ACTUALIZE CODING BOOTCAMP PRESENTS

HOW TO TEACH PROGRAMMING

THE ACTUALIZE METHOD

JAY WENGROW
PETER JANG

ACTUALIZE.CO

Introduction

Teaching is hard. A lot of people believe that someone who is an expert at a subject should be highly qualified to teach that subject. Yet, often the reverse is true - the more you know about a subject, the less you remember what it was like to learn the subject. For example, I am an expert at walking, having done it all my life (I almost never bump into things by accident). However, that doesn't qualify me to become a physical therapist for people who have been injured and need to relearn how to walk from scratch. My ability to walk is so intuitive that I have no idea where to begin if I had to teach someone. **This is called the curse of knowledge**.

While the curse of knowledge makes it difficult for educators to teach effectively, the curse of knowledge also makes it difficult for *learners* to learn effectively. Having a blank slate is often the best starting place for a beginner learning something new. Without a blank slate, it can be an uphill battle to fight against bad habits and misconceptions.

Programming is a difficult subject to teach for this reason. Anyone who is familiar with reading, writing, and arithmetic is capable of getting started with learning to code. However, programming requires a totally different way of thinking compared to reading, writing, and arithmetic. Most students aren't coming in with that blank slate **- their prior knowledge can actually make it harder for them to learn to code**.

So it takes a lot of careful thought to teach programming well. You'll have to train yourself to see every concept from a beginner's perspective to overcome both your and their curse of knowledge. You'll have to get students to unlearn what it means to read and think like a human, and instead learn what it means to read and think like a computer.

This book is a collection of approaches that we have found effective with our collective decades of teaching experience. It can help you design curriculum and classroom instructional experiences, no matter how large or small. We hope you find it useful!

HOW TO TEACH PROGRAMMING

PART ONE

CURRICULUM DESIGN

WHAT TO TEACH AND WHAT ORDER TO TEACH IT

Beginners are often paralyzed by choice.

