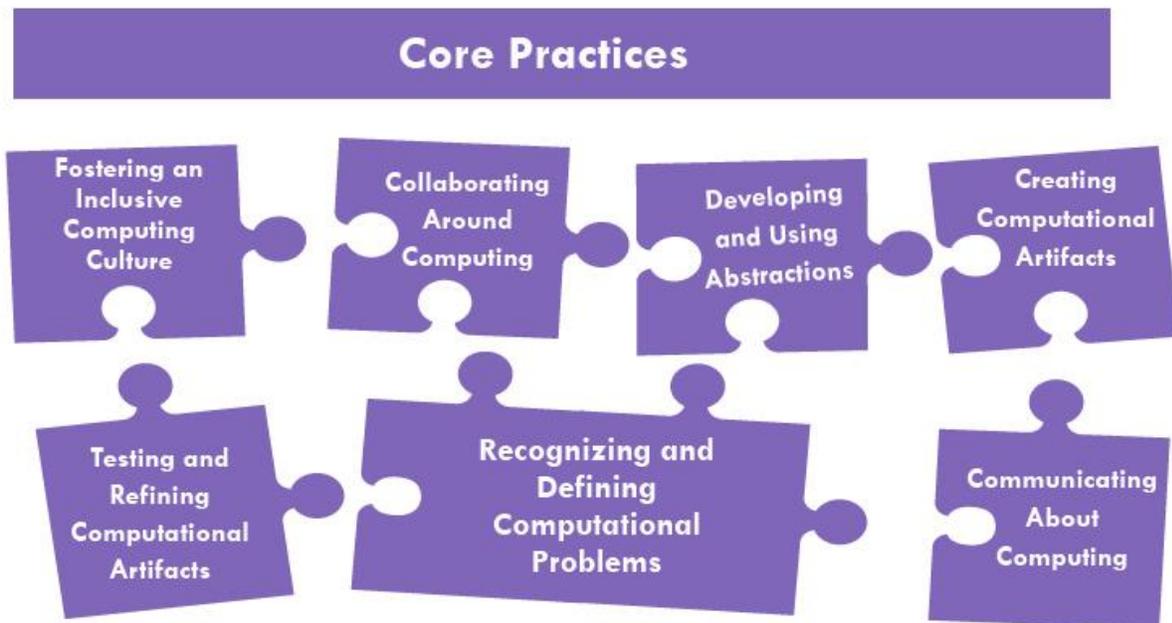


Practice Language⁵:**Practice 1: Fostering an Inclusive Computing Culture**

Building an inclusive and diverse computing culture requires strategies for incorporating perspectives from people of different genders, ethnicities, backgrounds, and abilities. Incorporating these perspectives involves understanding the personal, ethical, social, economic, and cultural contexts in which people operate. Considering the needs of diverse users during the design process is essential to producing inclusive computational products.

By the end of Grade 12, students should be able to:

1. Include the unique perspectives of others and reflect on one's own perspectives when designing and developing computational products.

At all grade levels, students should recognize that the choices people make when they create artifacts are based on personal interests, experiences, and needs. Students who are well-versed in fostering an inclusive computing culture should be able to differentiate backgrounds and skill sets and know when to call upon others, such as to seek out knowledge about potential end users or intentionally seek input from people with diverse backgrounds.

2. Address the needs of diverse end users during the design process to produce artifacts with broad accessibility and usability.

At any level, students should recognize that users of technology have different needs and preferences and that not everyone chooses to use, or is able to use, the same technology products. At the higher grades, students should become aware of professionally accepted accessibility standards and should be

⁵ K-12 Computer Science Framework. (October 2016) Retrieved from: <https://k12cs.org/wp-content/uploads/2016/09/K%E2%80%9312-Computer-Science-Framework.pdf>

able to evaluate computational artifacts for accessibility. Students should also begin to identify potential bias during the design process to maximize accessibility in product design. For example, they can test an app and recommend to its designers that it respond to verbal commands to accommodate users who are blind or have physical disabilities.

3. Employ self- and peer-advocacy to address bias in interactions, product design, and development methods.

After students have experience identifying diverse perspectives and including unique perspectives, they should begin to employ self-advocacy strategies, such as speaking for themselves if their needs are not met. As students' progress, they should advocate for their peers when accommodations, such as an assistive-technology peripheral device, are needed for someone to use a computational artifact. Eventually, students should regularly advocate for both themselves and others.

