

Ten Ideas in Programming

... a minimal introduction to programming

By Morten Misfeldt – Morten Aagaard Schultz – Karl Emil Kjær Bistrup

Adapted by Daniel Spikol for Kickstart 2025

Introduction

Ten Ideas in Programming is designed to communicate a minimal set of concepts in programming and computer science that I believe everyone should encounter as part of their education. These ideas range from concrete and low-level to more abstract. They don't attempt to cover computer science — but they represent a minimum set of ideas that serve two main purposes:



First, they offer a practical starting point for writing code and working with programs. Second, they provide 10 concepts that help read and understand simple programs. The goal is for the list to be understandable in a few minutes, and for the whole material to be readable within a few hours, including examples. I hope the list will also serve as a reference point to help you refine your conceptual understanding as you improve your programming skills.

I focus on just ten ideas to make the material approachable and contained. I believe that a simple “idea-oriented” introduction to programming will be beneficial when learning about programming in the age of AI, either as a stepping stone for further studies in computer science or simply to become more literate about the digital world we all live in. The order in which the ideas are presented is not entirely random. But the intention is that all ten ideas are introduced at once and then revisited as you gain more experience.

After introducing the ten ideas, I have augmented the material with five key practices of a programmer, highlighting the essential aspects we consider when creating computer programs. Where the ten ideas are concepts to be understood, the practices are focused on what we do when we program and why we do it. Before I go into exemplifying the ten ideas with coding examples, I have added a short section with some suggestions on how to work with LLMs when learning to program. Towards the end of the text, I have devoted a section to more details about the intentions of the text and about the specific conception of learning that it builds upon.

How we think about Programming

The first idea is **data**. Everything digital begins with data, which can take various forms, including numbers, text, images, and sensor readings. Data is usually stored in some sort of structure or sequence (say, a list) that supports computational manipulation and /or human interpretation.

- Numbers (42, 3.14)
- Text ("Hello World!")
- Images 
- Sensor readings 

The second idea is **function**. A function in programming is almost like the function machine from our primary school math class. A function is a “machine” that performs an action when asked. It can take inputs and provide outputs, similar to its mathematical sibling, but first and foremost, it performs actions when called.

The third idea is variables, which are used to store and access data. Variables are important for keeping track of values and enabling dynamic behavior in programs. A variable can be a tricky concept because, despite being used across mathematics, statistics, and computer science, its meaning varies in nuance. Most importantly, you almost always assign a value to a variable in programming, in contrast to, for example, solving equations, where you search for a value of the variable that solves the problem.

You use variables to store and refer to data, and functions to process that data in meaningful ways

The fourth idea is **(algorithms and) sequential processes**—computer programs perform their actions step by step, and these sequences of steps then transform data into something useful. Each step in the sequence can perform operations, such as adding or subtracting numbers, checking conditions, or calling functions. Typically, you will break down your program idea into a set of processes that work on data and variables.

The fifth idea is control-structures and **conditionals**. These are structures that guide decision-making and distinguish between different cases in computing, such that certain processes only run under specific conditions.

The sixth idea is **loops** or **iteration**, the simple yet powerful idea of repetition, which allows programs to perform tasks repeatedly a specified number of times or until certain conditions are met.

The seventh idea is **models** — representations of real-world or imagined systems, processes, or ideas, which we use in simulations, predictions, and reasoning. Models help us make sense of complexity and allow us to test, explore, and communicate ideas through computation.

The eighth idea is abstraction and decomposition. This involves identifying meaningful parts of a problem, breaking them down into manageable pieces, and creating the right levels of abstraction to reason about complex systems.

Closely related is the ninth idea: **functional thinking**. This means organizing your specifications and/or code in functions, inputs, and outputs. This is when and under what conditions some element of the program runs, and what the inputs and outputs of that process are. Constructing (or reconstructing) programs in reusable, well-defined functions is a great way to organize your thinking and abstractions.

Last, we turn to **data structures**—ways of organizing data to make processing efficient, meaningful, and scalable, from simple sequencing in lists and arrays to graphs.

Five Key Practices - How we do programming

We **specify**. Writing a computer program is first and foremost stating what you want the computer to do in an understandable way*. Therefore, one of the most important practices when learning to program is the practice of *specification*. This means explaining and specifying what we want a program to do, and how we want it to do it. Sometimes, we focus mostly on *what* the program should do—what output or behavior we expect. Other times, we pay more attention to *how* to build that behavior—what steps the program should follow. In all cases, explaining *what* and *how* is at the heart of programming.