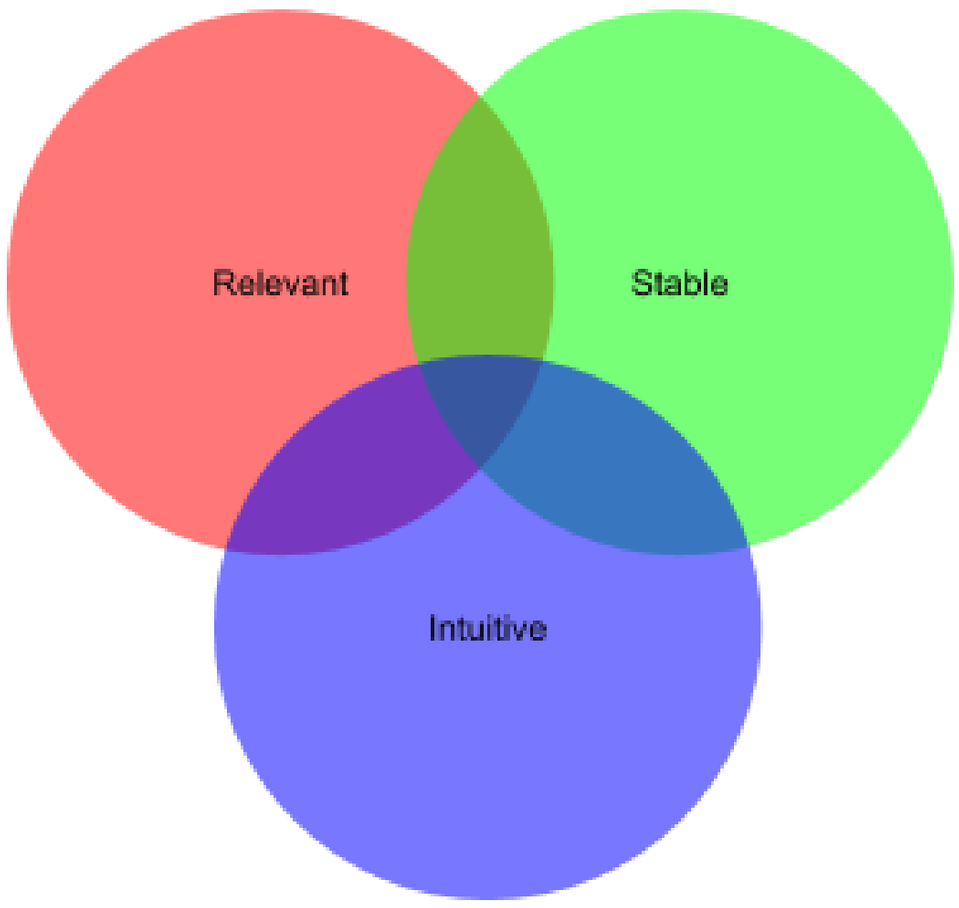
Choose teachable technologies

Choosing the right technology to teach a beginner can be a daunting challenge. Programming is a fast moving world; no one wants to learn a language or skill today that becomes obsolete tomorrow.

One trap a lot of curriculum designers fall into is to teach the newest tech. The problem is that the newest tech is by nature the least stable, which can lead to a terrible learning experience. Instead of focusing on understanding the fundamental concepts of programming, students will spend their time fixing installation issues and breaking changes with each new version of the technology.

Another trap is to teach the most performant tech. The problem is that performance often comes at the expense of readability. There's a saying in programming: Make it work, make it right, make it fast - in that order. Beginners shouldn't focus on all three - their energy needs to focused on making it work, understanding the fundamentals of how to communicate to computers through code.

