



The Canadian Primer to Computational Thinking and Code:

A Kids Code Jeunesse Introduction to
Algorithm Literacy

By Juliet Waters

With funding from

Canada 





Introduction	3
Part 1: Code to learn	5
Why learn to code?	5
Why this book?	7
No, really, you're not too old!	7
Part 2: Learning computational thinking	10
Learn the secret to computational thinking in 5 minutes	10
You're already doing it!	11
Algorithm literacy	14
Computational thinking in the curriculum	17
Why computational thinking?	19
Computational thinking practices	24
Computational thinking perspectives	25
Part 3: Learning to code	28
How do you teach when you don't know the answers?	28
Your cognitive toolbox	32
The Kids Code Jeunesse framework for learning to code	34
Five top strategies for getting unstuck	35
Visual block programming	36
Text programming	47
Part 4: Going further	56
Planning a workshop	56
Facilitating a workshop	58
Assessment	62
Conclusion: Code for life	66
APPENDIX 1: KCJ Lesson Plans	68
Introductory lesson plans for Scratch	68
Introductory lesson plans for Python	69
APPENDIX 2: Annotated resources	70
Acknowledgements	71



This work is licensed under a
[Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/).



Introduction

I wasn't even sure what code was when I made a new year's resolution one January to sign up for a challenge called Code Year – but I was curious. An educational start-up promised 52 weeks of simple tutorials and challenges. As the parent of a pre-teen, I was also anxious about the algorithms that were increasingly running our lives and how little I understood about them.

Within minutes of trying to navigate my way through the counter-intuitive punctuation and logic of a JavaScript “Hello World” tutorial, I was lost. Without the help of my son, Ben, I would have quit right there. He had an 11-year-old brain, less attached to old patterns of syntax, able to see new, different patterns more quickly. With his guidance, I made it past my initial confusion and frustration, and started to experience the first glimmers that coding might actually be fun.

Turns out I had some adult strengths that started to kick in too, in particular the perseverance I'd built over years of work as a journalist. I discovered that many of the satisfactions of writing were there in coding. The elegant loops of logic, the attention to detail, the mission of getting the maximum amount of impact from the fewest possible lines, the feeling of going from a few wispy abstract ideas to making something that works. All of these were familiar frustrations and joys.

Once I'd made my way past those first stumbling blocks, it wasn't long before I was better than Ben at the more complex challenges involved in building programs. By February, he'd given up. He'd found something new to learn, and I guess code didn't seem as cool once his mom knew how to do it.

But I was hooked. I kept on, bit by bit every week. By summer I'd learned about code libraries. I discovered one way that coding really was very different from writing: I didn't have to expend so much effort creating original work. Coders all over the world contributed their code to open-source repositories that made it easy and permissible to use other people's algorithms. Learning to code was essentially like getting a library card. Once you understood the basic protocols, the system, and where the artifacts were stored, you had access to a massive bank of other people's expertise that you could use for whatever you wanted to create. The last three months of the year, I learned the basics of Python, one of the most popular open-source languages in the world.



I was never going to become a professional programmer, but I discovered that what I loved most was the challenge of figuring out why the fundamentals were so hard to learn and teach. I now know that anyone who gets past the disorientation of those few first hours can learn enough to understand that code is just a language like any other. In fact, I'd even say it's simpler than most.

Close to a decade later, I'm proud to have participated in building Kids Code Jeunesse, a country-wide Canadian charity that has taught code to hundreds of thousands of kids, as well as to their supporting adults (teachers, parents, librarians, community organizers.) For the record, my knowing how to code did not discourage my son for life. Last year he aced his first college Python course, truly mystified why other students found it so hard. Learning even a little code early in life meant coding was a skill that returned to him like riding a bike. It was enough to build his confidence that he could take it up again. Also, knowing that his mother could learn to code means he'll never believe he's too old to learn anything!

But what about those adults, especially women, who never had this early exposure? The Canadian Primer to Computational Thinking and Code aims to be the voice of a friendly and persistent peer, there to remind all readers, whatever their age, gender, or background that they have skills, strengths and competencies that can be transferred to lifelong learning. This book exists also to remind you that there is nothing an 11 year old can learn that an adult can't learn too!

Juliet Waters,
Chief Knowledge Officer
Kids Code Jeunesse



Part 1: Code to learn

Why learn to code?

Many educators have heard the arguments for learning to code, but remain more anxious than curious. They've been told it will better prepare kids for future jobs and 21st-century competencies. They've seen their more tech-savvy or adventurous colleagues try it out. Maybe they've even tried it out themselves with a popular visual block coding language like MIT's [Scratch](#) (and yes, Scratch is a *real* programming language!) Maybe that was a great experience. Maybe they had mixed results, unprepared for what to do when their students outpaced them or got stuck. The idea of learning to code well enough to teach it seems daunting to many educators, if not impossible.

Educators have also heard the arguments against learning to code. That it will take away time from other more creative subjects, like music or art. They've been told the skill of coding is obsolete and that, with the advent of Artificial Intelligence (AI), computers will soon be doing all their own coding. They've heard that any computer language learnt today may not even exist by the time their students graduate. They have their own very good arguments for choosing to wait and see, top among them that there is only so much time and too little support for educators wanting to learn. With all the responsibilities they have as teachers, parents, and citizens, how possible is it for everyone to learn to code? And how important is it?

Coding is both possible and essential

To answer the last question first, consider this: in the time that has passed from when most of today's adults purchased their first smartphone to today, an entire generation of kids was born who will *never know a life not mediated by computer algorithms!* Algorithms, the sequence of commands that make up a piece of code, can help us harness the tremendous power of computation, but algorithms are also prompting this generation in their preferences and their decisions. Algorithms will collect data and learn more about this generation than ever before, often without their awareness. Algorithms have the potential to keep young people locked in habitual loops of infinite distraction.



We could take their devices away, but for how long? **Algorithms aren't going anywhere.** They will not only continue to run our world, but are already becoming more powerful, harder to see, and trickier to understand, as programming becomes more and more driven by data and machines, as opposed to human decisions. Our fear of algorithms is understandable. What lies behind our computer screens and browsers seems shadowy and complicated.

But how easy is it for everyone to learn to code? If you know you are not learning to code as part of a career in technology, but rather to gain a better idea of what code is and how it might help you go beyond digital literacy toward digital wisdom, **you could probably learn the beginner fundamentals in about 12 to 18 hours of study.** You don't need the 12 months it took me. Once you've learned those fundamentals, you might be surprised to discover that coding is actually fun!

Coding enriches our lives

Coding doesn't have to take away from music, art, or storytelling. In fact, all of these practices can be enriched and explored with code. It never ceases to amaze me how many artists I meet in the coding world and the ways they've found to combine their work and their passion. Fundamentally, **learning to code teaches us about learning**, and the ability to learn is arguably the most important 21st-century skill there is.

Willingness to learn and to remain a lifelong learner will become even more important as the complex data systems involved in machine learning and artificial intelligence become a bigger part of all of our lives. Learning to code strengthens what is called "computational thinking," which is basically the ability to think efficiently and creatively about the routines in our lives. Many adults, myself included, start out coding "for the sake of the kids," and discover that we were really doing it for ourselves.

Whatever the "why?" behind our path to coding, the even more important question is "when?" The answer to that is easy: now.



Why this book?

The Canadian Primer to Computational Thinking and Code will provide all that is needed to achieve the three following outcomes essential to gaining algorithm literacy:

1. Understanding what algorithms are.
2. Making something with an algorithm.
3. Having the language to learn more about algorithms and explore the basics of algorithms with others.

This book will offer activities and direct readers to projects that will help them develop algorithms to create their own “digital artifacts,” be these animation, games, or web pages. It will help readers understand fundamental coding concepts, so that they can reflect on what they are doing, explain it, or ask questions of peers. It will empower them to look around their world and assess the ways that algorithms might improve it, or the ways that algorithms might be messing it up. Above all, this book will empower readers to join or start the communities of practice we all need to become lifelong learners.

The hands-on activities and learning frameworks in this book are carefully chosen and designed as vehicles for **understanding the logic behind algorithms**. They provide a clear, well-lit path towards learning, practicing, and assimilating the basic concepts and practices that can be transferred to any coding language. They will teach the basic building blocks without the cognitive overload of too much new vocabulary. But above all, we hope they will inspire readers to go further.

No, *really*, you’re not too old!

One of the most damaging ideas and stumbling blocks to creating a society of truly empowered digital citizens is the belief that coding can only be learned by the young. So please know, **I am not a unicorn**. If I can do it, you can too!

When journalist Gary Marcus [investigated the “critical-period effect,”](#) which is the idea that there are certain things you can’t learn unless you start early in life, he found that the research behind it was much weaker than people realize. As well, some researchers have recently designed experiments that challenge these earlier assumptions.



Own your owl-like wisdom

In the world of “critical-period” studies, researchers use owls the way that other scientists might use rats. An experiment at Stanford showed that young owls were indeed faster at learning their way around a “virtual-reality” type of barn, fitted with confusing mirrors and prisms. Older owls, however, could still learn just as much—if they learned incrementally rather than all in one bite. This explains how I could learn the same basics as my 11-year-old son, albeit slower. It also explains why I was eventually able to go farther. As an older owl, I’ve been through the incremental learning loop many more times than my younger owl. If nothing else, I had the intuitive wisdom to know I can learn and keep learning.

In computational thinking, **this strategy of breaking overwhelming challenges into smaller, more achievable tasks is called decomposition.** As adults we do this all the time, whether it’s in the “to do” lists we make, or in the decision to read just a few sections of this book, rather than all at once. In the years I’ve been working at KCJ, we have “decomposed” the challenge of learning to code for Canadians from Gander, Newfoundland to Prince Rupert, B.C. We have taught algorithms to teachers in Iqaluit, students in Whitehorse, librarians in Montreal, and parents in Toronto. As important, we have trained programmers to talk to non-programmers in clear language that makes sense.

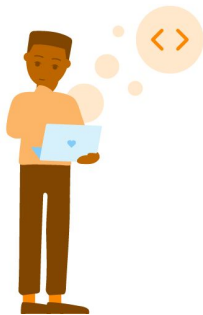




What teachers say about coding with KCJ

Since its inception in 2013, KCJ has been working alongside teachers in the classroom to develop lesson plans that have helped them to learn to code alongside their students. The KCJ team has taught hundreds of thousands of kids across the country to code, and has trained teachers on behalf of provincial ministries of education, major school boards, and as part of the federal government's Cancode program. In 2018 we trained over 6,000 teachers across Canada, and in partnership with the coding bootcamp [Lighthouse Labs](#), we delivered our popular Code Create Teach full-day teacher training workshops.

We asked teachers to fill out a short survey, and here's what they told us:



89%

say the activities are pertinent to the needs of classrooms and teachers



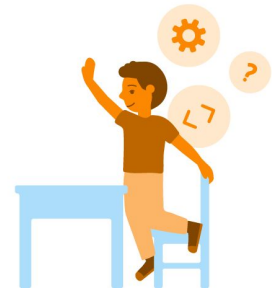
87%

say that they would integrate coding into their classroom projects



92%

see coding as an opportunity to try new ways of teaching / learning



91%

believe that coding will help to motivate and enrich student learning

So, no more excuses! Time to learn the **secret to computational thinking**.



Part 2: Learning computational thinking

Learn the secret to computational thinking in 5 minutes

Take five minutes to draw a house. Make it as childlike and rudimentary as you want, but take the full five minutes to draw as much as you can.

Once you're finished, scroll down to learn the **secret to computational thinking**.



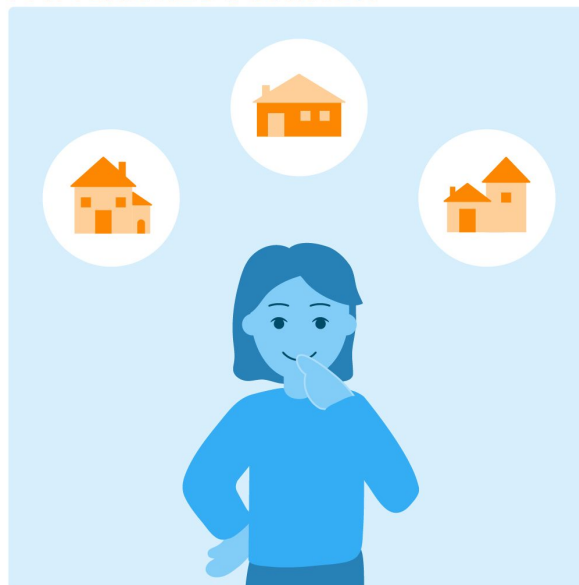
You're already doing it!

The good news is: you've been doing computational thinking as far back as you can remember, and even farther back than that! When you were a baby, building and filling the amazing database that is your brain, you didn't have the language to call it computational thinking. Now that you do, one of the easiest paths for understanding computational thinking is to explore how you've used it in the past. So let's explore it in one of the most universal early-learning experiences: drawing a house.

How drawing helps us understand computational thinking

Put yourself back in the place you were when you first learned how to draw a house. Maybe a stroller or a car seat, the place where you started filling your brain with enough examples of houses that you could recognize some commonalities, like walls, windows, and roofs. In doing that, you are employing a powerful computational-thinking strategy called:

PATTERN RECOGNITION





You don't put much effort into this. Humans do it intuitively. Scanning and storing patterns is how we learn languages, recognize faces, and make our way around a city.