Practice 2: Collaborating Around Computing

Collaborative computing is the process of performing a computational task by working in pairs and on teams. Because it involves asking for the contributions and feedback of others, effective collaboration can lead to better outcomes than working independently. Collaboration requires individuals to navigate and incorporate diverse perspectives, conflicting ideas, disparate skills, and distinct personalities. Students should use collaborative tools to effectively work together and to create complex artifacts.

By the end of Grade 12, students should be able to:

1. Cultivate working relationships with individuals possessing diverse perspectives, skills, and personalities.

At any grade level, students should work collaboratively with others. As they progress, students should use methods for giving all group members a chance to participate. Older students should strive to improve team efficiency and effectiveness by regularly evaluating group dynamics. They should use multiple strategies to make team dynamics more productive. For example, they can ask for the opinions of quieter team members, minimize interruptions by more talkative members, and give individuals credit for their ideas and their work.

2. Create team norms, expectations, and equitable workloads to increase efficiency and effectiveness.

After students have had experience cultivating working relationships within teams, they should gain experience working in particular team roles. As students' progress, they should become less dependent on the teacher assigning roles and become more adept at assigning roles within their teams. For example, they should decide together how to take turns in different roles. Eventually, students should independently organize their own teams and create common goals, expectations, and equitable workloads. They should also manage project workflow using agendas and timelines and should evaluate workflow to identify areas for improvement.

3. Solicit and incorporate feedback from, and provide constructive feedback to, team members and other stakeholders.

At any level, students should ask questions of others and listen to their opinions. As they progress in school, students should provide and receive feedback related to computing in constructive ways. For example, pair programming is a collaborative process that promotes giving and receiving feedback. Older

students should engage in active listening by using questioning skills and should respond empathetically to others. As they progress, students should be able to receive feedback from multiple peers and should be able to differentiate opinions. Eventually, students should seek contributors from various environments. These contributors may include end users, experts, or general audiences from online forums.

4. Evaluate and select technological tools that can be used to collaborate on a project.

At any level, students should be able to use tools and methods for collaboration on a project. As students' progress, they should use technological collaboration tools to manage teamwork, such as knowledge-sharing tools and online project spaces. They should also begin to make decisions about which tools would be best to use and when to use them. Eventually, students should use different collaborative tools and methods to solicit input from not only team members and classmates but also others, such as participants in online forums or local communities.

Practice 3: Recognizing and Defining Computational Problems

The ability to recognize appropriate and worthwhile opportunities to apply computation is a skill that develops over time and is central to computing. Solving a problem with a computational approach requires defining the problem, breaking it down into parts, and evaluating each part to determine whether a computational solution is appropriate.

By the end of Grade 12, students should be able to:

1. Identify complex, interdisciplinary, real-world problems that can be solved computationally.

At any level, students should be able to identify problems that have been solved computationally. As they progress, they should ask clarifying questions to understand whether a problem or part of a problem can be solved using a computational approach. For example, before attempting to write an algorithm to sort a large list of names, students may ask questions about how the names are entered and what type of sorting is desired. Older students should identify more complex problems that involve multiple criteria and constraints. Eventually, students should be able to identify real-world problems that span multiple disciplines, such as increasing bike safety with new helmet technology, and can be solved computationally.

2. Decompose complex real-world problems into manageable subproblems that could integrate existing solutions or procedures.

At any grade level, students should be able to break problems down into their component parts. As students' progress, they should decompose larger problems into manageable smaller problems. For example, young students may think of an animation as multiple scenes and thus create each scene independently. Students can also break down a program into sub-goals: getting input from the user, processing the data, and displaying the result to the user. Eventually, as students encounter complex real-world problems that span multiple disciplines or social systems, they should decompose complex problems into manageable subproblems that could potentially be solved with programs or procedures that already exist. For example, students could create an app to solve a community problem that connects to an online database through an application programming interface (API).

3. Evaluate whether it is appropriate and feasible to solve a problem computationally.

After students have had some experience breaking problems down and identifying subproblems that can be solved computationally, they should begin to evaluate whether a computational solution is the most

appropriate solution for a specific problem. For example, students might question whether using a computer to determine whether someone is telling the truth would be advantageous. As students' progress, they should systematically evaluate the feasibility of using computational tools to solve given problems or subproblems, such as through a cost-benefit analysis. Eventually, students should include more factors in their evaluations, such as how efficiency affects feasibility or whether a proposed approach raises ethical concerns.

