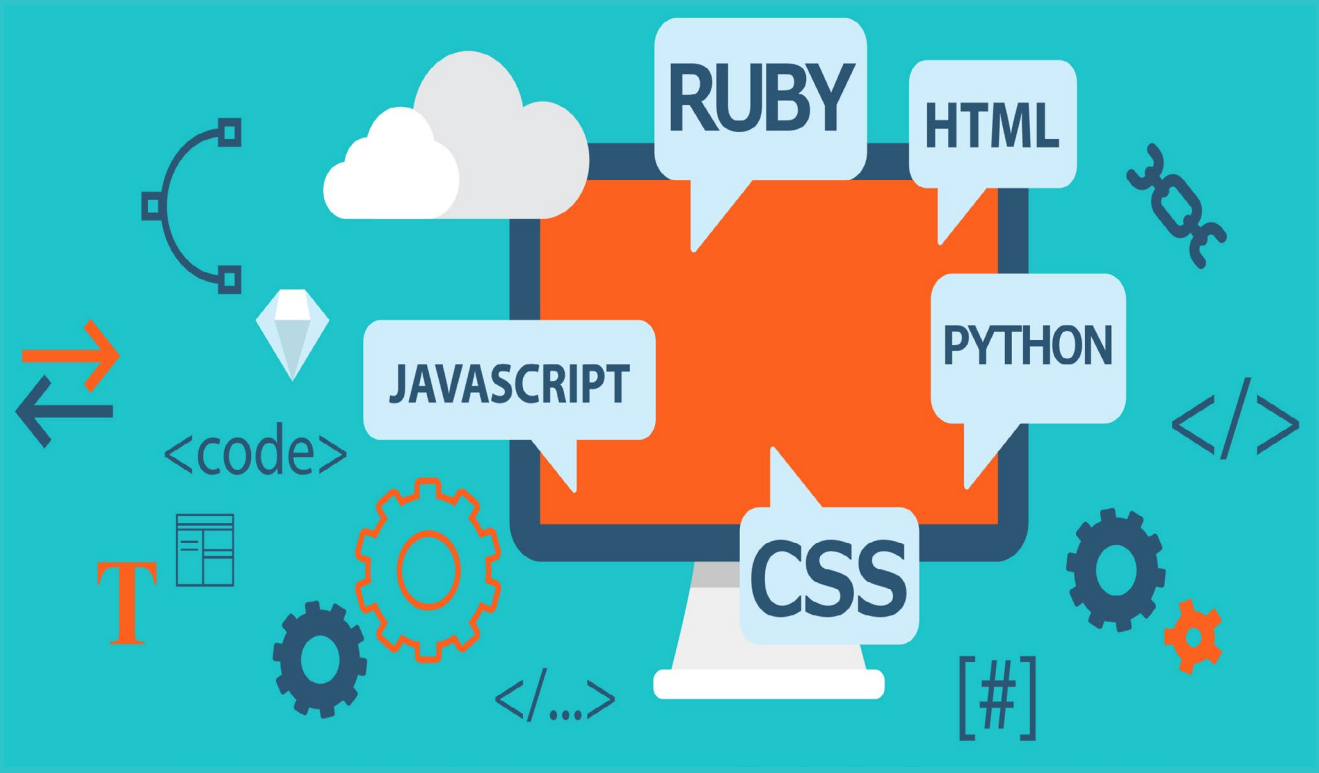
SO YOU WANT TO GO TO CODE SCHOOL



A Guide to Choosing a Code School and Succeeding When You Get There

By Katie Leonard

SO YOU WANT TO GO TO CODE SCHOOL

A Guide to Choosing a Code School and Succeeding When You Get There

By Katie Leonard

Table of Contents

Introduction

When I was 35, I decided to uproot my life, move to the other side of the continent and go to code school. It was the best decision I ever made.

My goal in writing this book is not to convince you to go to code school -- while it was a great decision for me, it isn't right for everyone. My goal is to help you figure out if code school is right for you and, if it is, to provide you with unbiased information as well as some of the tools you need to succeed.

To get a well-rounded perspective, I interviewed several other code school graduates about their experiences, some of which were similar to mine, some quite different. I also talked to some hiring managers who actively hire code school graduates to find out what they look for and what they expect. The information in this book is the combined wisdom from all of us.

Who should read this book?

If you are considering code school, this book is for you. It contains much of what I've learned on my non-traditional path to becoming a software developer, and answers many of the questions I had when I was considering code school. I hope it will be helpful to you. :-)

Why I went to code school

When I was 18 and considering options for my future career, technology did not make the list. I studied biology in college with a vague idea of becoming a doctor, and ended up in research. I really loved research because there were mysteries to solve and knowledge to gain. But with only a bachelor's degree, I found my prospects for advancement were limited.

Also, I wanted to do something different. I was working with some very talented software developers, and they inspired me to learn how to write small scripts to perform the repetitive tasks that occasionally cropped up in my work. I found that spending a few hours writing a script that would save me days of tedious labor was remarkably satisfying, so I decided to go back to school to learn how to write code.

My journey into programming started with a software development certificate through the continuing education arm of a local university. For two years I learned the basics of many technologies: network design, database design, operating system administration, project management, systems design, and basic Java programming and data structures. The courses covered a lot of ground, but were light on practical applications.

With my certificate in hand, I went on a few disastrous job interviews. Not only did I not understand the industry or the types of jobs I was qualified to perform, but I also bombed the technical interviews. Despite all I had learned, I was clearly unprepared for the realities of working in software.

In the meantime, I remained an avid programming hobbyist. I took some courses through Coursera and learned to build Android applications. But I

began to despair that I would never break into the industry full-time. That was when I learned about code school.

I was living in Montreal, Quebec, and a friend forwarded me an advertisement for a coding bootcamp in New York. They were offering $5K scholarships for women to attend during the summer of 2012. I made the short list for interviews, but the class was filled before I could make my pitch.

To be honest, I had done zero financial planning. While researching code schools, I realized that desire alone wasn't enough -- I needed money as well. So, I spent a year getting my financial house in order. Then, in the spring of 2013 I worked to find a summer program teaching Ruby on Rails. There were only a few to choose from, and they were all located in other cities. Moving would definitely be required.

I applied to another school in New York and asked if there were any scholarships available. The school was very new, and the most they could offer would be a $2K deduction from the $10K tuition. Also, the balance was due before the class started. Today that seems like an awesome deal, but at the time there was no way I could pay $8K upfront or live in New York without a salary for three months. So I kept looking.

I found one in Portland, Oregon. It was relatively new, but had enough funding to handle tuition deferral until after the program. Portland was a burgeoning tech center, and the founder/instructor was well-respected and active in the community. There were many great reasons to choose that school, but in the end, my decision was financial. I wanted to spend three months learning to code, but I had to eat and live in the meantime. I had family in Portland who were willing to put me up for the summer, and I could defer paying the tuition until I was (hopefully) employed on the other end. Fortunately, I got in.

Portland is a long way from Montreal. My partner and I packed our life into a storage unit and drove across the country, arriving on the day of my orientation. I literally unpacked the car and then went to meet everybody. Ready or not, I was about to get what I'd wished for.

My code school experience

In many ways, code school was exactly what I expected -- there were lectures, exercises, and projects, and we worked extremely hard to learn difficult concepts. When we were not in lectures, we were usually working in pairs.

Pair programming is a practice where you share a single screen, keyboard, and computer with another programmer. One of you controls the keyboard and writes the implementation, and the other does the "driving," which means you think about how to solve the problem. Pairing is a challenge for beginners, because you need to verbalize what you are thinking without knowing the right vocabulary. You are learning to converse in a foreign language without fully knowing the grammar, while solving logic problems.

The first day of class was stressful: navigating a new city, meeting the people with whom I would share the next three months of my life, comparing our abilities. But I was also excited to see how much I could learn in a short period of time.

My previous programming experience definitely helped in the first month, when we focused on programming essentials in Ruby and JavaScript. The exercises were designed to get us thinking like programmers by building functions that could perform simple tasks. Our day began at 9 am with a review of the homework from the night before, then a short lecture on a new topic. We broke up into pairs to spend an hour applying the new concept, and then came back together

as a class to discuss what we'd learned. After lunch we had time to revisit the exercise from the morning or to work on a project or homework for the next day.

The second month we dove into learning Sinatra and Rails, which are web application frameworks written in Ruby. A ***web application framework*** is a program that includes all the necessary components for building a dynamic website. Using Rails, we built websites that could store and retrieve information from a database. We published the websites online so that anyone could visit the site and view our work.

In addition to all of the programming work, we built our portfolios and polished our resumes. We wrote personal statements, ordered and compared business cards, and agonized over simple formatting changes. We were also tasked with going to meetups to mingle with successful developers. I was not super comfortable with meeting new people, but I went to one meetup every week and learned to extend my hand to at least one other person.

The last month of class was dedicated to building a demo project to showcase our new skills. For my final project, my teacher encouraged me to stretch a little further, and to write a programming language. He provided the resources I would need, and I spent the next three weeks grappling with the building blocks. Since I'm Canadian, I called my language "Eh?" (check it out on GitHub). It remains one of the toughest projects I've ever tackled, even after two years of professional experience. I learned a lot about programming languages and gained some attention from hiring managers for the boldness of my project.

The end of code school came abruptly because I felt like there was still so much to learn. I didn't emerge with a job offer, but several of my classmates did. Within

a couple of weeks, I scored some phone interviews. Roughly two months after I graduated from code school, I walked into work for my first programming job. I was grateful for the chance to prove myself.

Why I wrote this book

Going to code school was a hard decision to make. I left behind a comfortable job in a city where I had built a very full life to take a financial gamble on an uncertain path to a career I didn't fully understand.

Once I took the plunge, learning how to write software was the hardest work I had ever done. For three months, I worked from 7 am to 9 pm every day. I watched videos on my commute to school each morning, worked through code exercises all day, wrote blogs and coded side projects after school hours, and read technical books on my commute home at the end of the day. On weekends I sat with my laptop and worked until I could no longer focus on the screen. Code schools are sometimes called bootcamps for good reason -- it is hard work, and you will be frustrated, exhausted, and exhilarated most of the time.

I was uncertain of what I would find at code school, but I also had no idea what writing software would be like with a team of developers. I imagined it would be solitary work that required a lot of focus and little interaction with other people. I have come to learn that a lot of my ideas were formed in a vacuum, and many of them were wrong. I want to address the uncertainty I had before pursuing this path, and to help you decide if code school is right for you.

Being a software developer requires persistence, humility, and struggle, but the payoffs are plentiful. The "aha" moments of breaking through a difficult problem

are extremely satisfying, and building software that people love is both rewarding and lucrative. Definitely not easy. Absolutely worth the effort.

I am now a software developer at a successful company. I get to spend every day working with the brightest people in the field, writing software that people love to use, solving challenging problems, and I love it. Code school was an amazing, often terrifying experience, but it allowed me to get the job of my dreams. And I would make the same decision again.

Notes on vocabulary

For the purposes of this book, ***code school*** refers to any full-time, in-person classroom-style software development program. But keep in mind that there are many programs offering different delivery options and durations as well. This book is not intended as an endorsement or a criticism of any particular school.

My own experience is with the Ruby on Rails web framework, and some content is biased toward web development technologies. There is much that is relevant to mobile application development and data analysis programs as well, but your mileage may vary.

You may often hear the titles ***software developer***, ***software engineer***, and ***software programmer*** used interchangeably. According to the Bureau of Labor Statistics and Wikipedia, there are subtle distinctions between them, but even these are vaguely outlined. For simplicity, I use ***software developer***, but you should by no means assume that you should not apply for software engineering jobs once you are out of code school.

Chapter 1: What is a Code School?

Code schools are known by many names such as bootcamps, career accelerators, training schools, accelerated learning programs, and development schools. They offer programs of varying length, content, and delivery style.

While some code schools market themselves as 21st century computer science programs, the two paths offer training in very different skill sets. Computer science degrees are excellent for learning math, linguistics, and algorithms, but because technology changes so quickly, academic programs often have difficulty keeping up with current industry trends and practices. This is a known problem and is in fact one of the reasons code schools have become popular. Some computer science graduates are turning to code schools to augment their degree. In a Dice.com interview, Purdue University Associate Professor Dr. H.E. Dunsmore said: “The gap between the classroom and the workplace has been a problem.”

