teachers learning code

# DIGITAL TOOLBOX:

## Quick Start Guide

An educational program by
CANADA LEARNING CODE

# Table of Contents

## WELCOME



## CODING FOR BEGINNERS



## LESSON PLANNING

## APPENDIX: TEACHING WITH TECHNOLOGY



**For more great lesson plans, check out our website:**
canadalearningcode.ca/lesson-plans

# Welcome to the Digital Toolbox:
## Quick Start Guide!

### What is the Digital Toolbox series?

The Teachers Learning Code Digital Toolbox is a series of guides designed to give educators the general skills, knowledge, and confidence to introduce technology and coding concepts to their K-12 learners.

In this series, we share resources to familiarize yourself with, tips and tricks for teaching digital skills, and lesson plans to empower and teach the future generation of technologists across Canada.

### Quick Start Guide

The guide you are reading right now is our Quick Start guide and includes high-level information on how to bring computer science and coding into your classroom. This guide is designed to be used in conjunction with our more specific 'how-to' guides, which include not only the basics of our favourite technologies and coding tools, but also lesson plans ready-made for classroom delivery!

### Who are the Digital Toolbox guides for?

The Digital Toolbox guides were designed specifically with K-12 educators in mind, but can be used and adapted by non-traditional educators, such as program coordinators at community centres, homeschooling parents, Girl Guide troop leaders, etc., as well.

### What will the Digital Toolbox accomplish?

Through computer science education, we want to inspire all people in Canada to become empowered digital citizens who can understand, participate, and shape our country as creators and innovators of technology.

### Learning Objectives

After reading the Teachers Learning Code Digital Toolbox: Quick Start Guide, educators will be able to...

- Identify the skills and competencies covered by teaching computer science and coding
- Define foundational coding concepts and and explain them using relatable analogies
- Identify and modify (or "remix") elements of Canada Learning Code lesson plans
- Apply 4 strategies for modifying the curriculum to meet unique learner needs
- Support learners through the process of debugging and troubleshooting technological projects by using the debugging prompts and checklist
- Apply 7 strategies for the facilitation of technology-based workshops

### Who built the Digital Toolbox guides?

The Digital Toolbox series was developed by Teachers Learning Code under the national charity Canada Learning Code. The lesson plans have been developed in partnership with educators and industry-leading experts as indicated.

# Coding for Beginners

**What is Code?**

The simplest explanation for code is that it is a set of instructions that are given to a computer in order to execute a certain task. When we put these instructions together in a certain order, we call this an algorithm (a set of step-by-step instructions to follow in order to solve a problem).

Computers take direction extremely literally, which means that any instructions in your algorithm should be precise and specific. For example, when reading a peanut butter and jelly sandwich recipe (an algorithm in itself!), a human would likely understand that "Put the jam on the bread" is the direction to spread some jam on a slice of bread. A computer, however, might interpret this entirely differently. Should the robot put the jar of jam on top of the bread bag? This might seem silly to you, but you will soon see how literally a computer takes direction. A clearer set of instructions for our robot might be:

1. Turn the lid on the jam counterclockwise until it is completely loosened
2. Lift the lid off the jar of jam
3. Place the lid beside the jar of jam

... and so on. Watch a great demonstration of this concept at **bit.ly/pbandj-challenge**.[1]

Code is extremely versatile; we can use it to control computers/robots, build webpages, create video games, analyze large datasets, and more! The possibilities are endless. While we use different coding languages to complete different tasks (e.g. HTML & CSS for web-building, R for data analysis, Scratch for game building, MakeCode and python for micro:bit, and Javascript for flying robots!), each of these languages is really just a different way of communicating your instructions, or your algorithm, to the computer.

# Why Teach Computer Science and Coding?

In addition to teaching important skills and competencies, Computer Science education will also equip learners with the capacities and dispositions required to meet the needs of their times.

**Discovery:** Computer Science education will inspire learners to approach problems with curiosity and a sense of discovery. It will encourage learners to try new things, approach tasks with a growth mindset, and iterate as they master new skills.

**Critical Thinking:** Having a better understanding of how computers operate will equip learners to more critically engage the social, legal, ethical, and political implications of technology.

**Team Work:** Computer Science encourages learners to work together and develop projects, promoting effective communication, project management skills, and empathy for the perspective of others. It also encourages learners to constructively give and receive feedback and fosters a willingness to help and share with others.

**Citizenship:** Computer Science education will help learners understand the ways in which technology impacts society, helping them to become technological stewards who will harness the power of technology to improve the world around them.

**Resilience:** Creating digital artifacts will help learners to become more comfortable with trying new things, making mistakes, and learning through experience. By stressing the importance of continuous and unexpected learning, learners will ultimately become more resilient and see opportunity in failure.

