solve intricate problems goes beyond AI's automation.
2. **Maintenance and Flexibility**: Deep coding knowledge provides insights vital for software maintenance and adaptation. This knowledge is invaluable when AI-generated programs encounter issues.
3. **Safety and Reliability**: Similar to understanding basic car mechanics for safety, knowing coding basics is essential for addressing software malfunctions.
4. **The Creative Element**: AI lacks human creativity and innovation, which is crucial for envisioning and realizing novel software solutions.

The rise of AI in coding doesn't signal the end for human coders. Rather, I believe it just signals a collaborative future. AI's automation of mundane tasks gives human coders more space for creativity and innovation in software development.

Learning to code transcends mere programming. It's about understanding how various technologies work, honing problem-solving skills, and nurturing creativity. These skills remain invaluable, even as AI reshapes the landscape.

So I encourage you to embrace coding as it opens doors to a future where human intelligence and AI collaboratively push the boundaries of innovation.

Why You Should Consider Coding as a Career

Maybe when you were a kid (or now as an adult) you liked a computer game so much that you played it all the time.

And maybe you had the idea that someday you could develop a game like it.

I know people who started programming precisely for this reason: to be able to develop different computer games.

So, because of the pleasure you may have experienced, you may have decided to become a programmer yourself.

Over time, your reasons may change – and this certainly doesn't mean that everyone gets into coding only because of game development.

After all, there are many reasons why you might consider programming as an option for your career. Let's discuss some of these reasons now (and indeed there are many more).

You have the opportunity to develop something for fun

I mentioned the case of game development, but you don't have to start programming by building games.

Maybe you are an accountant, or you have a shop and want to register your goods through a program that you develop yourself…and so on.

Here are some practical projects that might inspire someone to learn to code:

- A custom note-taking application tailored to your specific needs.
- A personalized inventory management system for your shop.
- An app for organizing and tracking your daily tasks and goals.
- A scheduler for managing your college or university timetable.
- A problem-solving assistant for tackling challenging tasks.
- A daily weather update tool that sends forecasts to your email.
- A health app that suggests potential illnesses based on symptoms.
- A focus enhancer that limits access to social media during work hours.

There are many other programs that you could develop. In short, you can start programming because you like to solve the problems you face in your daily life.

And once you learn to code and can build these programs, then you can share them with your friends and family (and beyond).

You can solve problems for people all over the world

Here's something that might make you happy: having the opportunity to help someone else (or a lot of people). I am sure that when you have been able to help a family member or a colleague, you've felt happy and satisfied.

Now, imagine how good you'd feel if you had the opportunity to solve a problem that people are facing all over the world.

For example, you may have an idea to start a Facebook group for residents of your neighborhood asking each other for household tools that you may need.

Or maybe you want to propose that everyone pitch in on maintaining the common spaces in your neighborhood.

You could also use this group to bring up problems as well as solutions that you aim to realize.

If you know how to code, you can create your own tool and share it with your neighbors. You can make your platform tailored to the specific needs of your neighborhood.

And if it works in your community, you can then share it with people from other neighborhoods.

You can take things a step further and work on projects that help people working in different fields.

So, for example, in addition to helping residents of your neighborhood, you can help farmers be better informed about the weather conditions by building a weather reporting app.

Or you can come up with a recommendation system that helps them determin how to work the land and what crops to plant when.

Or maybe you want to help a barber who wants to visualize the hair styles of his clients before cutting their hair. Or you could even build programs to help a private medical clinic better manage their patients' records.

As I hope you're beginning to see, the possibilities are pretty much endless.

You can work on interesting challenges

A major advantage of programming is that you can use the knowledge you acquired to solve all different kinds of problems.

Often you may start to read a book, or watch a movie that you may not like, and you have the opportunity to leave that book or that movie unfinished and start another one.

The same thing applies to programming projects. You may not be very motivated to work on a side project, so usually you have the opportunity to switch and deal with another project.

You can do that even at work from time to time where you ask can ask your manager to assign you to a different project.

You can also learn during work hours

Many companies offer programmers space and the opportunity to learn during work hours when there is no work that needs to be done at the moment.

In fact, some companies cover the expenses for training, books, and various courses that qualify and prepare workers.

And at some companies, when you complete a certification, you'll get a reward of some type, whether financial or otherwise.

Skilled, prepared, and trained programmers are valuable resources for a company. In addition to being able to work on more complex projects, they also often have the opportunity to get better clients and offers. The company can also invest in these highly-skilled devs by promoting them.

You can work from home

The good thing about programming is that you can work from home (or anywhere) and be employed remotely. This means you don't have to physically move somewhere to engage in more interesting work or get a better offer.

During the pandemic, many companies offered the opportunity to work from home, and many companies will continue to offer this opportunity even after the pandemic.

This is a great relief for many people, as it saves them from the need to travel every day to the office, pay additional rent, live away from family members, and so on.

You can have a flexible schedule

Different people have different obligations outside of work, including family needs, medical appointments, or various commitments at certain times, which cannot be postponed for later.

So for many, having a flexible schedule, or even getting to leave work at certain times, is necessary. And many developers are able to have this flexiblity in their work schedules.

You can collaborate with and learn from intelligent people

If you're on a team, there will likely be people who have studied many different fields/areas of tech than you have. They'll also likely have rich experiences with various previous projects. This will present to you many opportunities for learning and collaborating with these team members. So take advantage of those opportunities when you can.

You may have team members in Germany, Singapore, or Brazil, or anywhere else in the world. And by working on the same project, you have the opportunity to benefit from their knowledge, their approach to problems, and their creative solutions.

The salary is often very good

Programmers often enjoy higher salaries compared to many other professions.

This isn't just a slight difference – in most countries, if not all, the average salary for programmers significantly exceeds the national average.

This trend reflects the high demand and value placed on programming skills in the global job market.

In addition to the opportunity for good compensation, there is also the opportunity to get raises or other financial incentives/benefits depending on your work.

Also, many tech companies implement bonus schemes for their employees. These bonuses are given for achieving business goals, exceptional performance, or extremely dedicated work.

They can be awarded to individuals or teams, depending on the company's policies.

Thus, based on individual or team achievements, securing big clients, or completing successful projects, employees may receive additional compensation in the form of bonuses.

This can be a significant motivational factor for many in the programming field.

You can often take paid leave

As much as we may be motivated to work and enthusiastic about turning our ideas into code and thus reality, we are human and need rest and relaxation. So make sure to take the time to disconnect from daily work.

In other words, rest is beneficial for everyone, especially for programmers, who may be stressed and engaged all day with work and pressure.