Code schools are more connected to the job market than are traditional learning institutions. The programming languages and frameworks most common in web application development aren’t taught in a typical four-year curriculum, and you don’t need a computer science degree to build mobile or web applications. “The 40,000-odd students who graduate each year with four-year computer science degrees are only a fraction of the 1.4 million coders the Bureau of Labor Statistics predicts will be needed by 2020,” writes

Dan Tynan in JavaWorld. With the increasing demand for skilled developers, code schools are stepping up to fill the gap.

If your goal is to enjoy a rewarding career building software, a four-year CS degree commitment (along with a $28K to $140K price tag) may look considerably less attractive than a three-month intensive training program (with a $10K to $25K price tag). Average starting salaries for recent computer science grads and code school grads are also fairly equivalent (around $60K), making code school a great way to sidestep the time and financial commitment of a degree and gain entry-level employment in software. All you need is your foot in the door, and the rest can be learned on the job.

History of code schools

LivingSocial's Hungry Academy pioneered the idea that you could take intelligent, ambitious people who have no technical background and train them to become contributing web developers. In 2012, then Chief Technology Officer Aaron Batalion described Hungry Academy as an experiment: "Can we take raw talent, add on technical experience, add on product development experience, and turn them into awesome members of the team?" The program began as a 5-month intensive course in web application development, followed by an 18-month apprenticeship. Senior Vice President of Technology Chad Fowler said: "We believe that intelligence and passion are far harder to hire for and much more important than a specific technical skill." Since then, code schools have been springing up all over the world. Today there are more than 200 in-person programs worldwide, and even more online options.

The students

While code schools can provide the training required without the need for a computer science degree, they aren't considered replacements for college degrees. According to Course Report's 2014 code school graduates survey, 89% of students admitted to code schools have a bachelor's degree or higher, and the rest have some college education. Most students choose code school as a means for career change and have six or seven years of work experience, bringing the average student age up to 29.

Code schools are looking for particular traits in their students. Dev Bootcamp is looking for people with the "desire to create beautiful and meaningful projects." MakersAcademy wants "passion and the ability to unlearn the word 'quit.'" While many claim not to be looking for students with prior coding experience, the rigorous admissions process may include a timed coding test or a technical interview. As some code schools have an admission rate of less than 10%, they will inevitably favor the well-prepared.

According to Course Report, 38% of code school students are women, which is a considerably higher proportion than the 18% found in computer science programs. Workforce diversity is linked to productivity and innovation, and as the industry scrambles to improve the gender discrepancy, code schools are responding to the demand for more female developers in tech. Many schools, such as Dev Bootcamp and Coding Dojo, offer scholarships for women and other groups that are underrepresented in the industry. Going one step farther, Ada Developers Academy and Hackbright Academy admit only women.

Racial diversity among code school students continues to lag far behind gender diversity, and it doesn't accurately reflect the racial composition of the U.S.

population. Sixty-three percent of U.S. code school students identify as white, while 17% identify as Asian American and a mere 1% as black.

It is important to look beyond the marketing messages. Code schools are not in business to drive social change -- they want to produce developers who are marketable to employers. They are incredibly profitable businesses. For example, the San Francisco chapter of Dev Bootcamp made about $4.1 million on tuition in 2014 alone. Code school revenues are not based solely on tuition -- they often net hefty recruiting fees from employers (up to 20% of a new hire's first year salary). Some schools extract the recruiting fee directly from the student by garnishing their first year's wages. Part of the reason code schools have mushroomed around the world is this amazing return on investment, which has attracted the attention of venture capitalists and regulatory agencies alike.

The bad press

The code school business model is based on the idea that anyone can learn to code, and that three months of hard work will result in a rewarding career in software development. Both of these claims come with an asterisk: while everyone can learn to code, not everyone will be admitted into a code school; and three months isn't enough time to learn the basics, in some cases.

Most criticism of the code school model can be found on social media, where former students or employees share their negative experiences. Exclusionary admission practices, artificial inflation of success statistics, and resistance to oversight are among the claims leveled at code schools, and they are not without some merit.

Lack of oversight and regulation of the industry means that it is difficult to penetrate the marketing messages to find real numbers on admissions and employment. The statistics you will hear from code schools themselves support their marketing goals. Hack Reactor boasts a 99% hiring rate with an average salary of $105K. Instructor Ruan Pethiyagoda writes in a Quora thread that "[much] of the credit for our success [goes to] to our admittedly strict admissions policy." They don't actually take people with no prior experience and turn them into software developers in three months. While many schools publicize a "zero to 60" curriculum, Hack Reactor is a "20-to-120 program," according to cofounder Shawn Drost.

Code school admissions typically evaluate students with the lens of a hiring manager or recruiter, choosing students who have demonstrated successes in the past, who are serious about learning to code, and who may already have coding experience. In this light, code schools take the role of pre-filtering candidates for their employer network, and the students benefit more from preferred access to desired employers than through the content of the course itself.

Another way code schools attain high success rates is by asking students to leave if they are struggling to keep up with the material. "They have financial goals to meet… [and] a product to produce. If I don't fit that mold, then I don't belong there." says former Dev Bootcamp student Khara Muniz in an interview with FastCompany. Muniz was asked to leave the code school because she was not able to keep up with the content. A "self-described visual learner," Muniz struggled with a curriculum and teaching style that did not fit with how she learns.

Most code school teachers are former developers and occasionally recent graduates of the code schools themselves. They are typically not trained

educators. Some code school founders have never worked as software developers, and many resist attempts to bring code schools under regulatory control. No bootcamp currently boasts accreditation, in spite of the 2014 crisis when the California Bureau for Private Postsecondary Education sent cease-and-desist letters to several coding bootcamps. Anthony Phillips, cofounder of Hack Reactor, told VentureBeat that he "would like to be part of a group that creates those standards...[but] what that looks like and what makes sense for our schools is not necessarily going to fit in the current regulations."

Dev Bootcamp co-founder Shereef Bishay said in an interview with USA Today that his school is more of an apprenticeship program than an educational institution. "If you are getting a 95% job placement rate, do you really care if the teacher has a diploma?"

Complying with state regulations would require registering a curriculum that would be evaluated against professional standards, as well as hiring instructors who have at least a bachelor's degree and three years' teaching experience. Raising the bar for educational credentials would be more costly for code schools, and their curriculum would no longer be protected by trade secret legislation. On the other hand, accreditation might stifle innovation and hamper their ability to respond to changing industry demands, which are the reasons why code schools exist in the first place.

In 2014, almost 7,000 students completed a coding bootcamp in the United States and Canada. That number is expected to grow to 16,000 by the end of 2015. Meanwhile, the demand for software developers will increase by more than 20% by 2022, creating nearly 250,000 jobs in the next 10 years. With only 40,000 computer science degrees granted each year, demand for trained software developers is predicted to far outstrip the supply.

There has never been a better time to work in software, but taking the plunge into code school is *not* a decision that should be made impulsively. It's not a golden ticket to an easy six-figure salary. The next section explores a few reasons why.

The challenges

Code school takes time

Most bootcamps range from 4 to 28 weeks long. The most common is a 10-week program that will demand a minimum of 60 hours of work per week. If you are a complete beginner, expect to spend even more. There is a vast amount of material to cover, and most code schools will help you hit the ground running by assigning pre-work in the weeks leading up to your first day. If you take a day off for any reason, you will fall behind. You should not expect the instructors to wait for you to catch up. Be prepared to immerse yourself completely and to not come up for air for two or three months.

> Can you commit yourself to 12-hour days, late nights of study, and weekends dedicated to code? Do you have the social support, childcare, and financial resources you need to remove obstacles to your learning?

Code school takes money

Three months of full-time study can seem like a prohibitively long time between paychecks. And it might be more than three months. Not all students finish code school with an employment offer in hand. An application process that results in a job offer can take six weeks or more, so it would be wise to budget for at least eight weeks of unemployment after graduation to cover a reasonable period of job search and interviewing. That means you need to plan to live without a paycheck for five or six months.

Code school tuition averages $11K, and can be as high as $21K, which is relatively inexpensive compared to a four-year computer science degree, but still nothing to sniff at. Many offer scholarships, deferred payment plans, or reimbursement from selected employers. Thanks to President Obama's TechHire Initiative, there will be more scholarships available and even student loans granted for code school programs.

Some schools offer guaranteed tuition, where they will pay you back if you are not employed as a developer after graduation. These are very attractive offers, but pay attention to the fine print. You may be required to hand over the entire sum before your first day of class, or at least to pony up a substantial deposit.

No matter how you slice it, with 10 or more weeks of code school and six to eight additional weeks of unemployment, attending code school a very expensive proposition.

> Do you have the financial resources to live for six months without an income? Check out the Coding Bootcamp Cost Calculator hosted by FreeCodeCamp to get an estimate of what code school will cost you, including tuition, housing, and opportunity cost (lost wages) while you study.

Code school does not guarantee you a job as a web developer

While there is increasing demand for software developers, many companies shy away from offering full-time developer work to someone without a few years of experience. Internships and contract-to-hire positions are ways that companies can test your abilities without the commitment of a full-time offer. These positions are usually low pay compared to a junior developer salary, and

the amount of structure and investment into your onboarding and education in such roles can be quite limited. But they are a great way to get your foot in the door.

There are also many different kinds of software jobs that require the technical knowledge you gain in code school, but not necessarily the coding skills. Technical support, quality assurance, and technical writing positions will give you valuable industry experience, and they may also lead to a job as a developer down the road.

> Are you prepared to take an intermediate position if necessary?

Code school takes determination

Learning is hard but satisfying work, and you can expect to spend the majority of your time as a student and as a developer trying to solve difficult problems. These are skills and tolerances that can be built over time, but will be easier if you find the end result sufficiently rewarding. Success requires a great deal of self-motivation and concentration, and a high tolerance for frustration. Expect to succeed sometimes and to flounder often. The hard work doesn't end when code school is over. Choosing a career as a web developer means choosing a career of learning and problem-solving.

> Do you like to solve logic problems that require attention to detail? Can you learn independently? Are you easily frustrated when you don't know the answer?

The rewards

Code school students are going to more than double in number this year, and that would not happen if the rewards of becoming a full-time developer did not greatly outweigh the challenges. Only 14% of code school students don't get a job right away, and on average those who do get a job enjoy a 44% increase in salary from their previous positions. If you succeed in code school, your skills will be in demand for many years to come. Here are some of the rewards.

Companies want to hire you

You can expect to graduate code school with a firm grasp of the fundamentals of software development -- enough to become a fully contributing member of an engineering team within three to six months after landing your first job. Companies are looking for developers who know the basics but are passionate about expanding their skill set, and who fit well into their company culture -- attitude and aptitude trump technical skill. Most of what goes into building and maintaining a web application can be learned on the job, but there is a range of skill and experience required, depending on the complexity of the problems that need solving.

Tasks at the lighter end of the scale do not require advanced education. Thus, they are perfect for juniors to dig into. From fixing bugs to improving existing features, junior developers are needed to take on the simpler tasks, freeing up experienced developers to tackle more difficult problems. Avi Flombaum, Dean of the Flatiron School in New York, writes, "if you're trying to build a better search engine, or solve the world's most complex data problems, you probably do need to recruit from the top 1%." On the other hand, if you are trying to build a new blog page or dashboard, hiring a "rockstar senior [developer] is like hiring

Picasso to paint your apartment." Code school will help you learn the basics of a technology stack, and most of the work of developing and maintaining a web application falls under the basic category.

Some companies perceive juniors as liabilities because they work more slowly, and need attention from more senior developers. There is a time and energy investment required to bring them to a contributing competence. However, there are many organizational benefits to be gained as well. A junior developer who is making a career change will bring a fresh perspective and different skill set than you would find in a new college graduate. Project management, public speaking, and team collaboration are just as important in software as they are in other industries, and companies want to hire candidates with experience in these soft skills.