But in your first attempt to draw a house, you don't start with a perfect replica of a castle, a cathedral, or even your own house. You don't draw directly from your own memory. You draw from an even easier pattern, other drawings of houses that you've seen in picture books, or posters at home or in a classroom. You recognize these as simple symbolic representations of what you've seen, easily recognizable without much detail. In doing this you are employing a computational-thinking strategy called:

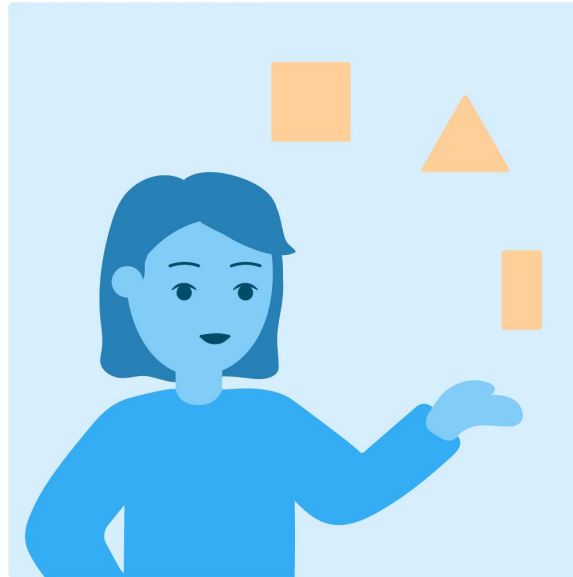
ABSTRACTION





Are you drawing this all in one unbroken line? Unless you were a preschool genius, probably not. You, or someone who teaches you, can break this mental model down into easy, smaller components that you can draw and combine with other shapes. In computational thinking, we call this:

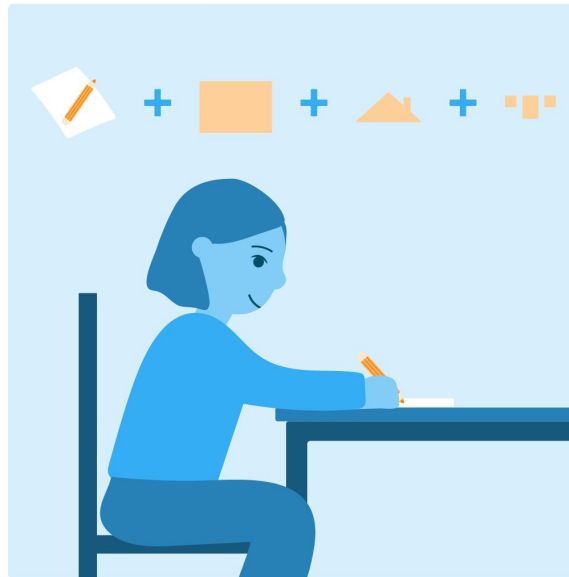
DECOMPOSITION



Do you start with the windows, or the roof, or the chimney? No, typically you start with the square or rectangle that represents the building, then the more interchangeable variables like the roof. This step-by-step, repeatable process – a routine employed by countless children across the world and over centuries – employs the most fundamental concept in computational thinking:



ALGORITHMS



Algorithms are sets of step-by-step routines that guarantee that we will get more or less the same result every time. We may input some variables, such as differently sized components, colours, and details to suggest the materials the house was built with, like brick or wood, but this basic algorithm guarantees that we will always end up with something that looks like a house.



Algorithm literacy

One of the best reasons to build your computational thinking is to become better at making and understanding algorithms. That skill is called algorithm literacy, and it's the first step of an adventure in gaining a more intuitive understanding of algorithms. In time, you will come to see and understand the algorithms around you, and start to conceive better ones that could shape the world in ways that matter to you.

But first you need to understand the basic building blocks of decoding and coding simple algorithms. You also need a fundamental vocabulary that will ensure you comprehend the elements of what you are doing.

In general, traditional literacy is the ability to read well enough to be able to develop your reading, more or less on your own, through more reading. Once you can decode the text on a page, and once you can recognize and comprehend the meaning of enough words, the text you are reading now becomes your predominant tool for learning more. In the same way, simply being aware of the presence of algorithms in our lives and the role they play in mediating our experience is important, but it's not enough on its own. No one would define literacy as the knowledge that books exist. You have to demonstrate you can read.

In this book, Algorithm literacy is defined as: knowing enough about algorithms so that you can decode simple sequences and comprehend the rudimentary, basic concepts common to all computer algorithms, regardless of what language has been used to create the sequence.



Choose your own coding adventure

Learning to write makes us better readers, even if we don't become professional writers. So it follows that learning to build algorithms can prepare us to become much more empowered users of digital tools. We need to go beyond basic digital literacy and develop a curiosity about what's happening behind our screens. Learning the concepts and practices of computational thinking and coding can help us build the courage to do that.

It's important to note that computational thinking is not coding. Yet **coding, the practice of putting routines into language simple enough for a computer to understand**, is a great way to encourage curiosity and strengthen our computational thinking. There's nothing wrong with launching right into code before we further develop our computational thinking skills or vocabulary. Many of us learned to write simple compositions before we were exposed to the rules of grammar applicable to more complex text.

If you feel courageous enough, or are by nature resistant to abstract concepts and prefer hands-on learning first, then feel free to skip the remainder of this chapter and dive right into [Part 3: Learning to code](#). Some people prefer to jump right into a cold pool rather than wade in.

Of course, there's nothing wrong with wading in either. If you want to read more about how to weave computational thinking into your curriculum without immediately taking on the challenges of code, then keep reading until the end of this chapter.

Even if you decide to skip the rest of Part 2, it's recommended that you return later to review the basic concepts and learn the basic practices of computational thinking. I also recommend taking the time to practice some unplugged exercises. Even professional programmers take a break from coding and work on their "unplugged" routines to recharge their mental energy.



Ping Pong to the rescue

If you want to try an unplugged activity that will let you practice algorithmic thinking with your students right away, without computers, try Ping Pong Rescue. This is an exercise KCJ developed as part of the student modules we created for the B.C Ministry of Education in partnerships with the Lighthouse Labs bootcamp. We have used this lesson plan across the country to help teachers wade into the practice of algorithmic thinking. All you need is a Ping Pong ball, a blindfold, and the [Ping Pong Rescue](#) lesson plan, which includes some simple assessment rubrics.

Go from unplugged to plugged

Writing a “choose your own adventure” story is another great way to explore the ways algorithms can take different paths and respond to choices. Once the story has been created, students will have the material that they need to start to code using a simple tool like [Twine](#). Try this other experiential lesson plan we developed with Lighthouse Labs for the B.C. Ministry of Education: [Unplugged: Choose Your Own Adventure](#).



Gander, NFLD. Teachers playing Ping Pong Rescue



Computational thinking in the curriculum

As we've now seen, computational thinking is a "thinking skill" we can use to solve problems or to create things. Sometimes it's known as "**algorithmic thinking**," and can be used without computers, such as when we create step-by-step directions for a recipe, carry out a fire drill, demonstrate the solution to a math problem, or, as we explored in the first activity, make a piece of art. We use it with computers to create software that automates the components of a task, a cool computer game, or a digital animation.

As with other thinking skills, like critical thinking and creative thinking, computational thinking can be improved, becomes more natural with practice, and gets better the more we use it in different contexts. We improve our critical thinking with practices like writing or public speaking. We improve our creative thinking by developing abilities in art, dance, or music. One of the best ways to improve our computational thinking is by practicing computer programming or coding.

Computation is human nature

It's worth repeating, however, that coding is not the only way. Humans have been using their remarkable abilities to compute solutions and create artifacts long before the first computer was ever invented. **When we learned to count with beads, we were computing.** When we designed complex games like chess, with multiple paths to an outcome, we were also computing. With the invention of the Jacquard loom in the early 19th century, which facilitated the manufacturing of textiles and weaving with a series of "punch cards" that could create different patterns, we were laying the foundation for the computers and computing languages that would one day power the computers we now use.

We credit the Victorian-era mathematician Charles Babbage with originating the concept of a programmable computer in the late 19th century, even though one wasn't actually built until the 20th century. However, we call Ada Byron Lovelace (the daughter of poet Lord Byron) the first computer programmer. The "poetical science" she imagined, described, and formulated for Babbage's "Thinking Machine," followed the same basic algorithm principles needed for the languages we would one day use with modern computers.



Ada Byron Lovelace, the first computer programmer

Computers are tools that help us leverage these age-old abilities. They help us, individually and collaboratively, to put our tremendous computing power to best use. But they are tools, and are only as effective as our ability to manage and optimize them.



As we enter the era of artificial intelligence, having algorithmic proficiency will increasingly mean knowing how algorithms can be driven by data as well as by the step-by-step programs we've created for them. As our algorithms become more sophisticated, computers are learning by themselves how to turn pattern recognition into algorithms. **That does not mean that we can or should abandon the project of algorithm literacy and proficiency.** As humans we still need to understand which algorithms are optimal when used with computers, but also which algorithms make the best use of our human talents.

Why computational thinking?

The softwares that humans create for computers using our powers of computational thinking are sophisticated. What computers contribute is the brute force and perseverance to do simple tasks quickly and tirelessly. This can be extremely helpful when the computer is doing the right task well. When it is doing the wrong task poorly, a computer can create a very big mess—even a disaster—far more quickly than any human can. In short: the better we are at computational thinking, the better we are at using and instructing computers properly.

By identifying the ways in which we use computational thinking in our lives and throughout the school curriculum already—even without computers or the latest in educational technology—students, teachers, parents, and other educators in the community can **build their curiosity, enthusiasm, and willingness to explore.** Together, they can discover when the best time is to start coding and how to best incorporate the practice of coding into activities and projects.

Another major objective of algorithm literacy is to use computational thinking and code to **build students' confidence and core competencies** to engage in deeper learning as well as lifelong learning. Computational thinking complements creative and critical thinking. It allows students to solve complex problems. It involves looking at challenges from different perspectives to develop potential solutions. It helps them engage in practices that assimilate skills and develop understanding.

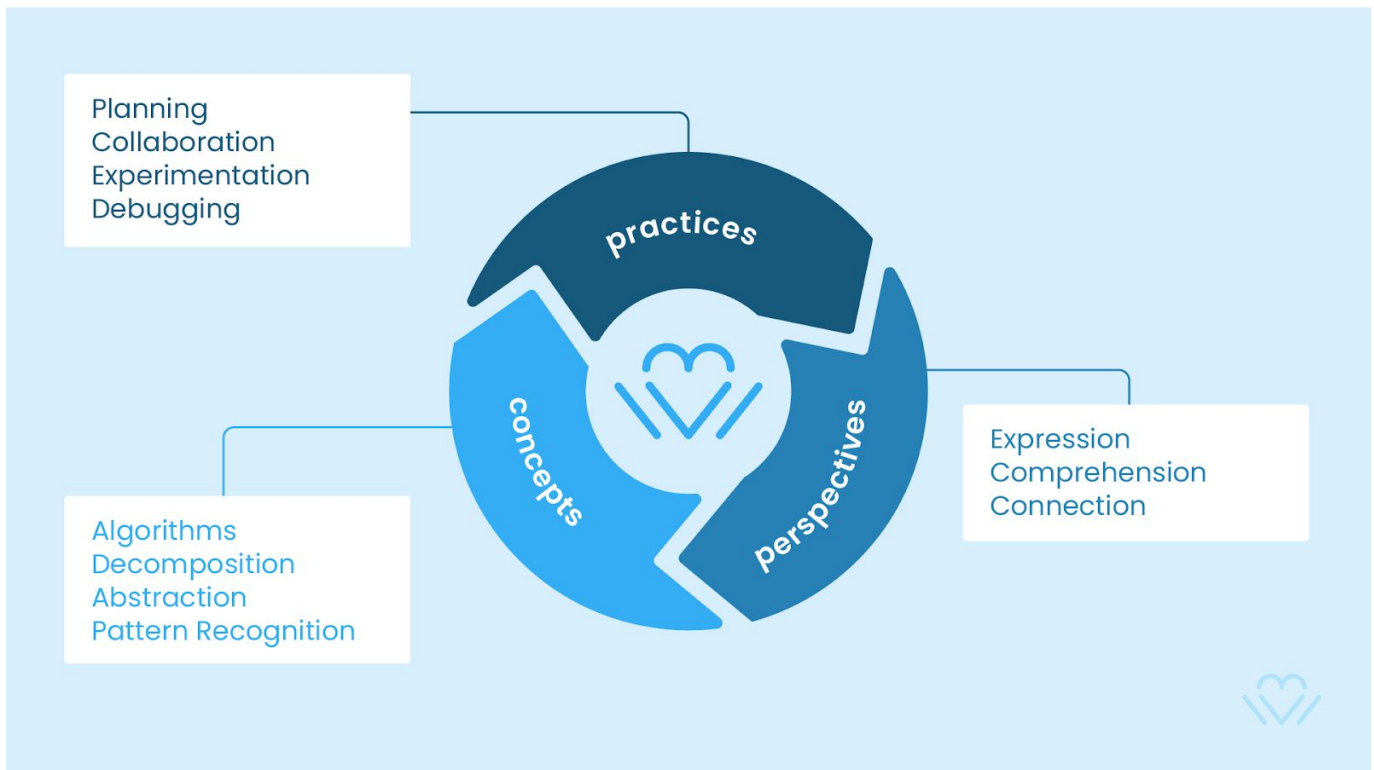
As our society becomes more and more dependent on computers, the stakes increase, and it becomes that much more important to be able to understand how computers work in order to use them safely and ethically. Every time we get into a car our safety depends not only on our driving skills but on how well the software works. This will only increase as we enter further into the era of AI and machine learning. The more insight average citizens have into the workings of the digital tools they use, the more prepared they will be for the



workforce and for the responsibilities of building a safer world. One of the quickest paths into developing curiosity about computing, however, is to spark the realization that it can be fun—even when it’s challenging.