In general, developers get fairly generous paid leave, allowing them to take vacations throughout the year. There are also companies that offer the opportunity for long periods of leave, as long as it does not harm the project and the work of the entire team.

Taking vacations isn't always easy for people of other professions, who often have much less flexibility or less generous paid time off.

Those are just some of the reasons programming may be a good fit career-wise for you.

But to be fair, let's also consider some of the downsides to see if any of them are dealbreakers for you.

Potential Disadvantages of a Programming Job

Perhaps you did not expect this issue to be addressed in this book which is aimed to inspire you to consider becoming a developer.

But I thought it was important to share potential downsides as well to help you make an informed decision.

Here, I'll discuss only some of the disadvantages that come from a career in programming. You might not experience all these specifically, and this may not reduce the chances of them happening to you. But if you're aware of them, perhaps you can avoid them more easily.

Programming can be stressful

Perhaps you have had the chance to speak with other developers and they told you that coding is stressful. Well, this can be the result of many situations.

For example, if you're fixated on solving a particular problem at work, you may hesitate to get up from your chair and walk around, which helps relieve stress. But you don't want to stop until you solve the problem. This can contribute to your stress.

So what can you do to help relieve stress, or release the emotional burden of an issue? Well first of all, it's good to do physical movement, including getting up from your chair, taking a little walk around the office (or your house, or wherever you're working), going out in the fresh air, and changing your environment for a few moments.

Remember: stress can be harmful both in keeping you from thinking clearly (which would help you more easily find a solution to the problem), and in consuming all your energy. All this can lead to a deterioration of your mental and emotional state.

You need to learn continuously

Learning is the only way to advancement. So if you always feel pressure to learn new things either directly or indirectly, this is a great opportunity you can use to advance your career.

New tech tools, programming languages, and platforms come out continuously. And you may be busy dealing with commitments outside of work that may also take your time and energy.

It doesn't matter how much experience you have so far in programming. The only thing that remains unchanged is the need to learn continuously. Learning new things helps you keep up with the job marke knt,ow what's required by your clients, and develop your own products.

But perhaps you don't enjoy the pressure to learn new skills all the time.

You might be someone who, as soon as you get into a job, want to feel comfortable and secure. This is ok, but it may result in having

lower ambitions and fewer or less exciting goals.

So if you're someone who doesn't always want to be learning new skills, this could be a negative side of tech and an unsuitable burden. But I hope that you, instead of seeing this as a negative side or disadvantage of programming, will embrace it as a worthy challenge.

You have a lot of responsibility

Your work as a developer may be used by many people. It may have a positive impact on them and make their lives easier.

But with this great power comes great responsibility. You may unintentionally cause negative consequences that may affect many people if you fail to catch an error in a program, for example.

It is not the same as making a mistake when designing a wristwatch, for example, that comes out with some problem like a scratched face or cracked leather band. If you release an application used by tens of thousands (or more!) of people – say, a medical device – that contains a mistake, it may risk people's lives.

This may seem a bit exaggerated, but it is worth remembering that many programs on hospital computers used by doctors and medical staff may have errors or bugs in them. As a result, this mistake could cause a doctor to make an incorrect diagnosis diagnosis and the wrong therapy which can be very harmful, even fatal to the patient.

So it's important to take your responsibilities as a developer seriously and complete your work carefully and thoughtfully.

You may need to work after hours

This is not unique only to programmers, since people who work in other jobs often need to work after work hours. But this can be

hard for devs, especially those with additional responsibilities in their lives.

As a developer, you may need to stay late at work for many reasons, such as:

- You have a short deadline for finishing a new task, new feature, or just for fully completing the project for your client.
- Some service that you have programmed has failed which needs an immediate fix so the client's service or tool doesn't go down. And you may need to be at your computer to fix it even though it might be after work hours, during a weekend, or during an official holiday.
- A security issue has arisen in a programming library that you use and you need to resolve it so that your clients and users are not affected by this problem. Mistakes are inevitable, especially in programming, but errors vary, and some can be very harmful, while some may not even be noticed.

Just make sure that you set boundaries and expectations with your team, your clients, and your manager so you don't end up working overtime all the time.

You have to sit for long periods in front of the computer

Developers often sit for long periods in a chair and often don't do enough physical activity. This can cause you to start experiencing back or joint pain, numbness, weight gain, or other potentially harmful health issues.

Or as you concentrate really hard on debugging a problem, you may have times when you even forget to close your eyes while working at your computer. This, of course, is not healthy, as it can damage your eyes.

Also, sitting such a short physical distance from your computer, which is usually less than a meter away, you may fixate your eyes in the same position for a long time, which can be harmful to your eyes as well.

To avoid these things, you can workout at a gym, do light exercises at home by following instructions on YouTube, and by generally moving your body more often like by getting up from your chair, walking around the office, eating healthy foods, stretching, and so on.

Your body mass may increase

Many programmers feel stress during work. As a result, they eat more than they need to reduce stress – and they may not even notice that they are overeating due to stress.

And again, since you may be spending long periods sitting and eating more than you need, you may start gaining some weight and losing muscle.

To avoid unwanted weight gain, try to be conscious of what you eat and how much. You can try to eat less unhealthy fat and reduce your sugar intake. You can also do some physical exercises, and make sure to eat little but often.

These are just some of the challenges programmers can face. But hopefully, now that you're aware of them, you can figure out how to avoid or deal with them.

Why Do Software Engineers Get Paid So Well?

One of the reasons we work is to make money to cover expenses and ensure that we can live a pleasant and dignified life.

Since we were kids, we've had to learn, complete our education, and gain experience so that we can more easily get a good job and make good money.

Since you are reading this, you have probably heard that software engineers are paid pretty well compared to many other professions.

The average salary of programmers – even those without formal education and relatively little work experience – is often much higher than that of people with jobs in other fields.

There are many reasons for this, but here I'll discuss what I consider to be some of the main ones.

High need for software engineers

In our world today, almost every company needs software engineers. This is because technology is a big part of our daily lives.

We use technology in our phones, cars, and at work. Companies in many areas like health, education, and fun activities need software engineers to make and look after their computer programs.

Because so many companies need these skills, there are a lot of jobs for software engineers. This makes companies offer big salaries to get the best software engineers to work for them.

Also, being a software engineer is not easy. It takes a lot of learning and practice.

Technology keeps changing, so software engineers have to keep learning new things. Not many people can do this well, so there are not too many software engineers.

This means that because so many companies want software engineers and there are not enough of them, they get paid a lot.

In simple words, software engineers make a lot of money because they are needed a lot and there are not enough of them who can do the job well.

