



## Software Engineering: Crash Course Computer Science #16

Crash Course: Computer Science

<https://youtube.com/watch?v=O753uuutqH8>

<https://nerdfighteria.info/v/O753uuutqH8>

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science! So we've talked a lot about sorting in this series and often code to sort a list of numbers is only ten lines long, which is easy enough for a single programmer to write. Plus, it's short enough that you don't need any special tools - you could do it in Notepad. Really!

But, a sorting algorithm isn't a program; it's likely only a small part of a much larger program. For example, Microsoft Office has roughly 40 million lines of code. 40 million! That's way too big for any one person to figure out and write!

To build huge programs like this, programmers use a set of tools and practices. Taken together, these form the discipline of Software Engineering - a term coined by engineer Margaret Hamilton, who helped NASA prevent serious problems on the Apollo 11 mission to the moon.

She once explained it this way: "It's kind of like a root canal: you waited till the end, [but] there are things you could have done beforehand. It's like preventative healthcare, but it's preventative software."

(Crash Course Theme)

As I mentioned in episode 12, breaking big programs into smaller functions allows many people to work simultaneously. They don't have to worry about the whole thing, just the function they're working on. So, if you're tasked with writing a sort algorithm, you only need to make sure it sorts properly and efficiently.

However, even packing up code into functions isn't enough. Microsoft Office probably contains hundreds of thousands of them. That's better than dealing with 40 million lines of code, but it's still way too many "things" for one person or team to manage.

The solution is to package functions into hierarchies, pulling related code together into "objects." For example, car's software might have several functions related to cruise control, like setting speed, nudging speed up or down, and stopping cruise control altogether. Since they're all related, we can wrap them up into a unified cruise control object.

But, we don't have to stop there. Cruise control is just one part of the engine's software. There might also be sets of functions that control spark plug ignition, fuel pumps, and the radiator. So we can create a "parent" Engine object that contains all of these "children" objects.

In addition to children objects, the engine itself might have its own functions. You want to be able to stop and start it, for example. It'll also have its own variables, like how many miles the car has traveled. In general, objects can contain other objects, functions, and variables. And of course, the engine is just one part of a Car object. There's also the transmission, wheels, doors, windows, and so on.

Now, as a programmer, if I want to set the cruise control, I navigate down the object hierarchy, from the outermost objects to more and more deeply nested ones. Eventually, I reach the function I want to trigger: Car, then engine, then cruise control, then set cruise speed to 55.

Programming languages often use something equivalent to the syntax shown here. The idea of packing up functional units into nested objects is called Object Oriented Programming.

This is very similar to what we've done all series long: hide complexity by encapsulating low-level details in higher-order

components. Before we packed up things like transistor circuits into higher-level boolean gates. Now, we're doing the same thing with software. Yet again, it's a way to move up a new level of abstraction.

Breaking up a big program, like a car's software, into functional units is perfect for teams. One team might be responsible for the cruise control system, and a single programmer on that team tackles a handful of functions.

This is similar to how big, physical things are built, like skyscrapers. You'll have electricians running wires, plumbers fitting pipes, welders welding, painters painting, and hundreds of other people teeming all over the hull. They work together on different parts simultaneously, leveraging their different skills, until one day, you've got a whole working building.

But, returning to our cruise control example: its code is going to have to make use of functions in other parts of the engine's software to, you know, keep the car at constant speed. That code isn't part of the cruise control team's responsibility. It's another team's code.

Because the cruise control team didn't write that, they're going to need good documentation about what each function in the code does, and a well-defined Application Programming Interface, or API for short. You can think of an API as the way that collaborating programmers interact across various parts of the code.

For example, in the Ignition Control object, there might be functions to set the RPM of the engine, check the spark plug voltage, as well as fire the individual spark plugs. Being able to set the motor's RPM is really useful. The cruise control team is going to need to call that function, but they don't know much about how the ignition system works.

It's not a good idea to let them call functions that fire the individual spark plugs, or the engine might explode! Maybe. The API allows the right people access to the right functions and data.

Object Oriented Programming languages do this by letting you specify whether functions are public or private. If a function is marked as "private", it means only functions inside that object can call it. So, in this example, only other functions inside of Ignition Control, like the setRPM function, can fire the spark plugs. On the other hand, because the setRPM function is marked as public, other objects can call it, like Cruise Control.

This ability to hide complexity, and selectively reveal it, is the essence of Object Oriented Programming, and it's a powerful and popular way to tackle building large and complex programs. Pretty much every piece of software on your computer, or game running on your console was built using an Object Oriented Programming language, like C++, C#, or Objective-C. Other popular "OO" languages you may have heard of are Python and Java.