Kids Code Jeunesse’s simple framework

We developed this framework for educators. It draws on the work of Canadian researcher Karen Brennan [in partnership with Scratch](#) co-founder Mitch Reznick of the Lifelong Kindergarten Group at MIT. The KCJ computational thinking framework includes the most basic and rudimentary concepts needed to begin the journey of becoming better computational thinkers.



Computational thinking



Computational thinking concepts and practices

As discussed at the outset of this chapter, when engaged in computational thinking, students may draw on one or more of the following concepts:

CONCEPT	DEFINITION
Decomposition	Breaking something down into smaller pieces
Pattern recognition	Finding similarities between things
Abstraction	Representing objects with a minimum of detail
Algorithms	Sequencing of events and commands

Computational thinking concepts are useful in all areas of learning, but they manifest differently in each discipline. Bringing this vocabulary into your curriculum and examining how you already use these concepts is a great start to developing the knowledge you and your students will need to feel confident when breaking down the challenges of coding.

On one hand, students may use decomposition to break down a math problem into smaller pieces so that it's easier to solve. On the other hand, they may use decomposition to break down the elements of a story or text to gain deeper understanding of different forms and genres. These strategies are often most engaging when applied to games played in physical education, to musical composition, or to the creation of artifacts in art class.



To put computational thinking concepts into practice, students may apply one or more of the following practices:

PRACTICE	APPLICATION
Planning	Outlining or prototyping potential solutions
Collaboration	Working in groups and asking peers for help in testing ideas or artifacts
Experimentation	Attempting solutions more than once to improve outcomes
Debugging	Fixing mistakes

In practice, computational thinking allows students to solve problems with solutions that can be replicated by themselves and others. It also allows students to not only use but take control of computers or any other kind of technology to express their ideas and solutions. But perhaps most importantly, the computational thinking process is highly iterative, meaning that students experience the benefits of a process that leverages fruitful repetition on the path towards a goal.

Computational thinking exercises and challenges should provide a safe environment to fail, build resilience, and support growth thinking. Students will inevitably make mistakes when they write algorithms; but more importantly, they'll be able to quickly see the results, make changes, and iterate until they have their desired effect.



Prepare students with unplugged activities

One of the best ways to prepare students is to explore “unplugged” activities, which require computational thinking in ways that transfer easily to coding. Here are a few activities that awaken strategies students already innately use in their everyday lives that are actually the fundamentals of computer programming!

- Pretending to be a blind robot that needs instructions from peers to find an object;
- Creating a game with specific instructions;
- Writing stories as a way to explore how we use repeatable sequences (such as beginning, middle, and end) to structure plot;
- Creating non-linear choose your own adventure type stories.

Computational thinking practices

Planning

Planning can be especially daunting for novice computational thinkers. Allow for warm-up, tinkering and pre-planning activities to stimulate ideas. Encourage students to quickly sketch paper and pencil prototypes as a way to think through a game or animation before starting to work on it. Prompt students to break projects down into small, achievable outcomes that clearly relate to the projected goal.

Experimenting

Programming requires iteration, which means trying something multiple times. It also means students must be willing to fail before improvement is possible. Students need time to tinker and reflect. Focusing your evaluation on process over product can encourage this type of learning. Student sheets or design journals can be used to record stumbling blocks and document steps taken to achieve an outcome. Some teachers even give partial marks for “failure,” which can be defined by the number of iterations attempted or by achieving a more interesting outcome because of an unexpected result. Encourage students to change their plan when appropriate.



Collaborating

Programmers rarely work in isolation. Making software is usually a team project, and programmers depend on feedback and advice from peers and mentors, even when working on individual projects. Encourage collaboration with student sheets that document both help received from and help given to peers. Or make time for a “think-pair-share” activity, where students take time to think about their project, pair with another student and share any problems or insights. Provide prompts that encourage peer mentorship, like sticking Post-It notes on a computer: orange for “feedback needed!” or green for “I’ve achieved my outcome, and happy to help others,” for instance.

Debugging

One of the best ways to teach debugging is to model mistakes yourself! Making mistakes in front of students, asking for their help in debugging a sequence, and showing interesting mistakes you made and what you learned does not demonstrate a lack of knowledge or competency. It does demonstrate that ALL programmers regardless of skill level must practice debugging. Professional programmers spend much of their working life debugging programs. The more complex the program, the more bugs are likely to pop up! Learn to frame being stuck as an opportunity to slow down and reflect.

Debug with a rubber duck

One of our favourite debugging practices is called “rubber duck debugging.” It means exactly what that implies: we explain our algorithm out loud to a rubber duck. Often just in explaining our problem and strategy to a passive learner, we slow down our thinking just enough to find the bug.

In our Code Create Teach workshops with Lighthouse Labs we introduced teachers to all of our favourite tools: paper and pen for planning and prototyping; blindfolds for a game of robot; Post-It notes for collaboration; and, of course, a rubber duck!





Computational thinking perspectives

The introduction of computational thinking concepts and practices inside and outside of the classroom will not only prepare students for coding through to post-secondary studies and the workforce, but it will also prepare them for the fast-changing challenges and opportunities presented in a digital world.

The ultimate outcome of teaching computational thinking, however, is not to create a new class of professional programmers. It is always to raise a new generation of digital citizens who feel confident that they can learn what they need to learn to be empowered in a digitally driven society.

As you work towards this outcome, it helps to nurture and encourage three essential computational thinking perspectives and practice applying them throughout your students' learning experience. Here they are in a simplified chart:

PERSPECTIVE	APPLICATION
Expressing	Seeing algorithms as a way to create (drawing, storytelling, etc.) as well as solve problems
Understanding	Seeing how programs and algorithms drive behaviour, both unplugged and digital
Connecting	Seeing how programs and algorithms can connect people and be a means of collaboration

Now let's take a deeper dive into *why* these perspectives have an invaluable role in both coding and everyday life.



Expressing

The more we see programming as a tool for creativity as well as problem solving, the more innovative and empowered we will be. Looking for playful ways to practice programming and creating meaningful artifacts is the best way to turn curiosity into enthusiasm and ultimately build a passion for learning.

Understanding

Algorithms are ways of structuring behaviour. They're part of our everyday routines, like researching something online and trusting search engine algorithms to give us the best information upon which to make judgments and decisions. Or, choosing our next movie based on a recommendation from our favourite streaming platform. As such, algorithms drive our behaviour by making things seem intuitive, to the point that we forget that we or someone else created these algorithms in the first place. Becoming aware of algorithms in our daily lives while questioning, examining, and reflecting on them is the start, but certainly not the end, of the lifelong project of applying computational thinking to the world we live in. Understanding that algorithms have and will always be inevitable is the most important first step towards developing algorithm literacy.

Connecting

Sharing our algorithms, adjusting our routines to be in harmony with others, responding collectively to a shared problem or social concern like a fire drill or collecting data on climate change: these are all ways that developing strong algorithms can help make the world a better place for everyone. The need to collect and share data while respecting our privacy and the privacy of others is another way that algorithms will connect us more profoundly. This connectivity is a good thing, but like all social connection, it requires shared values and codes of conduct for it to work for the good of all.



Further reading and exploring

["New frameworks for studying and assessing the development of computational thinking"](#)

Karen Brennan, Mitchel Resnick, MIT Media Lab

["A Pedagogical Framework for Computational Thinking"](#)

Donna Kotsopoulos, Lisa Floyd, Steven Khan, Immaculate Kizito Namukasa, Sowmya Somanath, Jessica Weber & Chris Yiu

["Computational Thinking and CS Unplugged"](#)

CS Unplugged, a non-profit based in New Zealand



Part 3: Learning to code

Computational thinking, as we've established, is something that you're already doing, and have been doing for a long time. Coding, however, is probably not, or you likely wouldn't be reading this book. Before we explore the fundamentals and some basic exercises, let's confront one of the biggest stumbling blocks that educators face in bringing code into the classroom. What are we going to do when students outpace us or encounter problems that we don't know how to fix? Or to put it another way:

How do you teach when you don't know the answers?

Through our Code Create Teach workshops, I've posed this question to thousands of teachers across Canada. It's a very broad question, with no single right or wrong answer, so it's helpful to break it down into three more focused questions: (pop quiz: [what computational thinking strategy](#) am I using when I break big ideas into smaller ones. *)

- What are you teaching when you don't know the answers?
- Why would we want to teach this way?
- Has anyone ever taught this way before?

Below are the most popular answers to these questions, as given to us by workshop participants. I also dive into why it's important as educators to have these answers on our minds.

Pop quiz

What computational thinking strategy am I using when I break big ideas into smaller ones?

Answer: Deconstruction of course!



What are you teaching when you don't know the answers?

How to learn. This is by far the most popular response. Teachers in our society have too often been trained to present themselves as experts in a certain subject, rather than as what they really need to be: experts at learning. Knowing how to code is not nearly as important as knowing how to help students figure out where to find the answers. Sometimes they'll find answers in the resources embedded in a popular coding platform like [Scratch](#). Other times it will be from a peer who seems to be learning quickly. And yes, quite often they'll turn to Google—real programmers use Google all the time too! There's no shame in using the learning of code as an opportunity to practice search engine skills.

How to deal with failing to find the answer. This popular answer points out that the inability to have or find the answer to something right away is not so much failure as it is an inevitable part of the human condition. This is why coding is often presented as a great way to develop perseverance, grit, and a growth mindset. It's especially essential in this Google-driven era where we expect to find answers in just a few clicks. Sometimes the best answer to the problem of not knowing the answer is to sleep on it and try again tomorrow.

How to gracefully admit that you don't know the answer. This is one of my favourite responses. Too much pressure is placed on teachers to present themselves as experts, even when they aren't. As a former journalist, I've interviewed many experts and found that as a general rule, a true expert knows what they're not an expert in — and is rarely afraid to admit it. Coding is a skill that will never stop providing opportunities to balance expertise with humility. Even professional programmers spend much of their time stumped. The more comfortable you are with humility, the better you will be at learning and teaching code, and the less afraid your students will be of learning it.

Why would we want to teach this way?

Because it shifts the focus to the process rather than the product of learning. There are some very good answers to why it can sometimes be better to teach something you don't know very well. If nothing else, you're less likely to deprive your students of the experience of truly coming to an answer on their own. It's why a teacher just learning to code can often be far better at teaching code than a programming expert.



Too often an expert can't help being an expert, which sometimes means they don't know when to hold back and give students space to learn. Experts can have an unfortunate tendency to give too much information, creating cognitive overload that makes it harder for students to find and learn the fundamentals. Teachers understand this as part of their training, so will often hold back answers to give their students space to figure things out at their pace. Sometimes this can even be easier for teachers to do when they don't know the answer.

Has anyone ever taught this way before?

I actually prefer to teach this way! I have never once posed this question to a room full of teachers and not had at least one teacher give me this answer. There is an education theory, particularly popular in technology, that teachers should be teaching less content and more learning through meaningful making and project-oriented lesson plans. This theory is called “constructionism” and it was first advanced by the MIT professor Seymour Papert.

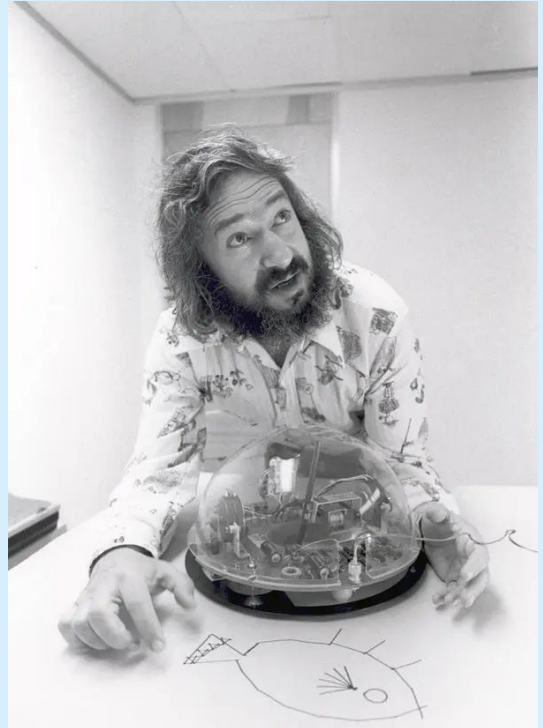
Constructionism is the basis of certain “code to learn” movements you may have heard of like the Maker Movement. Papert's book *Mindstorms* is one of the most influential texts in this pedagogy, and the idea behind it is that coding is one of the best ways to learn about learning itself regardless of the coding skills it develops. In teaching computers how to do what we want, we have no choice but to think about how we learn.