**Creativity:** Computer Science will encourage learners to explore their creativity, think outside the box, and develop innovative solutions to address issues that affect them, their community, and the world.

**Computational Thinking:** Computer Science will help learners employ computational thinking strategies to problems within and outside the digital world. See the 'Computational Thinking' section for more information.

**Fundamental Programming Concepts:** Computer Science education teaches the programming concepts that are the shared foundation of hundreds of unique programming languages. When learners understand these key concepts, they are able to switch languages more seamlessly. See the 'Coding Concepts and Key Terminology' section for more information.

# Computational Thinking

Computational thinking involves the thought processes and strategies involved in formulating a problem and articulating the solutions in such a way that a computer (or another human) could action on. It involves breaking big problems into smaller parts, and describing specific steps to overcome these smaller challenges. Computational thinking concepts aren't unique to coding—you'll notice they are the principles many people use in their day-to-day lives to solve problems of all sorts.

Computational thinking involves the following key steps:

### I. Decomposition

Decomposition involves breaking down large problems into smaller, more manageable, steps, just as we might break down a book report into different sections.

### II. Pattern recognition

Pattern recognition involves identifying the shared characteristics of these smaller problems to help us make predictions, create rules, and solve problems more generally.

### III. Abstraction

Abstraction involves isolating differences between our smaller problems in order to make one solution work for multiple problems. In this way, abstraction helps us decide what's important and what's not. It helps manage complexity, like when we decide what information is needed to help solve a math equation or word problem.

### IV. Algorithms

Creating an algorithm means creating a set of step-by-step instructions to follow in order to solve the identified problem(s). Algorithms are common in our everyday lives — a lesson plan is an algorithm for a class, while a recipe is an algorithm for making our favourite dish. Writing out step-by-step instructions in plain English is what we call 'pseudo code'.

### Coding Concepts and Terminology

There are hundreds of computer programming languages, and although they may look nothing alike, they are built on a shared foundation of key concepts.

As you will see, many of these coding concepts come up in daily life and aren't as daunting as you might think. Consider using the real-world examples provided when explaining difficult concepts to your learners!

# Coding Concepts and Terminology

There are hundreds of computer programming languages, and although they may look nothing alike, they are built on a shared foundation of key concepts.

As you will see, many of these coding concepts come up in daily life and aren't as daunting as you might think. Consider using the real-world examples provided when explaining difficult concepts to your learners!

| Concept/Term | Definition | Real-World Example or Analogy |
|---|---|---|
| Algorithm | A set of step-by-step instructions to follow in order to solve a problem | We all follow a similar algorithm when brushing our teeth. We put toothpaste on our toothbrushes, scrub our teeth with said toothbrush, and then rinse our mouths. |
| Arrays | A special variable that can store more than one value at a time; items are ordered by a number so that we can access them later | An array is like a photo album for your last family trip. The album allows us to store lots of related memories (photos!) in one place. When we want to look back at these specific memories, we just pull up the photo album. |
| Boolean Logic | 'and', 'or', 'not' are examples of boolean operators; values that are either true or false | We could use boolean logic to sort cats based on their fur colour.<br>Group 1: Cats that are black AND white<br>Group 2: Cats that are fully black OR white<br>Group 3: Cats that are only black, NOT white |
| Bug | An error in a program that prevents the program from running as expected | When you turn on your hose to fill your pool, you expect that water will come out of its opening. Kinks interrupt the water flow and prevent the hose from working as expected. In this case, the kink is a bug! |
| Coding Language | Means through which humans communicate with computers (e.g. HTML & CSS, Python, Scratch, JavaScript, etc.) | We all speak different languages to each other based on what we understand. In this guide, we are using English, but you and your family members might speak a different language. Just like we have different languages to speak to one another, the computer has different coding languages to use or interpret. |
| Command | An instruction for the computer | Just like we ask trained dogs to complete tricks or commands, we can command computers! |
| Conditional statements | Allow computers to make decisions based on certain conditions being met; if/else statements are commonly used in conditional statements | We use conditional statements when we make decisions about our lives all the time! For example:<br>If it is raining, then I will use my umbrella. If it's not raining, I won't! |
| Debugging | The act of finding problems or 'bugs' in our code and solving them | In the hose example, the debugging process would involve 1) looking along the entire length of the broken hose to identify any problems (a hole, a kink), and 2) taping the hole or straightening the kink! |
| Events | When one act triggers another to occur | When the clock strikes 12 PM, we eat lunch! At 3 PM, we leave school! |

