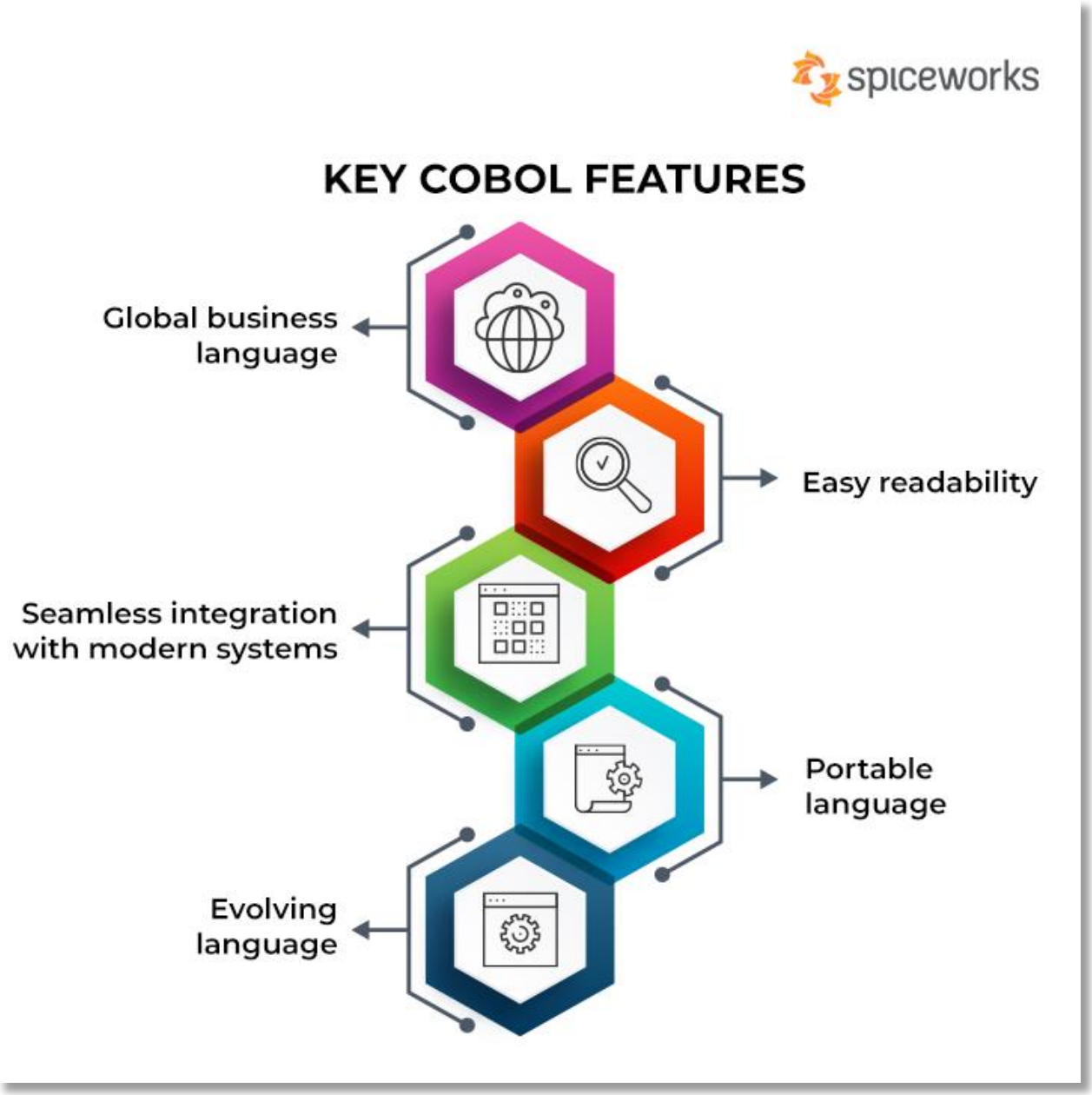
"The Code That Controls Your Money"
Published by Wealthsimple Magazine

MONEY & THE WORLD

NOTE: *This article is recreated here to be accessible to Americans as the original was only accessible in Canada… so, if you did not own a VPN app, you were unable to read it. I think it is a VERY good and VERY important article on COBOL.*

COBOL *(common business-oriented language) is defined as a standard programming language developed by a consortium CODASYL (Conference/Committee on Data Systems Language) in 1959 to support business and financial applications.*



The Code That Controls Your Money

Written By Clive Thompson on November 10, 2020

COBOL is a coding language older than Weird Al Yankovic. The people who know how to use it are often just as old. It underpins the entire financial system. And it can't be removed. How a computer language controls the financial life of the world.