Papert discovered that in learning to program, kids were strengthening their numeracy and even learning the fundamentals of more complex applications of math and engineering formulas. They were also hitting a zone that one student described as “hard fun,” that place where the development of these cognitive skills is both challenging and rewarding.



Seymour Papert: An icon in the world of education

If you search for Seymour Papert you will find a large bank of pictures and memes, and about as many pithy quotes as Lao Tsu. My personal favourite is: **“The role of the teacher is to create the conditions for invention rather than provide ready-made answers.”** By conditions for invention, Papert didn’t necessarily mean computers, iPads, micro:bits or the countless configurations of hardware that have been created since he said this. He also meant those conditions to include cognitive tools; internalized concepts kids would need to learn to build and invent.



Cynthia Solomon/MIT Media Lab



Your cognitive toolbox

The most basic set of cognitive tools a student or educator needs to get started are four concepts that you've already encountered if you've played any of the unplugged games suggested in the previous chapter. If not, you will quickly learn them once you dive into coding.

CODING CONCEPTS
Sequencing
Repetition
Variables
Selection

Sequencing is easy to conceptualize, though not always easy to put into practice. You learn about sequences when exploring algorithms, and when you create any set of commands. Computers, however, need sequences that don't always follow the same sequences as our ordinary unplugged algorithms. Once you start coding this will become very evident! Coding is basically learning how to build algorithms that computers will be able to execute.

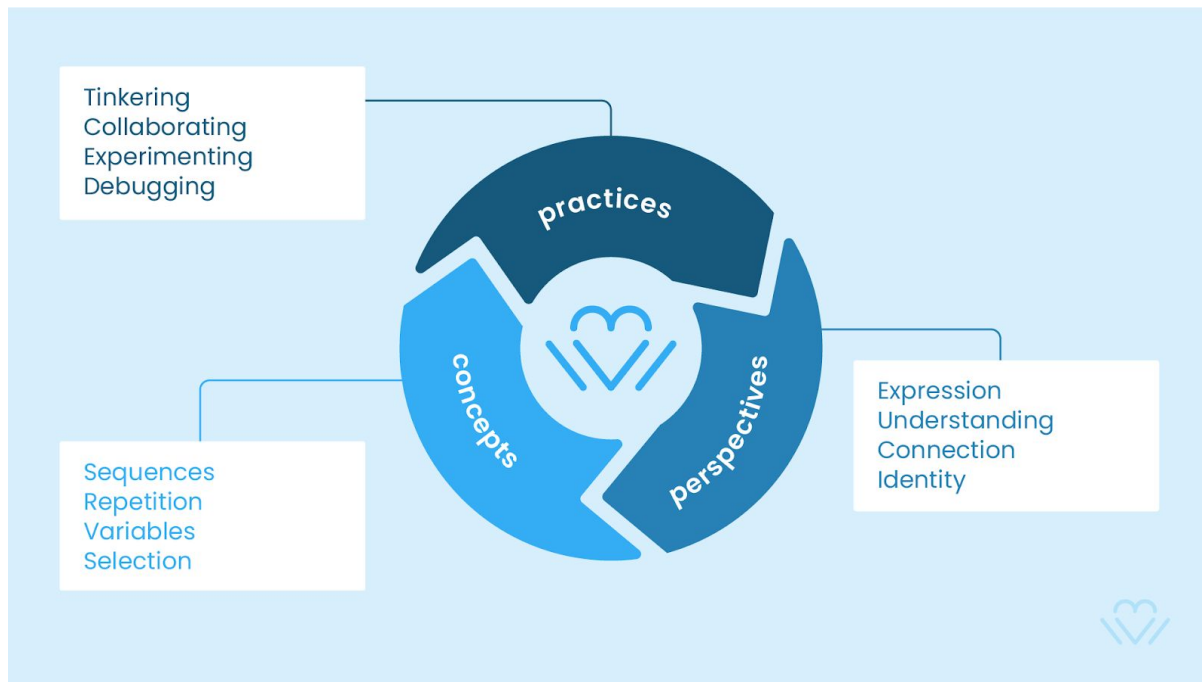
Repetition comes into play the moment you decide to make your sequence more concise by using an instruction like "repeat 3 times." This is the first step in writing your first programming loop. Again, loops can seem easy at the conceptual level, but putting them into practice is another story! Computers perform repetitions very quickly, which often gives the illusion that computers understand what they are doing. Much of the debugging process involves slowing things down so that we can see exactly what the computer is repeating.



Variables are employed to customize an algorithm. When we define a variable in an algorithm, we have a way of changing the inputs in our algorithms to create similar but different results. For instance, if I wanted to draw a house with a computer, I might create a variable called “door,” for which I could input a different colour or size of door every time I use my program to draw a new house. Again, computers can switch out variables very, very quickly. This takes some getting used to and can result in buggy programs if the variables haven’t been clearly defined and sequenced.

Selection is what we use when we want to create algorithms with different possible paths. We use selection with classic IF-THEN commands. You learned about them back in chapter two when I gave you the option of choosing your learning path. IF you wanted to jump into code THEN you came straight to this chapter. ELSE you continued reading Part 2 to the end. Or perhaps you decided to try an unplugged activity first. IF you came straight to this chapter but find coding a little more challenging than you realized, THEN you can return to the unplugged activities to slow the process down a little bit until you feel ready to return.

The Kids Code Jeunesse framework for learning to code



Learning to code

v. 1.2 – Last updated 17.04.2020
©Kids Code Jeunesse 2020



Sequencing, repetition, variables, and selection are certainly not the only concepts in coding, but they are the most basic. **Coding is the art of putting these concepts and others into practice to communicate with computers.** If we could learn these concepts only by reading about them, coding wouldn't be a skill.

Practice is the only way to turn these concepts into understanding, but frameworks help give us a working map. Our framework for learning to code builds on the [framework for computational thinking](#) introduced in the previous chapter. As you can see, the practices and perspectives are pretty much the same. "Tinkering" has replaced "planning," as a more hands-on approach, but learning to code is developed in very much the same way as any other competency.

Practice shifts perspective

Putting concepts into practice leads to shifts in perspective, which in turn, enrich our concepts and practices further. For example:

- Creating something with code helps us see code as a vehicle for **expression** as well as problem solving.
- Coding helps us develop a better **understanding** of algorithms and other things built with algorithms, such as the internet.
- Coding helps us develop a sense of **connection** between not only other pieces of code, but the people who build and use that code.
- Even more than computational thinking, coding helps us change our sense of **identity**, in that it can lead to a shift from feeling powerless and mystified by technology to being someone who knows how to code and understands that the language of technology can be learned.

Five top strategies for getting unstuck

With the framework for learning to code fresh in your mind, we're now ready to explore a few more code-specific answers to the question we posed at the outset of this chapter: how do you teach when you don't know the answers?

In 2018, Canadian researcher Karen Brennan of the Harvard Graduate School of Education surveyed expert Scratch coders under the age of 17 on the strategies they use to get unstuck.



Here are their top five. **Write these down on an index card and you will have an answer to pretty much any problem a new coder might ask you about.** It may not be the answer your student wants, but it will still be an answer.

Kids' top 5 strategies for getting unstuck:

1. Read through your code
2. Experiment, tinker, and play with your code
3. Look for examples
4. Work with someone else
5. Be persistent, but know when to take a break

Survey of Scratch community done by Harvard Edu

With these tricks up your sleeve, and a stronger sense of what exactly it is you are learning and teaching, you now have **the tools you need to start your adventure in learning to code!**

Visual block programming

Seymour Papert did more than create the theory of learning known as constructionism, he also created the first kid-friendly programming language: Logo. Logo used programming to move a turtle around a screen in order to draw simple shapes and surprisingly complex patterns. By programming in Logo, kids were in fact building visual models inside their head of how a programming loop works; how a computer variable changes at lightning speed; how sequences have to be adjusted to the limited way that computers understand the world. These **basic mental models are as important as learning the syntax of a programming language** that might be on its way to obsolescence before your students have graduated high school.



Learning from Scratch

Twenty years later, the [Lifelong Kindergarten](#) group at MIT Media Lab built on Papert's language in creating the kid-friendly computer language [Scratch](#). The turtle was replaced by a cat, which in turn could be replaced with any number of programmable "sprites" from a huge library of graphics. Text programming, and all the stumbling blocks kids and adults might have in learning it, was replaced with visual blocks that looked and snapped together like Lego.

Kids could take the same exploration-through-play approach learned through generations of playing with iconic Lego blocks to create, take apart, and re-create. They could transfer the computational thinking skills they'd already developed through play and, without having to worry about syntax errors, master and internalize basic, intermediate and even advanced programming concepts.

Though intended for after-school programs, for coding beginners as young as 8 years old, this language is robust and sophisticated enough that it's used in Harvard's Introduction to Computer Science class. Scratch is now supported by a powerful and moderated platform at [Scratch.mit.edu](#). The blocks are available in over 25 languages, and Scratch has been used by kids around the world to create almost 100 million projects. It has become **one of the largest and definitely the most global practice community for kids**. The language has been steadily moving into classrooms, with many teachers using it to introduce digital skills like visual animation without even realizing that this is code!

Scratch programming in the classroom

That said, bringing Scratch into the classroom is not as easy as dumping a bunch of Lego blocks on the floor. To paraphrase Ollie Bray, Lego's global director of play, **technology doesn't make teaching easier; it makes teaching different**.

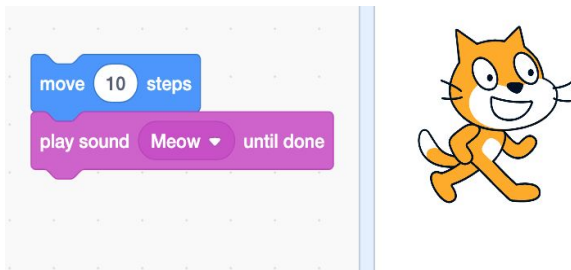
A good example is the challenge of sitting a class down to learn through the self-directed Scratch tutorials embedded on the platform created in partnership with Google in 2018. The idea behind the tutorials is to liberate teachers from the need to know anything about code. Kids are encouraged to simply learn the skills they need through the tutorials.

One of my favourite ways to teach Scratch to educators is to take them step by step through the first self-guided tutorial as though we were in an ideal world where kids followed tutorials like robots. Then I take them through what usually happens when you sit 25 kids down at the

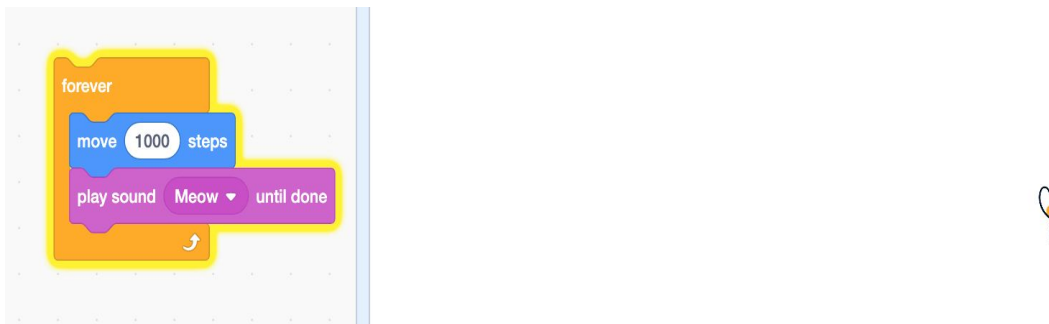


same time to follow it. Because, of course, kids are the opposite of robots. If they're programmed for anything, in fact, it's curiosity and finding out what happens when they don't follow instructions.

Kids have no problem putting together the simple program that makes the cat move and then meow on the Scratch "stage."



The problems start in the next few minutes when kids discover that you can put that simple sequence inside a "repeat" block and make it move and meow again and again. It's amazing how quickly they discover that they can make the cat move and meow a thousand times, or better yet – put it inside the block that will make it move and meow "forever."



I've never timed how long it takes for cats to start zooming off the stage, invisible except for the tip of a tail but still locked in a loop of rapid-beat meowing. But I do know that it's not long before a teacher ends up literally and virtually herding 25 ear-splitting cats. It is the rare teacher who will stop, savour this moment, and think "How wonderful! My emerging migraine indicates that my students have just learned one of the most powerful concepts in code: repetition!"



DIY your own reset button—and more—with computational thinking

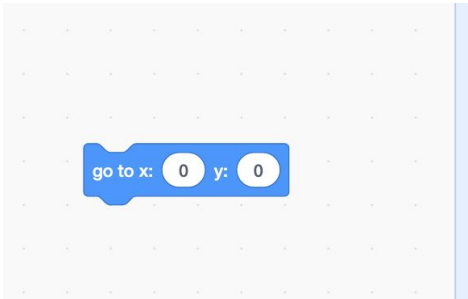
Like most people who have been trained to be users rather than makers of software, the first thing educators and students do is look for the reset or undo command. It will likely take them quite some time before they discover that there isn't one. The Scratch platform looks like any other online software platform, but **the point of Scratch is to interact, not as a user but as a programmer.** The key difference between non-programmers and programmers is that non-programmers look for the reset button while programmers know to make their own reset buttons.