# Coding Concepts and Terminology

| Concept/Term | Definition | Real-World Example or Analogy |
|---|---|---|
| Function | A named section of a program that performs a specific task; can be used over and over again | Think of a function as the chorus of a song. When you are writing a song, you only have to compose the chorus once because it is repetitive. After composing and labeling it as the chorus, you can write "Chorus" anywhere you want it to be repeated within the song, and people will know what you mean! This saves time. |
| Index | The specific location of an item in an array; the first item in an array has an index of 0 | When your teacher takes attendance, they are reading from a list (or an array) of all the students in your class. Depending on your last name, you will appear at a certain place (or index!) on this list. |
| Input | Information the computer senses from its environment or information the user supplies to the computer | When we want a certain bag of chips from a vending machine, we press a letter and number (e.g. A5). This is known as the input. |
| Loops | Allows us to run the same sequence multiple times, as long as a certain condition is met | As long as it is still dirty, I will continue to wash my dish. I will only stop when the dish is completely clean!<br><br>Some loops can go on forever! A real life example of this is time. Time goes on... forever! It never stops. |
| Modularizing | Exploring connections between the whole and its parts | In modular code, the larger program is divided into smaller parts for easier management and readability. Each of these smaller parts has its own role within the larger program! Think of this as a team working on a school project. If one person tried to complete the entire project, it would be much more difficult to manage. Dividing up the tasks makes it easier for everyone to function and for the project to be completed more efficiently (as long as the team members are willing to communicate!)<br><br>We also use modularization when solving math equations like =4×7+5÷2. In order to solve this problem, we break it into smaller chunks like (4×7) and then (5÷2). |
| Operators | Mathematical and logical expressions<br><br>Relational operators are used for comparison (e.g. == (equals), != (does not equal), >= (greater than or equal to), etc.)<br><br>Arithmetic operators are used for calculations (e.g. + (addition), - (subtraction), etc.) | Symbols dictate our actions all the time! When we see a stop sign, we stop. We know how to leave buildings by looking for the EXIT sign. We know what the '+' or '-' symbols mean when we are calculating the answer to a math problem, and our computers use these same (or very similar!) symbols. |
| Output | Information the computer supplies to its user | After we press a letter and number (e.g. A5) in our vending machine example, the machine releases our treat as the output. |

# Coding Concepts and Terminology

| Concept/Term | Definition | Real-World Example or Analogy |
|---|---|---|
| Parallelism | Making things happen at the same time | Can you rub your belly and tap your head at the same time? This is a real-world example of parallelism where our brains are instructing our arms to do 2 separate things at the same time! |
| Program | An algorithm that has been coded to be run by a machine; written in a coding language (synonymous with 'code' and 'software') | Think of a program as a secret family recipe that was passed down from your grandparents. The recipe is the algorithm and the fact that it was written down to be prepared by someone else makes it similar to a program (Keep in mind that programs are only run by computers, not people!) |
| Remixing | The process of taking an existing project or idea and making it new by changing or adding to it | DJs remix popular songs by changing the tempo, adding additional sounds, sampling from other songs, etc. |
| Sequence | An ordered series of steps for a task; computers read and perform tasks in order from top to bottom | Think back to our algorithm example. Have you ever tried to brush your teeth by rinsing your mouth first and scrubbing your teeth with toothpaste second? The sequence or order of the commands matters! |
| State | State in a programming sense is just the same as state in a non-programming sense (ie. the TV is on or off). Variables have states, values don't. For example, 42 is 42 and there's nothing you can change about it | The state of a TV or light switch can be on or off! |
| String | A type of data that represents text | Any line of text can be a string. A word, a sentence, a line of gibberish, etc. |
| Syntax | A set of rules for physically writing the coding language such that computers are able to understand; Scratch and the MakeCode editors blockly structure removes the need for syntax | Different human languages (like English, Spanish, and French) have different vocabularies and figures of speech, as well as spelling and grammar rules that we must follow. When we don't follow these rules, other people may find it difficult to understand what we mean. Using proper syntax for coding languages ensures the computer is able to understand our commands! |
| Variable | Allow us to store a single piece of information | We can think of a variable as our piggy bank or wallet, which stores our money. The amount of money we have can change depending on our actions. Did we pay rent or go on a shopping spree? If we want to know how much money we have at any given time, we just need to look inside our piggy bank or wallet. |

# Debugging Strategies

It can be tempting to jump in and solve your learners' problems when they get frustrated, but troubleshooting and making mistakes is an important part of the learning process.

When something goes wrong with a project, be calm and use it as a teaching opportunity to show your learners how to debug technology and think through problems. Set expectations! Let learners know that encountering a block of code that doesn't work as expected is simply a part of programming. Debugging is coding!

## Prompts

Consider using these prompts to encourage and motivate learners who might be having trouble getting started with the debugging process[2]:

"Let's break this down. What should we do first?"

"If I'm the user of this project, how am I supposed to be interacting with it?"

"If you read the code you're building like you are reading a sentence, what do you notice? Is anything missing?"

"What are you trying to do? What do you think isn't working? What have you tried already?"

"Go [here]. Do you see anything there that might be useful for this problem?"