When Thomas first started programming, it was 1969. He was a kid just out of high school in Toronto, without any particular life goal. His father was a carpenter, but good luck following in his family's footsteps; Thomas was all thumbs. "My father knew I couldn't hammer two pieces of wood together," he laughs.

So his mother suggested something weird and newfangled: What about… computer programming?

Computers, in 1969, were still strange new curiosities, the size of big cabinets. But companies around the world were realizing they were invaluable for any task that required a lot of rapid-fire accounting, like tallying up payroll. Jobs were on offer to anyone who could learn even a *little* coding. So Thomas found "some fly-by-night, little pop-up school" in downtown Toronto, and over the next two months, learned the hot computer language of the day: COBOL (Common Business-Oriented Language).

After he graduated, he got hired in the check-sorting department of a major Canadian bank. (He doesn't want me to name it, banks are secretive; "Thomas," I should mention, is a pseudonym, if you hadn't guessed that already.) Thomas wasn't yet a programmer for the bank then, but over the next few years he made it clear he wanted to be, and his employer paid for him to do a bunch of honest-to-goodness college courses in coding, and in 1978 he began a long career at the bank as a programmer.

Thomas loved it. It was like constant puzzle-solving, a game of mental chess. He'd sit at his desk, writing out his code by hand, then give it to a "punchcard operator" who'd put holes in cards to represent his programming instructions. Twice a day they'd feed those cards into the huge "mainframe" computers at the bank. It would take hours for Thomas to find out if his code had actually worked correctly, or whether he'd made a goof that grounded things to a halt. If he did, he'd pore over the error statements, rewrite the COBOL, and try again.

Over the next few years, Thomas became good at COBOL and wrote thousands of invaluable lines of code. When the bank issued payments, it was his code, every day, helping them tally it all up correctly. As the '70s and '80s and '90s wore on, he and his coder colleagues probably wrote tens of millions of lines of COBOL. There's one system he's particularly proud of, a lightning-fast program that can process "anywhere between three and five million transactions a day. That's my baby!" He wrote his first bits of that program in 1988.

And the thing is that the code is still running today.

Thomas retired from the bank in 2007 at about 60, and when he left, the bank was still relying on the system, which by then was 20 years old and written when Thomas had a lot more hair and when Phil Collins's "Groovy Kind of Love" was a chart-topping hit. These days, the code is over three *decades* old. It's still crunching millions of records a day. Indeed, he believes most of the code he and his peers wrote back in the day is still running because the bank can't function without it.

In fact, these days, when the phone rings in the house Thomas retired to — in a small town outside of Toronto — it will occasionally be someone from the bank. *Hey*, they'll say, *can you, uh, help… update your code? Maybe add some new features to it?* Because, as it turns out, the bank no longer employs anyone who understands COBOL as well as Thomas does, who can dive in and tweak it to perform a new task. Nearly all the COBOL veterans, the punch-card jockeys who built the bank's crucial systems way back when, who know COBOL inside and out — they've retired. They've left the building, just like Thomas. And few young coders have any interest in learning a dusty, 50-year-old computer language. They're much more excited by buzzier new fields, like Toronto's booming artificial intelligence scene. They're learning fresh new coding languages.

So this large bank is still dependent on people like, Thomas, who is 73, to not only keep things running, but add new features and improvements.

Will his COBOL outlive him?

"Probably."

> **COBOL democratized coding. Companies could take everyday people and train them to be useful COBOL programmers in a few months.**

That bank is not alone. COBOL programs — some written so long ago that colour TV wasn't even a thing yet — **are everywhere in our daily lives.**

CONSIDER: Over 80% of in-person transactions at U.S. financial institutions use COBOL. Fully 95% of the time you swipe your bank card, there's COBOL running somewhere in the background. The Bank of New York Mellon in 2012 found it had 112,500 individual COBOL programs, constituting almost 350 million lines; that is probably typical for most big financial institutions. When your boss hands you your paycheck, odds are it was calculated using COBOL. **If you invest, your stock trades run on it too. So does health care: Insurance companies in the U.S. use "adjudication engines'" — software that figures out what a doctor or drug company will get paid for a service — which were written in COBOL.** Wonder why, when you're shopping at a retailer you will see a clerk typing into an old-style terminal, with green text on a black background? It's because the inventory system is using COBOL. Or why you see airline booking agents use that same black screen with green type to change your flight? "Oh, that's COBOL — that's *definitely* COBOL," laughs Craig Bailey, a senior engineer at Faircom, a firm that makes software to help firms manage those old systems.