Making a reset button is actually quite simple to do in Scratch. First recognize that you need a reset sequence. Then decompose that task into the smallest elements. (If you want to open up Scratch and try it out first, the next section might make more sense. But you might want to read through the computational thinking process here first. You can also follow the step-by-step instructions provided in the lesson plan at the end of this section.)

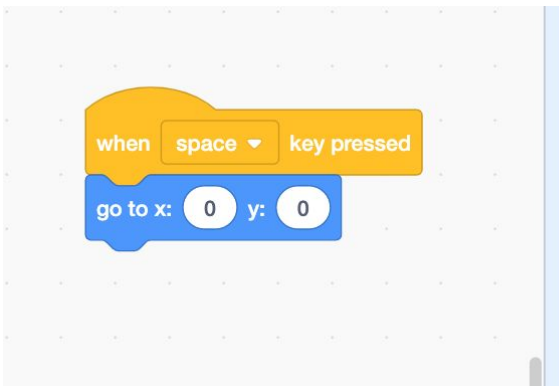
In a programming loop you always want to establish what your programming object is going to do first. If you were designing an algorithm for a robot to reach a specific destination to retrieve a ping pong ball, you would want to establish first where it needs to begin. Otherwise the algorithm would always end in a different destination, not at the position of the ping pong ball.

Introducing programming loops in Scratch

All Scratch sprites move along a stage that is plotted like a cartesian grid (that basic XY graph we all learned, even if we don't remember what it was called.) You don't need to explain what a cartesian grid is to your students—in fact, I advise against this until a later class. They are learning enough already in this class. You only need to introduce them to the “go to x:0, y:0” block, which returns the sprite to centre stage once you've clicked on it.



A set of yellow “event” blocks act as the triggers for each sequence you build. Choose one that will trigger a sequence when the space bar on a keyboard is pressed. Snap in your “position” block and you have your reset button.



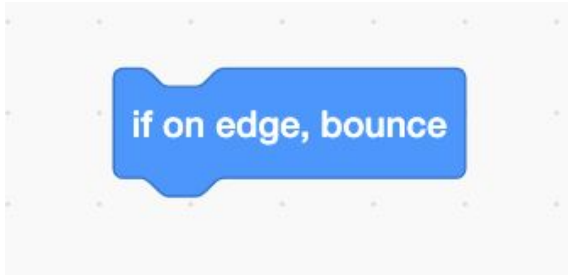
In coding this is called **“initializing” your loop**. One of the first and most valuable algorithms an educator can learn and teach is how to create an initialization sequence that will set up the default settings for any loop created with a “repeat” block. In fact, in text programming you cannot have a functioning loop without determining where your loop begins and ends. So, the sooner you learn this in Scratch, the better.

This solves the problem of getting your sprite out of the wings and back to centre stage for students who have input too many repetitions to contain it.

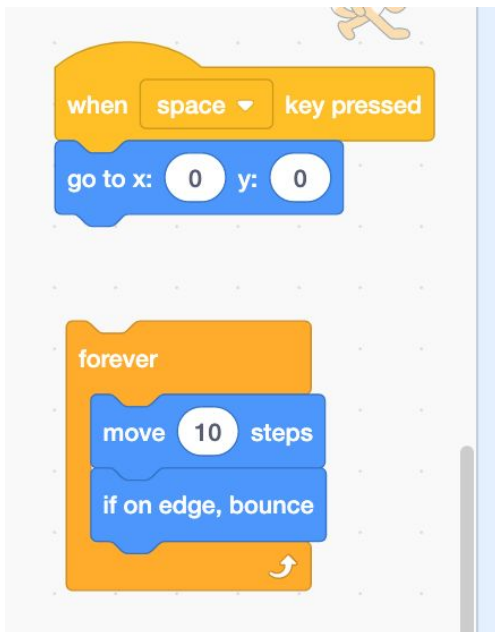
If it is still in a “forever” loop, however, it’s simply going to go right back to the wings unless you set a condition that actually makes it possible to keep moving forever.



Here's a single block that introduces the logic of selection:



Put this in a "forever" block. You have set a condition for your sprite: if it reaches the edge of the stage, it will turn around and come back. (It is highly recommended that you respectfully ask your students to remove any meow blocks from inside any and all forever loops. Forever.)





It's worth considering that in this simple program, we've touched on all four concepts in our cognitive toolbox:

1. We've used **sequencing** to create the program.
2. We've used **repetition** to create a programming loop.
3. We've input **variables** by initializing where the sprite will begin.
4. We've used **selection** to issue the conditional command "if on edge (of stage), bounce."

Your job, if you choose to take it, is to explore these concepts further through play.

You'll find several basic lesson plans organized around the four concepts in [Appendix 1](#) of this book.

If you want to explore Scratch as an animation tool, [this introductory lesson plan](#) has been classroom tested in hundreds of classrooms in Quebec's largest school board, Commission Scolaire de Montréal (CSDM): Let's move (On Bouge)

But before you head into the wilderness and limitless potential paths of Scratch animation, you might want to try Scratch out with a skill that we've already established you're familiar with: drawing.

Creating the right conditions for innovation

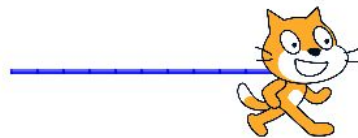
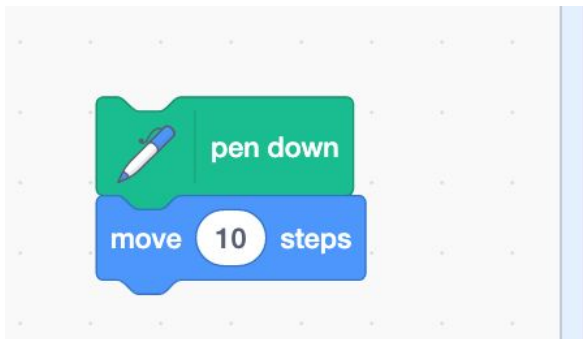
The great thing about Scratch is that as a fully developed programming language and animation platform, it gives kids so many ways to create. This is also its greatest challenge. With such a wealth of options, kids can fly off in so many different directions, it can be very difficult for one owlish adult to keep up with what they are learning. Excellent Scratch guides like Karen Brennan's "[Creative Computing](#)" suggest challenging kids to create Scratch programs using a limited number of blocks. This creates a "sandbox" that will keep at least new learners out of the more powerful and potentially complicated blocks for a while. This is a great option in groups of kids who have never tried Scratch. The growing popularity of the language, however, makes that situation increasingly rare.



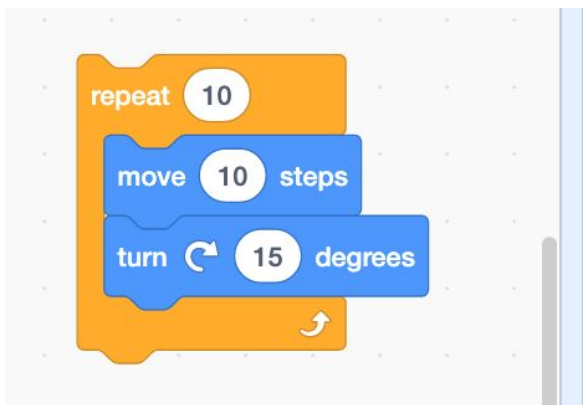
Into the Scratch playground

We've found that one of the best ways to create a Scratch "sandbox" that doesn't limit the potential of kids who might already have experience with Scratch animation, and want to explore its more enhanced playground features, is to return Scratch to its Logo roots by bringing in the "pen" extension.

Once a "pen" block is used, all "move" blocks become "drawing" blocks. With a simple algorithm created from the simplest "move" blocks, students can learn how to draw a line (I have shrunk this cat down to 30% of its size, otherwise we could not see a 10 pixel line):



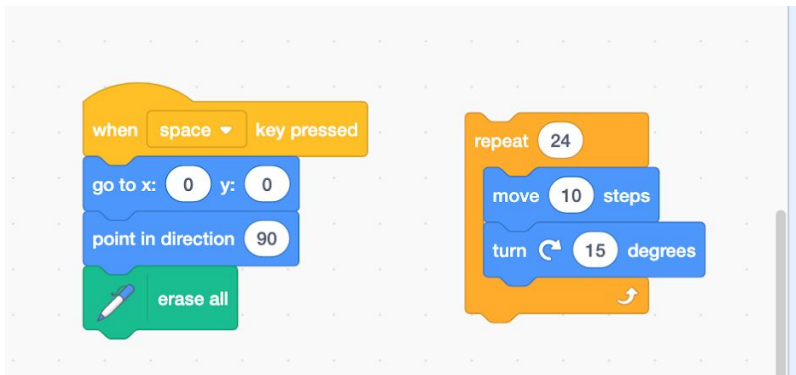
Add a "turn 15 degrees" block and a "repeat" block and this straight line becomes a curved line.



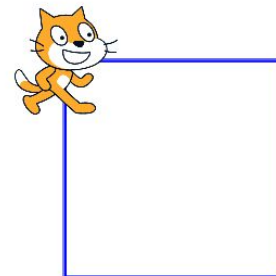
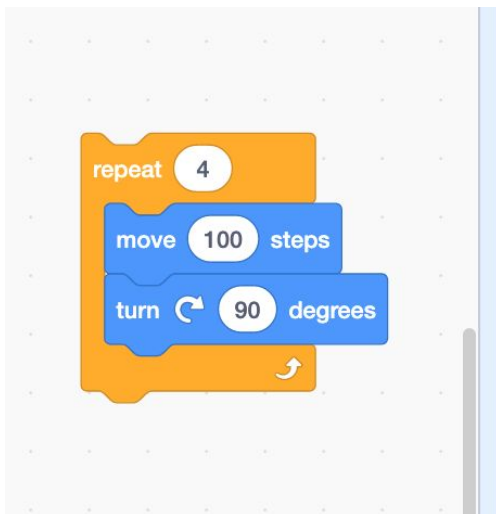


Kids as young as seven can learn how to create a simple circle by experimenting with the “repeat” block. No geometry needed!

A reset sequence with an “erase all” block will return the sprite to a default position and default angle (90 degrees to the horizontal keeps the cat standing up straight). Students can experiment again and again, until they learn the number of repetitions needed to create a circle.



Or try new shapes:

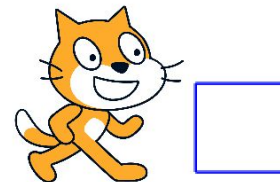
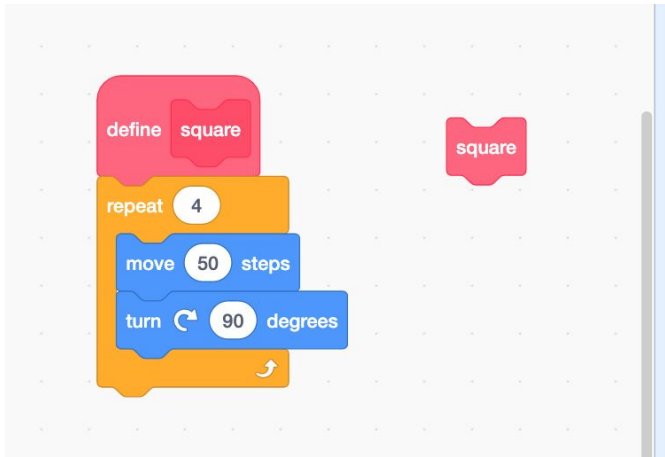


Once shapes have been created we can use the pink “my blocks” to simplify our algorithms even further by naming them. Giving sequences names so that they can be used without the



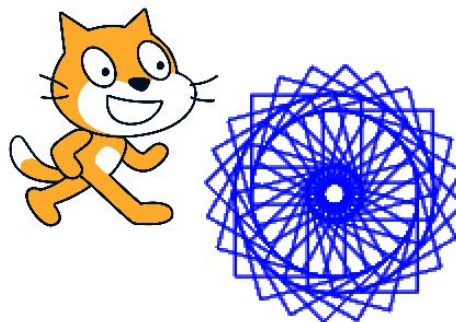
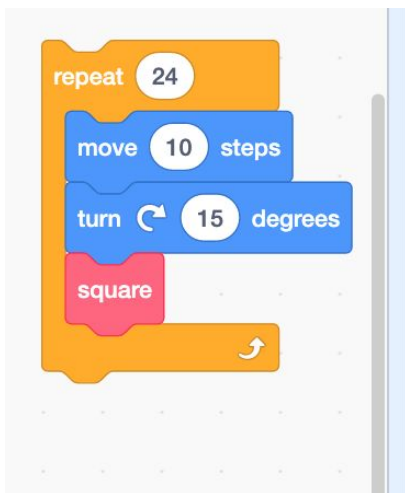
tedium of re-writing instructions is the basis of “modularization.” It’s the advanced beginner stage of building sequences.

Learning to modularize early encourages kids to keep algorithms concise and readable, which makes them easier to debug.



Now the fun can really start!

By placing a “square” block inside the algorithm they used for a circle, kids will learn why they might want to program drawings as opposed to doing them by hand.