Junior developers are also good for the careers of senior developers, because they provide the opportunity for them to develop leadership, mentorship, and management skills. There are obvious benefits for the careers of juniors as well, not least of which is the opportunity to learn from more skilled developers. Having a range of visible stages of career growth within the company reduces developer turnover and keeps a company's investment concentrated in-house.

Fially, hiring junior developers is healthy for code. Having less experienced programmers improves readability and maintainability by reducing complexity. If code is not understandable by someone with a small amount of training, it will be difficult to change and easy to break. Explaining engineering decisions also forces senior developers to challenge their assumptions and spur collaboration toward creative problem-solving.

You get a great salary

Software development is a profitable career. The national average for a junior developer hovers around $60K, but your salary will vary by city. There are several technology hubs around the United States and Canada, but they are by no means the only cities where you can find a job. Remote work is becoming more common, and your company will compensate you according to the standards of your locale, as well as for your level of experience.

City	Minimum	Median	Average
San Francisco, CA	$60,000	$90,000	$90,000
Seattle, WA	$71,000	$95,000	$100,000
Portland, OR	$50,000	$65,000	$80,000
Chicaco, IL	$49,000	$72,000	$68,000
Las Vegas, NV	$48,000	$62,000	$62,000
Austin, TX	$46,000	$60,500	$60,000
Vancouver, BC	$62,000	$75,000	$80,000
Toronto, ON	$77,000	$80,000	$82,000
Montreal QC	$55,000	$75,000	$75,000

These ranges are self-reported by full-time software developers. Junior developer salaries will likely be a bit lower. Checkout GlassDoor, Payscale, and Salary.com to find the average salary for a software developer in your city.

What jobs can you get after code school?

Almost three-quarters of code school students intend to get a job as a developer after the program, yet only 63% of graduates start their career with a full-time job or an internship. There are many entry-level jobs in software that offer rewarding career paths of their own or that can be a gateway to developer roles in the future.

Junior developer: Landing a job as a junior developer is the holy grail for most code school students. Working in a production-size code base with a team of experienced developers will provide you with valuable experience and support as you learn the ropes of becoming a professional developer.

Intern: Many larger companies will not hire untested developers without putting them through an internship first. As an intern, you will spend much of your time learning and connecting with mentors. When your internship comes to an end, you will have gained a lot of experience, and you can use your newly expanded professional network to get back into the job hunt. Many companies also hire their interns as developers.

Quality assurance engineer: The qualifications for QA engineer are similar to those of a developer: strong communication skills, the ability to learn fast, and familiarity with development processes. As a QA engineer, you are a product expert. You understand the way users interact with the product and how the features interact with one another. Combining attention to detail with a holistic view of the system, QA engineers ensure that users continue to trust the product you are delivering. You work alongside development teams and may write code in the form of integration or front-end testing.

Technical writer: Writing technical specs and documentation will require all of your technical knowledge as well as language, teaching, and research skills. The technical writing field is expected to grow by 15% in the next 10 years, and delivering web-based product support will be par for the course. Use your new coding skills to build documentation websites and forums, as well as to generate useful content.

Technical support engineer: Technical support is growing at a similar pace to technical writing. Interacting successfully with a product is essential to customers' happiness, and companies have a vested interest in troubleshooting where these interactions go wrong. As a technical support engineer, you act as liaison between the end user and the development and management teams by reporting and triaging bugs, working with developers to resolve issues, and communicating with customers. Some companies pair their support staff with experienced developers to train them on the job and will only hire junior developers who are already tenured in support.

Chapter 2: How to Pick a Code School

There are more than 80 code schools offering full time programs in the United States and Canada. So, which one should you choose?

The answer to this question will be different for every student. Depending on your previous experience, financial considerations, and time constraints, there is a school that will deliver the education for you. With so many in-person and online options available (including one that combines travel and study called Ruby on the Beach), how do you determine if a code school is worth the tuition they are charging or if it will be closing its doors forever after you graduate? Four schools from CourseReport's 2014 Code School Survey were no longer operating the following year. There are many factors to consider when deciding which code school is right for you.

What are your goals?

Before you decide which code school to apply to, you should know what you want to get out of it. Deciding what success looks like is a great way to make sure you are making the right choices along the way. My goals for code school were to find a 12-week program that would teach me web application development, to build a network of contacts in the tech community, and to get a job as a junior developer. Twelve weeks was my target because I wanted a comprehensive curriculum, and three months is the longest I could afford to be unemployed (though as I mentioned earlier, it took longer than that). I eventually chose my

code school for financial reasons: I could live with family and defer the tuition until after I graduated. The program was the right length, and the school taught the technology I wanted to learn: Ruby on Rails.

Do you want to design websites? Build web or mobile applications? Are you interested in management or solving technical problems? Do you want to live in San Francisco or Chicago? Having a clear idea of what you want to accomplish will help you narrow down the choices.

Once you've identified your goals, it's time to do some research. Course Report is a great tool for finding programs in your area of interest, be it by location, duration, or content. You should whittle down the options to a short list by evaluating candidate schools with a the following criteria in mind.

How awesome is their website?

If a code school is going to teach you about web development, they'd better be good web developers. Their website should be modern and well designed, and it should load quickly. You can tell at a glance whether the person who built the website is using design techniques from last century or is familiar with the latest patterns. Information should be easy to find, with a relatively high amount of detail. You won't find day-by-day breakdowns of course content, but you should be able to determine what technologies they teach and their general approach. Their application process and admission standards should be clear, and they should include a few student testimonials and frequently asked questions (FAQs).

How long is the program?

Programs vary greatly in length: some are as short as four weeks (Code Fellows), some as long as seven months (Turing School of Software and Design). If you can commit to a longer school, you will be better prepared to enter the job market as a more experienced developer. You will learn not only how to build and deploy a web application, but also how to architect and scale. Ten or twelve week courses are more common: you will be taught the basics, but you need to be prepared for a lot of learning on the job.

Avoid bootcamps that are fewer than six weeks long unless you already have considerable programming experience. Shorter programs will not be able to give a beginner the breadth of training they need to become employable as a developer. You can't learn the same amount of material in one month as you can in seven, but depending on your previous coding experience, four weeks may be all the start you need.

What do they teach?

There are several broad categories of curriculum available: mobile app development, web frameworks, data, and design. A school that specializes in mobile app development will teach Android, Objective-C, or Swift. Web application schools remain the most popular and offer more language options, but the majority of schools (35%) teach Ruby on Rails. JavaScript web frameworks are gaining traction as many companies move towards building single-page applications. MakerSquare changed their curriculum from Ruby on Rails to entirely JavaScript in 2015. If you choose a program in Ruby on Rails, make sure that the curriculum includes some JavaScript.

Data science and database workshops are growing in number, although many of them require students to have two to three years of programming experience. A smaller number of schools also offer training in user experience design and product management. The table below outlines a few of these offerings.

Category	Content	Examples
Mobile app development	iOS Android Swift	Code Fellows DaVinci Coders Lighthouse Labs DevMountain CodeAbode
Web applications	Ruby/Rails Python/Django PHP/WordPress AngularJS jQuery Node.js	Epicodus Hackbright Academy Coding Campus Dev Bootcamp LearningFuze
Data and databases	Data Science/R Hadoop MongoDB SQL	The Iron Yard Galvanize NYC Data Science Academy BitBootcamp
Design	Digital marketing Product management User experience design	Interface Web School General Assembly TradeCraft DevPoint Labs

What is the structure of the program?

Going from zero experience in programming to job-readiness requires a lot of time and planning. By the end of the course, you'll need to know all the building blocks required for a web application, but there is a wide range of opinions about how each block should be learned. A good code school should have an established cadence for how to make the material approachable and incremental.

Pre-work: There is too much material to cover in a few short months, so look for a code school that assigns pre-work to bring the entire class to an appropriate baseline before the course begins. Pre-work ensures that people who have more experience will not be waiting for the rest to catch up before moving on. Getting a head-start on material you need to cover means you'll have more time and energy to tackle the harder parts.

Pair programming: Working through exercises with another person can help you leapfrog your way to an understanding of programming fundamentals. Many employers are beginning to encourage more pair programming, because it generates higher quality code than a single developer can produce on their own. While having a co-learner can be helpful, it's also important for you to struggle through problems on your own. Ideally there will be a balance between independent and group work.

Project work: Most of your time in code school should be devoted to project work, so start thinking about software that you would like to use, but doesn't exist yet. Find a code school that emphasizes independent final projects over group projects. An independent project will force you to build and understand every part of your application. It's a great mini-portfolio and a concrete way to demonstrate to future employers that you are capable of learning and applying

difficult concepts. A group project, on the other hand, emphasizes delivery of a product rather than a process, and makes it difficult to highlight individual contributions.

There is also a variety of delivery methods that will appeal to different learning styles. Some code schools use a combination of lectures, exercises, and group work. Others include a combination of in-person and online tools, where students work through problems on their own. Ask your prospective bootcamp what learning styles they cater to and whether the curriculum is taught using current educational theories. Also, think about how you like to learn. Do you like to learn in groups or do more self-directed study? Your preferred mode of learning will play a large role in your code school success as well as in your future career, so make sure you select a school that fits your learning style.

Who are the teachers?

Your code school teachers are your first mentors. You will draw on their experience to learn not only programming concepts, but how to navigate the industry, how to interview for technical jobs, and how to connect with others in the tech community.

It's important that your teachers have industry experience, and their credibility often lies in how well they are respected in the community. Google your instructors and follow them on Twitter.

Does the instructor write a blog? It's a good sign if they write about technical concepts or teaching. They should also have an updated GitHub profile. GitHub is a social network and repository for open source code. There's an activity chart on every user's page that indicates how active they've been in the last year.

Avoid code schools that hire their recent graduates as teachers. While they may be qualified to help you through the course material, they won't have an industry insider's perspective.

When you begin your job search, you will be borrowing the credibility of your code school as well as its instructors. You'll need to leverage your teacher's reputation as a programmer, so it's important that their recommendation is something employers will value.

Ask your prospective code school about their teachers:

- Are they previous bootcamp grads or professionals dedicated to training and inspiring the next generation of developers?
- What is their teaching style?
- Are they actively involved in coaching their students, or do they prefer lectures and homework to a more hands-on approach?
- Do they have any teaching experience or training?

If you have the opportunity, sit in on a class. Try to talk to the teachers directly and ask them what brought them to the bootcamp and why they like teaching.

Who are the students?

You will learn as much or more from your peers as from your instructors, so it's important that your class is made up of people you want to learn from. Make sure that your code school has stringent selection criteria. The admissions committee should select students who have a high potential for success, not just the ones who can afford to pay the tuition, so you can reasonably assume that everyone in your cohort will have the same level of drive as you do. It's good for a code

school to not accept students who don't seem fully committed to seeing the program through, and it's good for the morale of the class as well.

Ask your code school about their admissions standards:

- What are their rates of admission?
- What are the criteria they are looking for?
- Are the students more mature, and looking for a career change, or are they recent college graduates?
- How much experience do they have on average? Any complete beginners?
- What skills must be demonstrated for admission?

Your code school class should ideally be small in size, from 15 to 30 people. If the class size is any larger, ask what the instructor-to-student ratio will be. One instructor for every 10 students is the minimum. Otherwise, you will have reduced access to help when you get stuck. If the class is smaller than 15 to 30 students, it's a sign that your school is either losing traction in the market or is not fully established. Don't go to a code school that cannot attract students.

Who are the graduates?

Many code schools boast 90+% employment for their graduates, but not all of them report their numbers. It's also unclear when the measurements were taken. Are the averages for the last year? The last cohort? How long after the program? According to Course Report's annual graduates survey, 63% of students are employed full-time after graduating from code school and 75% are employed as programmers, but these numbers don't distinguish between internships, contract, and full-time jobs. Keep in mind that currently there is no governing

agency that code schools are required to report to, so these numbers may not be true measures of the average code school student's success.

Ask the code school for the most current employment statistics for the last year's graduates and for the last cohort for which they have data. If they don't have data for the last cohort, ask why (they may be hiding something). The answers to these questions will tell you how mature the code school is and whether they deliver on what they promise. The number of graduates who were hired by companies in the employer network will also tell you how well the school is respected in the industry.