No one quite knows how much COBOL is out there, but estimates suggest there are as many as 240 billion lines of the code quietly powering many of the most crucial parts of our everyday lives. **"The second most valuable asset in the United States — after oil — is the 240 billion lines of COBOL," says Philip Teplitzky, who's slung COBOL for decades for banks across the U.S.**

We're often told that tech thrives because of its new, pioneering innovations — its willingness to do bold new things with code, to "move fast and break things," as a young Mark Zuckerberg famously plastered on the wall at Facebook. And it's certainly true that every day we see wild new code released, written in fresher, newer languages. If you've seen that crazy new AI that can write sentences like a human, its creation relied on Python, the well-known new computer language. When Facebook unleashes some new features on its browsers' app, the coders are often using JavaScript, another hot one.

But in older, massive industries central to the economy? COBOL's still omnipresent. It makes it hard to innovate. How can you tinker, bolt on new features, using an ancient language that energetic young coders have no interest in? If big old banks aren't the firms pushing forward with services like Venmo or Square or other fizzy "fintech" products, it would follow that COBOL is part of the problem. But if that's the case why, exactly, is Thomas still being dragged out of retirement to keep it alive? Why can't we do without it?

It's partly because COBOL got there first — and was a tool fit perfectly for its task. COBOL was, in many ways, the spark that lit our modern computer age.

Programmers began devising COBOL in 1959. When it was finally released ten years later in 1969, it was the first language to make computers widely useful for everyday life. In the late '50s, computers had just left the "experimental" stage. Everyday companies had begun pondering whether having their own computer to crunch numbers could be valuable. The problem was, before COBOL came along, coding was cryptic and difficult to learn. Programmers often wrote software using some variant of what are called "assembly" languages, where the commands could be awfully abstruse. (For example, the command "**LXA A,K**" means "take the number loaded into location A of the computer's memory and load it into the 'index register' K.") Worse, computer makers often devised their own special languages for their computers. If you wrote some great code for a machine, it couldn't run on a computer made by another company.

A new generation of ambitious programmers thought this was crazy. One was Grace Hopper, a rear admiral in the U.S. Navy — who'd cut her teeth on an early experimental computer — and a firecracker of a personality. (She's the one who popularized the phrase: "It's easier to ask forgiveness than ask permission.") Hopper thought programming languages ought to more closely resemble English so that they'd be easier to learn and to read. In 1955, she devised a language called "FLOW-MATIC" that aimed to do just that; to move a number from location A to location D, for instance, you'd simply write "**TRANSFER A TO D**".

In 1959, a computer programmer named Mary Hawes decided her industry needed to devise a language that would be as easy to write as FLOW-MATIC, and one that could run on any machine. She assembled a committee of experts — including many from the nascent

business-computer industry — to start creating the language, working together under the Defense Department. The goal was to make a language that the average corporate manager at a company could read and understand, even if they weren't trained as a programmer.

That decade of work — heavily propelled by many female superstar contributors, such as the computer science pioneer Jean Sammet — produced a language, much like FLOW-MATIC, that was easy on the eyes. To add two numbers, for example, you could write "**ADD Num1, Num2 GIVING Result**". To run a calculation three times, you'd write "**PERFORM 3 TIMES.**"

"**It's really hard to overstate the importance of COBOL**," says Mar Hicks, associate professor of history at the Illinois Institute of Technology and author of Programmed Inequality. "It was doing something absolutely critical in computing. It was filling this niche that had gone unfilled in the early years of computing. And it changed the way that you could think about writing programs."

It changed *who* could write it, too. COBOL democratized coding; companies could take everyday people and train them to be useful COBOL programmers in a few months, and to become experts in a year or two. This was crucial given that companies desperately needed more warm bodies to write software.

"You could pick people up out of the street," says Jon Pyke, a British coder who learned COBOL back in the 1960s, "and basically and teach them how to do it."