## Checklist

Ask your learners to use the following checklist when debugging[3,4,5]:

Have you tried...

- ☐ **Reading your code aloud?**
  Reading your code aloud, in a similar way to how you would read a sentence, can be helpful for identifying gaps in logic or missing components.

- ☐ **Asking a friend to review your problem? Or working on the problem together?**
  Sometimes a fresh set of eyes is all it takes.

- ☐ **Isolating and testing your code?**
  Revert back to stable code that you know works. Then, try testing out the additional lines of your code by removing one command at a time. Add the commands back one-by-one until you replicate your problem.

- ☐ **Thinking like a computer?**
  Are your steps specific enough? Remember, computers aren't very smart. We have to tell them exactly what to do.

- ☐ **Looking it up?**
  Google is your friend! If there is something wrong with your WiFi, laptop, keyboard, code, etc., try typing the issue into Google and see if there's a solution! There often will be!

  If Google isn't useful, all digital tools usually have a 'Help' or 'Reference' section you can explore.

- ☐ **Trying something new out?**
  When in doubt, try it out! There is always the undo and redo button!

- ☐ **Taking a break?**
  It can be difficult to think clearly when we are frustrated.

- ☐ **Resetting the machine?**
  Though resetting may not be an ideal solution when you are coding, if you are working with unresponsive hardware like robotics, this might be the solution.

# Lesson Planning

There are many ways to bring coding to your youth! You could start an after-school coding club or incorporate coding into pre-existing classes. For example, instead of having your learners write a report on a historical figure or incident, you might ask them to create a website instead! The possibilities are endless!

## Minimum Requirements

You will need:

- [ ] **An Educator**
  To supervise learners and facilitate sessions and coding challenges.

- [ ] **Volunteer Mentors (optional, but recommended)**
  To support learners with their questions as the educator leads the session. Consider asking more advanced learners to play this role.

- [ ] **Content**
  Consider using the lesson plans in our Digital Toolbox series or the lessons on our website at https://www.canadalearningcode.ca/lesson-plans/.

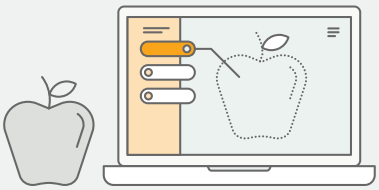
- [ ] **Access to Hardware**
  - [ ] Laptop or desktop computers (at least 1 per pair of learners)
  - [ ] Extension cords and power bars
  - [ ] Projector, projector screen, and appropriate dongles to connect (or another way to display your screen to learners)
  - [ ] Hardware (as specified in lesson plan or 'how-to' guide)

- [ ] **WiFi connection (optional, but recommended)**
  Some tools can be used without an internet connection, while it is a strict requirement for others.

- [ ] **How should our room be set up?**
  Each room should have 1 educator computer and a projector/screen to display the slides to your group. Laptops for learners should be set up such that the educator's screen is visible from any seat.

# Suggested Structure of a Lesson

We recommend that each lesson include these main sections:



### 1. Introduction

The introduction sets the stage for the lesson. Start the session with an icebreaker and a review of classroom expectations. At Canada Learning Code, we refer to these expectations as our Coders' Code (see 'Managing Behaviour' section for additional details on this concept). After this, pose essential questions related to the topic of the workshop and provide any background information necessary. Wrap up this section by showing the learners an example of the main project they will be building!

### 2. Code-Along

During the Code-Along, you should introduce or review the main tool being used through an exploratory learning activity. Give learners a chance to play around with and test out the tool, providing a bit of structure through 'challenges'. Demonstrate problem-solving and resourcefulness while solutioning by asking (rather than telling) learners where to find things, verbalizing your thought process from A to B, and redirecting questions towards other learners, or towards the reference.

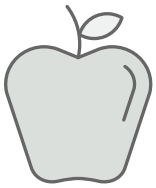
### 3. Work Session

During the Work Session, start by giving the learners the chance to work with you to build the main project by answering questions and completing challenges with your guidance. At this point, the Work Session may look similar to the Code-Along. As they get comfortable, provide learners with more independent time for finishing the project and adding customizations (remixing!) on their own. Educators should walk around to support learners as they build. If learners complete the project, challenge them with the add-ons or extensions found in the lesson plan.

### 4. Wrap-up

As you wrap-up the lesson, give learners the chance to see what their friends have created! You might have learners present their project at the front of the room, or lead a Gallery Walk, where learners open their laptops, tuck in their chairs, and walk around the room to explore each other's final projects. Before you send learners off, be sure to review any coding concepts covered, as well as learning outcomes achieved!