Instead of focusing on popularity or performance, the best curriculum designers choose what technologies to teach based on balancing three criteria: **relevance**, **stability**, and **intuitiveness**.



Relevance is based on current industry usage - is the technology being used by many companies today? Many beginners focus only on this criteria, and choose to learn the most popular of the relevant technologies. Popularity is important, as it will help students get jobs in the industry. However, the most popular technologies are often not the most teachable technologies, based on the other two criteria.

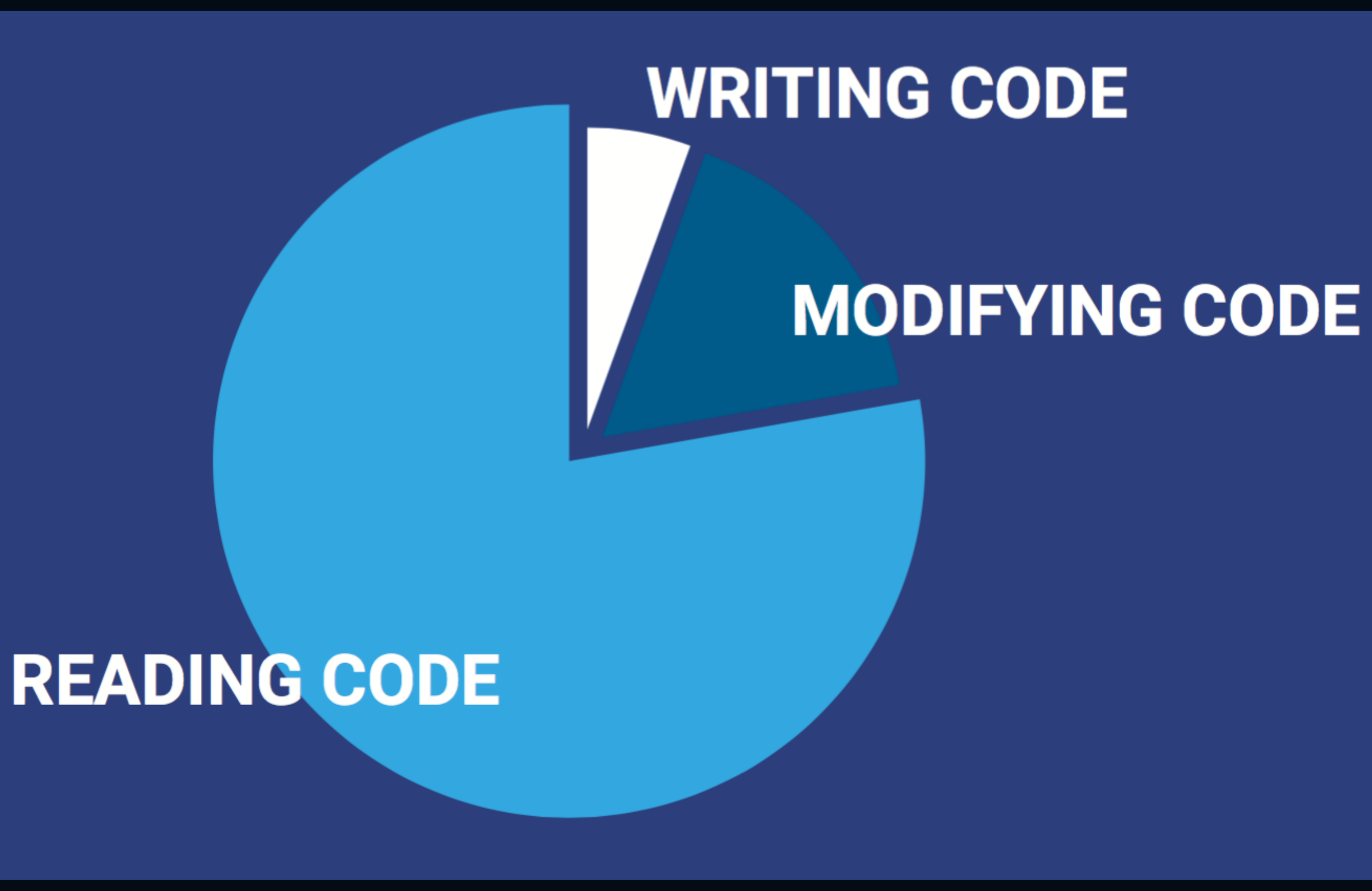
Stability is based on age and maturity - is the technology at a version that's relatively stable or are there constant breaking changes with each release? If you teach a technology that's unstable, then students will spend time chasing bugs and issues that aren't relevant to the concepts that they're trying to learn. There are many stable technologies that are widely used in the industry to choose from. They may get less attention in the daily news precisely because they are more stable, but they are far better candidates in terms of teachability.