Try it yourself!

Practice this kind of Scratch coding with this [simple spirograph lesson plan](#). Or develop more skills through the projects at [Code Club Canada](#), a bank of projects we've made available to volunteers across the country. If you start a Code Club in your school, you'll also have access to several fundamental concept posters like the ones shown below!

SEQUENCING

code club explains... computer science

Computers are powerful, but they are not very intelligent. They will only do exactly as they are told, in the order they are told to do it.

For computers to complete tasks, we need to give them a set of instructions in the correct order.

This is called **sequencing**.

We use sequencing to build programs to tell computers how to do clever things.

We use sequencing every day too!

For example, if you are making a cup of tea, you will follow a sequence of steps.

You have to complete the steps in the right order - you can't add water to the cup until the kettle is boiled.

Can you think of any other examples?

Try out sequencing with this maze!

Can you write a sequence using these instructions to get the robot to the cat?

You can find more mazes like this in Scratch: [jumpto.cc/maze](#)

forward
turn left
turn right

Want to know more about Code Club? Check out [www.codeclub.ca](#)

REPETITION

code club explains... computer science

When you spot groups of repeated instructions in your code, instead of writing them again and again, you can use repetition.

Repetition is a way of telling the computer to repeat instructions:

- A set number of times
- Until a condition is met
- Forever

For example, a computer game might repeat background music forever, until the player quits the game.

We use repetition every day too!

If you were sharing out a deck of cards with a friend, you would need to deal a card to your friend and then to yourself. You would keep repeating these actions until there are no more cards left.

Try using repetition to make animations!

Repetition is really useful when you want to make a character walk, jump or fly.

Check out [jumpto.cc/animation](#) to see how to use repetition to create animations.

Can you make your own animations?

Want to know more about Code Club? Check out [www.codeclub.ca](#)



VARIABLES

code club explains... computer science

A variable is a place in a computer's memory to store data, like a number or some text. Each variable in a program is named, so that the stored data can be fetched, used and changed later.



A computer game uses variables to store data about the game such as the scores, number of lives remaining and time remaining to play.

These variables are updated as the game is played. For example, the score will go up, whilst the remaining time to play goes down.

We use variables every day too!

When playing a board game sometimes you will write down each person's scores. You'll then update scores as you go along.



Try out variables with this voting program!

Voting programs work by adding one to the value held by the variable each time you click to vote.

There is a Scratch program at jump.to/cc/food where you click to vote on your favourite food.

Can you add more food to vote on?

when this sprite clicked
change apple by 1



1 1 2

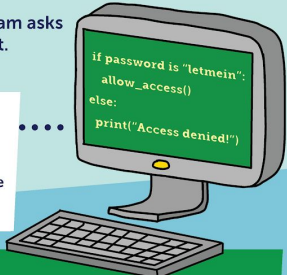
Want to know more about Code Club? Check out www.codeclub.ca

SELECTION

code club explains... computer science

Selection is when a computer program asks a question to decide what to do next.


You can use if and if...else statements to make decisions in your programs. For example, when you enter your password into a computer, it decides whether to give you access.



```
if password is "letmein":  
    allow_access()  
else:  
    print("Access denied!")
```

We use selection every day too!

When you look out the window in the morning, if it's raining, take an umbrella with you, else leave the umbrella at home.




Try out selection with this quiz!

Selection is used when creating quizzes too!

You can find a Scratch quiz program that uses selection at jump.to/cc/quiz

Can you create your own quiz questions?



when this sprite clicked
ask What is the capital of Germany? and wait
if answer = Berlin then
say Well done! for 2 secs
else
say Better luck next time. for 2 secs

Want to know more about Code Club? Check out www.codeclub.ca

Text programming

Bringing text coding into a classroom is daunting, even if you're a coding expert. But with the right resources, some students and teachers find they actually prefer it to block coding.

While Scratch is a great choice for visual learners, studies have shown that **coding engages the language centres of the brain even more than the numeracy centers**. Kids who are naturally drawn to text and reading find a special joy in learning a new language that looks a lot like basic English, yet is just different enough to make them feel like they're mastering something new.



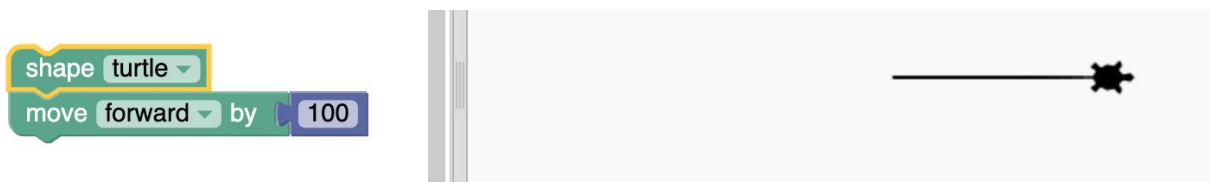
Fun text-programming fact: Girls in particular, who may not have grown up with their own Lego set or other construction sets, seem to take to **Python**. Python is a coding language that was created to feel as close to natural syntax as possible, while using the same basic concepts encountered in all programming languages.

Transitioning from blocks to text code

KCJ was very fortunate early in our journey to discover a tool created specifically to decompose this particular challenge for teachers.

[Trinket.io](https://trinket.io) is a platform designed to ease the transition between visual block code and text programming. It uses a “code library” of logo-like turtle commands that have been pre-coded in Python. Students use the commands in this library to draw art in very much the same way we used the sprites in Scratch to draw shapes.

Here’s a basic sequence in Trinket block code to make a turtle-shaped pen move:



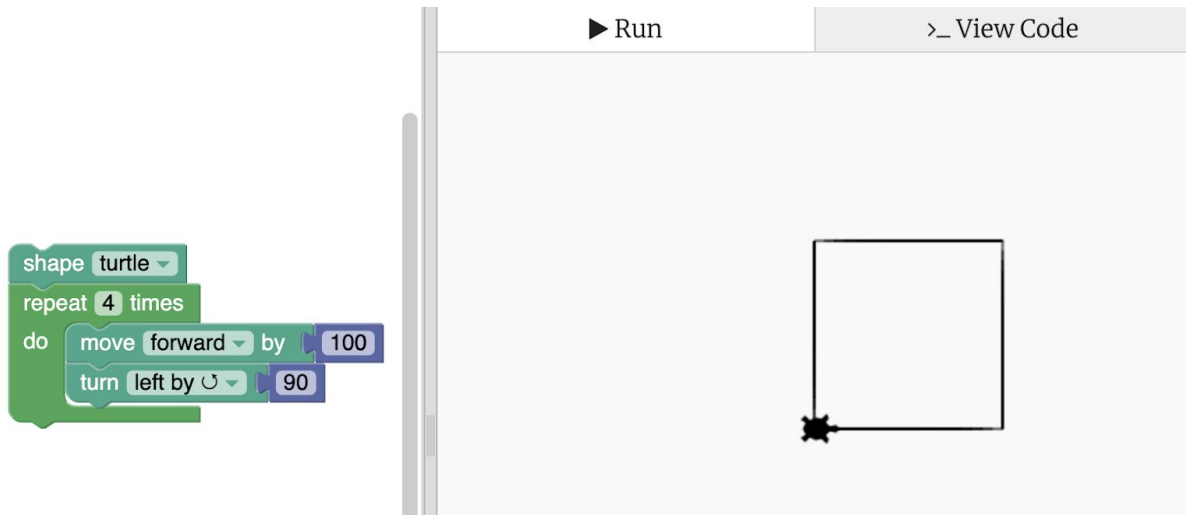
Trinket allows users to easily toggle between block code and text code. What we have coded with blocks is instantly transformed into readable text code. It’s a little like the process of reading words associated with pictures before venturing into the challenge of learning how to write.

Here’s what the sequence looks like in Python code:

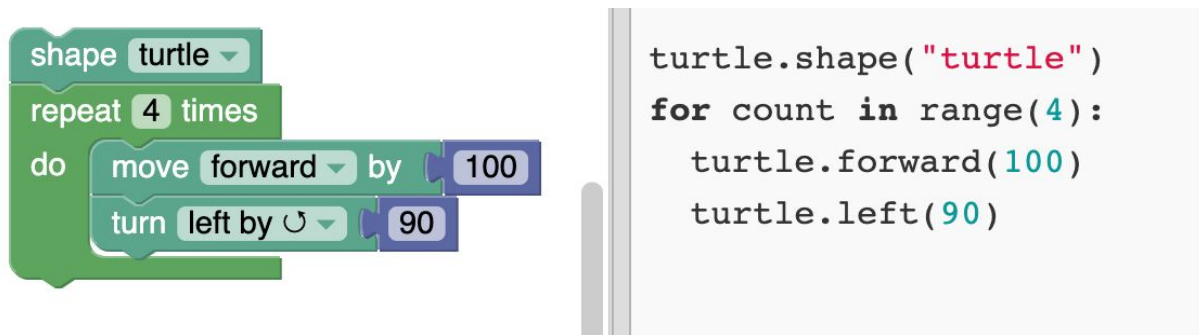




Now let's look at a square drawn with Trinket block code. As you can see, this is very similar to what we coded in Scratch:



Let's see how that programming loop reads in Python:



The most complicated line in that code is the “for loop” that commands the turtle to repeat the movement it must make four times to form a square. If you’ve had some experience building a spirograph in Scratch, however, it’s not too hard to intuit its mechanics. This loop command instructs the computer to repeat the commands contained in the loop sequence “for” as many times as it must until it has reached a count of four.



Another way of looking at Python coding

Imagine the computer as a creature that is very slow at understanding what it means to do something four times, so must count out loud each time. Since this creature is very fast at executing this command, it seems like it understands the command even better than you do, despite the fact that it is still silently counting out loud.

The truth is, the computer-creature doesn't understand what it's doing and never will. So, we must write sequences that spell it out. In this case the sequence is: move the turtle forward, turn left 90 degrees.

Defining a square

We can save ourselves time from the tedious task of spelling everything out. As in Scratch, we can use the visual blocks to create a block called "square."

The diagram illustrates the equivalence between a visual Scratch block and Python code. On the left, a Scratch 'to square' block contains a 'repeat 4 times' loop with two actions: 'move forward by 100' and 'turn left by 90'. Below this is a custom block icon labeled 'square'. On the right, the corresponding Python code is shown:

```
def square():  
    for count in range(4):  
        turtle.forward(100)  
        turtle.left(90)  
  
square()
```

In text programming we call this creating a "function." A function is quite simply a name that we give to a sequence of code. Once we've done this, we don't need to keep writing it out.

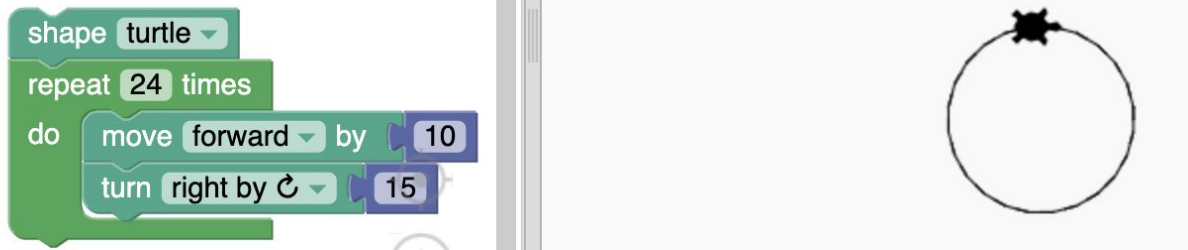
As you can see in the Python code, when we give the computer the command `square()`, the computer will then run through the sequence.

This gives us a sequence we can now use to build the same spirograph we built in Scratch.

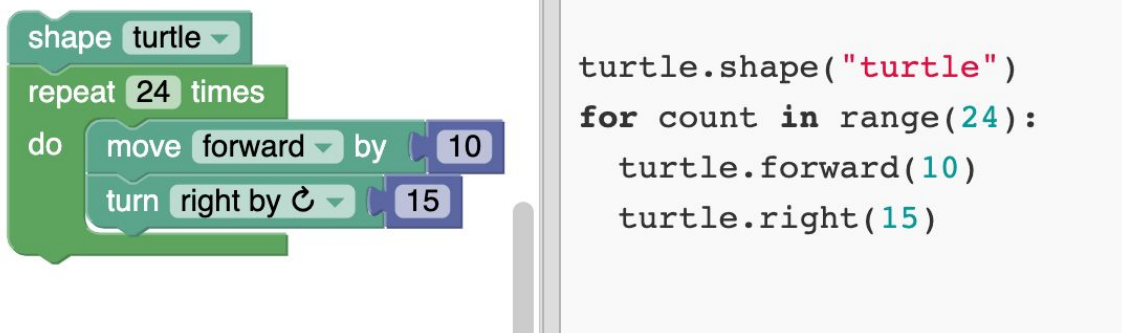


Building a Spirograph in Python

To build the same spirograph we built in Scratch, we'll also have to code a circle. Let's code it exactly as we did in Scratch, with the "move" and the "turn" blocks.