# Educator Preparation Checklists



### Before each lesson...

*Prep Time: 2 hours*

- ☐ Review the lesson plan and any supporting materials (slides, example project, solution sheet, handouts, etc.)
  - ☐ Review relevant coding/programming concepts and ensure you are confident describing them to your group
  - ☐ Code and/or build (if applicable) the main project, ensuring you are comfortable with all outlined steps (consider sharing this project with your group as an example!)
  - ☐ Run through the lesson to minimize the risk of tech issues
- ☐ Create online accounts (if applicable)
- ☐ Prepare for WiFi failure
  - ☐ Print a copy of the lesson plan and solution sheet
  - ☐ Download offline software and transfer the files to a USB
- ☐ Print learner resources (if applicable)
- ☐ Locate and gather technology and any additional resources
- ☐ Set up equipment
- ☐ Write Coder's Code, WiFi password, and any required login credentials somewhere easily readable by learners (e.g. whiteboard, chart paper, etc.)

### After each lesson...

- ☐ Clean up your space
- ☐ Take some time to reflect on the experience:
  1. What problems or obstacles did you encounter in this lesson? How did you solve these problems?
  2. What obstacles do you anticipate learners might encounter next time? How can you prepare to ensure learner success?
- ☐ Get feedback from your learners—what worked for them and what didn't?

# Lesson Plan Modifications

## When to modify the lesson?

Small adjustments to the content may be necessary if you notice your group is not able to fill, or complete the lesson within, the recommended time frame. You may also want to remix the content to fit within another subject.

## How to modify the lesson

There are several ways you can modify our lesson plans to meet the unique needs of your particular group.

## For advanced learners, consider:

**1. Add-ons and Extensions**

Add-ons and extensions are additional challenges and tasks that can be attempted by the learners once the main project has been completed. These are a great way to challenge advanced learners and fill extra time. All of our content has add-ons and/or extensions built into the lesson plan, but feel free to pose your own challenges, as well!

**2. Remixing**

Following each Code-Along segment, there is a Work Session where learners are encouraged to personalize the main project. We call this 'remixing' the project! If your learners speed through the Code-Along, you can lengthen the Work Session, encouraging learners to get creative. Consider using prompts to brainstorm additional ideas as a group. For example, ask: "How can we make our project more [fun, challenging, silly, etc]?" or "What is this project missing?"

## For learners that may be falling behind, consider:

**1. Pair-Programming**

Pair-programming is a software development technique where two developers work together on a project. Consider pairing a more advanced learner with a learner that is falling behind, or pairing two learners at a similar level. In either scenario, having two learners collaborate on the project will allow them to move through each task at a faster pace and will encourage problem-solving within the pair. See the 'Facilitation Strategies' section for more on pair-programming.

**2. Lengthened Workshops**

If the allotted time is not enough to complete the main project, consider lengthening your sessions (within reason). Be sure to add in some breaks if you are going to take this approach! A single workshop could also be covered over multiple days if you feel it would be beneficial for your particular group. When splitting workshops into several days, take some time at the beginning of subsequent sessions to review concepts covered previously. For example, you could review these concepts yourself, or you could ask the learners to remind you what they learned or to demonstrate one thing they learned to the rest of the class.

Finally, when remixing or customizing content for your learners, consider the language or program as the tool through which learners might demonstrate their learnings in any given subject. To do this, it is helpful to recognize that much of Canada Learning Code's content (canadalearningcode.ca/lesson-plans) can be further broken down into three sections: the subject(s), the project, and the programming tool or language. For example, in the original Wildlife Soundscapes lesson plan, learners explore the boreal forest ecosystem (the topic) by building a digital soundscape (the project) in Scratch (the tool).

To customize this content further, swap one of these three components. You could remix the project by asking learners to demonstrate their understanding of a specific ecosystem by animating a short story in Scratch. Alternatively, you could remix the subject of this lesson by having learners explore Canadian history by creating a digital soundscape in Scratch that includes the sights and sounds of a specific decade.

If you treat these lesson plans as a starting point and aren't afraid to get creative and have fun with your remixing, the possibilities are virtually limitless.


# Tips for Teaching Code & Having Fun While Doing It

1. Ensure you are familiar with the tool, but don't worry about being an 'expert'—allow your learners to teach one another.

2. Have a clear vision for what you want to accomplish and find a champion in your school to co-teach with you! It is all about integrated learning.

3. Have a growth mindset or 'fail-forward' approach. Spread the belief that abilities are not dictated by talent alone, but can be developed through hard work and perseverance.

4. Bring outside experts in. Invite guest speakers and volunteers from the community to lead mini lessons and be there as extra support.

5. Let your learners and their creativity be your guide—what do they want to explore more? What do they want to learn?

# Facilitation Strategies

New to facilitation or nervous about leading your session? Consider some of our favourite facilitation strategies:

**1. Pair-Programming**

Pair programming is a software development technique where two developers work together on one program. At any given moment, one developer is actively writing the code (the 'driver'), while the other reviews the code for errors (the 'navigator'). The developers will switch roles frequently, getting a chance to flex their skills in both areas.