You help turn ideas into reality

When you want to build a house, first you seek the help of an architect who comes to your land, analyzes the environment, and then, after some time, comes up with a detailed plan for every part of the house.

Then, this plan is taken by a building engineer who manages a team of workers and the entire construction process until the construction of the house is completed and the house is ready.

This is highly skilled work that requires a lot of training and expertise. It's the same with being a developer. You get paid relatively a lot because it reflects the high level of skill you have and the amount of effort and work it takes to do your job.

Software engineers are able to turn an idea that a client has into a computer program which then can bring benefits to people all over the world. This is a very valuable skill, and such it is rewarded accordingly.

Your solutions can connect people all over the world

When you think about a major highway, consider how it connects regions and transforms travel and commerce.

For instance, a major road might link two major cities, greatly increasing the flow of tourists and business travel between them.

Building such a highway requires a massive investment and a lot of effort, but once completed, it saves people significant time and opens up new opportunities.

This concept is similar to the creation of computer programs.

When a program is developed, it has the potential to connect millions of people worldwide who aren't in the same country or even on the same continent.

Take social media apps, for example. They're computer programs designed for global use, helping people like you, your family, and friends communicate, meet new people, or stay updated with the latest news.

Or consider Google's search engine, which processes over 3.5 billion queries daily on a variety of topics. It can help people anywhere, as long as they have an internet connection, solve their problems and learn new things.

These programs, much like a well-constructed highway, facilitate connections and interactions, making life more efficient and connected for countless individuals.

Software Engineers Work in Many Areas

A special thing about software engineers is that they can work in many different areas. They are not just stuck in one kind of job.

This is really important and one of the reasons they get paid a lot.

Imagine a software engineer making a program for a doctor's office to keep track of patients.

Then, maybe the next week, they might make a system for a hotel to book rooms, or help a school manage its information.

Software engineers can help lots of different businesses by making computer programs that solve problems and make things easier.

Different companies, like those in health, hotels, or schools, might not know much about technology. But they know it helps them do better.

So, they pay software engineers well to make good technology for them.

This is because software engineers can do a lot to change and improve how different businesses work.

It's more than just writing code; it's about making tools that really help businesses in many ways.

You are constantly improving your skills

Many developers constantly work to develop new tools to facilitate their daily work and make it easier or less time-consuming.

They write programs to do repetitive tasks for them, so they can then spend that saved time implementing logic and more complex solutions things for their projects.

So as a programmer, you'll always be improving your skills and getting faster at providing solutions to problems. You'll adopt the most powerful tools that help you perform your job faster and get more done.

A project that might have taken much longer, you can now finish in less time by using these new tools you have learned. This means that businesses who use your services as a programmer are able to go to market faster with useful products that serve customers better.

In other words, software engineers are paid so well because they have such specialized and valuable skills that allow them to help many people around the world.

How to Prepare for Your Coding Journey

Your Ego is Your Enemy

When you begin thinking about starting your journey into coding, it's easy to feel overwhelmed.

You might envision complex systems and software like those at Google or Amazon and wonder how you could ever understand such complexities.

Often, when we set out to become proficient coders, we find that our biggest obstacle isn't the code itself, but our own egos.

Here, by ego, I mean an unhealthy belief in our own importance: our arrogance and our self-centered ambition.

It's like a stubborn voice inside us that insists on being the best without considering the collaborative, challenging, and iterative nature of learning to code.

Ego can mislead us into thinking we're more knowledgeable than we are, hindering our ability to learn and collaborate effectively.

It stands in the way of truly mastering coding, building productive relationships with fellow learners, and recognizing or creating opportunities for growth.

Ego is a constant threat, not just in professional life but as we embark on learning something new like coding.

It can prevent us from building great software, advancing in our learning goals, and recovering from setbacks.

To guard against ego, we need to be aware of its influence at every stage of our learning journey.

> "If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle."
> — Sun Tzu

How to Overcome Ego in the Early Stages of Learning to Code

You might be starting with high enthusiasm, perhaps influenced by the success stories of self-taught coders or the allure of becoming a tech prodigy.

In such cases, ego might convince you that success will come easily, that you'll quickly master programming languages or skip the usual learning curves.

Remember, the most impressive achievements in coding come from dedication and hard work. Consider the builders behind WordPress, Microsoft, and Google, for example.

Their success stories are marked by relentless effort, not just innate talent or luck.

When starting your coding journey, don't fall into the trap of overestimating your abilities. Commit to the process, dedicate time to practice, and embrace the grind of learning.

Collaboration is key, as many groundbreaking projects are the result of team efforts.

Consider how major tech innovators worked together: Google's founders met at Stanford's computer science department, Microsoft

began with Bill Gates and Paul Allen working together, and WhatsApp was a collaborative creation.

Learning to code is similar – it's about joining a community, sharing knowledge, and growing together.

Ego and Ongoing Learning

Even after landing your first coding job or completing a few projects, don't let ego convince you that you've learned all there is to know.

The tech field is ever-evolving, and continuous learning is vital.

Remember, the journey in coding doesn't have an endpoint. It's a continual process of growth and adaptation.

Your ego might make you resist new challenges, shy away from learning new languages or frameworks, or avoid seeking help when stuck.

Yet, these are the very experiences that enrich your coding journey and lead to true mastery.

History is full of examples where initial failures or setbacks led to great success.

Bill Gates and Paul Allen had a failed venture before Microsoft, and many successful apps and platforms were born out of earlier unsuccessful attempts.

Embracing Humility and Persistence

To thrive in your coding journey, humility and persistence are your greatest allies. Accept that failures and challenges are part of the process. Use them as stepping stones, not roadblocks.

Be open to continuous learning, seek feedback, and remember that collaboration enhances your growth.

As you progress, remember that ego is the enemy of learning. Stay grounded, focus on the process, and cherish the journey of becoming a skilled coder.

Your unique contribution to the world of coding lies not in being the best from the start but in being open to growth and learning from every experience along the way.

> "The first principle is that you must not fool yourself — and you are the easiest person to fool."— Richard Feynman

In learning to code, let your curiosity and passion for the craft be your guide, not your ego.

Keep your focus on the work, the learning process, and the joy of coding.

Identify the whispers of ego early on and counter them with discipline, humility, and a commitment to lifelong learning.

Take Responsibility for Your Learning

Do not blame others for your lack of understanding or struggles when learning to code. Simply accept that you are the common denominator of all the problems and difficulties in your life.

Are you not understanding a particular lesson?

Do not blame the course, the book, or the instructor. Learning is your responsibility.

Every teacher tries their best, and we live in a time when you have plenty of options to choose and learn from.

So there is no excuse. If you work hard, you'll get there.