Finally, are there graduates that you can contact to learn more about their experience? If prior graduates are unwilling to be contacted, or are distancing themselves from the school, consider that a hint that you may be in for a bad experience. Find out if they've ever invited a student to leave the program because they were unable to keep up with the work. Get as much detail as you can about their efforts to ensure success among their beginner students, which will tell you how much they will invest in you.

Where is it located?

Geography is as important as curriculum, because the program you select will have a sphere of influence that is unlikely to stretch beyond the city where it is located. The network of developers you connect with during the program, as well as the employers connected with the code school, improves the likelihood that your first job offers will come from companies in the same city. Choosing a school that has multiple locations is a good strategy for keeping your options open once you graduate. For example, Dev Bootcamp has locations in San

Francisco, Chicago, and New York, and will often encourage people who are on their wait list for San Francisco to apply to Chicago instead. They get access to the same employer network, but don't have to wait a year to be admitted.

What are the career resources?

The amount of time spent on career preparation varies widely among schools. While it's important to polish your resume, prepare code samples, and develop your online presence, spending too much time preparing your first job application will eat away at your skill-building time. If you can, get some of these resources started in advance so you don't waste precious coding time trying to write a personal statement or selecting the right business card (see Chapter 6 for more tips on how you can prepare in advance).

Is there an employer network? Code schools promise to teach you how to build software and to provide you every opportunity to gain employment as a software developer. One way they can fulfill the latter is by developing their employer network. Some employers even work with school organizers to develop curriculum. They may contribute funding and often host tours, give talks, and attend career fairs. Companies in the employer network are pre-sold on the idea of hiring junior developers, which means you can spend more time demonstrating your skills instead of persuading them that you will not just be drawing resources. The more connected your code school is in the community, the larger the network you will have to draw on when you are job hunting. If the school does not have an employer network, back away slowly. Without buy-in from the industry, the school does not have the credibility you need to get your foot in the door.

How extensive is the employer network? Are they actively recruiting students? How many graduates have they hired? The employer network could be large if the barrier to entry is low -- some schools offer free or low-cost memberships to employers in exchange for the equivalent of a first-round draft pick. If the network is not hiring graduates, take your business elsewhere.

Will there be an opportunity to meet and demonstrate your work to potential employers? Without a career day or job fair, you will be stuck with cold calling your resume after code school is over.

Does the school have relationships with recruiters who will help place you in companies outside the network? Are there work-study, internship, or work placement opportunities? Some schools organize a two-week internship at a local company, so you can extend your learning in a professional environment.

Will there be dedicated career counselors? What about invited guest speakers or company tours? Your school should bring in experienced developers who can help you decide where and how to take the first step in your career.

What does it cost?

It's not a bad idea to sort your short list of code schools by their cost. You want to attend the best school you can afford, but the prestige of your school won't matter once you get your first job.

Code school is expensive, and some require you to pay the complete tuition fee upfront. Others have financial aid packages that allow you to pay the full amount over two or three years. Some schools offer scholarships for underrepresented

minority groups, so if you belong to one, be sure to ask if there is any compensation available. There may even be crowd-sourcing opportunities that your school can recommend (check out The Crowdfunding Centre).

Consider not only tuition, but also living costs while you are studying full-time. Some students have left code school with almost $20K in debt. Currently, most code schools don't qualify for federal or state financial aid. In addition, the job hunt can take time, so be sure to plan for two or three extra months of financial padding.

Are books and other learning materials included in the cost? Are there computers available to rent, or do you need to purchase your own? If you are relocating, can the school help you arrange inexpensive or subsidized housing?

What other resources are available?

Think about your physical needs. You will be spending eight or more hours per day in a classroom. What are the facilities like? Are there shared workspaces and rooms? Are there quiet areas where you can work alone? How available is the learning space? Will you be able to access the room at any hour or stay late to work on a project? Are there large monitors, or will you be hunched over a laptop most of the time? You are embarking on a relatively sedentary career, so it's never too early to get into good ergonomic habits.

This may seem small compared to the curriculum, but the facilities are important. My school shared space with a development shop and we couldn't stay in the classroom later than the last developer. This time constraint became a problem when we had project deadlines.

Most schools offer an open house to prospective students, and that is a great time to check out the space where you will practically be living for three months. Is there a kitchen available? Coffee or snacks? Easily accessible bathrooms? Is the school located near reasonable food options, or will you need to bring your own? If you can't take a tour, ask a former student.

Your choice of code school has a big impact on your success. Students apply to an average of 1.6 schools. This low number may be because students are doing a great deal of research before beginning the application process, but it could also indicate that many people choose a code school in a non-deliberate way. Having your goals in mind as you begin your research will help you avoid being wooed by attractive marketing messages.

Chapter 3: What You Will Learn

Full-time immersive web development courses all share a common general curriculum. The quantity of material you need to cover can be overwhelming, but if you tackle one foundational concept at a time, you'll be surprised how much you can learn. You shouldn't expect to be comfortable with the entire stack by the end, but you will emerge with a good overview and some experience with everything you need to begin your career.

This chapter outlines the basic knowledge and skills you can expect to learn in code school.

Web development languages

HyperText Markup Language (HTML) and ***Cascading Style Sheets (CSS)*** are the foundation of all web pages. HTML determines the structure, and CSS dictates the style, color, and layout of the page. While HTML and CSS have strict rules governing structure, they aren't programming languages in the traditional sense because they don't contain any internal logic. There are numerous books and online tutorials to get you started (see the Resources section), and being comfortable with HTML and CSS is a prerequisite for many code schools.

Programming languages

While static web pages are built using HTML and CSS, interactive pages require JavaScript. ***JavaScript*** is a programming language designed to respond to events by manipulating the structure and content of a web page after it has been loaded in a browser. The events might be user generated, such as clicking on a dropdown arrow in a menu, or time dependent, such as a popup window that appears as you are reading a news article. JavaScript is transforming the way we write applications for the web by creating new ways to retrieve, display, and manipulate information.

Basic programming concepts are common to most mainstream languages: creating variables, methods, and objects; order of operations; flow control (if/else); loop control (while/for); and scope. These concepts, along with basic problem solving, will dominate the first few weeks of the course.

Most code schools teach either Ruby or Python. These languages power the logic behind applications, and they are popular for beginners because they have a descriptive, English-like syntax. Many well-known websites are written using Ruby, including Bloomberg, AirBnB, and Shopify.

Mastering the basics in these languages will allow you to build applications using a popular web framework.

Web frameworks

A web framework is a software program that produces web pages dynamically by storing and retrieving data from a database to generate content. Writing programs that generate HTML is more efficient and less error prone than writing

pages by hand. Your code school will specialize in one web framework: Rails is the most common, but you may also learn Sinatra, WordPress, Django, Meteor, or others.

Test-driven development

Test-driven development (TDD) is a technique for thinking about the behavior of your program before you implement it. Tests are code that exercises the important parts of your application. In TDD, you first write a test that asserts some behavior from your program, and then you write the code that satisfies the test. The cycle is often called Red-Green-Refactor, because the test you write will fail initially (Red), will pass when you implement the behavior (Green), and will give you confidence to make changes without breaking any essential functionality (Refactor). Writing tests first will result in clean and clear code.

Source control

Part of your code school experience may include a group project. Collaborating with a team means working in the same code base. How do you ensure that you aren't overwriting one another's changes?

Using a tool for tracking the changes each programmer makes to code is essential for working together harmoniously. Git is one example of source control software, and you will interact with Git or a similar tool every day of your development career. Git is like a series of cartoons that make up an animation in a flip book -- each page contains a snapshot of what your code looks like with some slight modification. The last image is the most up-to-date version of the code, but you can also flip back to any snapshot along the way, compare two snapshots to see how they are different, or add new ones to continue the story.

How to learn

More important than any language, tool, or process is the ability to learn quickly. There are many different approaches you can take to picking up a new concept, depending on how much you want to understand how everything works. For example, I like to learn through experience, and will race to build my first program and use the code in context. Whatever your learning style, it's important to know where to go for answers.

A Google search is a great place to start. Experienced developers use Google search and StackOverflow constantly, so if you are stuck with an error message you've never seen before, pop it into the search bar to see what turns up. A word to the wise: if an answer was posted more than two years ago, it may no longer be relevant.

Also, don't be afraid to go directly to the source code. Open source libraries are written in the code that you are learning how to read. Take it slow. Read each line and ask: What is it doing? Why is it doing that? Beginner tutorials will only take you so far, and you will need to be resourceful when troubleshooting difficult problems. Learning is something you will do throughout your career in programming, so get comfortable with not knowing the answer right away.

How to solve problems

Algorithm design will help you learn to code, and it will come in very handy as you prepare for your first technical job interview. Before implementing a solution in code, you should have a plan in mind. In plain language, write out the steps you would take to find the solution. Take this opportunity to think about tests and alternative solutions. Once your plan is clear, write the code, and then look for ways to improve its structure and readability.

How to apply for a technical job

Writing a solid resume and cover letter is just the beginning of landing a technical job. You need to generate a portfolio of work that showcases your coding skills, as well as pass many rounds of phone screens and interviews. Mock interviews, code challenges, and resume coaching are a few of the ways your code school should help you prepare for the barrage of technical and culture fit questions you will face.

Chapter 4: What You Won't Learn

Code schools are designed to get you contributing productively in a short period of time. By trimming down the content of a computer science degree, and removing the parts that are mostly irrelevant to web development, it is easy to produce a program that stacks up a lot of coding solutions with no context. There are many tools and concepts that will make your life as a developer easier, but most of this learning is left up to you.

This chapter outlines knowledge and skills that developers use all of the time, but that you likely won't learn in code school.

How to work with legacy code

Starting a hobby project from scratch gives you full creative control. You decide how to organize your files, how to design your data, and how to name each component. After a few weeks of development, you will inevitably find yourself rearranging, renaming, and rewriting parts of your project. Multiply the impact of these early decisions by the dozens of developers, features, and technologies that are incorporated into a successful product, and you can understand what it might be like to work with legacy code.

Legacy code is code that was written with different technologies, developers, and constraints. It can be weeks, months, or years old. The main property of a legacy code base is that you are unlikely to be able to understand all of it

at once. Consequently, you will often be asked to solve a technical problem without having all of the information beforehand. Learning how to follow an error message and trace your way through the message chain to find the source of the problem is a skill you can't learn until you work in a complex code base with many other contributors.

How to use debuggers and developer tools

No code is completely bug-free. Finding bugs in code requires shining a light into a line of code as it's running. You will need some experience working with debugging tools on the server side as well as the browser side. ***Debuggers*** give you the opportunity to see what happens in your code while it's running. This is mainly useful when something goes wrong. You can create a breakpoint just before a broken line of code and the debugger will give you an interactive console session where you can check the state of your application before the error occurs. Getting to know your debugger will increase your speed, so look for tutorials and tricks that will help you learn its features. For server-side Ruby, become familiar with 'pry' (for Python, it's 'pdb').

For JavaScript and CSS, learn an in-browser debugger, like Chrome Developer Tools or Mozilla's FireBug. These tools give you access to everything that happens to your application when it's being rendered in the browser, and you can troubleshoot JavaScript, CSS, and layout issues without having to refresh the page. Check out the tutorials at discover-devtools by CodeSchool.

Regular expressions

Pattern matching is a fundamental tool for many developers, and ***regular expressions*** are a programming language specifically designed for the task. You will find some form of regular expression in every language.

Regular expressions can be used for many different kinds of tasks:

- ***Validating user input.*** Is the email address the user entered valid?
- ***Route identification.*** Which part of the program should run when someone clicks on a new link?
- ***Testing.*** Is the output of the method I am testing similar to what I was expecting?
- ***Flow control.*** Which program should I run if the user enters *A* in the form instead of *B*?

DevOps

Once you leave code school, you will have touched every part of the full stack, or so you think. You will have created a database along with the server and pages for your application. You will also have deployed your application to a free hosting service such as Heroku or Meteor. The application you develop by following a few tutorials, however, can't be used simultaneously by hundreds of users.