There is a lot of research that supports that learners in this age group perform well together when working with peers on tasks![6,7,8] Learners will move through the material at a faster pace and will save troubleshooting time by catching errors as they go. The collaborative nature of pair-programming means they will be exposed to unique perspectives and will default less to educators for direction when they encounter a problem. Finally, pair-programming challenges the stereotype of developing being an antisocial career choice, which can be beneficial for promoting diversity within the field.

**2. Scaffolding**

Scaffolding is used to set learners up for success by moving them towards greater independence in the learning process over time. Scaffolding starts with full educator instruction to the learner, then to educator-guided instruction *with* learner participation, and ends with independent learning *for/by* the learner.

This model is reflected in our recommended structure of a lesson. For example, during the introduction, the educator demonstrates the completed project (instruction to the learner). During the Code-Along, the educator demonstrates how to use the tool and works with the learners to complete various challenges (facilitation-guided instruction with learner participation). Finally, during the Work Session, learners are given time to build independently, having acquired the skills to add or try new things earlier on in the lesson (independent learning *for/by* the learner).

**3. Inside-out Approach**

The inside-out approach is a facilitation strategy that is useful when we want learners to look beyond the 'what' of a concept and move towards understanding the 'how' and the 'why'! For example, we don't want our learners to stop with the knowledge that loop blocks (like 'forever' and 'repeat' blocks) make things happen more than once. We want them to see how the two blocks are different and why they might use one over the other.

Coding tutorials often show us, step-by-step, how to write a program from top-to-bottom. When using the inside-out approach, we often start with the innermost blocks and ask guiding questions to determine what to do next. In this way, the inside-out approach is important because it ensures learners are not just blindly following instructions. It teaches the importance of sequence (a fundamental coding concept!) and creates opportunities for learners to put blocks in the wrong place, problem solve, and use their creativity and decision-making skills to learn how and why something works.

# Facilitation Strategies

**4. Student-Led Discovery**

We want learners to be curious. We want them to break things, to try something new to see if it works, and to be okay with changing direction. Your learners need to be engaged in the education process and one way to do that is by asking questions and taking suggestions throughout the workshop - even if you know said suggestion is not the right answer!

If a learner doesn't know something, have them ask their elbow partner or someone at their table, or when in doubt, try it out! There is always the undo button.

**5. Team Teaching**

Team teaching is exactly what it sounds like: two or more educators working together to help the learners work through the content. If you have two educators running a session, you might decide to collaboratively lead each session, or decide things run smoother with one educator leading and the other acting as a mentor.

Regardless of the specifics, team teaching gives your learners different perspectives on how to solve problems and means they will have more chances for one-on-one support. It can also be comforting for new educators to know they are not alone!

**6. Learner Mentorship and Co-facilitation**

Some of the best educators are not subject matter experts, but are simply a few steps ahead of their learners! This is because they can often relate to and teach to the common pitfalls of current learners, having recently overcome these same issues themselves.

If you have advanced learners in the room, consider asking if they would be interested in mentoring other learners or joining as your 'co-facilitator' for part of the workshop. Mentorship and co-facilitation have the potential to benefit many individuals within your group. Not only does it foster the development of leadership skills within your advanced learners, but it will solidify their understanding of key concepts. For groups with a wide range of prior experience, having additional mentors can help level the playing field and reduce the burden on educators.

**7. Hands-Off Approach**

Whenever possible, we want to keep our hands off the learner devices. Even though it might be tempting to quickly jump on and fix something for them, it is important that the learners get into the habit of problem solving on their own. Let us be clear: this does not mean you should not help your learners. It simply means we will not do things for them that they can troubleshoot themselves! For learners that are really stuck, you can always verbally direct and point them in the direction of the right answer without physically touching the keyboard and trackpad/mouse to complete the problem yourself. At the very least, they will have gone through the motions themselves!

# Managing Behaviour in the Presence of Technology

If you've ever seen youth with technology, you might be nervous about the prospect of directing their focus away from the screen and onto you!


Here are our recommendations for maintaining active engagement during your sessions:

**1. Coders' Code**

The Coders' Code is a collaborative social agreement that highlights the type of behaviour expected from all participants during the lesson.

Consider creating your own Coders' Code with your group at the start of the workshop. Ask your learners what they need from you to have fun and learn. What do they need from each other? What do you need from them?

Coders' Code Example

We will...

- listen to whoever is speaking at the moment.
- show patience and kindness to others in the room.
- ask questions when we are having trouble.
- support others who may be stuck.
- try our best to complete each task!

Agree upon 4-5 expectations with the group and write them somewhere the learners can see (chart paper, whiteboard, etc.) You can refer back to the Coders' Code when behavioural issues arise, especially if they violate the expectations everyone agreed upon! Finally, the Coders' Code is a living document so things can be added or removed, as needed!