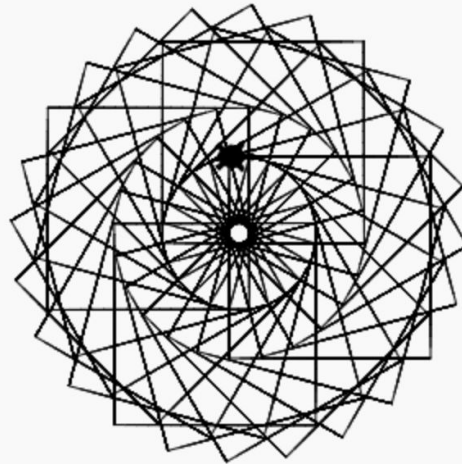
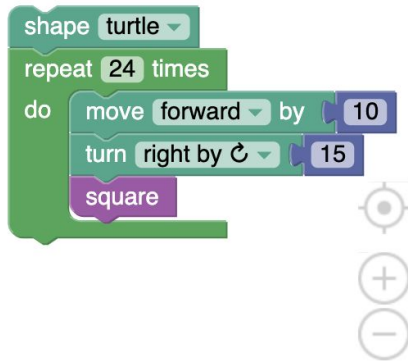


Here's that code in Python:

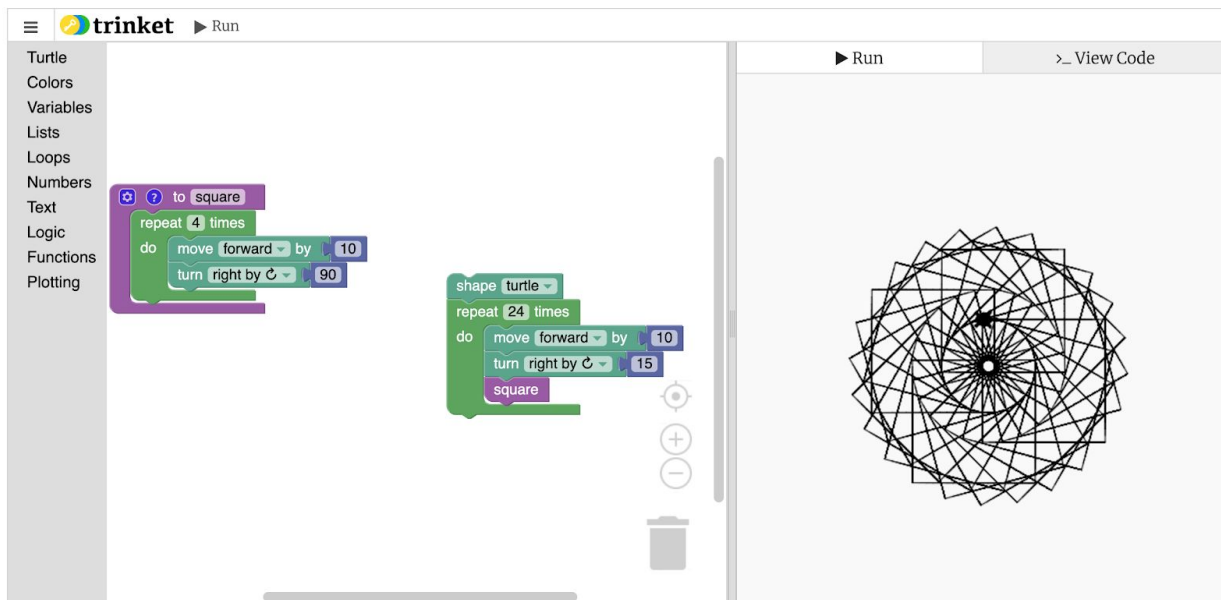




Now let's place our "square" block inside our "circle loop" to create a spirograph!



Here's that spirograph on the Trinket.io platform:





In human language, these commands would be:

- Use a pen shaped like a turtle
- Complete the following sequence 24 times:
 - Move the turtle forward by 10 pixels
 - Turn the turtle to the right by 15 degrees
 - Complete the square as it's been defined for the "square" function

Let's look at what that code for the spirograph reads like in Python:

The image shows a side-by-side comparison of a Scratch script and its Python equivalent. On the left, the Scratch script starts with a 'shape turtle' block, followed by a 'repeat 24 times' loop containing 'move forward by 10' and 'turn right by 15' blocks, and a 'square' function block. On the right, the Python code defines a 'square()' function with a 'for count in range(4):' loop that calls 'turtle.forward(100)' and 'turtle.right(15)'. The main script sets 'turtle.shape("turtle")' and uses a 'for count2 in range(24):' loop to call 'turtle.forward(10)', 'turtle.right(15)', and 'square()'.

```
import turtle

"""Describe this function..."""

def square():
    for count in range(4):
        turtle.forward(100)
        turtle.right(15)

turtle.shape("turtle")
for count2 in range(24):
    turtle.forward(10)
    turtle.right(15)
    square()
```

Complex, but not actually very complicated.

The variable "count" in our loop command for the square is given a different name, "count2" so that there will be no confusion between that variable in the outer and inner loops.



It's worth pointing out here that we have used three of the four basic concepts in our coding framework: sequences, repetition, and variables. As a bonus, we've also learned to create a function (though the truth is we already learned that by creating our own blocks in Scratch.)

Try out more coding in Python!

Here's a fun [Rock, Paper, Scissors](#) project that will start you and your students writing Python text, coding without blocks, and exploring the challenges of selection and coding with conditional statements.

Recreating the [spirograph built in Scratch](#) is one way to introduce text coding into the classroom. Another is to try out some of the Python projects at [Code Club Canada](#). Or visit the "learning section" on the Trinket website.

You're now well on your way to becoming an advanced beginner. Before you know it you'll be exploring more intermediate, lofty-sounding concepts like boolean operators and data structures. These concepts and more can be explored through Code Club Canada, where you can also find this handy basic [skills map](#):



Python 1

Python 2

	Project 1: Modern Art	Project 2: Popular Pets	Project 3: RPG	Project 4: Robo- Trumps	Project 5: Where is the ISS?	Project 6: CodeCraft
Sequencing	✓	✓	✓	✓	✓	✓
Variables	✓	✓	✓	✓	✓	✓
Selection		✓	✓	✓		✓
Repetition	✓	✓	✓	✓	✓	✓
Boolean Operators			✓			✓
Data Structures		✓	✓	✓	✓	✓
Functions	✓		✓			✓
File Handling		✓		✓		
Web services					✓	

v. 1.2 - Last updated 17.04.2020

©Kids Code Jeunesse 2020



Part 4: Going further

Planning a workshop

KCJ has delivered thousands of workshops for kids, teachers, and families across Canada and we've learned much of what we know from great partnerships with other organizations like Lighthouse Labs and international friends like the Lifelong Kindergarten Group at MIT Media Lab.

From our bootcamp friends, we've learned **the importance of iteration**; being willing to try something out on the ground as soon as possible and making it better through feedback and experimentation.

From Lifelong Kindergarten we've learned the **three fundamentals of structuring a successful creative learning experience**: low floors, high ceilings, and wide walls. What this means is every workshop should have challenges and tools easy enough for all learners to access (low floors), the flexibility to allow learners to achieve beyond expectations (high ceilings), and the latitude to bring their personal passions and originality into whatever they are doing (wide walls).

We incorporate this pedagogy into all our lesson plans, applying the principles of inclusive design to ensure that the lesson plans are accessible and can be easily localized to all types of learners. We also offer our [framework for coding](#) to make it easy for teachers to feel confident that their students are achieving learning outcomes that will help make coding and the art of coding intuitive enough to prepare them for more advanced concepts and strategies.

Using concepts, practices, and perspectives

In crafting our lesson plans we like to mix and match concepts, practices, and perspectives to assure that students are guaranteed the best chance of assimilating the skills taught.



A sample lesson plan

Main coding *concept* to explore: [repetition](#)

Main coding *practice*: [experimentation](#)

Main *perspective* to seed with this lesson: [expression](#).

A student completing this workshop should have a good idea of how repetition is used to improve code, should be able to demonstrate that she experimented with repetition in the program she wrote, and should now know that she can use repetition in coding to create code that is powerful, efficient, and creative.

Obviously, this doesn't mean that students won't learn anything beyond the main concept practice, and perspective, but having central learning outcomes and being able to convey those expectations helps keep a workshop focused and manageable. It also makes it easier for you to assess whether or not your floors are low enough, your ceilings high enough, and your walls wide enough.

At its best, a successful workshop should result in a diversity of artifacts that demonstrate competency:

- A grasp of the concept,
- An awareness of how to put that concept into practice,
- An understanding that this skill can be meaningfully applied in other contexts.

Choosing the floor, the ceiling, and the walls

Once you've chosen your learning outcomes, follow these steps:

1. **Set the scene with a quick unplugged activity that will activate students' relevant computational thinking skills.** For example, in our workshop called [Let's Move](#), we recommend two students play a quick game of blindfolded robot in front of the class and ask students to note how they could use repetition to improve the commands given. This kind of activity seeds the concept of using repetition to make a code



shorter and more efficient, demonstrates that mistakes are part of the process, and frames programming as a party game where participants are free to laugh and play towards achieving the results they want. (As an added bonus, if for some reason you encounter a technical stumbling block with the coding platform, students can extend the unplugged game while you figure out the glitch.)

2. **Choose three short activities that range from floor to ceiling**, encouraging students to bring their own interests into the workshop. Here are three examples of activities:
 - Activity 1: Build a very basic loop and the simplest initialization condition. Choose a sprite and/or a background that expresses a favourite animal or place.
 - Activity 2: Expand the loop with other variables, like size or changing colour. Make sure that the initialization sequence has a default size or colour to assure that the sprite can be returned to its original state.
 - Activity 3: Add other sprites. Experiment with putting loops inside loops and/or make a block that gives a name to a sequence and allows you to simplify your code.
3. **Leave time for reflection.** It's highly possible that students will not make it through all three activities, for any number of reasons. Even if you only make it through activity 1, however, don't skimp on giving students time to think about what they've learned, discuss any insights they may have come to, and think about how they use repetition and experimentation in other parts of their lives.
4. **Make connections between computing and society.** Consider reflection prompts that will get kids thinking about the ways computers use loops and repetition in our lives for good, but might not always result in positive outcomes. Video games and the addiction loops created by many digital activities can be an interesting subject of conversation, especially in the context of having created something playful. Kids might open up in ways that will make them feel less defensive about their current habits.

Facilitating a workshop

This list of best practices for facilitating a coding workshop comes from practices that our KCJ facilitators use every day to support teachers in the classroom.



Best practices for the first hour

The first hour of coding with kids is challenging and exciting, but also a little nerve wracking, if you're not prepared. Use Papert's idea of "hard fun" as your compass. If you guide your students into the right zone they will be happy to take on challenges and recognize those challenges as part of the learning process. These practices aim to set the right tone, create realistic expectations, and lay the groundwork for the hard fun we all want to have.

1. Understand your role

We've learned over the years to avoid the trap of trying to present ourselves as party magicians dazzling the class with cool tricks. Better to put in place the conditions to allow students to make their own magic. Present challenges that are fun and flexible, set expectations that are realistic, and trust your students to be good co-learners and even co-teachers.

2. Make unplugged ice-breakers count

Use an introductory unplugged activity, however short, to establish the tone of the workshop. If kids are relaxed and engaged right away, chances are you will be less nervous about introducing new material. If a Scratch session includes a robot game, bring a blindfold. Robots *pretending* to be blind are never as fun. Whatever is being presented, this 10-minute unplugged activity should encourage laughter, mistakes, and active learning.

3. Establish a code of conduct

Set expectations early, before kids go online. Establish clear guidelines about the importance of protecting privacy, treating other participants with respect, and posting only content that is free of abusive language or violent imagery. Establish a protocol to command attention, e.g., 1-2-3, hands on the head, or screens lowered while important information is being conveyed.

Also establish a clear protocol for asking for help. Coloured Post-It notes (e.g., pink for "I need help", green for "I'm ready to move on and learn more") allow students to indicate that they are stuck, without having to stop working towards a solution, and can help assess when it's



time to change activities. Keep in mind, this is a hands-on learning workshop. There will be creativity, exploration, and inevitably noise. To learn, kids need plenty of time to play. So provide structure, but don't be rigid.

4. Present in small increments

Be mindful of focus and pacing. Every new idea, block, or tip introduced is something that both students and educators have to absorb and understand. It used to be believed that average working memory could contain seven pieces of information. Scientific consensus now is that that number is closer to four. KCJ workshop plans “chunk” learning into groups of 4 to 5 blocks. Don't risk cognitive overload. Every new block introduces a new layer of potential bugs, so keep it simple. Allow students time to practice with hands-on tinkering. Don't try and teach all the necessary blocks at once.

5. Live code

Be ready to write code in front of the class. Practice beforehand and know (roughly) where you'll have to click to build the programs in the lesson you'll be teaching. Be comfortable taking questions, trying and discarding bits of code, and truly being *live*. Live coding — including side-tracks, pauses, stumbles, and mistakes — conveys a subtle but important message to the students: coding is always a process of trying, building, and learning. No programmer proceeds in a straight line from zero to complete.

Use every trick to make code readable. Zoom in to magnify the workspace (both Scratch and Trinket can do this) so the code can be clearly seen on the screen/board/projector. Point out that block instructions are colour-coded, which helps beginners find the blocks they're looking for. When it's time for students to actively build something, give very clear instructions of what to do next. Make sure everyone knows where they are supposed to be focused, and what they should try to achieve, particularly when opening “low floor” activities.