CURRICULUM DESIGN

Intuitiveness is based on the code itself - does the code make concepts easy to express for a beginner? A beginner faces a tremendous amount of information that they are learning at once. Similar to using unstable technologies, if a language has an unforgiving syntax, then a beginner can spend hours fixing missing semicolons and other minor issues, which prevents them from focusing on the actual programming concepts they need to focus on.

It's important to remember that students will learn many different technologies over the course of their careers. The decision being made here is not what technologies are worth learning, but rather what technologies should be taught first. By choosing the appropriate first technology to teach, your students will be able to gain a strong sense of the fundamentals and will be able to teach themselves subsequent technologies as the industry continually evolves.

CURRICULUM DESIGN



Programmers spend more time reading code than writing code.

Teach reading code versus writing code

Programmers spend the majority of time reading code, not writing code. Yet, most programming curriculum for beginners focuses exclusively on teaching how to write code - creating variables, writing loops, making functions, etc.

A beginner often falls into the trap of writing code they simply do not understand. It’s bad if they write code they don't understand that doesn’t work; it’s worse when they write code they don't understand that *does* work. Without the ability to read code, the ability to write code becomes almost meaningless.

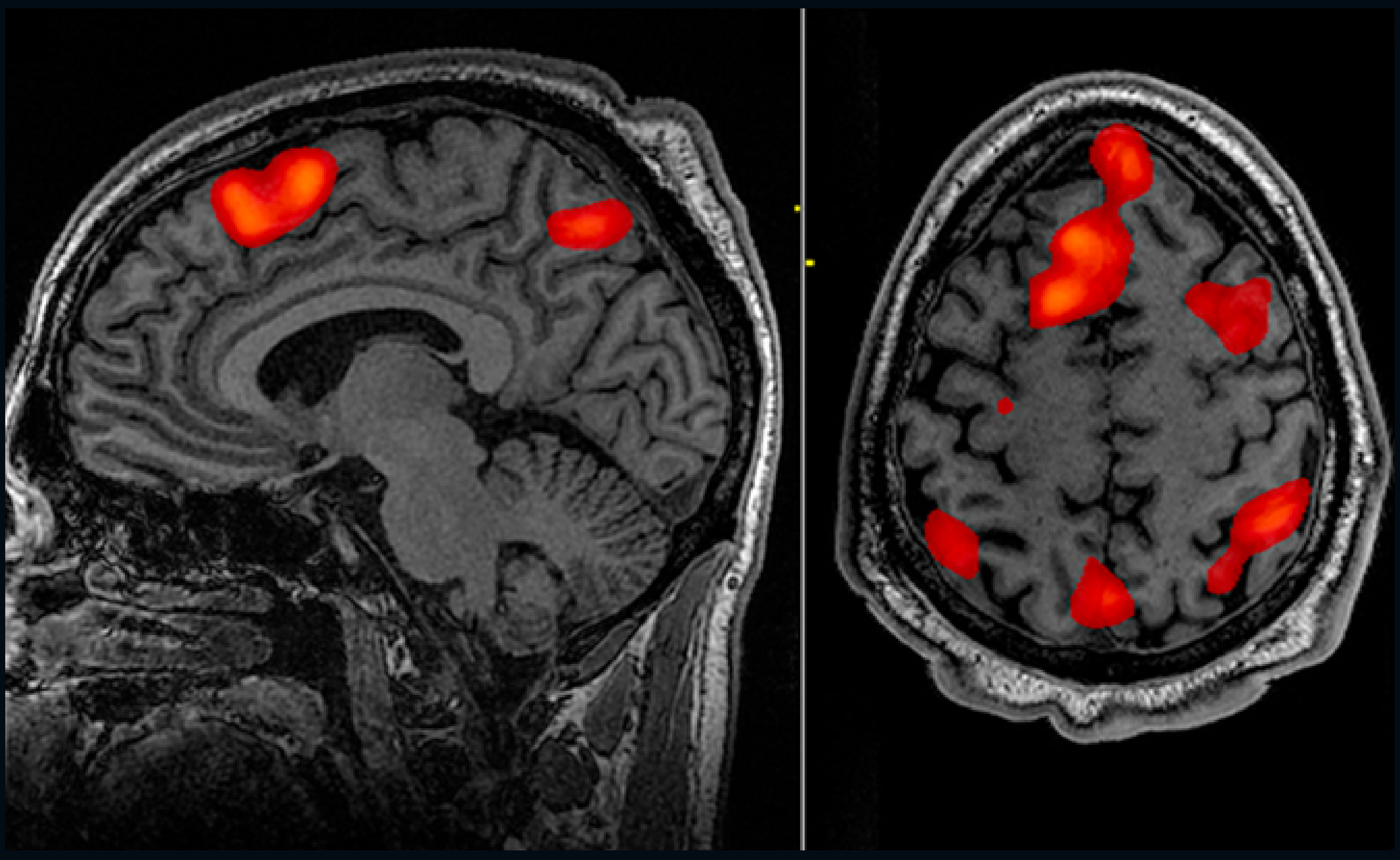
While it is important to teach how to write code, it is at least as important to teach how to read code. But how can you do it? Take a look at the following exercise:

```
# Write comments next to each line in the order that they are run.
# The first several comments are done for you, your job is to complete the rest.

booyeah = 0                    # 1
2.times do                     # 2
  booyeah = booyeah + 10       # 3
  2.times do                   # 4
    booyeah = booyeah + 4      # 5
    3.times do                 # 6 9
      booyeah = booyeah + 1    # 7 10
    end                        # 8
  end                          #
end                            #
p booyeah                      #
```

This problem requires students to read the code and demonstrate their knowledge of how it works through writing comments. If a student doesn't understand the nature of loops, they won't be able to answer this question correctly. An instructor can quickly identify any misconceptions and guide the student towards a correct understanding of the concept.

This style of exercise is one way to emphasize reading code as opposed to writing code. This is an absolutely critical component of programming education that most people miss out on!



A learner can only focus on one thing at a time effectively.