Practice 4: Developing and Using Abstractions

Abstractions are formed by identifying patterns and extracting common features from specific examples to create generalizations. Using generalized solutions and parts of solutions designed for broad reuse simplifies the development process by managing complexity.

By the end of Grade 12, students should be able to:

1. Extract common features from a set of interrelated processes or complex phenomena.

Students at all grade levels should be able to recognize patterns. As they progress, students should identify patterns as opportunities for abstraction, such as recognizing repeated patterns of code that could be more efficiently implemented as a loop. Eventually, students should extract common features from more complex phenomena or processes. For example, students should be able to identify common features in multiple segments of code and substitute a single segment that uses variables to account for the differences. In a procedure, the variables would take the form of parameters. When working with data, students should be able to identify important aspects and find patterns in related data sets such as crop output, fertilization methods, and climate conditions.

2. Evaluate existing technological functionalities and incorporate them into new designs.

At all levels, students should be able to use well defined abstractions that hide complexity. Just as a car hides operating details, such as the mechanics of the engine, a computer program's "move" command relies on hidden details that cause an object to change location on the screen. As they progress, students should incorporate predefined functions into their designs, understanding that they do not need to know the underlying implementation details of the abstractions that they use. Eventually, students should understand the advantages of, and be comfortable using, existing functionalities (abstractions) including technological resources created by other people, such as libraries and application programming interfaces (APIs). Students should be able to evaluate existing abstractions to determine which should be incorporated into designs and how they should be incorporated. For example, students could build powerful apps by incorporating existing services, such as online databases that return geolocation coordinates of street names or food nutrition information.

3. Create modules and develop points of interaction that can apply to multiple situations and reduce complexity.

After students have had some experience identifying patterns, decomposing problems, using abstractions, and taking advantage of existing resources, they should begin to develop their own abstractions. As they progress, students should take advantage of opportunities to develop generalizable modules. For example, students could write more efficient programs by designing procedures that are used multiple times in the program. These procedures can be generalized by defining parameters that create different outputs for a wide range of inputs. Later, students should be able to design systems of interacting modules, each with a well-defined role, that coordinate to accomplish a common goal. Within an object-oriented programming context, module design may include defining the interactions among objects. At this stage, these modules, which combine both data and procedures, can be designed and

documented for reuse in other programs. Additionally, students can design points of interaction, such as a simple user interface, either text or graphical, that reduces the complexity of a solution and hides lower level implementation details.

4. Model phenomena and processes and simulate systems to understand and evaluate potential outcomes.

Students at all grade levels should be able to represent patterns, processes, or phenomena. As they progress, students should understand that computers can model real-world phenomena, and they should use existing computer simulations to learn about real-world systems. For example, they may use a preprogrammed model to explore how parameters affect a system, such as how rapidly a disease can spread. Older students should model phenomena as systems, with rules governing the interactions within the system. Students should analyze and evaluate these models against real-world observations. For example, students might create a simple producer–consumer ecosystem model using a programming tool. Eventually, they could progress to creating more complex and realistic interactions between species, such as predation, competition, or symbiosis, and evaluate the model based on data gathered from nature.

Practice 5: Creating Computational Artifacts

The process of developing computational artifacts embraces both creative expression and the exploration of ideas to create prototypes and solve computational problems. Students create artifacts that are personally relevant or beneficial to their community and beyond. Computational artifacts can be created by combining and modifying existing artifacts or by developing new artifacts. Examples of computational artifacts include programs, simulations, visualizations, digital animations, robotic systems, and apps.

By the end of Grade 12, students should be able to:

1. Plan the development of a computational artifact using an iterative process that includes reflection on and modification of the plan, taking into account key features, time and resource constraints, and user expectations.

At any grade level, students should participate in project planning and the creation of brainstorming documents. As learning progresses, students should systematically plan the development of a program or artifact and intentionally apply computational techniques, such as decomposition and abstraction, along with knowledge about existing approaches to artifact design. Students should be capable of reflecting on and, if necessary, modifying the plan to accommodate end goals.