It's important to remember that code, before being compiled, is just text. As I mentioned earlier, you could write code in Notepad or any old word processor. Some people do. But generally, today's software developers use special-purpose applications for writing programs, ones that integrate many useful tools for writing, organizing, compiling and testing code.

Because they've put everything you need in one place, they're called Integrated Development Environments, or IDEs for short. All IDEs provide a text editor for writing code, often with useful features like automatic color-coding to improve readability. Many even check for syntax errors as you type, like spell check for code. Big programs contain lots of individual source files, so IDEs allow



## Software Engineering: Crash Course Computer Science #16

Crash Course: Computer Science

<https://youtube.com/watch?v=O753uuutqH8>

<https://nerdfighteria.info/v/O753uuutqH8>

programmers to organize and efficiently navigate everything.

Also built right into the IDE is the ability to compile and run code. And if your program crashes, because it's still a work in progress, the IDE can take you back to the line of code where it happened, and often provide you additional information to help you track down and fix the bug, which is a process called debugging.

This is important because most programmers spend 70 to 80% of their time testing and debugging, not writing new code. Good tools, contained in IDEs, can go a long way when it comes to helping programmers prevent and find errors. Many computer programmers can be pretty loyal to their IDEs, though - but let's be honest, VIM is where it's at, providing you know how to quit.

In addition to coding and debugging, another important part of a programmer's job is documenting their code. This can be done in standalone files called "read-me's" which tell other programmers to read that help file before diving in.

It can also happen right in the code itself with comments. These are specially-marked statements that the program knows to ignore when the code is compiled. They exist only to help programmers figure out what's what in the source code. Good documentation helps programmers when they revisit code they haven't seen for awhile, but it's also crucial for programmers who are totally new to it.

I just want to take a second here and reiterate that it's the worst when someone parachutes a load of uncommented and undocumented code into your lap, and you literally have to go line by line to understand what the code is doing. Seriously. Don't be that person.

Documentation also promotes code reuse. So, instead of having programmers constantly write the same things over and over, they can track down someone else's code that does what they need. Then, thanks to documentation, they can put it to work in their program, without ever having to read through the code. "Read the docs," as they say.

In addition to IDEs, another important piece of software that helps big teams work collaboratively on big coding projects is called Source Control, also known as version control or revision control. Most often, at a big software company like Apple or Microsoft, code for projects is stored on centralized servers, called a code repository. When a programmer wants to work on a piece of code, they can check it out, sort of like checking out a book out from a library.

Often, this can be done right in an IDE. Then, they can edit this code all they want on their personal computer, adding new features and testing if they work. When the programmer is confident their changes are working and there are no loose ends, they can check the code back into the repository, known as committing code, for everyone else to use.

While a piece of code is checked out, and presumably getting updated or modified, other programmers leave it alone. This prevents weird conflicts and duplicated work. In this way, hundreds of programmers can be simultaneously checking in and out different pieces of code, iteratively building up huge systems.

Critically, you don't want someone committing buggy code, because other people and teams may rely on it. Their code could crash, creating confusion and lost time. The master version of the code, stored on the server, should always compile without errors and run with minimal bugs. But sometimes bugs do creep in.

Fortunately, source control software keeps track of all changes, and if a bug is found, the whole code, or just a piece, can be rolled back to an earlier, stable version. It also keeps track of who made each change, so coworkers can send nasty, I mean, helpful and encouraging emails to the offending person.

Debugging goes hand in hand with writing code, and it's most often done by an individual or small team. The big picture version of debugging is Quality Assurance testing, or QA. This is where a team rigorously tests out a piece of software, attempting to create unforeseen conditions that might trip it up. Basically, they elicit bugs. Getting all the wrinkles out is a huge effort, but vital in making sure the software works as intended for as many users in as many situations as imaginable before it ships.

You've probably heard of beta software. This is a version of software that's mostly complete, but not 100% fully tested. Companies will sometimes release beta versions to the public to help them identify issues. It's essentially like getting a free QA team.

What you don't hear about as much is the version that comes before the beta: the alpha version. This is usually so rough and buggy, it's only tested internally.

So, that's the tip of the iceberg in terms of the tools, tricks, and techniques that allow software engineers to construct the huge pieces of software that we know and love today, like YouTube, Grand Theft Auto 5, and Powerpoint. As you might expect, all those millions of lines of code needs some serious processing power to run at useful speeds, so next episode we'll be talking about how computers got so incredibly fast. See you then.

Crash Course Computer Science is produced in association with PBS Digital Studios. At their channel, you can check out a playlist of shows, like Physics Girl, Deep Look, and PBS Space Time. This episode was filmed at The Chad & Stacey Emigholz Studio in Indianapolis, Indiana, and it was made with the help of all these nice people and our wonderful graphics team Thought Cafe. That's where we're going to have to halt and catch fire. See you next week.