> That older code can not only be good, but in crucial ways superior to newer code, is at odds with a lot of Silicon Valley myth-making.

The other thing about COBOL is that it was *fast*. It had been designed specifically to do mammoth amounts of "transactions" really quickly. If you're a retail chain, you need to count up your sales and recalculate your inventory every night. And you don't have much time to do it — perhaps a couple of hours in the evening, after your business day ends, while your computer staff works late.

Banks, too: During the day, they're frantically accepting transactions, requests from customers to take money in and out of their accounts. At night they have a few hours to balance all those books. If you've wondered why a check you've deposited won't clear for a while, it's partly because both banks need to run their mammoth COBOL jobs after the day staff has left. At Citibank, Teplitzky's code ran through a huge center with 248 mainframe computers.

"You have a six-, eight- hour window where you have to do, if you'll pardon the expression, a shitload of work — you have to do all transactions in a certain order," he tells me. "It takes big, big iron to run a billion transactions through a six-hour batch window. It's a screamer."

COBOL was optimized for precisely that task: processing gazillions of transactions. Computer languages often have a sort of cognitive or creative bias; they were each created with a particular type of task in mind. Python is excellent for data science and AI; Fortran was created to render math formulas in code; JavaScript was created to help programmers make websites interactive.

COBOL? It was customized for working on those mainframe computers, which themselves were designed specifically to crunch bazillions of transactions, reading and writing data streams at a brisk pace. It was like a high-octane fuel designed specifically for a sports car. Over the years, COBOL "compilers" — the software that takes the English-like syntax of computer code and transforms it into the ones and zeros that a computer chip can execute — were refined more and more, so that COBOL's "compiled code" became exceptionally speedy. Which means that part of the reason COBOL underlies so many crucial things we do is because it's actually pretty good at it.

"They've had 50 years to get this right," notes Bill Hinshaw, who runs COBOL Cowboys, an agency that provides COBOL programmers.

The sheer age of those COBOL systems is, oddly, actually something that works in their favour. Because they're old, they have been relentlessly debugged. When a program is first written, it inevitably has problems. Sometimes it's a typo, a misplaced command; other times, the user does something the programmer never expected, and things crash. When you get a new app, if it's buggy and crash-prone, this is why: the creators sent it out into the world with lots of these little flaws. **It can take days, weeks, or years to discover all the problems.**

But those COBOL programs that run the world? They've had decades for coders and users to uncover all the problems, and to fix them.

Adriana Stern (not a pseudonym this time!), another coder I spoke to who worked for large Canadian banks, started her career in the '80s, when the systems were still ironing out some odd bugs. One day she found that a particular bank terminal in Quebec was sending the system accented letters — and the original programmer had never expected that to happen.

"So when the system tried to interpret it, it would choke," she tells me. In another case, a different COBOL program kept crashing, and she finally realized it was because a new customer's name had a single quotation mark in it — which the program accidentally thought was an instruction saying "the end of the data set," grinding the code to a halt.

Stern worked for banks for 30 years, and she figures 85% of her work wasn't writing bold new features for the bank — it was "maintenance." Think of it like a sort of digital plumbing, fixing leaks, making everything run gradually more and more smoothly.

"It was hard work — you're burning the candle at both ends," she told me.

This is precisely why those COBOL systems are now so reliable. They've been debugged more than just about any code on the planet. **A fizzy new TikTok-style app can launch and enjoy massive popularity even with a lot of bugs. If the "like" count on your latest**

post is slightly wrong, *eh, no big deal*. In contrast, if a major retailer miscounts its inventory, or a bank suddenly can't send money? That causes financial chaos at scale.

"The entire GDP of the world is in motion in the [banking] network at any moment in time," as Teplitzky notes. "A bank turns over twice its assets every day, out and in. A clearing bank in, say, New York, it could be more… So a huge amount of money is in motion in the network and in big backend systems that do it. They can't fail! If they fail, the world ends. *The world ends.*"

COBOL is not merely fast; it's also "stable, stable, stable", as Thomas tells me. **One of the processes he developed takes, every month, a file of about 2.4 million government pension and puts the proper amounts in people's bank accounts. "We verify them and check them in 11 minutes. It hasn't failed in 20 years."**