2. Create a computational artifact for practical intent, personal expression, or to address a societal issue.

Students at all grade levels should develop artifacts in response to a task or a computational problem. As they progress, student expressions should become more complex and of increasingly broader significance. Eventually, students should engage in independent, systematic use of design processes to create artifacts that solve problems with social significance by seeking input from broad audiences.

3. Modify an existing artifact to improve or customize it.

At all grade levels, students should be able to examine existing artifacts to understand what they do. As they progress, students should attempt to use existing solutions to accomplish a desired goal. For example, students could attach a programmable light sensor to a physical artifact they have created to

make it respond to light. Later, they should modify or remix parts of existing programs to develop something new or to add more advanced features and complexity. For example, students could modify prewritten code from a single-player game to create a two-player game with slightly different rules.

Practice 6: Testing and Refining Computational Artifacts

Testing and refinement are the deliberate and iterative process of improving a computational artifact. This process includes debugging (identifying and fixing errors) and comparing actual outcomes to intended outcomes. Students also respond to the changing needs and expectations of end users and improve the performance, reliability, usability, and accessibility of artifacts.

By the end of Grade 12, students should be able to:

1. Systematically test computational artifacts by considering all scenarios and using test cases.

At any grade level, students should be able to compare results to intended outcomes. As students' progress, they should test computational artifacts by considering potential errors, such as what will happen if a user enters invalid input. Eventually, testing should become a deliberate process that is more iterative, systematic, and proactive. Older students should be able to anticipate errors and use that knowledge to drive development. For example, students can test their program with inputs associated with all potential scenarios.

2. Identify and fix errors using a systematic process.

At any grade level, students should be able to identify and fix errors in programs (debugging) and use strategies to solve problems with computing systems (troubleshooting). As students' progress, they should become more adept at debugging programs and begin to consider logic errors: cases in which a program works, but not as desired. In this way, students will examine and correct their own thinking. For example, they might step through their program, line by line, to identify a loop that does not terminate as expected. Eventually, older students should progress to using more complex strategies for identifying and fixing errors, such as printing the value of a counter variable while a loop is running to determine how many times the loop runs.

3. Evaluate and refine a computational artifact multiple times to enhance its performance, reliability, usability, and accessibility.

After students have gained experience testing, debugging, and revising, they should begin to evaluate and refine their computational artifacts. As students' progress, the process of evaluation and refinement should focus on improving performance and reliability. For example, students could observe a robot in a variety of lighting conditions to determine that a light sensor should be less sensitive. Later, evaluation and refinement should become an iterative process that also encompasses making artifacts more usable and accessible. For example, students can incorporate feedback from a variety of end users to help guide the size and placement of menus and buttons in a user interface.

Practice 7: Communicating About Computing

Communication involves personal expression and exchanging ideas with others. In computer science, students communicate with diverse audiences about the use and effects of computation and the appropriateness of computational choices. Students write clear comments, document their work, and communicate their ideas through multiple forms of media. Clear communication includes using precise language and

carefully considering possible audiences.

By the end of Grade 12, students should be able to:

1. Select, organize, and interpret large data sets from multiple sources to support a claim.

At any grade level, students should be able to refer to data when communicating an idea. As students' progress, they should work with larger data sets and organize the data in those larger sets to make interpreting and communicating it to others easier, such as through the creation of basic data representations. Eventually, students should be able to select relevant data from large or complex data sets in support of a claim or to communicate the information in a more sophisticated manner.

2. Describe, justify, and document computational processes and solutions using appropriate terminology consistent with the intended audience and purpose.

At any grade level, students should be able to talk about choices they make while designing a computational artifact. Students should provide documentation for end users that explains their artifacts and how they function, and they should both give and receive feedback. For example, students could provide a project overview and ask for input from users. As students' progress, they should incorporate clear comments in their product and document their process using text, graphics, presentations, and demonstrations.

3. Articulate ideas responsibly by observing intellectual property rights and giving appropriate attribution.

All students should be able to explain the concepts of ownership and sharing. They should identify instances of remixing, when ideas are borrowed and iterated upon, and give proper attribution. They should also recognize the contributions of collaborators. Eventually, students should consider common licenses that place limitations or restrictions on the use of computational artifacts. For example, a downloaded image may have restrictions that prohibit modification of an image or using it for commercial purposes.