01 - Pseudocode

Pseudocode: One common method for specifying programs is writing *pseudocode*—a description of your program in plain English (or whatever language you prefer).

```
FUNCTION make_fish with parameter fishX:
    // Step 1: Draw the fish's body
    Draw an oval at position (fishX, 200)
        Make it 120 pixels wide
        Make it 75 pixels tall

    // Step 2: Draw the fish's tail
    Draw a triangle with three corner points:
        Point 1: (fishX - 60, 200)  -- closest to body
        Point 2: (fishX - 90, 170)  -- top of tail
        Point 3: (fishX - 90, 230)  -- bottom of tail

    // Step 3: Draw the fish's eye
    Set eyeSize to 15
    Draw a circle at position (fishX + 30, 190)
        Make it eyeSize pixels wide
        Make it eyeSize pixels tall
END FUNCTION
```

02 Testing and Debugging

We **test and debug**. When we have an idea for part of a program, we usually try it out on its own before adding it to the larger project. This helps us check if it works as expected.

Writing code in a precise syntax is not easy, and dealing with complex logical structures is sometimes even harder. So, we make mistakes. Everyone does. Often. Making mistakes and fixing them is a normal and important part of programming. When trying to figure out what's wrong, we localize the problem by isolating the parts we think might be causing the issue. By doing this, we can test, correct our mistakes, and improve our program step by step.

03 Ways of Thinking

We **organize our thinking** about programs in layers, **distinguishing 1) data, 2) computation, and 3) interaction**. In a program that asks the user to type their name on the screen and then responds by generating a friendly message like “Hello, Morten!” The interaction layer is just this – you write your name and receive a personalized greeting. The data layer consists of the text string “Morten Misfeldt” that I provide in the input field, and the computational layer is somehow able to fetch out my first name from this text string and provide it as input for the greeting that the program responds with.

These three layers—data, computation, and interaction—are distinct but closely connected. Keep these three layers in mind, and it will help you organize your thinking about the programs you are building.

04 Documenting, Modularization, and Reuse

We **document**, **modularize**, and **reuse** our code. When you're just starting out, you might write simple programs from scratch or edit examples your teacher gives you. But quickly, you'll discover that programming is a cumulative process—programs grow over time, when you work with them. In professional software development, programs often contain thousands of lines of code. Keeping track of changes and

understanding what each part does is essential. But even in your first real project, you'll see the benefits of organizing and documenting your code.

If you've already written something that works for a particular task, then use it again for similar problems. When you do that, you will end up thinking about not only the specific problem you are engaged in (say, creating a red box with the text "No" on the screen) but rather thinking about the type of problem that you work on (e.g., creating a colored textbox on the screen). This way of thinking saves time, reduces errors, and helps you think more clearly about how your whole program fits together.

Commenting Code Writing short explanations in your code helps you (and others) remember what you were trying to do. For example: What is this part supposed to do? Why are you doing it this way? This is called comments and is typically distinguished from the code by some special character like `//` this is a comment `//` in the file.

```
function make_fish (fishX) {  
  // Draw the basic fish shape  
  // Draw fish body (fishX , 200) with a width of 120 and height of 75  
  ellipse(fishX , 200, 120, 75);  
  // Draw the fish tail using a triangle  
  // The triangle points are at (fishX - 60, 200) , (fishX - 90, 170) ,  
  // and (fishX - 90, 230)  
  triangle (fishX - 60, 200, fishX - 90, 170, fishX - 90, 230);  
  // Draw the fish eye  
  // The eye is a small ellipse located at (fishX + 30, 190)  
  // with a size of 15 x15  
  eyeSize = 15;  
  ellipse(fishX + 30, 190, eyeSize , eyeSize );  
}
```

05 Collaboration with people

Finally, we **collaborate with our users**. Programs are developed for people — to help them do the activities they want to do in a more enriching, engaging, or more efficient way. An important part of programming is to think about use. If you're trying to design a solution, it's a good idea to understand the problem, and that often means understanding the practice that the program will support. This also means paying attention to how what you have developed is used by other people. There's always a slight difference between what a designer or programmer has in mind and what users do. That's why it's important to consider both the imagined use during development and the actual use of the program after release or prototype.

There are many ways to do this. In modern software development, data about user behavior is often collected directly from the software. But you can also simply sit down with the users, talk to them, understand their needs, or, when your first version is ready, observe how users tweak it to fit their needs.