Have you recently found that a piece of code you wrote isn't functioning as expected in your project?

Even though this may be a rare occasion or something that you may not predict that could happen, you should still take responsibility.

First, admit that you made a mistake. Then, go and figure out how to fix the problem.

Use this as an opportunity to learn and document what went wrong and what you could do better in the future.

Do you feel that you aren't able to keep up the pace with learning new things?

I believe you already know that you are responsible for that as well.

It can be hard and very time consuming, especially if you have kids or other family to look after and also want to have a social life.

Still, your growth is your responsibility.

You have to own that.

You should try to carve out time a few times a week to improve your skills, whether that's during your day job when there are no tasks to work on and you are free to learn, before leaving for work, in the evenings, or whenever is realistic for you.

You could try to wake up one hour earlier before work and invest that period of time in learning new things, or you could also set aside a few hours on a Sunday morning to do that.

I'm Afraid to Start Programming. What Should I Do?

Starting your journey in programming can be daunting, and it's natural to feel fear or self-doubt. But remember, even the most successful figures in tech and science have faced these challenges.

Consider Sergey Brin, the co-founder of Google. He achieved remarkable success but has spoken about feeling like an impostor, experiencing doubts about his contributions to the tech industry. His story shows that such feelings are normal, even at high levels of achievement.

Sheryl Sandberg, former COO of Facebook and a tech industry leader, also discussed her struggles with impostor syndrome. Despite her significant accomplishments, she's talked about moments of self-doubt and questioning her abilities. This highlights that everyone, regardless of their success, can experience these feelings.

Astrophysicist Neil deGrasse Tyson has been candid about his own experiences with impostor syndrome, reflecting on moments of self-doubt in his science career. His openness helps demystify the fears associated with stepping into new, challenging fields.

Dr. Maya Angelou, an esteemed author and poet, also expressed similar feelings. Despite her numerous accolades, she felt doubts about her worthiness and achievements. Her journey is an inspiring example of overcoming self-doubt to achieve greatness.

Wayne Gretzky's famous quote, "You miss 100% of the shots you don't take," resonates deeply here. If fear holds you back, you'll never discover your potential in programming.

Starting in programming can be intimidating, and fears of failure are common. However, it's crucial to begin despite these fears. Remember, the tech experts and leaders you look up to started as beginners, just like you. They faced fears, made mistakes, and learned from them.

So, if your goal is to excel in programming or software engineering, embrace the challenge. Start learning and don't let fear or self-doubt deter you.

Everyone's journey starts with a single step, and feeling uncertain is simply a part of the process towards becoming skilled and confident in your abilities.

Strategies and Tips for Learning to Code

Can you Learn Coding in a Single Night?

A software bug in a Therac-25 radiation therapy machine caused the death[1] of five patients after receiving a massive dose of X-rays.

Knight Capital lost half a billion dollars[2] in half an hour when a software bug allowed computers to sell and buy millions of shares with no human oversight.

These and many other stories tell how seemingly unimportant bugs can actually cause disasters.

Software is becoming more and more important – which means that developers have more and more responsibility to be very careful and really good at what they do.

So is it possible to learn programming in just one night?

As funny as it sounds, there was a similar question asked on Quora more than 5 years ago.

Unfortunately, I cannot find that exact question anymore, but it stuck in my memory since then.

[1] https://www.newscientist.com/gallery/software-bugs/
[2] https://www.newscientist.com/gallery/software-bugs/

Maybe, the person who asked the question was trolling, or had an exam the next day and was hoping to get encouraging answers to pull an all-nighter and study before the exam.

I don't remember the other answers, but I read one answer there which was really wise and quite funny.

The answer was something along these lines:

> *Take a laptop and go to the north pole. A night there lasts 6 months. That's how you can increase your chances of learning programming in one night.*

We live in a time where we want everything in a matter of seconds.

We want fast food, fast cars, six-pack abs in 6 days, and so on.

That's the mindset that we have most of the time, and we expect the same thing in other areas as well.

But true mastery comes from a lot of work and dedication.

Take, for example, Peter Norvig, the director of research at Google, who suggests[3] that you to learn to program over the course of 10 years because rushing isn't going to be worth it:

> *"In 24 hours you won't have time to write several significant programs, and learn from your successes and failures with them. You won't have time to work with an experienced programmer and understand what it is like to live in a C++ environment. In short, you won't have time to learn much.*
>
> *So the book can only be talking about a superficial familiarity, not a deep understanding. As Alexander Pope said, a little learning is a dangerous thing."*

[3] https://norvig.com/21-days.html

If you mention that to people, they may not like it, but that's the actual reality.

You cannot just jump from printing a few "Hello World" statements in the console to building the next Google a few days later.

I am not denying the fact that you could implement a really nice application in a few hours – but the ability to do that doesn't come that fast.

Sure, you might have heard of the developer who implemented 10 applications in 10 hours in one sitting[4]. But that's not the result of one-night of learning. He worked hard and learned a lot before he managed to pull that off.

It takes time to learn something really well.

The more you do something, the more likely you are to get better at it. The more you practice, the more you'll be able to improve your performance – and the cycle repeats.

Now that you've read and hopefully internalized that, do not get too overwhelmed with the long journey ahead of you.

Will Smith describes it really well:

> "You don't set out to build a wall. You don't say 'I'm going to build the biggest, baddest, greatest wall that's ever been built.' You don't start there. You say, 'I'm going to lay this brick as perfectly as a brick can be laid.' You do that every single day. And soon you have a wall."

Now, instead of asking yourself whether you can learn programming in 1 night, a better question would be:

What is one small but valuable thing that I could learn tonight?

[4] https://dev.to/florinpop17/10-javascript-projects-in-10-hours-coding-challenge-316d

Avoid Tutorial Hell

Have you ever spent hours watching tutorial videos and then felt like an expert, only to realize you couldn't actually do what the tutorial showed?

This is a common trap, and it's called "Tutorial Hell." It refers to the false sense of learning you may get when you read through a bunch of tutorials without actually trying to build something yourself.

When you watch tutorials without practicing, you might think you're learning, but you're really not getting the full experience.

Just watching something passively is not the same as truly understanding and being able to use that knowledge.

Learn to Code by Coding

Learning to code is something that many people try to do just by watching videos or reading articles.

But if you really want to learn how to code, you have to get your hands dirty with the actual code.

It's not enough to sit back and watch someone else do it. **You need to write the code yourself**.

This is the only way to turn the theoretical knowledge from videos into practical skills.

Move Beyond Passive Learning

When diving into a specific framework or technology, it's easy to fall into the trap of passively watching or reading about it.