This idea — that older code can not only be good, but in crucial ways superior to newer code — is at odds with a lot of Silicon Valley myth-making. **Venture capital-backed startups** [like Elon Musk] **usually tout the shiny and novel. Founders do not prance around boasting about how old their codebase is. Quite the opposite: They brag about their code being cutting-edge, pounded out in all-night sessions by bleary-eyed genius 21-year-olds. But as nearly every programmer will tell you, the newer and more recently written the software, the more likely it is to be a hot mess of bugs.**

A good example of this could be witnessed during the pandemic. **In the early days of Covid-19**, businesses shut down en masse. **Laid-off employees swarmed online to apply for unemployment benefits**, and the **websites** for many state governments **crashed under the load**. In New Jersey, the governor told the press that their COBOL systems desperately needed help to deal with the new demands. "Literally, we have systems that are 40-plus-years-old," he noted.

But **technologists who were working behind the scenes to fix the problems knew that the number-crunching COBOL wasn't the problem. That old stuff was working fine**. No, **it was the newer stuff that had crashed** — the programs powering the website itself.

"The thing that went bananas was this web application in between the mainframe and the outside world. That was the thing that sort of fell apart," says Marianne Bellotti, a programmer and writer who worked for years on government systems, and who observed New Jersey's system. But it's too embarrassing, as the historian Hicks points out, to admit that "oh, our *web* systems broke down."

Bellotti's **seen the same thing happen with other government agencies, like the IRS.** She was called in once to help with an IRS web app that wasn't working. When they investigated, they found that, indeed, **the problem was in newer programs**, "this chunk of poorly written Java code". **The mainframe running COBOL, in contrast, was racing along like a Ferrari.**

"The mainframes," she says, "were responding within milliseconds."

Being "stable" and old, though, can create a paradox — a curse of success. Because **when code runs nicely without anyone needing to check up on it,** eventually people

drift away. They stop looking at it, stop inspecting it. And that **means they stop understanding how, precisely, it works.**

Certainly, **they know *that* it works.** Hey, it's functioning every day, processing millions of transactions in a snap! But nobody quite knows why or how. **COBOL has become an inscrutable mystery, a daemon that performs its tasks dutifully, but in a manner no one quite comprehends.**

This can become a big problem when, years later, you really would like to change something or add a new feature.

Dave Guarino saw this up close. He's a software developer who worked for years for Code For America, a nonprofit that takes talented coders and gets them to help governments update their ancient services. A few years ago he was helping write a new web app so that Californians could more easily apply for food stamps. The web app floated on top of California's older software systems, as it were; users would interact with the app, and it would pass along their requests to the decades-old code running on California's mainframes.

And that's where a problem occurred. At one point, his team wanted to build a way for food stamp recipients to book a meeting with a government official. The old California systems already had a section that would accept a request like that. But in the field where you'd input "when are you free to meet?" the older system only let you type 40 characters — and it wouldn't let you use hyphens, so you couldn't use a short form of language, like "M-W," to show you were free Monday through Wednesday.

What a pain, Guarino thought. So he met with the person who managed that old software system. "Unfortunately, yes, those are real constraints," the guy told him. And it was a COBOL problem; it had been written decades ago. "So what can you do? Can you make the field bigger or whatever?" Guarino asked. "And he was just like, straight up — *no! There's nothing we can do."* **That COBOL code — nobody was ever going to touch it.** The state didn't have enough money to pay for the enormous staff time it'd take to dive back into that codebase.

They were also **likely terrified that if they tried to change something crucial, they'd break it.** This is the other paradox of COBOL's success. Because it's fast and it's stable, over the years and decades, **governments and banks grew to rely on those old systems**. So even **if you *want* to change them, it's too dangerous to try**. At the bank Stern worked at, you could lose hair over the stress of tinkering with truly ancient, mission-critical code.

"It was a high level of risk to fix things because you could damage something that was already working," she tells me. So most of the time, instead of intensively rewriting old code, they'd just add small new bits of code, patching things around the edges. "People kept adding on little pieces and little pieces, and it started to look like a little Frankenstein," she laughed. Which, of course, only made the system potentially more inscrutable and messy to later generations.

Very, very occasionally, though, **some design decision made decades ago that turns out to be so truly awful that banks and companies need** — suddenly, in a panic — **to dive**

in and gut renovate genuinely old COBOL. This is what happened with the infamous "**Y2K bug**".