Your experience will not prepare you for how applications are deployed and managed at business scale. This is a job for a team of Developer Operations (DevOps) engineers, who are focused on servers, databases, and network devices. They use a different stack of technologies than the ones you learn

in code school. Getting familiar with some of the tools they use will help you understand what is happening on the server when your application starts up, connects to the database, and begins to serve traffic.

- Linux and command line tools
- SQL for direct interaction with databases
- Scripting: Bash (Linux/Mac) and PowerShell (Windows)
- Puppet (or Chef) for configuring servers
- Capistrano for deploying to multiple servers at once
- Monit for managing processes
- Unicorn (or Puma or Thin) for growing your web server

There are even more technologies for managing clusters of servers and configuring networks and load balancers. While your first job is not likely to require interacting with this deeper stack, you will eventually need to troubleshoot issues that occur below your application layer. Keep an eye out for opportunities to learn more about these tools.

Agile vs. waterfall methods

Managing the development of a software project takes several forms. ***Waterfall software development*** is a set of practices for planning, designing, and delivering a software product. It is called waterfall because the project progress moves in one direction only: gather user requirements, develop technical specifications, describe the architecture using unified modeling language (UML), implement the specifications, and, finally, test the implementation. In this scenario, making changes gets increasingly more difficult and expensive as the project progresses, and if any stage takes longer than expected, it's the quality assurance engineers at the end of the process who suffer. Following the waterfall model, it can take

months if not years to bring a product to market, and much can change in the meantime. You may deliver a product that no longer serves the needs of the users you were targeting.

Agile software development shortens the cycle between implementation and delivery. The users are involved at every stage of the product development, and you deliver the minimum viable product required for the stakeholders to decide if you are moving in the right direction. Work is broken up into small chunks called ***sprints***. Within each sprint there will likely be design work, architecture decisions, implementation, and testing. At the end of the sprint, you demo the latest chunk of work and plan the next. This allows for fast iteration and continuous user acceptance, which ensures that the product you deliver at the end of the project is what your users wanted all along.

Every company takes a different approach to software development. While there are many tools and techniques for agile and waterfall methodologies, most companies use the tools that work for them and discard the rest. Don't get hung up on the details. Your first job in technology should teach you all you need to know.

Statically typed languages

There are several categories of programming languages. Ruby, Python, and JavaScript are ***dynamically typed***, which, for simplicity's sake, means that you can make changes to the program while it is running. A statically typed language, such as Java and C++, ensures that the program is defined, checked, and compiled into its final form long before run time.

Ruby and Python are easy to learn and fun to write in, but the trade-off for ease-of-use is that they are relatively slow to run. Large companies like Twitter

and Facebook serve data to millions of users simultaneously, and Ruby and Python can't keep up with the demand. At that size and scale, companies must turn to languages such as Java, Scala, and .NET. These languages are incredibly powerful, but they take longer to learn. There are no code schools for teaching beginners professional-grade Java in three months.

The language you learn in code school will impact the kind of jobs you can pursue right out of the gate, but remember that your learning doesn't end when you leave code school. Once you are comfortable with one language, it's easier to pick up another. Learning a statically typed language will deepen your understanding of programming and broaden your professional horizons.

Chapter 5: How to Get into Code School

As soon as you decide to go to code school, you should start putting together your application. Some schools have an 8- to 12-month waiting period for their programs. Applications may be accepted during specified weeks prior to the next class, or there may be rolling enrollment. Read the instructions carefully -- it's the first opportunity you have to prove that you are detail oriented and committed to success.

The application

Your application to code school could take many forms. Most schools will ask for your resume and cover letter to start, so make sure you get those as polished as possible. Emphasize aspects of your career or education where you have completed projects, interacted with developers, or acquired coding experience yourself.

Include all of your experience

Include any online courses or technical books you've studied. Any initiative you've already taken to learn how to code will show your dedication. Coursera, Codecademy, and Stanford Online offer many free classes in programming and data science. Online tutorials also count. Try the W3Schools website for HTML, CSS, and JavaScript. TryRuby and Rails for Zombies provide practical and free experimentation.

Emphasize the experience you have, however marginal, but don't exaggerate. Don't put a language or technique on your resume that you are not prepared to discuss in an interview.

Don't skip the creative component

Your resume is just a starting point. Many code schools will ask you for a creative piece to go along with your application. This could be an infographic, a video, or a poem or short story describing who you are, what you are passionate about, what drew you to code school, and why you are interesting. Don't consider this exercise optional (even if it is). This is your opportunity to differentiate yourself from the crowd of applicants. As you get creative, keep in mind that code schools are looking for students who will be marketable to employers: they are looking for curiosity, passion, ambition, and dedication. Make sure that your video or infographic highlights these qualities in you. (Check out the embarrassing infographic I submitted.)

Call upon your inner nerd

Your acceptance to bootcamp depends on your demonstrating the kind of interest and tenacity you will need for coding. Do you play video games or card games like bridge? Design your own knitting patterns or soap box derby cars? Can you play an instrument or a sport? Any interests you pursue outside of your career will help you get into code school, because they demonstrate your ability to learn, to persevere, and to continue growing after you graduate. Include these interests on your resume and in your infographic or video, and use them in your interview to show your versatility, dedication, and ability to focus.

Prepare for a technical test

Code schools will not consider applicants who have never written any code. How do you know you'll like it if you've never tried it? Some schools will

also ask for a technical interview, but more often you will be given a small piece of coding homework to demonstrate your technical ability. It may be a logic problem or a layout problem, but it's more likely going to be something like FizzBuzz.

FizzBuzz challenges you to write a program that will print out each number from 1 to 100. If the number is a multiple of 3, it should print "Fizz"; if it's a multiple of 5, it should print "Buzz"; and if it's a multiple of both 3 and 5, it should print "FizzBuzz." This challenge tests your knowledge of basic programming using loops and if/else statements. There are many solutions, so don't worry about trying to get to the "right" one. The important quality to demonstrate is that you can build up a solution by breaking a problem down into components and devising a strategy to solve each part.

The interview

Once your application has been reviewed, you might be offered one or more interviews, which will be your opportunity to convince the school that you will work hard, study hard, and interview well when it comes time to apply for a job. Your interviewer is looking for why you want to go to code school, what got you interested in programming, and why you chose this school in particular. They want to know how likely you are to succeed at that code school and in the job you get afterward, because your success is their success.

Don't forget that the interview is also your opportunity to ask questions about the school. Find out who your interviewer is in relation to the school: Are they a former student (Hackbright Academy calls on alumni to help vet the next class of candidates), a teacher, or a director? Each of these roles will give you a different perspective on what you can expect from your code school experience. Former

students can tell you about the learning environment and share how they came to code school and how that has contributed to their success. Teachers can give you a clear idea of the content you will be learning. And directors and founders can share more about their mission, ideals, and history.

Ask questions about drop out rates, conversion rates (how many students attend the school after they're accepted), and application rates (how many people apply and how many are admitted). You'll be able to tell when you are getting canned marketing responses if you've done your homework about the school before hand.

Chapter 6: How to Succeed at Code School

Code schools are six or more weeks of high-intensity learning. Even if you have coding experience, you will struggle. There will be frustration. There will be self-doubt. There will be moments of despair. Bootcamp is hard.

There will also be incredible moments of "aHA!" that will come after every struggle. Each new difficult concept you conquer will drive you on to the next one, until you are riding a sine wave from frustration to elation and back again. Welcome to programming! This chapter outlines several strategies that will help you succeed.

Work hard

There are no shortcuts to learning how to be a developer. Code school is a full-time day job and a full-time after-hours job. Expect to spend 70+ hours studying every week for the duration of the program. There are lectures and group work by day and homework and research on nights and weekends, all followed by an arduous job search. You will not have much time to rest and recharge, so come to school with strategies for how to focus and how to steal moments of rest. Take breaks during the day, eat healthy food that will sustain you, and drink a lot of water. Code school is not a sprint, it's a marathon.

Keep everything else as stable as possible

As with all things, you get out of code school what you put into it, but you can improve your odds of success by minimizing common sources of stress that can distract you from your studies. If you're moving to a new city to attend a bootcamp, you will have the additional strain of navigating a new environment and being away from home and family. Sometimes this can't be helped, but bootcamp can be a crash course in frustration and self-doubt. Having a stable support system will help you focus on what you need to learn. Avoid any additional major changes in your life that will distract you from your ability to focus.

Learn how and where to ask for help

Learning how to Google effectively will usually get you un-stuck. For particularly tricky problems, you may need to ask for help. Before you ask a question, make sure you've done your research. One way to do this is to write a draft of your question.

A good question has four parts: summary, context, reproduction steps, and a recap of the investigation so far. Following this pattern is useful whether you are asking in person or online, and going through each stage will often lead you to discover gaps before you request someone else's time:

- ***Summary.*** One sentence that describes both your goal and the problem.
- ***Context.*** A brief description of why this is a problem.
- ***Reproduction.*** All the steps required to reproduce the problem, with code examples if they are relevant.
- ***Research to date.*** Don't ask someone a question without researching it first. You do not want to hear "Let me Google that for you."

Don't just tell someone you are stuck on a problem. Show them. Drafting a question with enough detail is even more important in online forums like StackOverflow and Google Groups, because your listener has no context for your problem. You have to get them up to speed before they can help you. Checkout StackOverflow for examples of how to draft a good question.

When you get the answer you need, follow up to learn how they found it. Was it from previous experience? Some documentation you were unfamiliar with? Is there any way you could have come to the answer on your own? The answer to your question is not as important as how to get to the answer, because you will be filling your toolbox with ways to solve similar problems in the future.

Have side projects

Not all developers spend every waking moment working on open source projects or inventing the next big framework. We all have lives, families, and other pursuits. However, experienced programmers likely have solid work-related contributions they can point to during an interview. First-timers need some proof that they can apply what they learn. Side projects show potential employers what you are learning and that you are interested in programming, not just the salary that comes with it.

Build your network

Like in all industries, a strong network is a huge asset when searching for a job in tech. But since you're just starting out you'll be relying heavily on your code school and your fellow students to land that first job. After that, your next job in programming is more likely to come from someone in your network than

from hitting the street with your resume. If you can claim an acquaintance with someone from a company you want to work for, that is one more point in your favor.

Go to meetups. If you are shy about meeting lots of new people at once like I am, start by setting a goal: extend your hand to one person you haven't met before. By making an internal bargain about what would make the evening a success at the outset, you can save yourself a lot of self-chastising afterward for not being more outgoing. Try introducing yourself to the invited speaker, if there is one -- they are usually well connected and respected in the community.

Get started long before your first day of code school

Applying for code school is a big decision. You are choosing a career that prizes the ability to learn above all other skills, and you are eager to get started. So don't wait until the first day to start your journey -- there is plenty that you can do to prepare in advance.

Build your online presence

You need a public image because the developer and recruiting communities rely heavily on online communication tools. You may choose to maintain different accounts to keep your professional and private spheres separate. Keep in mind that a hiring manager is going to dig deeper than the professional persona you present, and it's not difficult for a determined and even mildly tech-savvy person to find the forums, Instagram, and Twitter accounts that you contribute to.

Choose a username

Think carefully before choosing a username. Try not to pick one that obscures your identity or would be embarrassing for a classmate to tweet -- the goals are visibility and professionalism, not anonymity. Don't pick a username that's too long or too difficult to spell.

Choose a photo

Choose a photo for your avatars. You are going to school for three months, but it's all about building toward a new career. While a cartoon character may be acceptable on some social media sites, it's discouraged for a more professional site like LinkedIn.

Set up your social networks

LinkedIn is a social network for business professionals that programmers use for keeping in touch with co-workers. Hiring managers and recruiters are also active users, and they will likely scan your profile. You can also use LinkedIn to research the company you're applying to and the hiring manager who will interview you.

Twitter is for connecting with new acquaintances, or developers you admire and haven't met yet. Keep your professional goals in mind when you engage in public conversations -- your thoughts and interactions are just as available to potential employers as they are to your followers.

Stack Overflow is a Q&A knowledge repository. Ask a question on StackOverflow, and you will very quickly get a response from someone who has the domain knowledge you need. While an account is not required to access the knowledge on the site, you need a profile before you can ask questions, vote for good responses, or identify a solution. Earn badges and extra privileges as you become more experienced on the site, and begin answering questions yourself.