**2. Politeness Mode[9]**

You can prevent screen distractions when you are speaking at the front of the room by using 'Politeness Mode'. When in 'Politeness Mode', laptops are closed halfway and the monitors of desktop computers are turned off, so the computer screen is not visible. Consider using this approach when walking through a new step at the front of the room. Have learners open their laptops or turn on their screens once it is their turn to complete a task!

**3. Transparency on Timing**

Create a predictable environment by writing down the schedule for each session (including break times!) somewhere visible, or using timers for periods of independent work, so learners know how much time they have left for each task. Try using **timer-tab.com** to project a countdown that everyone can see![10]

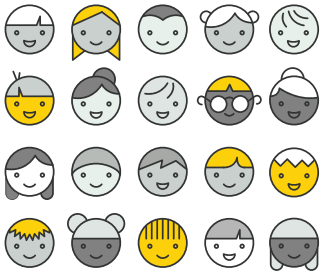
**4. Call-and-Response**

If you lose the attention of your group, don't bother trying to raise your voice! Use the call-and-response technique where you say something and the group has to respond.

Call-and-Response Examples:

Educator: "To infinity..." Learners: "And beyond!"
Educator: "Stop!" Learners: "Collaborate and listen!"
Educator: "If you can hear me...

- Wave like the queen!"
- Give your partner bunny ears!"
- Raise your left hand... raise your right hand... give yourself a high-five!"
- Snap 1 time. Snap 2 times. Snap 10 times!"



When working with technology, it is helpful to add some accessible physical components to keep learners' hands up and away from the keyboard when you have something important to demonstrate.

**5. Learner-driven Lessons**

Avoid giving a lecture or talking at your learners. Ask them questions, guiding them towards figuring it out and giving you the solutions instead!

Examples:

- **Think-Pair-Share**
  When asking for feedback, give everyone a chance to share their thoughts by using the Think-Pair-Share technique. First, give learners 30 seconds to think about their own answer to the question. Next, give them 1-2 minutes to share their ideas with 2-3 people nearby. Finally, ask for a few individuals to share what they discussed with the group. This is less intimidating than asking learners to share with the group right away.
- **Voting**
  Need to make a decision? Have learners vote by giving a thumbs up or thumbs down.
- **Demos**
  Ask learners to come to the front and demonstrate their solutions for the class using the educator's computer. Did another learner approach the same problem in a different way? Ask them to show the class their approach! Did someone figure out something really cool? Have them share with the group!

**6. Model Good Behaviour**

Some learners will get very frustrated when their computer isn't working the way they want it to. Try to reduce this frustration by modeling good behaviour. Show learners how to positively respond to technical issues and verbalize your thought process while trying to debug problems. See the 'Debugging Strategies' section for more information.