The Y2K bug emerged from an old design decision. When the early COBOL programmers wrote dates into their software, they used two digits: 1971 was "71", for example. That was because the machines back in the '60s and '70s had very little storage room in their memory. Removing two characters was a big deal. "All programs were very memory conscious — every byte used to be expensive," as Thomas tells me. Plus, the coders in the '60s and '70s never dreamed their software would still be in use 30 years later, when the year 2000 approached.

But as 2000 drew near, the two-digit dates became a huge dilemma. In the new millennium, the COBOL software wouldn't know whether "00" meant 2000 or 1900. If a bank calculated interest on a deposit made on "01", it might wrongly assume the deposit was made in *1901*, and issue the customer 99 years of free interest. A huge number of bank and retail and payroll transactions all rely on dates, so billions of lines of programs needed to be updated. As 2000 approached, banks called their old-timers out of retirement, paying them to pore through the codebases, find every place dates were used, and fix things.

"We spent two-and-a-half years prepping for Y2K," Thomas chuckles. "That's one of the reasons that a lot of the programing guys like me know our systems so well. Because we had to go through every program."

Even so, at Thomas's bank, they didn't have time to *truly* fix the problem. In some cases, banks and firms didn't actually change the code to use a full four-digit date like "2016". Instead, they used a hack: a "sliding rule." They'd pick a year far enough in the future, like 2045, and make it the new breakpoint. So if the COBOL sees a two-digit date that's greater than 45, it assumes it's in the 1900s — so, "87" means 1987. And if it sees a number lower than 45, it assumes it's 2000s — so, "33" means 2033.

This means, as Thomas notes, that **the Y2K problem isn't, for them, entirely fixed. They just kicked the can down the road**. Come 2045, they may well be in a panic again. Which means that still more COBOL will need to be fixed by COBOL experts.

Assuming any are still alive. Craig Bailey, of the software firm Faircom, was working with some clients to help them try to migrate off their old COBOL systems. They'd work with the client, picking the brains of the older, retired employees who originally wrote the systems — but have occasionally had an old-timer die in the middle of the process.

"Literally, we get a call on a Monday morning saying, 'Oh my god, project's on hold — so-and-so passed away," Bailey says.

> A paradox of COBOL's success is that because it's so stable, even if you want to change it, it's too dangerous to try.

Banks need to hope that those old-timers hang on as long as possible. Because there aren't a lot of new young kids learning COBOL these days.

"We get calls from companies all [the] time, saying, 'Hey do you have anybody who's got any skills in COBOL?' They're desperate," says Marilyn Zeppetelli, a former IBMer who worked on their mainframes, and who now is a professor at Marist College.

Marist is one of the few universities that regularly teaches COBOL. Many computer science programs don't, or certainly don't, promote it. Indeed, the **academe has long snubbed COBOL.** When the language took off in the '70s, elite computer scientists were scornful, arguing that COBOL encouraged terrible styles of coding that were falling out of favour. One example was the "GOTO" statement: COBOL lets you tell the program to suddenly jump from one line to another, say from line 899 to line 217. To be fair, the computer scientists had a point! This type of coding produces janky, disorganized programs that can be onerous to read ("spaghetti code," as they call it), and languages that came after COBOL mostly abandoned GOTO. Either way, the libel stuck. For people serious about pushing the frontiers of computing, COBOL was a loser's language, a backwater.

"The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence," as the famous computer scientist **Edsger Dijkstra wrote in 1975. COBOL was more of a working-class language, a blue-collar intrusion into the priesthood of coding.** Plus, When cheaper desktop-sized PCs arrived in the '80s, they became the exciting new place to run code. Anyone could have one on their desk; learning COBOL required you to have access to a huge mainframe computer, which were mostly just at banks or major retailers. "When the smaller and mid-range machines really took off in popularity, [universities] moved all their education to those platforms, and the mainframe kind of fell by the wayside," Zeppetelli notes. These days, smartphones have made COBOL even less relevant to students: **"It just doesn't seem as sexy as some of the other platforms."**

With a small incoming talent pool, **many banks and governments and retailers long ago began to rely on outsourced COBOL labour**. They keep **a small core of coders on staff who know the language**, and when they need something new written, hire firms that have phalanxes of COBOL coders, like Bill Hinshaw's "**COBOL Cowboys**", or **firms in India**.