GitHub is the most popular place for developers to collaborate. It's an online repository for code that uses Git version control to track changes. One of the great advantages of GitHub is the ability to create a pull-request, which is a workflow for proposing changes to some code that is stored online. This means that you can browse through thousands of code bases and make improvements. There are also systems for tracking tasks, inviting collaborators, and writing documentation. All of your practice code should be saved on GitHub.

If you're nervous about putting your code out on the web for everyone to see, put those fears to rest. You are not the first to save rough code on GitHub, and until you are being actively evaluated by hiring managers, no one will be paying attention. Don't be paralyzed by perfection -- more activity is better than perfectly clean code. Before you begin sending out job applications, however, clean up a few of your repositories and make sure they are at the top of your list. Include a README file that describes each program's purpose, usage, and installation.

Create a portfolio

Your portfolio is your personal landing page and your opportunity to showcase your growth as a developer. Whether you choose WordPress, Blogger, or GitHub pages, you will need a place to park your online presence.

Pick a simple but modern-looking theme, and make sure the layout is readable on mobile devices. At minimum include your picture, email, resume, a short personal statement, and links to your GitHub, Twitter, and LinkedIn accounts. Your personal statement is a brief section about who you are, what drew you to learn web development, and what sets you apart from other candidates.

You should also include a page to advertise the projects you're working on. A brief description and a link to the code on GitHub will be enough for any potential employer. Also link to websites you have built, and use images wherever you can.

Start blogging about what you are learning

Consider adding a blog to your portfolio. Much of your time as a developer is spent communicating, not writing code, so before you skip over this section, here are some reasons why you should blog:

- Show that you are capable of learning technical content.
- Demonstrate communication skills to potential employers.
- Improve your own learning by teaching others.
- Keep a record of achievements.
- Display an interest in growing your skills and the skills of others.
- Draw attention to your online presence and availability to employers.
- Help someone who is struggling with a similar problem.
- Win fame and fortune, and maybe a movie deal.

Blogging dos

- Do write for an audience that has never programmed before.
- Do use the active voice.
- Do include images, code snippets, and pull quotes to break up visual monotony.
- Do give detailed context so that the reader can understand where you got stuck and how you broke through.
- Do write about aha moments. If you discovered a good analogy that helped you understand a difficult concept, share it with the world! Someone else is struggling with the same thing.

Blogging don'ts

- Don't worry about breaking new technical ground -- all you need is an understanding of a technical concept.
- Don't make it a blow-by-blow account of how hard you are finding code school, or life.
- Don't write a post that relies on another post for context. Make each post self-contained.
- Don't write a five-paragraph essay. But if you do, use pull quotes, bullet lists, and headings to make it easy to scan.

People don't read technical blogs as if they were novels. A user is likely to end up on a page of your blog because of a Google search for a particular problem. Make sure all the information required to understand the solution is available in the post. Your code school will promote you any way they can, so make it easy for them by providing content they can tweet about.

Start learning to code

Your code school will provide an aggressive curriculum for learning all the tools a web developer needs to get started, but there is room for getting a head start, and plenty of resources available. Ask for some pre-work if there is none assigned, or get started with online tutorials.

- Learn HTML and CSS.
- Get familiar with Git and source control.
- Learn the terminal on your computer.
- Improve your touch typing with Typing.io.
- Pick a text editor and learn it inside and out. Common editors are Sublime Text, Atom, Vim, and Emacs.

- Learn keyboard shortcuts and extensions that will improve your workflow. Try ShortcutFoo for practice with common editors and operating systems.
- Practice algorithm building: Pick a programming challenge and describe in words how you would determine a solution. Try Code Kata and Code Wars for community challenges.

Start networking

If your town has a tech community, find out where they meet and what they talk about. Get some business cards that include your name, as well as your Github, Twitter, and LinkedIn identities on them. They don't need to be fancy, and 50 or so will do. What you need is something to hand out at networking events that will keep you connected to recruiters, hiring managers, and other developers.

Learning to write web applications is not something you can accomplish in a few short months. So the more preparation you do before you go to code school, the more advanced you will be when you finish.

Chapter 7: Life After Code School

Congratulations! Completing a coding bootcamp is not a cakewalk -- you have worked hard and persevered for the last 10 or more weeks and are now ready to apply all that you've learned.

Writing applications, preparing for the interview, and building your network require different skills than the ones you have been focused on. Code school does not really prepare you for the question, "where do you see yourself in five years," because for the last five months all you have been thinking about are short-term learning goals. Learning the ins and outs of your new industry will begin with the job search, and you should be thinking about where you want to work even before code school is over.

What kind of company you might work for

Your future success will be determined by how much support you receive in the first months and years of your new career. While companies are constantly clamboring to hire more developers, some aren't a good fit for juniors because they don't have the resources to continue training them. It can be hard to know what you should look for in a company when your top priority is to land a job, but here are some guidelines that can help you find a good fit.

Goldilocks and the three job offers

This startup is too small

Startups are designed for growth, and they value innovation and risk. Working at a startup is a great opportunity for a junior because you will be thrown into the deep end of creative problem-solving. Of course, being thrown into the deep end means you will inevitably swallow a lot of water, and a startup does not have time to teach you how to swim.

Working for a startup has many advantages. You will get to perform many more varied tasks than you would at a more established company. You may wear the hats of a developer, a QA engineer, a DevOps engineer, and a user experience designer. You will participate in planning and design conversations that would be done behind closed doors in a more established company. There is a steep learning curve for wearing so many hats, but the more responsibility you take on the faster you will learn.

Startups are exciting workplaces. Building something the world has never seen before offers a universe of possibilities, but there's a lot of work to do to get a new business launched and generating revenue. To compensate you for the expected long hours, you are generally awarded some equity in the company in the form of stock options.

Options are not stock. Options are a promise that you will have the right to buy public stock at a lower-than market value. They are usually vested over a long period of time -- typically four years. If you are awarded 1000 stock options when you're hired, but leave after 3 years, you are only entitled to exercise 750 options, and the rest will remain with the company. Your options will only be converted to stock once the company is publicly traded. Until then, you may purchase your options, but you cannot liquidate them

without the permission of the company board of directors. There is a lot to learn about equity shares in startups, so don't count on an options award as a source of income or compensation for lack of other benefits.

While startups have their advantages, many don't have the time and resources to invest in junior developers. You will be responsible for keeping your head above water, and the compensation may not be worth the stress (unless they have a killer IPO, and then you will enjoy fabulous wealth).

This enterprise is too big

An established company has grown out of the "we have no revenue" desperation that comes with starting a new business. Thus, they can focus some attention on growing teams and individual careers. Working at a mature company means that you are likely to have a mentor who can help you define and work toward your career goals. You will also have opportunities to learn from developers with many different backgrounds and skill sets, which will grow your network quickly.

Your project will likely be limited in scope or confined to relatively mundane tasks that don't require the speed or cost of a more senior developer. You might find it difficult to grow out of that role, as you may not have many opportunities to prove you are capable of more complex tasks.

Some well-established companies will only hire untested junior developers into internships or contract-to-hire positions. These arrangements carry very little risk for the employer and can furnish you with some important professional experience. However, most of the advantages are on the employer's side: they may be less committed to your success, but if you persevere, they will have scored a trained developer. These positions usually pay an hourly wage and don't provide benefits. Some internships

are unlimited, meaning they have no ending date, so you could be stuck in intern-limbo until you can prove yourself worthy of a full developer's salary. In the meantime, you will be expected to perform to a developer's standard without the compensation, and in competition with other interns instead of in a co-learning environment.

Enterprise companies have the time and resources to train junior developers, and you don't have to worry about them closing up shop overnight. On the other hand, you may have to take an internship to get your foot in the door, and you could run into bureaucratic processes that hinder your advancement.

This mid-size company is just right

A large company will have the resources to support a junior developer, while a startup will stretch your capabilities faster and farther than you can imagine. Often, a mid-size company can provide the advantages of both. They retain an air of agility without the threat of financial ruin after every quarter, and they have resources available (like an HR department) to guide you through technical and professional difficulties. A mid-sized company can provide good compensation, benefits, and stock options, and they are often more willing to both hire and invest in newer developers.

If you have the luxury of choice, I recommend taking your first programming job at a mid-size company. They are challenging, stimulating places to work and can provide you with the training and opportunities you need to advance your career.

What about freelancing or remote work?

The idea of being able to work from anywhere and to define your own schedule lures many people into the software industry. However, this path

doesn't provide the benefits of working with other developers, such as learning good habits that will help you in your career. Like working for a startup, working for yourself requires a great deal of self-driven learning. If you are interested in becoming a freelancer, first get the experience of working with a team of developers at a company, and build your portfolio and network for when you want to venture out on your own.

Whether you find yourself as an intern or a full-time developer, at a startup, mid-size, or enterprise company, be sure to take the driver's seat when it comes to your own learning. It's easy to become complacent doing repetitive tasks, so speak up if you're interested in learning something new. Ask your manager for opportunities for you to grow, and find a mentor if you can. There are many resources available -- be proactive.

The application process

The application and interview process for the average developer job takes three to six weeks, so it would be wise to start looking around before the end of your program. Some students in my class had job offers before they left code school, and some (myself included) endured several months before landing full-time employment. Others found internships, technical support roles, and management opportunities.

Tailor each application for the company and job you are applying for. Read the information on the company's website, especially their blog. It's an excellent resource for learning the type of culture the company is trying to develop, the market they are targeting, and the people they employ. Use the product if you can, and find out how their customers use it. Read help forums and the company documentation, and look for questions that are posted on Quora or StackOverflow.

Use the information you gather to craft a cover letter that expresses how your skill set will help them achieve their corporate mission. A cover letter is never optional. If the website for submitting your application will not allow you to upload a separate cover letter, include a personal statement at the top of your resume. It's important to share your story in prose that does not include bullet points.

Don't leave your previous work experience off your resume because you believe it's irrelevant to software development. If you've worked on a team, or have an aptitude for leadership and communication, these skills are important to share with your next employer. If you're using code school to change careers, you are already demonstrating the kind of intellectual curiosity and problem-solving

that a software developer needs to succeed. Include everything, and use the opportunity to display your soft skills alongside your new technical skills -- you will need them all.

The phone screen

Once your application has made it through the gears of the hiring machinery, your first interaction with your prospective employer will likely be a brief phone interview. They will arrange the time in advance, which will give you plenty of opportunity to prepare.

Review all of the research you've done on the company and spend an hour learning about your interviewer. Be prepared to ask some questions or you will seem like a candidate who wants just any job, not a candidate who wants this job. Your questions can be about the product you will be building (What does your team build? Who are your customers?), the team you will be working with (How many people do you manage?), and the interviewers themselves (Why do you like working for the company? How long have you been a manager?).

For more tips on how to nail the phone screen, how to make the most of your first 90 days on the job, and an insider's look into the life of a manager, pick up *Managing Humans* by Michael Lopp.

The code challenge

Some hiring managers will ask you to solve a code challenge before meeting you in person. It's important to read the challenge carefully, as your ability to follow instructions is being evaluated along with your code. Just like when you put together your application for code school, do more than is expected. Use all the best practices you've learned, including writing tests and detailed documentation about how to run your code. You may be asked for a technical phone screen as well.

The culture fit interview

Your next interview will be in person. You will have about an hour one-on-one with your future manager, and the goal will be to determine your culture fit. Culture fit is a mysterious qualitative criteria. It includes things like:

- Are you easy to get along with?
- Do your values align with the company's?
- Will you adapt to the processes and behaviors that have led to the company's success?
- How well do you handle stressful moments, such as a job interview?

Prepare for this interview deliberately. Start thinking of examples from your life that demonstrate your ability to see a project through to the end, where you had to overcome some difficulty, or where you demonstrated leadership or teamwork. Pull anecdotes from every area of your life, not just your career. Any hobby where you have scratched deeper than the surface will suffice, because you will become animated when you talk about it. This is the spark that your interviewer is looking for, so don't hide it because you think it's irrelevant to learning how to code.

The technical interview

This interview is about proving yourself capable of solving technical problems, not to find out if you already know the solution. For me, it was the most intimidating part of the application process, but there are a few ways to prepare for the gauntlet.