Instead, take an active approach.

Think of a project that you find interesting and would like to create. This could be anything from a simple website to a small app. The key is to choose something that excites you and motivates you to learn.

The project you choose will become your testing ground. It's where you'll apply everything you've learned from watching tutorials.

This hands-on practice is invaluable.

As you work on your project, you'll encounter real problems and challenges.

Solving these will deepen your understanding of the coding concepts and techniques you've learned about. You'll learn so much from googling, researching, trying and failing, and trying again.

It's in facing and overcoming these challenges that true learning happens.

Remember, the path to mastering any skill, especially coding, lies in active participation.

Watching tutorials is a great start, but it's the practice that makes you proficient. Learning by doing.

Your personal project is not just a test, but a journey towards deep understanding and skill development.

So, close the video player, open your code editor, and start building.

Your journey of true learning begins with the first line of code you write.

How to Understand Complex Topics by Using the Feynman Technique

In the world of software engineering, there's a vast difference between simply knowing the name of something and truly understanding how it works.

You might know what a certain machine or a piece of software is called, but do you really grasp how it operates and accomplishes tasks?

This distinction is crucial in a field as complex and ever-evolving as software engineering.

One effective way to bridge this gap between superficial knowledge and deep understanding is through the 'Teach-Back' Technique.

This method, often associated with the famous physicist Richard Feynman, revolves around the idea of teaching what you learn to someone else.

What is the 'Teach-Back' Technique?

Imagine you're explaining a new concept or tool you've learned in software engineering to a friend or a colleague who isn't familiar with it. This could be a programming concept, a coding language, or an advanced topic like blockchain or machine learning.

First, begin with an explanation: start by breaking down the topic into simple terms. Your goal is to make it understandable to someone with little to no background in the subject. This exercise forces you to clarify your thoughts and solidify your understanding.

Next, identify any gaps in your knowledge: as you explain, pay attention to moments where you stumble or feel uncertain. These are indicators of areas where your understanding is still shallow. Return to your study materials, whether they are books, articles, or online resources, and delve deeper into these topics.

Remember that simplicity is key: in your explanation, avoid jargon and complex language. The ability to simplify a concept without losing its essence is a sign of true understanding. This approach not only benefits your learner but also reinforces your grasp of the subject.

Sharing helps you solidify your knowledge: once you're comfortable with your explanation, consider sharing your newfound knowledge more broadly. Writing a blog post, creating notes, or even recording a video can further solidify your understanding and benefit others.

Reaping the benefits: this technique of teaching others not only deepens your knowledge but also enhances your effectiveness as a software engineer. It's a powerful tool for learning and a testament to the saying, "To teach is to learn twice."

Remember, the true test of understanding in software engineering is your ability to convey complex ideas in simple, accessible terms.

The more effectively you can explain a concept, the better you understand it.

Embrace the role of being a teacher, even if your student is just an imaginary one, and watch as your understanding and proficiency grow.

This approach is not just about gaining knowledge – it's about mastering it.

Don't Memorize the Syntax

Say you move to a new city and you don't know where the nearest grocery store is. You don't beat yourself up — this is expected. You just use Google Maps to find it.

After you've been there several times, it's highly likely that you won't need to use Google Maps anymore (unless you want to check traffic or see if there are any accidents on your route – things like that).

So you naturally, over time, learn where things are – but it's not that you're intentionally *trying* to memorize every road, every possible path to a certain destination.

There may be cases when you're only interested in going to a certain place once.

It's the same with certain methods that you may need to use in a programming language. New software developers may feel bad that they're not memorizing sufficient syntax — that they still need to refer to the documentation.

They see this as a sign that they're bad developers.

Here are a couple of common questions you may have when you're starting your journey into programming:

- How will I be able to learn all this syntax?
- How long will it take for me to master it and write code without referring to the documentation?

Then when you're able to memorize some frequently used syntax, you may think that you've become good at programing.

You get the impression that it's the amount of a language or the number of programming languages and frameworks you are able to memorize that really matters.

To put things in perspective, let's compare the syntax of finding the length of a string in different programming languages:

- `seq.length`, JavaScript lists, Java arrays.
- `seq.size()`, Java sequences, jQuery expressions (deprecated).
- `len(seq)`, Python.
- `seq.count()`, Django querysets.
- `SeqType'Length`, Ada arrays.
- `length seq`, Haskell.
- `(count seq)`, Clojure.
- `sizeof(seq)/sizeof(seq[0])`, C, statically allocated.
- `strlen(seq)`, C, null-terminated.

As you can see, each language or technology has its own version of finding the length of a string. This should show you that it's really difficult to memorize the same function in 12 different ways.

It's very unlikely that a developer who's used multiple programming languages can memorize all the syntax that they have used in the past. They may even not bother trying to learn it in the first place. So why is this? How does anyone actually write code?

Is Syntax So Important That I Have to Memorize It?

To answer this, let's see what some senior developers say.

Here's a comment[5] from a senior dev at Google:

"Hello, my name is Tim. I'm a lead at Google with over 30 years coding experience and I need to look up how to get length of a python string."

And here's one from a developer who works on commercial airline control systems:

"Hello, my name is JP. I put Node.js in a commercial flight airplane. I still don't know the difference between `.substr()` and `.substring()`"

And JP went on to further comment on that previous tweet:

"@tdierks even for this post I had to google how to write substring with or without camelcase"

Jon, a lead Android dev at Phunware, says he can't read an input stream without copying and pasting code from Stack Overflow:

Another Tim commented on the original Tim's post:

"Hi, mine is Tim too. I've been coding since 1979 and I still have to look up `java.lang.String` methods all the time. [lovely chain]"

[5] https://twitter.com/tdierks/status/835912924329836545

Lastly, I'll leave you with Umer's comment:

> I wrote 255 lines of code that included a working server and a client. I queried google 23 times mostly landing on StackOverflow, Netty 4 website, GitHub, and JavaDocs. If you do the math, that averages out to 1 query every 10 lines of code! I had no idea. — Umer Mansoor[6]

These insights should give you courage — you don't have to be ashamed that you can't remember every detail of the syntax.

This is something that many junior developers may be concerned about. The truth is, you really do not have to memorize everything as you go.

Not even tech recruiters care about that. Here's the response that an engineering director at Google gave regarding this:

"I always tell candidates that I don't care about anything that an IDE could help you with."

Here's another quote from a comment[7] on Hacker News:

> I not only use Google frequently, I use it to search for things I myself have written in the past.
>
> I can't count the number of times I've Googled for a programming question where the answer is found on a Stack Overflow page I've written. If it's a particularly old answer that I've completely forgotten, I've even thought to myself, "Wow this genius sounds just like me!" as well as "This idiot has no clue what he's talking about!" .