Teach one concept at a time

Good curriculum design is about taking a complex topic and breaking it down into isolated smaller concepts. When a student learns something for the first time, they must spend nearly all of their mental effort (known as cognitive load) focusing on the new concept. Teaching several concepts at once can easily overload a student's ability to learn.

It can be argued that any time a student doesn't understand something you taught, it is because you introduced too many concepts at the same time. The best curriculum designers can take any topic and break it down into isolated parts (no easy feat).

Let's examine what this looks like in the following simple coding example:

```
# BAD INTRO: introduces multiple new concepts
# (RESTful conventions, instance variables, databases, rendering a view)
def index
  @recipe = Recipe.all
  render :index
end

# GOOD INTRO: introduces one new concept
# (rendering a view)
def example_method
  render 'example.html.erb'
end
```

The “bad intro” code isn’t bad code at all for experienced web developers! However, this shouldn't be the first example of Rails controller code you teach to someone new to web development. In those four lines, there are a multitude of concepts being introduced, including RESTful naming conventions, instance variables, database communication, and rendering a separate view. Note that this is very difficult for experienced programmers to notice, since they are so familiar with each concept (the “curse of knowledge” mentioned at the beginning of this book).

The “good intro” code begins with just rendering a view. Once a student grasps this concept, you can layer the other concepts one at a time when appropriate. This provides a better foundation for students to deeply understand each concept.

In order to teach one concept at a time, you do have to understand a student’s prior knowledge. An intermediate developer may be familiar with a set of concepts, so the one concept you would teach would be different compared to a complete beginner, or an advanced developer. This makes designing good curriculum a challenge; the best curriculum developers are able to balance all these factors with the singular focus of identifying the appropriate isolated concept to teach at each moment in time.

A demonstration is worth its weight in gold.

Provide high quality examples with each concept

It’s important to provide examples to accompany each concept. Examples are one of the backbones of student learning - a well chosen example helps a student both solidify a concept as well as clarify any misunderstandings. Conversely, a poorly chosen example can actually increase misunderstandings.

One of the keys of making high quality examples is to look out for coincidences. In math, 2 x 2 = 4 doesn't make for an ideal example to first introduce multiplication, because, coincidentally, it is the same result as addition (2 + 2 = 4). Instructors often don’t notice these coincidences (due to the curse of knowledge), so it’s critical to take extra care here.

One can argue that providing an example takes away from the student discovering and learning on their own. We don't agree with this perspective - if your goal is for students to discover something on their own, then the concept you’re teaching is not a programming concept - it’s the broader concept of problem solving. Problem solving can be treated as an isolated concept, and an instructor can and should provide examples and approaches to problem solving if they truly want students to improve in that area. In the end, it’s your job to identify the isolated concept to teach and identify high quality examples to go with it.

Scaffolding helps learners reach new heights.

Provide scaffolded exercises for every new concept

Exercises are another key part of student learning. A student must practice the concept on their own to identify their own understandings and misunderstandings of a concept, regardless of how well the concept is explained.

A good exercise has an isolated focus on the concept being taught. Each exercise should gradually increase in difficulty. Here is a general pattern for teaching a new concept with examples and exercises:

- Explain a new concept
- Demonstrate an example of the new concept
- Assign exercises for the concept
  - Level 1: Identical to the demo with a slightly different context
  - Level 2: Requires synthesis with previously taught concepts
  - Level 3: Requires a yet-to-be-taught skill, generating curiosity for the next topic

It is important to always follow steps 1 - 3 when teaching a new concept. Which exercises to use depends on the concept being taught. Nevertheless, it is critical that each new concept is accompanied by at least one exercise (with repetition throughout as needed).

Good instruction requires a variety of techniques.

Provide different modes for different goals

Different modes of exercises work best for different educational goals. Here are some different goals with corresponding exercises:

- New concepts - individual scaffolded exercises
- Fundamental skills - individual deliberate practice exercises
- Review various concepts - pair programming short exercises
- Synthesize related concepts - pair programming or group real-world exercises

There are some companies that use pair programming for 100% of their work. While this may make sense for that company in terms of professional work, it doesn't make sense to pair 100% when learning how to code. Pairing is an enormously effective strategy for certain types of learning, but is far less effective for others. A good mix of individual, pair, and group exercises is a key part of a well balanced learning environment.