6. Know when to shift attention towards learners and away from the educator

As the workshop progresses, it may be harder to command attention, as more students focus on their emerging creations and become less interested in the stretch challenges you're presenting. This is to be expected and shouldn't be taken as either a failure on behalf of the educator to engage or the students to pay attention. It's a good sign that students



want to start driving their own learning! This is the time to start asking students to explain to their peers how they resolved a problem, demonstrate an interesting animation, or reflect on lessons learned. Students will always pay more attention if they expect the attention might shift to them.

Best practices for the second hour

The second hour of a workshop presents a new set of challenges: how to reinforce what is being learned, how to keep students engaged, and how to manage the gap that may start to grow between different kinds of learners.

1. Switch rooms (metaphorically)

At hour two, start with a new “low floor” (that is, a simple challenge in a different category of activity). This allows fast learners to try something new, while allowing slower learners a new chance to try something easy. If this means sacrificing a challenge activity in hour one, so be it. These activities can be returned to in future classes. A good workshop should engage *everyone*, not just the vocal students, the quick learners, and the students who already have some coding experience. Check to make sure that girls are receiving equal attention, particularly those who might be sharing computers with boys.

2. Look for opportunities to revisit and build on past learning

This helps students start to weave new skills into a more robust understanding. For example, ask students if they can remember the right blocks for an initialization sequence or another basic programming technique covered in the first hour. It’s also not a bad idea to revisit the code of conduct or protocols established at the outset.

3. Teach students what they need to help themselves

Keep your own hands away from students’ keyboards and resist the temptation to give them the solution right away. Give them suggestions, and let them think the challenges through. Continue encouraging learners to teach one another. Point to resources that can be used to discover answers on their own, including the help/tutorial section of the platform used.



4. Make mistakes

A lot, and in front of the whole class. Learn how to do this gracefully. It demonstrates that even “experts” make mistakes. Having the right attitude will help compensate for what is not yet known. Don’t be afraid to ask students if they see the problem. Once kids are busy with a new challenge, they’ll forget an awkward moment.

5. Make time to pause and reflect with the class

In the second hour, making the time for pausing and reflecting (and the insight and inspiration that are seeded in this time) is especially important. Students need time to think about what they’ve learned, demonstrate successes, and validate this new and exciting activity. Reserve 10 minutes at the end of the session and give the class one or two prompts, such as: problems and solutions they encountered; how they feel about what they accomplished; ideas generated; or how this activity might have changed their perspective on coding or computers.

Assessment

KCJ’s approach to assessment

At Kids Code Jeunesse our goal is for students to learn basic concepts and practices in computational thinking and code. Our lesson plans present both easy and stretch challenges that leave plenty of room for making, tinkering, experimenting, and collaborating. The language and platform can vary—Scratch, Python, even JavaScript, or a physical computing tool like micro:bits, to name a few—but the concepts and practices are universal.

We’ll focus here on two ideas for rubrics that assess the concepts and practices. One is for the student’s own self-assessment and the other is for an educator who has become comfortable assessing beginner code. Both types of assessment are crucial for the development of self-efficacy, particularly in competency-driven curriculum. Educators who are in the process of learning to code themselves may prefer to lean towards



self-assessment with the objective of eventually being more equipped to assess the end product of a coding project.

As discussed in the computational thinking practices section of Part 2, student sheets or design journals can be used to record stumbling blocks and document steps taken to achieve an outcome. Some teachers even give partial marks for productive “failure,” which can be defined by the number of iterations attempted, or by achieving a more interesting outcome because of an unexpected result. Encourage students to change their plan when appropriate.

The example of a self-assessment rubric presented here was developed by the researchers at Harvard’s [Scratch Ed](#). It links to other rubrics in other computational practices like debugging and modularizing. The educator-led rubric was developed by Mike Deutsch, director of Education Research and Development at KCJ.



Rubric 1: Self-assessment of computational practices

Computational practice (experimentation & iteration), student self-assessment:

EXPERIMENTING AND ITERATING	LOW	MEDIUM	HIGH
Describe how you built your project.	Student provides a basic description of building a project, but no details about a specific project.	Student gives a general example of building a specific project.	Student provides details about the different components of a specific project and how they were developed.
Describe different things you tried out as you were working on your project.	Student does not provide specific examples of what they tried.	Student gives a general example of trying something in the project.	Student provides specific examples of different things they tried in a project.
Describe revisions you made to your project and why you made them.	Student says they made no revisions, or only states they made revisions but gives no examples.	Student describes one specific revision they made to the project.	Student describes the specific things they revised in the project and why.
Describe a time when you tried to do something new.	Student provides no examples of trying to do something new.	Student provides a general example of trying to do something new in the project.	Student describes specific new things they tried in a project.

Source: Harvard ScratchEd

More computational practice rubrics can be found here: bit.ly/CTrubric



Rubric 2: Educator led assessment

Computational concepts, educator assessing a student coding project:

	4	3	2	1
Sequencing & loops	Code uses loops and sequences in complex ways (e.g., loops inside other loops).	Code is properly sequenced and uses loop blocks correctly.	Code is properly sequenced, but does not use loop blocks.	Sequences are wrong, and there is little (or mostly ineffective) use of repetition.
Selection & logic	Selection (if/then conditional statements) and logic are used in complex ways and/or in combination with other concepts like sequencing, loops, and variables.	Selection and logic are clean and work well.	Selection and logic work partially.	Selection and logic do not work.
Abstraction - variables	Uses variables (storing, setting, and using values) in complex ways: in combination with other concepts like sequencing and loops, or to simplify repeated code.	Uses variables properly to store, set, and use values.	Uses variables, but not correctly. May be storing values but not using them or updating values correctly.	Does not make effective use of variables.
Abstraction - algorithm	Uses modularization (functions and naming of sequences) in complex ways and/or in combination with other concepts.	Algorithm is complete and uses modularization to simplify and move towards complexity.	Algorithm is complete and does what it intends to do, but is one large block of code, no modularization.	Algorithm is messy or at least partially unbuilt.



Conclusion: Code for life

KCJ has received many testimonials over our six years, but there's one in particular that convinced me we were heading in the right direction. This was from a teacher from School District 71 in Victoria, B.C.

"Before the workshop I had ordered two books about Scratch for the library. When I opened them, I was dismayed to see that I had no understanding of the content and I wondered how students would make use of the resource. When I got home and looked at the books again it was like learning to read for the first time. I knew exactly what the authors were explaining to students. I feel confident that I can bring this to our classroom, thank you!"

I know how this teacher felt. It's the change from anxiety—what I felt the first time I looked at a line of JavaScript and tried to complete a tutorial with no help—to a feeling of clarity. For me, clarity started to emerge when, thanks to my son, I started to decode that jumble of text. I knew then that with the right amount of effort, applied incrementally, I could continue to learn more.

As someone born well before the Internet became such a ubiquitous place on the planet, learning to code felt like more than just learning a skill. I truly felt like I had changed my status from digital immigrant to digital citizen. It's what I imagine any immigrant feels when they learn to read in the official language or languages of their new country.

I got to this point of "algorithm literacy" through what is called the "bottom up" path, the path that many professional programmers and computer science undergraduates take. I took it on faith that what I was learning was useful, but it took about three months before I was able to start seeing programming from the "top down." It also wasn't until I learned Python that I was able to see the concepts that translated across programming languages.



Let curiosity lead you and your students

Increasingly, educators are being asked, even mandated, to learn coding and teach it from a sense of obligation to future generations who will need 21st-century job skills. But it has never been, nor should it be, the job of public school teachers, particularly at the elementary school or middle school level, to teach children job skills. Moreover, these teachers are not given and don't have the luxury of 12 weeks of study to reach a lofty peak of learning where they can recognize the concepts being taught.

It is our experience that **educators are far more responsive when they are encouraged to learn coding from a sense of curiosity**, or as a way to increase their own sense of technological competence or digital citizenship. They feel more confident when they have a map that fits with the many maps they are supplied with in their various provincial curriculum plans. Most provinces have some kind of framework for skill development that follows a “know (concepts), do (practices), and understand (perception)” cycle. It is our hope that the guide we have provided will help educators to map these skills to their own provincial plan.

Change perceptions, change the world

We have kept this primer simple because we believe that educators need a plan that will make enough sense to them after 12 *hours* of learning – not 12 *weeks*. This might not sound like enough time to demystify something that has been as mystified as code, but learning code does not have to be complicated.

We are not teaching students all the skills they need to become professional programmers. Instead, we're teaching them the skills they need to believe that they could become professional programmers, if that was something they wanted someday. The job of an educator teaching code is first and foremost to find a way to enjoy programming, and then through candid assessment and the help of good self-assessment tools, help students build the willingness and confidence they need, to know they can improve.

Particularly if you are a primary or middle school teacher, your job is to change students' perceptions of what a coder looks like. As long as we continue in a society that fuels the misperception that coding is somehow a skill intended to be learned only by a certain gender, age, or socio-economic group, then we are doomed to remain stuck in this cultural loop of technological inequity.



Changing that perception starts with educators having the courage to present themselves as the kind of people who can learn to code, and who create classrooms where girls and boys of every background enjoy coding together.

Changing the world starts with you making a decision to learn to code. And once you learn that feeling of empowerment, it's like riding a bike – you become a coder for life. That is a feeling that you and your students will never forget!

APPENDIX 1: KCJ Lesson Plans

Introductory lesson plans for Scratch

Let's Move

This introductory lesson plan uses Scratch to explore basic coding concepts like sequencing and repetition. For a stretch challenge, students can be directed towards a project like Code Club's [Lost in Space](#), which explores a variety of different loops. Repetition blocks and working with a cartesian grid make for some easy curriculum ties to basic math and algebra.

Let's Draw

This lesson plan explores the same basic concepts, but with the pen blocks in Scratch 3.0. It's a great way to put a few constrictions on the blocks without limiting the range of challenges that kids with more programming experience can attempt. It also serves as a great introduction, or tie-in to basic geometry. Go further with this [drawing logos](#) project KCJ created in honour of Canada's 150th year birthday.



Let's Play

This lesson plan explores the use of data and the coding concept of variables, first by creating a “pedometer” that will measure the number of pixels a sprite can run, then by keeping score in a simple game. Students looking to go further can try the Code Club project [Ghostbusters!](#)

Let's Talk

Explore conditional logic and new concepts like “input/output” by making a simple chatbot that asks how you are and responds to your answer. Go further with Code Club’s [Chabot](#) project.

Find many more [Scratch lesson plans](#) at the Kids Code Jeunesse website

Introductory lesson plans for Python

Python is a popular, general purpose, text-based programming language. With Python students will learn more about core coding concepts like loops, variables, and conditions. They can build their own creative projects with greater speed and power than block code.

These lesson plans in [Python](#) offer a great start in text coding. Supplement them with Code Club’s Python projects.



APPENDIX 2: Annotated resources

[*Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers and Play*](#)

Mitchell Reznick, MIT press 2017

By one of the co-creators of Scratch, a wonderful book on the constructionist philosophy behind the code to learn movement. Reznick argues that in a skills-driven education landscape where kindergarten is becoming too much like the rest of education (think flashcards), we should consider making all education more like kindergarten. Creative thinking is better sparked by project-based learning focused on interests, collaboration, and above all playfulness.

[*Creative Computing Curriculum*](#)

Harvard Graduate School of Education 2019

Designed by a team of educators led by Canadian researcher Karen Brennan, this is a comprehensive, practical, and research driven guide to introducing Scratch into the classroom. Includes frameworks, and assessment rubrics.

[*How Learning Works: Seven Research Based Principles for Smart Teaching,*](#)



ed. Susan A. Ambrose, Michael W. Bridges, Michelle diPietro, Marcia C. Lovett, Mary K. Norman. Jossey-Bass 2010.

This exceptional book is a meta-study of recent research into learning. Each of the seven principles is based on consensus emerging from research in the science of learning. Principles like “students’ prior knowledge can help or hinder learning” apply especially to this challenge of teaching and learning programming. Includes helpful appendix on self-assessment, concept maps, and rubrics.

Acknowledgements

This primer would not have been possible without the many contributions made by KCJ instructors and employees, and without our partnership with Canada’s leading technology bootcamp Lighthouse Labs. It certainly would not have been possible without the opportunity given to us by the the Ministry of Education of British Columbia to create and test student modules across the province, the opportunities provided though funding from le Ministère de l'Économie et de l'Innovation du Quebec, and without the significant funding we received from the Federal Government of Canada’s CanCode program in bringing what we learned across the country.

In particular, Yasmin Ahmad did foundational work towards building our first computational thinking framework, which was partially funded by the Canadian Internet Registration Authority (CIRA). Lead instructor Meggie Carrier contributed much towards our computational thinking materials and best practices. Director of Research and Development Mike Deutsch devised our assessment rubrics. Wendy Hoy and Don Burks created the unplugged activities that were part of our B.C student modules and lesson plans.

KCJ would like to thank anyone internal and external who has contributed feedback and editing.

We welcome any further feedback, which can be sent to juliet@kidscodejeunesse.org.