[6] https://tiantiankan.me/a/5cb4c6c2f4b9a5fab76893b2
[7] https://news.ycombinator.com/item?id=11603078

A couple of years ago, I read a post on a Facebook group from a senior developer mentioning that when recruiters ask him whether he has experience with a particular technology for which he hasn't, he responds, "That's just another tool."

He means that he may not have had the chance to work with it in the past but he has the confidence that he can learn it.

You might not need to spend months before you're able to work with a new tool. You may only need a few hours to read its documentation and then learn more as needed along the journey without the need to memorize everything.

You do not expect to have all green lights on when you start your journey to your destination. You pass through the current green light and stop at a red light. You wait until the green light is on before proceeding.

Nowadays, we get new languages and frameworks, or considerable changes to existing ones, so that trying to memorize the syntax is both difficult and not that important. As one commentator on a blog post brilliantly put[8] it:

> "Great engineers know how to formulate good queries.
> Yet interviewers expect walking dictionaries."

So, to sum it all up: get really good at googling[9] and learning as you go :)

Keep a To-Learn List

You might have heard of to-do lists for keeping track of daily chores or tasks.

[8] http://disq.us/p/17uk3bk
[9] https://www.freecodecamp.org/news/how-to-search-google-like-a-pro/

But as a software engineer, where learning is as important as doing, a "to-learn" list becomes your roadmap to growth and discovery.

Think of a "to-learn" list as your personal guide in the world of technology.

Just as a to-do list reminds you to buy groceries or pay bills, a to-learn list keeps track of all the exciting things in technology you want to explore.

This could be a new programming language that's taking the world by storm, a revolutionary tool that can change the way you code, or the latest trends in fields like artificial intelligence.

Your to-learn list is more than just a collection of cool things – it's a set of goals.

It's like having a bucket list for your professional development, full of adventures waiting to be embarked upon.

Whether it's mastering a new coding language, diving into insightful books, or enrolling in online courses, this list is your ticket to a world of endless learning.

A to-learn list keeps your curiosity alive. It's like having a treasure map where X marks the spot for new knowledge.

Each item on the list is a stepping stone to understanding the deeper workings of technology, beyond just using it.

This list also acts as a motivator. Imagine each new skill you learn as a level-up in a game, making you more capable and skilled in your job.

It's a reminder that your journey in software engineering is filled with exciting milestones waiting to be achieved.

Remember, there's always more to learn, and that's a good thing.

It keeps you humble and open-minded, acknowledging that the tech world is vast and ever-evolving, and no one knows it all.

This perspective is vital for continuous growth.

As you check off items on your to-learn list, you'll find yourself becoming a problem-solving wizard.

Each new skill or piece of knowledge adds to your ability to tackle complex challenges, making you an invaluable asset to your team.

The more you learn, the more valuable you become as a software engineer. This means being able to handle tasks more efficiently and creatively, boosting your productivity and making you a star performer in your field.

Your "to-learn" list is a powerful tool in your journey as a software engineer.

It's a living document that grows and changes with you, reflecting your aspirations and curiosity.

By maintaining and updating this list, you ensure that your journey in software engineering is always moving forward, filled with continuous learning and personal growth.

So, embrace your "to-learn" list, and let it guide you to new heights in your career, keeping you curious, motivated, and ever-evolving in the dynamic world of technology.

Practical A*spects of* C*oding*

How to Deal with Overwhelming Projects

Big software projects, like those of Google, Amazon, or WhatsApp, often seem like modern marvels.

They can be awe-inspiring, reflecting complex engineering and innovative ideas.

If you ever consider the teams behind these giants, it's natural to feel a mix of awe and intimidation. The scale and impact of such projects can seem overwhelming.

Remember, the builders of these platforms likely faced similar feelings at the outset. But they didn't allow apprehension to hinder their progress.

Instead, they channeled it into productive action, laying brick by brick the foundation of what would become technological milestones.

As a software engineer, you might not be tasked with building the next global tech phenomenon right away, but every project, no matter its size, can feel daunting at the start.

Here's a roadmap to help you tackle even the most intimidating software engineering projects.

Vision: Seeing the End from the Start

One of the most critical steps in managing a large project is to have a clear vision. This means having a well-defined picture of the end goal. The clearer this vision, the easier it will be to navigate towards it.

Try to articulate this vision in simple terms, maybe through a diagram or a straightforward description.

Your goal should be to make this vision so understandable that anyone, regardless of their technical background, can grasp it.

For instance, if you're developing a complex web application, outline its core functionalities. Envision the user experience, the interface, and the key features that will define your application.

Breaking It Down: The Art of Reverse Engineering

With your end goal vividly pictured, the next step is reverse engineering this vision into actionable steps.

This process involves deconstructing your final goal into smaller, manageable tasks. These tasks should be quantifiable, allowing you to plan and schedule them effectively.

In plotting your course, set clear milestones. These are significant checkpoints that signal progress towards your ultimate objective.

Take a web application as an example: begin by charting out the various stages of development. This could include initial design, front-end development, back-end setup, integration of functionalities, testing phases, and final deployment. Then you can tackle each one, one at a time.

Starting Point: The First Small Step

Every monumental project begins with a single, often small, step.

Do not fall into the trap of believing you can conquer the entire project in one go. Large-scale projects demand time, effort, and persistence.

Start with a manageable task or component of the project. Completing this initial task will provide a sense of accomplishment and momentum.

Progress, however incremental, is a powerful motivator. It propels you forward, fueling your drive to tackle the subsequent phases of your project.

If you find yourself hesitating to start, it might be because the task at hand seems too daunting. In such cases, break down the task further into smaller, more approachable segments.

Reflective Progress: The Importance of Review

Regularly step back and review your progress.

This practice is essential for ensuring that you're on the right path toward your goal.

It's easy to get caught up in the minutiae and lose sight of the broader objective. A periodic review helps you realign your efforts with your initial vision.

This review process is not merely a check-in. It's a strategic evaluation of your direction and methods. It might lead to course corrections, refinements of strategies, or even a reassessment of your goals.

The Path Through

It's common in software engineering to encounter phases of doubt and to feel stuck in a cycle of inaction.

Waiting for the perfect moment or for challenges to ease on their own is a futile approach. Instead, the key to conquering large projects lies in embracing responsibility and actively engaging with the tasks at hand.

No matter the size of the project, its completion is always achievable through careful planning, dedication, and unwavering focus.