Patterns only emerge after repetition.

Repeat, repeat, repeat

A concept that is not practiced over time will be forgotten. Good curriculum design is about **identifying the right level of repetition** for each concept. Students often require multiple exposures to the same concept in order to eliminate misunderstandings. An often overlooked fact is that human beings spend about 30% of their time daydreaming. This means that when you explain a new concept, there's a great chance that several students didn't even hear the explanation. The only way to overcome this issue is to provide multiple exposures to key concepts.

Sometimes proper repetition involves spacing out concepts across several days instead of cramming everything into a single sitting.

Choose the right thing to focus on.

Use the right level of explicitness

When introducing a concept, the goal is to make the concept as clear as possible within the code. If you rely on syntactical magic, the concept is hidden from view. But code can be too explicit as well. Let's look at the following example code:

```
# BAD INTRO: uses implicit render defaults, makes the method's responsibility unclear
def index
end

# GOOD INTRO: uses explicit render, makes the method's responsibility clear
def index
  render "index.html.erb"
end

# BAD INTRO: uses overly explicit render, adds details unnecessary to teaching the concept
def index
  render file: "/u/apps/warehouse_app/current/app/views/products/index.html.erb"
end
```

All three versions of the code work the same way from the perspective of the computer. An advanced programmer may prefer the first version, as it is terse and uses a shortcut to automatically render the appropriate file. However, if the goal is to teach students about writing methods to render a file, it's too implicit. In this case the second version is more explicit and is preferable. The third version, however, is too explicit - it introduces more detail, but the detail is not related to the concept being taught.

Newer isn't always better.

Don't teach new shortcuts if you can reuse concepts

As students learn concepts for the first time, their brains identify patterns which become generalizations they apply to future scenarios. **Forming generalizations is the foundation of good learning**. Poor learners focus on memorizing everything. Without generalizations, each new problem becomes a new concept to memorize. Strong learners focus on forming generalizations, as multiple problems can be solved by applying a single generalization.

Shortcuts add cognitive overhead. Instead of making it easier to perform a concept, it adds a new concept to learn. Let’s look at the following example:

```
# BAD INTRO: introduces new shortcut method (.update)
recipe.update(
  name: params[:name],
  description: params[:description]
)

# GOOD INTRO: reuses already taught OOP concepts (class attribute writers, model save
method)
recipe.name = params[:name]
recipe.description = params[:description]
recipe.save
```

The first version is the way most experienced developers would write the code. It's so familiar to them that they may not even recognize that they're using a shortcut with the .update method (which both changes attributes in the object and then saves the changes to the database). The second version makes these two steps more explicit by separating them into two steps. It's arguably more verbose, but by reusing individual concepts that have been already taught, it helps students form the correct generalizations and practice the concepts they know.

This isn't to say that shortcuts shouldn't be taught. Using shortcuts effectively is an important part of being a successful programmer! Shortcuts should be taught, but only after students get a chance to thoroughly practice the long way first. A shortcut should be considered its own isolated concept within a curriculum - it should be introduced separately from the original concept itself at the appropriate time.

History provides the foundation of learning.

Teach the historical context behind the concept

Teaching without historical context makes it more difficult for students to understand any given material. It's particularly important to note that programming is a relatively young industry, where technologies and best practices are constantly shifting and evolving. Without the appropriate context, the sequence of concepts can seem arbitrary and nonsensical.

There are countless examples of concepts that seem arbitrary without historical context. Why do we use HTML to make complex web applications? Why are some languages compiled while others are interpreted? Why do developers use the backend language Node.js for working with frontend JavaScript?