**7. Student-Led Discovery**

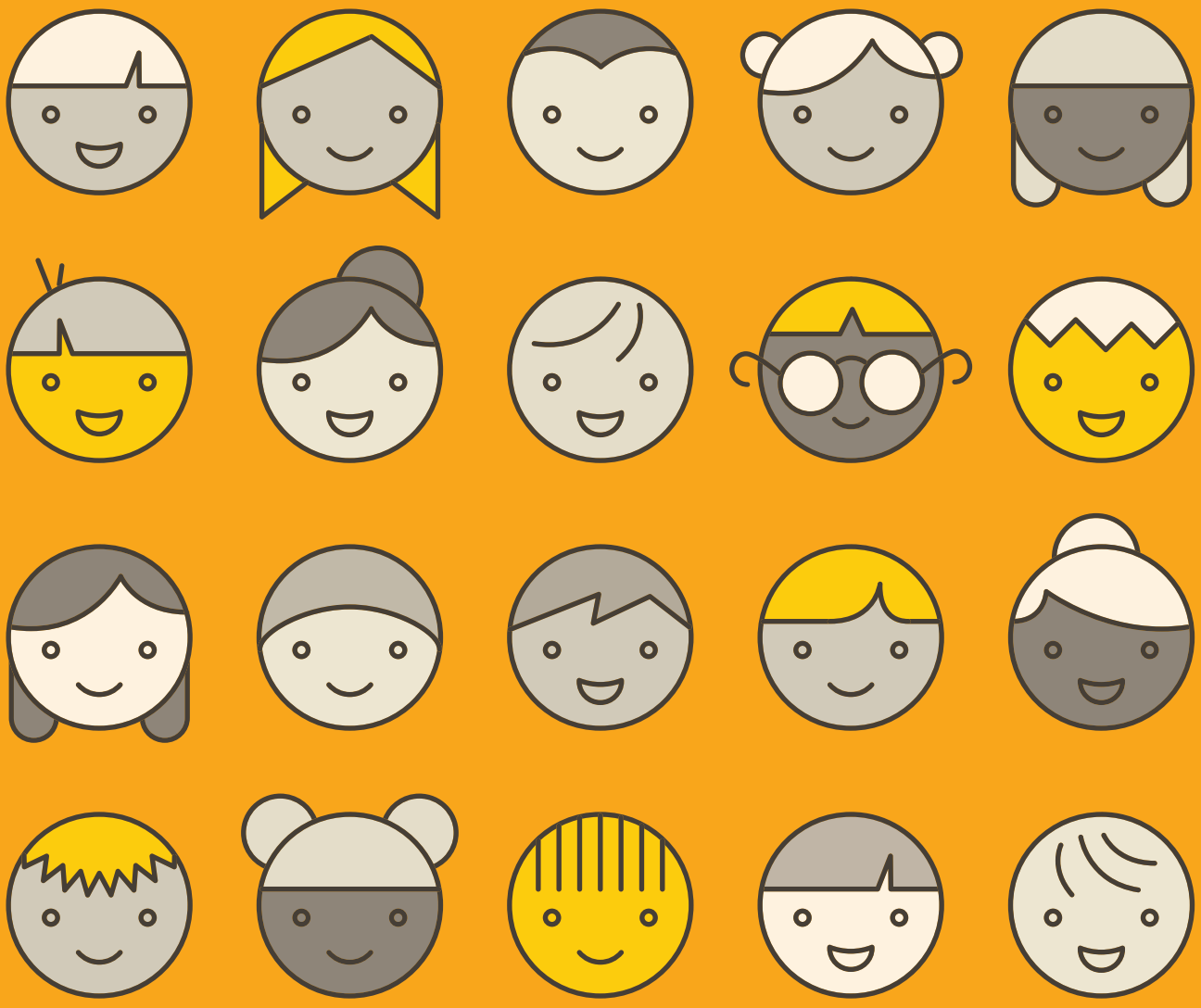
While each workshop introduces coding concepts and specific challenges and tasks in the Code-Along portion, allow learners to explore during the work session, running with their own ideas and questions while remixing. They will feel more engaged when they have autonomy over their projects. See the 'Facilitation Strategies' section for more information on Student-Led Discovery.

# Resources


1. "Exact Instructions Challenge PB&J Classroom Friendly | Josh Darnit." *YouTube*, uploaded by Josh Darnit, 18 Apr. 2017, https://www.youtube.com/watch?v=FN2RM-CHkul. Accessed 11 Mar. 2020.
2. Gustafson, Ingrid. "Overheard In the Classroom." ScratchEd, 20 Jul. 2016, http://scratched.gse.harvard.edu/resources/overheard-classroom-ingrid-gustafson. Accessed 10 Mar. 2020.
3. Roach, Emily. "Celebrating Mistakes and Embracing Process." ScratchEd, 3 Mar. 2017, http://scratched.gse.harvard.edu/stories/see-inside-classroom-emily-roach. Accessed 10 Mar. 2020.
4. ScratchEd Team. "Debugging in Scratch: Resources and Strategies." ScratchEd, 18 Apr. 2017, https://scratched.gse.harvard.edu/resources/debugging-scratch-resources-and-strategies.html. Accessed 10 Mar. 2020.
5. Tanguay-Carel, Matt. "Frustrations of Programming & How to Avoid Them." Codementor, 30 Nov. 2016, https://www.codementor.io/@matstc/avoid-frustration-as-programmers-ge54ddszr. Accessed 17 July 2017.
6. Goel, Sanjay, and Vanshi Kathuria. "A Novel Approach for Collaborative Pair Programming Executive Summary." Journal of Information Technology Education, vol. 9, 2010.
7. Lye, Sze Yee, and Joyce Hwee Ling Koh. "Review on Teaching and Learning of Computational Thinking through Programming: What Is next for K-12?" Computers in Human Behavior, vol. 41, 2014, pp. 51–61, doi:10.1016/j.chb.2014.09.012.
8. McDowell, Charlie, et al. "Pair Programming Improves Student Retention, Confidence, and Program Quality." Communications of the ACM, vol. 49, Aug. 2006, pp. 90–95, doi:10.1145/1145287.1145293.
9. Wilson, Greg. Data Scientist and Professional Educator at RStudio, Inc. LinkedIn. https://www.linkedin.com/in/greg-wilson-a26510b6/?originalSubdomain=ca. Accessed 13 Mar. 2020.
10. Brillout, *Romuald*. *Timer Tab*. https://www.timer-tab.com/. Accessed 7 Mar. 2020.

DATE:

DATE:




**GET IN TOUCH**

30 St Patrick St
Toronto, ON  M5T 3A3

info@canadalearningcode.ca
canadalearningcode.ca


teachers learning code