Some firms, worried that it'll be too hard to find COBOL adepts in the years to come, **try to rewrite their entire system in a new language**. It is **nearly always a hellish task**: You have to think of every single thing your complex, decades-old software does, and recreate each tiny step in a new language. Three years ago the *New York Times* rewrote its COBOL-based newspaper-circulation system in Java; it was successful but took longer than expected **due to the "vexing" challenge** — in the coders' words — **of making sure the new system did what the old one did.**

And they were the lucky ones. The **Commonwealth Bank of Australia tried** to rewrite a core system in a fresh language; the project **cost twice as much as they expected, $1 billion in Australian dollars**. Len Santalucia, the longtime mainframe expert, once worked with the financial institution DTCC to investigate the possibility of converting their COBOL to Java.

"They probably have about seventy-five million lines of COBOL code," he tells me, "and they found out that it would cost them so much that it would take, maybe, a couple of lifetimes to recover. It was ridiculous. And they have more money than God."

So the banks shrug, and figure, *screw it.* **If it ain't broke, don't fix it.** Keep the old COBOL running. "These programs have been running day in, day out, 24/7 for 30 and 40 years. So why would we change it?" as Thomas says.

And in the meantime, the banks just try to encourage as many people to learn COBOL as possible. "You'd have a job for life," Thomas laughs.

The problem for banks, though, is that while their COBOL may be stable, their customers' expectations aren't. As you probably realize, the landscape of the financial industry is shifting quickly. Transactions are increasingly happening on Venmo-style apps that let people ping money to friends; services like Coinbase let people buy cryptocurrency; there are new lending apps like Tala and Upstart. People now expect ever-easier ways to manage their money via software.

This is where banks, which should have inherited advantage in moving money around, have it harder. It's difficult for them to roll out buzzy new features quickly, because they have to deal with their Jurassic "technology stacks," notes Denis Ryan, a former banker who's now the chief growth officer for Showoff, an Irish firm that has built fintech apps. Those old COBOL-fueled backends store data in disparate chunks — "they have a lot of silos," he notes. And **it's dangerous, of course, to tinker much with the old code: "You've got resource pain, technical pain, operational pain, risk pain."**

But a startup can do whatever it wants. There are no old systems. They're in what programmers lovingly call a "green field" situation. Instead of buying hundreds of thousands of dollars' worth of mainframe computers to store and process their data, they just rent space on a "cloud" system, like Amazon's. They can write code in new languages, so they can hire nearly any eager young computer science student. And they don't even need to build everything themselves: When Showoff is crafting a new fintech app, it might use an existing service to handle a tricky task — like using Stripe to process payments — rather than trying to create that software themselves.

"That takes away quite a lot of the operational pain from the team, so that they can scale," he notes, "and work on the product without having to worry as much about infrastructure." They don't, in other words, have any COBOL to worry about.

The problem for banks, though, is that while their COBOL may be stable, their customers' expectations aren't.

COBOL will probably never die. But that hasn't stopped many coders from predicting, over and over again, that it is about to meet its doom. Indeed, the first warning that COBOL was dead came from before the language was even released.

In 1960, the committee that was devising COBOL was only one year into its work — but one member, RCA executive Howard Bromberg, was worried they were moving too slowly. If they didn't get COBOL out faster, he reasoned, the business world would move on! Computer manufacturers would release their own unique languages, and business programming would descend into the land of Babel.

So Bromberg decided, "in a fit of pique," to send a message to the head of the COBOL committee, Charlie Phillips, who worked for the Defense Department. Bromberg bought a tombstone, which was topped with a granite icon of a "sacrificed lamb," and had "COBOL" carved on it. ("What kind of a name is that?" the tombstone-maker asked him.)

Then Bromberg put the tombstone in a crate and shipped it off to Phillips at the Pentagon. "There were rumours all over the industry that COBOL was dying," as Grace Hopper later recalled.

60 years later, the tombstone is sitting in the Computer History Museum in Mountain View, California, and **COBOL still runs the world**.

NOTE: ***Clive Thompson*** *is a journalist who writes about science and technology; his latest book is "**Coders: The Making of a New Tribe and the Remaking of the World**". He is a contributing writer for the New York Times Magazine and a monthly columnist for Wired magazine.*

SOURCE: ***Wealthsimple makes powerful financial tools to help you grow and manage your money. https://www.wealthsimple.com/en-ca***