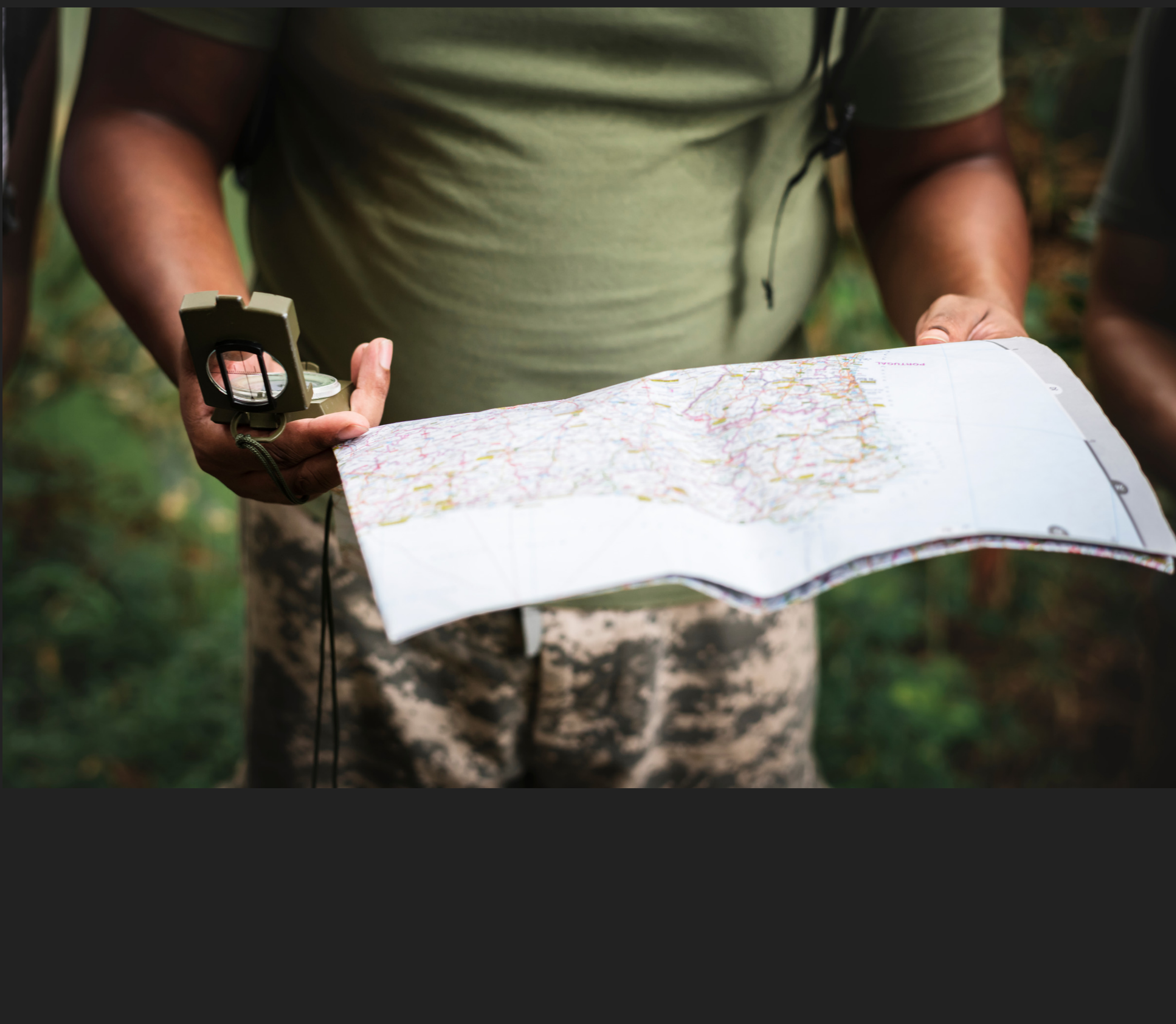
Without understanding the context behind some of these concepts, it can significantly impede a student's ability to learn. At some points in a curriculum, teaching programming is equal parts history as it is the concept itself. Having an appreciation of that history will give students a better sense of the concept and a more well-rounded outlook when examining new technologies, which alone would be worth the price of admission.

HOW TO TEACH PROGRAMMING

PART TWO

CLASSROOM INSTRUCTION

HOW TO TEACH AND EXPLAIN CONCEPTS IN THE MOMENT



Sometimes learners need to find their own way.