Remember, in the realm of software engineering, *the only way out is the way through.*

Each step, no matter how small, is a move towards the realization of your project. Embrace each phase of the journey with commitment and resolve, and watch as your ambitious projects turn into accomplished realities.

How to Choose Projects to Build

As you start learning and trying to apply your skills, you'll likely have a lot of project ideas that you're eager to start building.

Maybe you've listed them in a project management tool or jotted them down in your notebook. You have various app concepts, software solutions, or coding experiments you're excited about. But there's a common challenge: deciding which project to tackle first.

This indecision can lead to procrastination. You might find yourself opting for activities that are more immediately gratifying, avoiding the commitment to a single project.

"Perhaps there's something more crucial I should be working on?" you wonder. "Should I start this app or that automation script?

Maybe I should brainstorm more ideas and something will click."

Let's pause for a moment and consider an old fable that might shed some light on this predicament.

Learning from Buridan's Donkey

There's a story about a donkey that was exactly in the middle of some hay and water. It couldn't decide whether to eat or drink first, so it didn't do either and got really hungry and thirsty.

The donkey didn't think about just picking one thing first and then doing the other thing next.

As software engineers, we often face a similar dilemma with our projects. We have numerous ideas and concepts, but none of them will come to fruition without action.

It's essential to recognize that we can't work on every project simultaneously. Just as we can't be in two places at once, we can't simultaneously code different applications. And this physical limitation shouldn't hinder our progress towards our goals.

Just start with one project

Is it the perfect choice? Maybe, maybe not. But that's not what's critical at this moment.

The project you choose to work on probably isn't a life-or-death decision. Pick one and begin. You can always return to your other ideas later.

Start by outlining the features of the app or software. Code the first function or interface. Celebrate this progress, however small. Then, continue developing piece by piece.

Repeat this process for your other projects.

Don't let a plethora of choices impede your progress. Over-analysis leads to paralysis, stifling your movement towards completing your projects.

Stop overthinking. Start coding.

Let the world benefit from your work, one completed project at a time.

How to overcome procrastination

This is a common challenge for software engineers: "I know I need to start coding, but I just can't seem to begin."

"The blank IDE screen is intimidating. I can't get my fingers to start typing."

Consider how Ernest Hemingway, a renowned writer, dealt with writer's block:

> "Sometimes when I was starting a new story and could not get it going, I would sit in front of the fire and squeeze the peel of the little oranges into the edge of the flame and watch the sputter of blue that they made.
>
> I would stand and look out over the roofs of Paris and think, 'Do not worry. You have always written before and you will write now. All you have to do is write one true sentence. Write the truest sentence that you know.'
>
> So finally I would write one true sentence, and then go on from there."

Adapt this approach to your coding. Start with a simple feature or function.

If that's all you accomplish for the day, so be it. Consistent small steps lead to significant progress.

Even a modest amount of coding each day maintains consistency. And often, once you start on a small task, you'll find yourself naturally progressing to more substantial parts of the project.

Inspiration from History

Consider Isaac Newton, who, during the bubonic plague lockdown in 1666, retreated to his family home from the University of Cambridge.

Without modern tools or online classes, Newton used this time to delve into complex mathematical problems, eventually laying the groundwork for calculus.

His focus and dedication led to groundbreaking discoveries.

In software engineering, it's easy to get caught up in finding the perfect tool or environment.

But remember, the real work comes from dedication and effort, not from having the latest technology or software.

> "Real professionals don't hide behind their tools — real professionals use what's around them. It's the amateurs who try to nerd out about their tools — it's all a distraction from really doing the work." — Derek Sivers

Let's move beyond excuses. Let's get back to coding.

Learn by Building Things from Scratch

Every time you start a new project, think of it as adding a brick to your knowledge fortress.

You'll encounter the essential elements of software – like algorithms, data structures, and design patterns. These are the ABCs of software, and grasping them deeply turns you into a more skilled and versatile engineer.

When you build from scratch, you're the architect and the builder. You get to decide every detail, customizing your project to fit your interests and needs.

It's like cooking a meal where you choose every ingredient to suit your taste, making the entire process deeply personal and satisfying.

Starting from zero means you'll walk through every stage of creating software – from the first sketch of your idea to making it work and showing it to the world.

This complete journey is like reading a book from cover to cover, giving you a full picture of how software comes to life.

Building from the ground up often means exploring the depths of the technologies you're using. Imagine diving into the ocean to discover what's underneath – that's what you do with each technology, understanding its secrets and subtleties.

As you build, you'll face various puzzles and challenges. Each problem you solve sharpens your mind, not just in coding but in finding smart, elegant solutions to tricky situations.

There's a special kind of pride and confidence that comes from seeing a project evolve from a simple idea to a working program.

It's like climbing a mountain – with every step, you feel stronger and more capable.

The hurdles you overcome while crafting software from scratch are similar to those you'll face in the professional world.

Be it fixing an unexpected error, adding a complex feature, or managing your time – these experiences are like rehearsals for the real-world stage of software engineering.

Embarking on a project from scratch is a cornerstone of your journey as a software engineer. It's an exercise in patience, persistence, and creativity.

Remember, it's not just about the software you build. It's about the skills you hone, the knowledge you acquire, and the confidence you develop.

Each project is a step towards becoming not just a coder, but a true artisan in the realm of technology. So, begin with an empty file and a spark of an idea, and let the magic of creation unfold.

Build Something that Already Exists

In the thrilling journey of software development, starting a new project often feels like opening a treasure chest filled with endless possibilities.

While it's tempting to dive into creating something unique and complex, the path of learning often calls for a simpler, more familiar approach.

Now, we'll explore the idea that not every project must be groundbreaking. Sometimes, the best learning comes from simplicity and familiarity.

Imagine starting with a blank canvas. Instead of painting a complex masterpiece right away, you first practice with simple strokes and familiar patterns. This is the essence of learning in software development.

By keeping projects simple and manageable, you avoid the trap of overwhelming complexity, allowing you to focus on the fundamentals of coding and problem-solving.

Think of existing open-source projects as a library filled with books you can learn from. These projects, created by others and shared on platforms like GitHub, are like guides offering valuable lessons.

They're a starting point, showing you how things are built and encouraging you to explore further.

Picture yourself as an apprentice painter, learning by replicating the works of masters. Similarly, in software development, cloning an existing project[1] is a hands-on way to learn coding and problem-solving[2].

After recreating the project, compare your work with the original. This comparison acts as a mirror, reflecting your strengths and areas for improvement.

Think about an app or a website you use daily. What if you tried to build a version of it yourself? This approach makes learning fun and relevant.

Working on something you're familiar with keeps you engaged and helps you understand the project's intricacies more deeply.