Practice problem-solving

There are many resources, such as CodeKata, LessThanDot, and TopCoder, to find small problems that you can use to prepare. Pick a problem every day and

outline the steps you would take to solve it. Indicate where you would need to look something up, where you would add a test, and where you would ask a clarifying question. Stay away from actual code during these exercises -- practicing problem-solving doesn't necessarily mean you have to come up with a solution, and the same goes for the technical interview. Your interviewer isn't looking for a solution (although it's a bonus if you arrive at one). They're looking for your ability to tackle problems that are unfamiliar to you, and for evidence you are a clear and logical thinker.

Practice asking questions

To implement a good solution, you must first understand the problem. Diving into code is the last step in problem-solving, not the first. Asking questions about the problem will demonstrate to the interviewer that context is important to your solution, and that you think about problems from multiple angles before you decide on the solution strategy.

Assumptions are a great place to start looking for questions. If it is an input/output problem, you could ask "Can I assume that the input is finite?" Your solution for an infinite stream of input could be very different from a static file that can be loaded into memory. If you are asked to build a widget on a web page, you could ask "How has the user arrived at this page? How long is the process that brought them here?" If it is the end of a very long process, you may consider building a more streamlined widget, using data from sources already collected, etc. Identify and question assumptions before diving into code. You will understand the problem better, and your interviewer will be impressed with your seeking the bigger picture.

Practice in front of other people

Thinking through a problem can be a non-linear process. Your thoughts may jump around to different parts of the code. You may leave off halfway down one

path because a different idea surfaces. Talking through a technical problem, on the other hand, requires more clarity and purpose. Arriving at a solution may require many stops and starts, and you will need to be able to communicate your rationale at every step. Some code schools will provide practice time to help you prepare for the technical interview, but even if yours doesn't, there's nothing stopping you from working with other students to simulate the interview experience as closely as possible.

Use the whiteboard

The technical interview is about proving you have strategies for tackling new problems. Being able to think through the steps you will take is only the beginning. Being able to communicate your strategy is what will help you succeed in the interview, and providing visual cues of the connections you are making will help. If you are given a whiteboard or a pad of paper, use it to illustrate your approach. You don't need perfect syntax, but you do need to be clear and concise.

Use the internet or use pseudocode

The internet is a vast knowledge repository that most professional developers can't do without. If you are asked to solve a programming challenge that requires producing working code, ask if you can use the internet. If the answer is no, and you get stuck on syntax, fall back to writing pseudocode.

Bring your own computer

Your technical interview will be some combination of coding and using the whiteboard, but you may also be asked to walk the interviewer through some of your own code. Bring your own computer. You could work on the one provided, but you have likely set up your development environment and keyboard shortcuts in a way that makes you comfortable. Your comfort with your own tools will

make your ideas flow easier and will be a point in your favor. You also have all of your current projects loaded locally, so you do not have to rely on a network connection to show off your code.

Some interviewers want to see the work you are most proud of. Make sure that you have a couple of pieces of code handy, and tailor them to the position you are interviewing for. Spend some time polishing a couple of projects by refactoring and including tests. Have your sample projects loaded and ready to launch when they are asked for.

There are many ways to run a technical interview. You may be asked to write code using a pencil and paper, a whiteboard, or a computer. It is important to practice problem-solving in each of these mediums, including how to move forward if you are stuck on one part of the problem. Practice verbalizing your approach -- even if you don't solve the challenge, having a plan to execute shows that you would get there eventually.

The full-day interview

The manager may have enough information after the phone screen, culture interview, and technical interview to make a decision about hiring you, but they will still need the team's input. If you're invited for a full day of interviews, it's a good sign that the manager believes you are a fit for the job. It also means that you have your work cut out for you. The full day will be a combination of culture fit and technical interviews with future team members, other managers, and human resources.

If your manager is a good communicator, you will know who you will be meeting in advance. Do your research about each of your interviewers. Find out how

long they have been with the company. Read their blog, and review their Twitter feed and LinkedIn history. Prepare for each encounter as you would for any other interview.

Pay close attention to your physical needs throughout the day. If you are being passed from one interviewer to another, ask for a couple of minutes to use the restroom, get a drink of water, or eat a granola bar. Don't be shy to take notes during the interview if it will help lighten your memory load. You will be evaluated anew by each team member, and it's hard to remain fresh after three hours of interviewing. Asking for a couple of breaks will help you be as collected during your last interview as you are during your first.

The job offer

Once you have survived the interview process, how do you evaluate the job offer? As a junior developer, you may not feel that you have much negotiating power, but you do. Your prospective employer is wagering on your potential as a developer, and it's important that you know your own worth.

Compare the compensation they are offering against the national and city averages (see The Rewards and check GlassDoor for the most current data). If your offer is on the lower end of the range, try to counteroffer for something more in the middle. If your employer is unwilling to budge, ask if they will cover all or part of your code school tuition. If they are not already shelling out the equivalent to your code school or a recruiter, they may be open to the idea.

There are many types of compensation a company can offer in addition to your salary. Do some research and compare the offer with averages around the city. If you are contracting with a non-public company, they may sweeten the deal with stock options. As mentioned before, options are not cash -- they are a promise

that you will have the opportunity to purchase stock at a price and a time to be determined. Options help employees feel like they are owners and that they are invested in the company's success. The most common pattern is a "one-year cliff and a three-year drip," meaning that a quarter of the options you are promised in your contract will belong to you after one year, and you will receive a little more every month. This is a great strategy for companies to insure their investment in you and to keep talented developers from leaving before they have received all of their options.

Don't think that the offer will be revoked if you take a day or two to accept it. The hiring manager would not extend you an offer unless they were certain you are a good fit, and they will be eager to move you through the hiring process and start your training. You have some power to negotiate, so take your time in evaluating the offer.

The salary boost that comes with a development career will not disappoint: before bootcamp, students reported earning an average of $53K, and that number increased to $76K after graduation. In addition, 15% of students received some form of tuition reimbursement for their code school. These job placement rebates can range from a few thousand dollars to the entire tuition, although that is becoming less common. Code schools will charge employers in their network who hire their graduates a recruiting bonus, up to 20% of the student's first year salary. Keep this in mind as you negotiate.

Beyond the job offer

After you land your first job, there will be no time to sit on your laurels. Every day will bring new challenges. Here are a few strategies to help you meet them.

Keep coding

There are many free resources you can use to continue your education after you leave code school. Coursera, EdX, and Stanford Online all offer courses in data theory, algorithms, and web development that you can use to expand your skill set. Work through tutorials, build hobby applications, contribute to open source software, and publish your code online. The software industry values the self-taught developer, and keeping your portfolio current is the only way to convince a future employer that you love coding enough to keep it up.

Do scary things

Not like skydiving or alligator wrestling. Write blogs, contribute to open source projects, start your own programming language, or become visible on social media. You don't need to be an expert to have an opinion. Don't stop taking risks after you land your first job, either. Managers want to see your curiosity and interest in solving hard problems, and they will endeavor to throw ever greater challenges your way.

Become an expert

Find an area of your work that people are avoiding and become an expert in that. Not only is tackling a difficult problem extremely satisfying, but you will become known as the person who can do that thing we all hate. Your peers will rely on your expertise. They'll praise your courage, and you'll impress your managers.

Pick the hardest ticket

Take the hardest task from the your team's to-do list. Don't sift through them until you find something you think you are familiar with. This is your best opportunity to stretch your skills. Start with research. When you get stuck, ask a more experienced developer for help. This will get you interacting with other teams and learning how to solve tough problems.

I work with a lot of people who don't have formal computer science degrees, and while their backgrounds vary, they all share a love of problem-solving and a willingness to work hard. Attending code school was the best decision I ever made. I moved from a field where I had limited earning potential and no opportunities for job growth to one where I am challenged every day. The message I am trying to communicate is to take responsibility for your own learning -- code school will bring you so far, and the rest is up to you.

Note

I've been asked why I haven't mentioned specifically the code school I attended. So here you go: I went to Portland Code School. But the PCS that exists today is very different from the one I attended. Midway through my program, both the director and the founder announced they would be leaving the school at the end of the session. The school reorganized and changed its offerings, and the program has evolved over the last two years. I haven't kept tabs on what is happening at PCS, so I can't comment on the current curriculum or community.

My goal in this book is to share my code school experience and provide advice and resources for prospective students, not to endorse or criticize any particular code school.

Appendices

Appendix A: What It's Like to Be a Developer

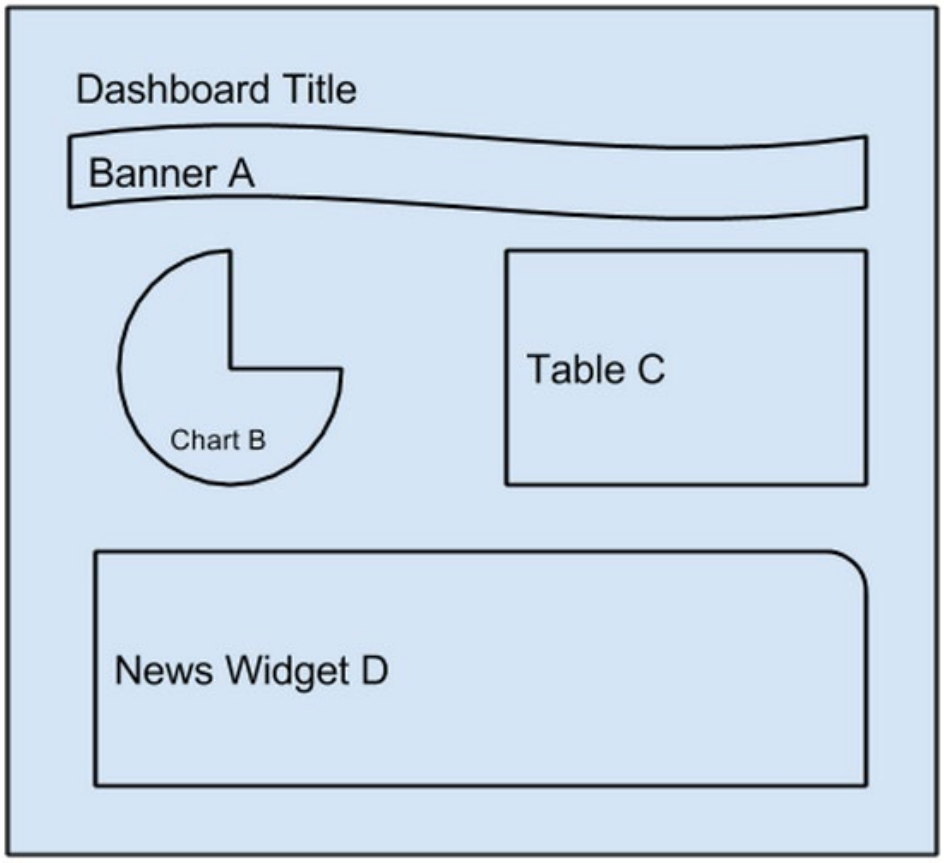
Before you can decide whether code school is right for you, decide if being a developer is right for you. Unless you have experience, or a network of friends already in the tech sector, it's difficult to discover what the industry is like on the inside. Gone are the days of lone-wolf coders who sleep all day, code all night, and work alone in a dark room. The day-to-day life of a modern software developer requires a lot of collaboration. Here's an example of a what your day might look like when you work in one of these ultra-collaborative companies.

On your first day as a new software developer at AwesomeWidget Company, your inbox is flooded with a dozen meeting invitations. The first meeting of the day is Sprint Planning.

> ***Sprint planning*** is a weekly meeting where the team gets together with managers to prioritize tasks for the week. It's a part of Agile software development where work is divided into chunks called sprints. In the meeting you look back on the previous sprint and either carry over tasks you didn't accomplish or re-prioritize based on changes that have arisen in the meantime.

In your first sprint planning meeting, you meet the other members of your team as well as the project stakeholders. You learn that the team is charged with

building a new dashboard for the executives of the company. They already have the design of the user interface, and it's your team's job to break it out into stories.



> ***Stories*** are a way to describe the tasks required to complete a project. They should be a small unit of work that has clear requirements and acceptance criteria.