In your learning journey, focus on projects that challenge and enhance your skills, rather than trying to invent something entirely new. It's like practicing scales in music – they may not be glamorous, but they're essential for growth.

Prioritize learning and skill development over the pursuit of a groundbreaking idea. As you embark on new software projects, remember that your primary goal is to grow as a developer. It's not just about building something novel – it's about building your capabilities and confidence.

By replicating existing projects and focusing on the learning process, you turn each project into a valuable step in your journey.

This approach not only enriches your understanding but also lays a strong foundation for your future as a skilled software engineer.

So, grab your tools and start building – not just software, but a path to mastery in the art of development.

[1] https://www.freecodecamp.org/news/javascript-game-tutorial-stick-hero-with-html-canvas/

[2] https://www.freecodecamp.org/news/learn-how-to-create-an-instagram-clone-using-react/

Beyond the Basics

Develop a Love Learning

Software development is one of the most in-demand professions of our time.

There are constantly new job openings, which attract both youngsters and already employed people from a wide range of professions.

They know that you can get a really good salary working as a software engineer, and so they start their careers with great ambitions.

Even if you get a job in software development, if you really want to become great, it helps to love learning and programming.

It is one of the few professions in which you constantly have to learn something new. It's not just the existing concepts you need to know. You have to adapt to the constant influx of new technologies.

You have to adapt to the changes in the market and learn whatever new things are currently considered as valuable. In short, you will have to consistently learn a lot.

In order to keep your brain constantly engaged in learning, you will have to be curious and stay humble.

Often, programmers get employed on the basis of their existing knowledge and start to think they don't need to learn new things anymore. But you will only be able to excel as a software engineer if you are curious and love learning, as the need to learn never ceases.

In November 1915, after writing only two pages of what he referred to as "one of the most beautiful works of my life," Einstein sent his 11-year-old son Hans Albert a letter.

He praised his son for his learning efforts, saying, "That is the way to learn the most…When you are doing something with such enjoyment that you don't notice the time passes."

"Curiosity has its own reason for existing," Einstein explains. "One cannot help but be in awe when one contemplates the mysteries of eternity, of life, of the marvelous structure of reality."

Become curious about how a framework works and is structured. When you learn something thoroughly, you gain a clear picture of its mechanisms and functionalities.

For example, don't simply accept that scikit-learn is a great framework for machine learning — learn how it works, behind the curtains. And consider the possibility of contributing to and improving it, as it is open source.

Also, try not to be overly concerned with money. Of course, we all need money to pay our bills and buy food for ourselves and our family. These are basic needs, and we need to meet them – so money is critical. But try to cultivate a passion for learning and developing for the sake of it. This will make the financial compensation all the more rewarding.

Try to develop a love of programming and feel grateful that you have the opportunity to positively impact the lives of millions of people with the lines of code that you write.

Einstein believed that "love is a better teacher than a sense of duty."

What we can learn from this as software engineers is the importance of not just working for a good salary but working because we love learning and our work in general. If we can develop an intrinsic curiosity and drive that keeps us engaged after work or during the weekends when our boss is not watching, all the better.

Look at the Bigger Picture

In the fast-paced and exciting world of software engineering, it's crucial to understand that being a software engineer is about much more than just writing code.

Imagine you're not just a builder, but also an architect and a visionary. Your role is like being a key player in a big team, where your ideas and work help the team win.

Understand your impact

When you're a software engineer, you might spend a lot of time working on specific coding tasks – like solving a puzzle piece by piece.

But, it's important to remember that you're actually helping to complete a much larger puzzle.

Let's say your company is like a big ship, and you're not just fixing parts of the ship – you're helping it reach its destination.

Imagine you're making an app for a local bakery. Instead of just focusing on making the buttons work, you should also think about how your app will help the bakery get more customers and make their life easier.

Seeing the whole, not just the parts

As you grow in your career, you'll start to see not just the lines of code you write, but the whole project, like looking at a whole forest instead of just one tree.

This means you'll start to understand how your work affects everything else and can make decisions that help the entire business.

Think of a puzzle game on your phone. If you only focus on one small part, you might miss a better solution that solves more puzzles at once.

From coding to contributing

When you start focusing on the most important tasks, you become more than just someone who writes code. You become someone who solves big problems.

This is like moving from being a player in a game to being the one who helps plan the strategy of the game.

Imagine your company is like a garden. At first, you might be planting individual flowers (writing code). But later, you start to plan where to plant trees and how to make the garden more beautiful (strategic thinking).

More than a coder

By aligning your work with what the business really needs, you become a key player on your team. Your work starts to have a big impact, and you become someone everyone relies on.

It's like being a star player on a sports team who not only scores goals but also helps the team win championships.

If you're working on a website, don't just make it look good. Think about how it functions, how it can attract more visitors, and how it can help the business grow.

Being a software engineer is an amazing journey. It's not just about writing code, but about using your skills to help the business and its customers.

When you start to see your role in this bigger way, you grow not just as a programmer, but as a key member of your team.

This approach will help you develop both personally and professionally in the exciting world of software engineering.

Remember, you're not just building software – you're building success stories.

Embrace Lifelong Learning in Software Engineering

Think of the field of software engineering not as a one-time race but as an ongoing marathon with new and exciting paths to explore at every turn.

Your formal education, whether you got a college degree or participated in a bootcamp or taught yourself to code – it's just the starting line.

What lies ahead is a thrilling adventure of continuous learning and growth.

Picture yourself as a detective in the world of technology. Each new challenge is a mystery waiting to be solved.

At first, these puzzles may seem daunting, but as you learn more, you'll find joy in piecing together the solutions. This journey enhances your confidence and turns complex problems into exciting quests.

In the fast-paced tech landscape, what's hot today might be forgotten tomorrow. So while it's important to stay up-to-date and aware of the latest trends, don't forget to work on the fundamentals and learn the mainstay technologies.

So while you'll want to learn new programming languages and frameworks, and get comfortable with the latest software, be practical about what you focus on.

As you learn and grow, your value as a software engineer will skyrocket.

This isn't just about earning a bigger paycheck – it's about getting your hands on more exciting and challenging projects.

Just like a sought-after artist, your up-to-date skills and knowledge can help make you a hot commodity in the tech world.

Imagine each new skill you learn as a key. The more keys you have, the more doors you can open. This could lead to better job offers, working on groundbreaking projects, or even guiding a team.

Your knowledge and experience are like a map, guiding you to a rewarding and dynamic career.

Good Luck!

Thank you for reading this handbook! I hope you now feel better equipped to break into the world of tech, grow your skills, and work towards your first – or next – developer job.