Your team decides on five stories to complete the dashboard:

1. Add the page to the current website and a link from the home page
2. Add a banner to the page
3. Build the Chart widget
4. Build the Table widget
5. Build the News widget

The stakeholders decide that the chart and news widgets are more important than the banner and table widgets, and increase their priority:

1. Add the page to the current website and a link from the home page
2. Build the Chart widget
3. Build the News widget
4. Add a banner to the page
5. Build the Table Widget

Your team can finish about three stories per sprint, so #4 and #5 are put into the backlog for the next sprint. After sprint planning, you follow your teammates directly into a standup meeting.

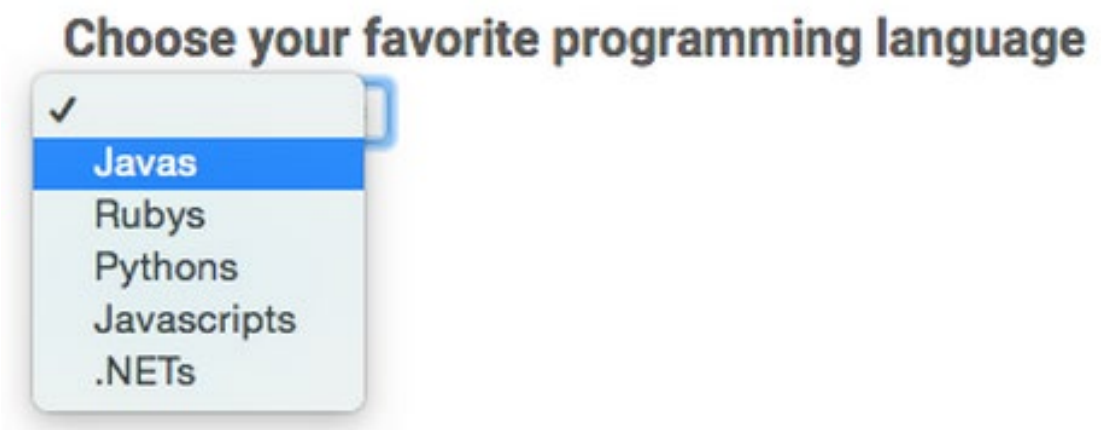
> ***Standups*** happen daily and last around 10 minutes. You each take a turn sharing what you worked on yesterday, what you are working on today, and whether you are blocked by anything.

Your manager begins the standup by welcoming you again to the team. She describes her last work day (several design meetings about the new dashboard and scheduling for the coming quarter) and what she expects to accomplish today (entering the sprint tasks into project management software, interviewing another new hire). Everyone takes their turn sharing. After standup, the team disperses to their workstations to get started on the current round of tasks for the sprint.

Since this is your first day, and you are unfamiliar with the code base, your manager suggests you take a look at the bug backlog and see if you can fix a few. Unexpected failures or results from a piece of software are called ***bugs***. Fixing bugs is like being a detective: you gather clues, follow leads, and occasionally end up lost. A single line of code can hold multiple bugs, and since codebases are

often hundreds of thousands of lines of code, software developers spend more time troubleshooting and maintaining existing code than writing new code. Bug hunting is a task you should come to enjoy.

The first bug in the list looks like an easy one: the text in a dropdown selector is improperly pluralized:



You find the place in the code that's generating that dropdown and fix the pluralization. Your manager tells you that it is AWC policy that all new code must be approved by at least one developer, so you seek out one of your teammates to help you through your first code review.

> ***Code reviews*** can be formal or informal. They allow your team to gain consensus on style, design, performance, and testing. Your code design decisions will have an impact on the project as a whole, so it's a good policy to always get feedback from a teammate.

Quality Assurance engineers can be engaged at every phase of the development process. They may use a combination of automated and manual tests to verify fixes and to make sure that no changes are introduced that will break other

parts of the product. Once your teammates have approved your changes, you need to get the bug fix into the next release.

> ***Deploying*** new code is a different process at every company. The application may need to serve thousands of requests per second, and there are limits to hardware capacity. Web applications are deployed frequently, and there are software tools to make the deploy process easy.

Now that your bug has been fixed, verified, and deployed, you can prepare to demonstrate it for stakeholders at the end of the sprint.

> ***Demos*** occur on a weekly or bi-weekly basis. They are your opportunity to share the week's accomplishments. While many stakeholders are involved in the project and sprint planning, they are often disconnected from the work as it's being performed. Regular demos help to keep them in the loop, allow for design changes, and provide an opportunity for you to explain both progress and unexpected delays.

Daily life as a software developer involves a lot of collaboration. There are many levels of design, technology, product, and architecture decisions that should not be made without consensus. The job is far from routine because of the diverse set of problems to solve, but there is a regular cycle of sprint planning, code review, and demonstrations.

Appendix B: Glass Ceilings

Interns and junior developers are often given simple, mundane tasks in web development. Things like code maintenance and support aren't as sexy as building new products, but they are the first rungs on a career ladder that you can climb to greater technical challenges. You may be surrounded by successful developers from various educational backgrounds, but you should still be prepared to hit your head on a few glass ceilings along the way.

One of those ceilings is the barrier between technical support and software development. Some companies use their support staff as farm teams for engineering. By the time you are ready to dive into code, you will have spent one or two years pairing with developers and learning the product inside and out. This is a good model for growth within a company, but not for moving between companies. The longer your tenure in technical support, the less likely you are to convince a new company of your engineering abilities. Systems administrators have encountered similar barriers to breaking into software development, as their specialized skills tend to pigeonhole them into filling a particular kind of role.

Moving from an internship into a full-time developer role is another ceiling that can be difficult to break through (it even has its own name, the "intern glass ceiling"). Internships have become the "new interview," and while some companies are looking to find full-time employees, others employ interns as part-time workers for short-term projects. Many recent graduates are bouncing from internship to internship, and some choose to settle for jobs that are adjacent to their desired position. If you continue to perform only low-complexity tasks in one internship after another, future employers may question your desire and ability to take on further responsibility.

Similar to the internship ceiling is the mountain of credibility lying between junior developer and senior developer. Senior developers are capable of demonstrating leadership on the team. They should be expert communicators who work well with management, as well as mentors for less experienced developers. Technically, they can design robust software and anticipate difficulties at scale. With three to five years of experience, many developers with a degree in computer science will be considered senior. Regardless of experience leading up to your career in software, as a code school graduate you will need to demonstrate proof of these qualities a little longer and a little louder than your degreed peers.

One of the ideals of the tech community is that software development is a meritocracy and that your capabilities will be judged based on your skills and experience alone. While code schools are feeding the industry's desperate need for skilled developers, there is some very public debate about whether the code school shortcut is good for the industry. In his Slate article "'Everybody' does not need to learn to code," software developer Chase Felker writes: "Frankly, just the idea that you can learn to code in a year gives me the creeps: I would be terrified if someone with only a couple of classes were writing programs for me," mainly because of all the context they are missing from not having a computer science degree. Some companies feel that a "three-month course cannot possibly provide the engineering fundamentals required for being a productive member of a development team."

As you gain experience in the industry, you will encounter these attitudes more frequently. You can combat this prejudice in a couple of ways. You can find an ally who will champion your cause and bring your accomplishments to light, or you can look for opportunities on a more progressive team or company. If you're experiencing a general attitude of exclusion toward interns or junior developers, it might be a good idea to take what experience you have and find employment elsewhere.

Appendix C: Impostor Syndrome

There is a lot of writing on the web about impostor syndrome, which is the feeling that you are unqualified for, or incompetent at, your job, and that it's only a matter of time before you are discovered to be a fraud. Learning to code in a bootcamp is a struggle. When you get your first job and begin working with more experienced developers, it's easier to imagine that programming just comes naturally to them. Don't forget that all developers had to pass through the same lessons you learned. The difference between you and someone you look up to as an expert is just experience, not intelligence or natural ability.

Alicia Liu writes of her experience in "Overcoming impostor syndrome" on her Medium blog. "Unlike learning other skills where one can expect to be reasonably competent after sufficient practice, programming largely consists of constantly failing, trying some things, failing some more, and trying more things until it works." Before I began working in the industry I thought that I would have to learn everything I needed to know and then I would spend the next chunk of my career applying what I learned. It didn't take long to catch on that even the most senior developers still use the docs for simple operations. They know how to find the information they need, and they are expert experimenters.

Code school students are prone to feeling impostor syndrome because of the nature of the education we chose to pursue. We took a shortcut into engineering, bypassing the traditional degree route that lends legitimacy to any job application. It's important to remind yourself that graduating from a coding bootcamp is the start of the journey, not the end. Your education will never be over. Give yourself permission to be a beginner, keep a record of what you learn, and take time now and again to reflect on how far you have come since you first decided to learn to code.

Appendix D: What Else is Out There Besides Being a Developer?

Learning to program at a code school is a great opportunity to acquire new skills and to make your current skills relevant in tech. Becoming a software developer is the most common goal, but there is a wide variety of roles that technical people can play in software that aren't about writing code. Whether you left a job in customer service, teaching, or research, the skills you gained in your last career will help you fill a niche in your new one. Here are a few paths in tech that don't involve writing software, but may require some experience with coding, frameworks, and methodologies.

Product manager

Code School + Business Management

As a product manager (aka product owner), you decide what features to build by studying the market and learning your customers' needs. The result of your research is a product roadmap and business strategy for the next year and beyond. Experience in the field is helpful for making good business decisions.

Project manager

Code School + Project Management

A project manager follows the product road map feature by feature. In this role you will be responsible for marshaling resources and delivering products. Excellent communication skills are a must, as is a technical familiarity with the system under construction. You need to ask the right questions, identify risks, validate estimates, eliminate blockers, and keep everyone informed of progress.

Engineering manager

Code School + Project Management + People Management

High-functioning teams do not manifest by chance -- they are the result of deliberate choices made at the intersection of project needs, culture fit, and career growth. As an engineering manager, you don't just assemble the team. You also schedule projects, implement (or remove) processes, grow careers, and communicate up and down the management chain. In addition to all of these tasks, you must be familiar with the job the team is performing.

DevOps

Code School + An Entirely Different Stack

Every developer should know DevOps tools. You will not only manage hardware and configuration, but user access, capacity planning, deployments, and databases. You need to learn many different software tools for distributing code and monitoring server processes. Your HTML and CSS skills will not be useful in a DevOps role, but you will be the one responsible for keeping the app running as the company grows.

Not everyone leaves code school with a passion for building software, but everyone should leave with new skills they can apply to an awesome career in tech. No education or experience is wasted, so don't forget to leverage your current skill set as you are learning a new one.

Resources

Books

- *Learn to Program*, by Chris Pine
- *New Programmer's Survival Manual*, by Josh Carter
- *Managing Humans*, by Michael Lopp
- *Team Geek: A Software Developer's Guide to Working Well with Others*, by Brian W. Fitzpatrick and Ben Collins-Sussman
- *JavaScript: The Good Parts*, by Douglas Crockford
- *Ruby on Rails Tutorial*, by Michael Hartl
- *Test-Driven JavaScript Development*, by Christian Johansen

Online tutorials

HTML, CSS, Javascript

- W3Schools
- Codecademy
- Treehouse

Ruby

- Try Ruby
- Ruby Koans
- Ruby Monk

Rails

- Rails for Zombies
- Rails Tutorial

Python

- Learn Python the hard way
- LearnPython

Command line tools

- Git immersion
- Learn code the hard way
- Windows command prompt tutorial
- ShortcutFoo

Debugger Tools

- Chrome dev tools

Acknowledgements

Thank you for purchasing the book, and taking the time to read about all the people who contributed to making it happen. Producing a book, even an ebook, takes a village. I have talked with many aspiring developers, code school graduates, and hiring managers, and have had many mentors along the way. Many thanks to Merlyn Albery-Speyer, Katherine Wu, Rebecca Campbell, and Chuck Lauer Vose for their mentorship. To Kate Morrow and Maureen Dugan for being my first readers and early supporters. Thanks to Jóhann Hannesson and Jonan Scheffler for their ideas and insight. Finally, thanks to Krista Garver -- my editor, and partner in all adventures.