Don’t teach best practices too early

As a teacher, it's tempting to teach best practices as a way to help beginners avoid writing bad code. But learning these high-level concepts doesn't happen by avoiding mistakes - it happens by experiencing mistakes and learning from them. And what's worse, teaching best practices takes the focus away from what a beginner should be focused on at their current stage in learning. If someone is still struggling with syntax, teaching them about high-level design patterns is likely to do more harm than good.

If you teach a novice about the best practice of DRY (Don't Repeat Yourself), one of two things will happen: Either they won't believe in its importance and generally ignore the rule. Or, they will believe in it wholeheartedly without fully understanding why it's valuable, and apply it to every situation imaginable. This is where you'll see a Chair class inheriting from a Cat class because both have four legs. The bottom line: It's not hard to teach a rule, but it's incredibly hard to teach when it is appropriate to apply it.

So what's the ideal solution? Instead of teaching best practices, try creating experiences for a student such that they can see the problem and discover a strategy on their own. The challenge is to create valid experiences and present it at an appropriate time in their learning. Unfortunately, this takes a lot longer than teaching best practices directly. But without the experiences to go with it, it's not teaching - it's pontificating. Good teaching takes patience.

Only after a student experiences the same problem in different contexts will they be able to discover both a strategy to address it and the contexts in which it is applicable. At that point, the only thing you need to do as a teacher is to introduce a name for that strategy. And now it's not some mystical rule, it's simply a name - a concise way to describe both the strategy and its appropriate application. Best practices aren't learned so much as they're earned, and an effective teacher knows when to guide rather than teach to make it happen.

Teaching requires a completely fresh perspective.

HOW TO TEACH PROGRAMMING

Demo teachable code instead of readable code

Software developers strive to write code that is as readable as possible. But what does it mean for code to be readable? One definition that sums it up well is:

Readable code is code that clearly communicates its intention to the reader.

There's an important point to emphasize in this definition: Code readability depends on who's reading it, which means it's subjective. It's not an objective measure or score you can use to evaluate code. What's readable to an advanced developer is very different than what's readable to a beginning developer.

When you are teaching a concept to a beginner for the first time, it's important to keep the subjective nature of code readability in mind. Abstractions, design patterns, shortcuts - these are all advanced tools used to make code maintainable, but to a person who is not familiar with them, the code becomes nearly impossible to understand.

To avoid confusing beginners when teaching new concepts, you should instead focus on using teachable code. We can define teachable code as follows:

Teachable code is code that makes the specific concept being taught as explicit as possible.

CLASSROOM INSTRUCTION

In other words, it's objectively making the specific concept as readable as possible for an absolute beginner. The next sections will showcase specific examples that highlight the difference between teachable code and readable code.

It's important to note that teachable code is utilized when presenting a concept to students for the first time. Once you feel that the concept has been grasped, it makes perfect sense to show students a more "normal" or readable way of writing the code. When you demo code this way, you are showing the progression from basic code to advanced code. You can evolve your coding style to match student understanding - instead of showing advanced code to beginners and overloading the number of concepts they need to learn at once.

Visual differences will help avoid confusion.

Demo different variable names for different things

Beginners have trouble reading code when every variable has the same name. Using different names helps beginners understand the role of each variable in the code.

```ruby
# BAD INTRO: uses same variable names for inputs and attributes
def initialize(name, age)
  @name = name
  @age = age
end

# GOOD INTRO: uses different variable names for inputs and attributes
def initialize(input_name, input_age)
  @name = input_name
  @age = input_age
end
```

For a student who's just learning about classes and methods, the first example is surprisingly confusing. The curse of knowledge makes it hard to appreciate how much is actually happening here. Nearly every variable is named the same thing, and it's hard to distinguish which variable does what.

The second example makes it clear which variable is the input (by adding an overly explicit prefix) and which is the instance variable. This code would generally be considered to be less readable due to its overly explicit nature, but it helps students distinguish between the various pieces being presented here.

Avoid abstractions at first.

Demo overly explicit variable names

Variable names can help remove some of the subtle abstractions inherent in code. For example, http parameters in the Ruby on Rails framework are parsed from multiple sources (query parameters, url segment parameters, form input parameters) into a single hash called params. When using the params hash, it is unclear where the parameters are coming from. This makes it hard for beginners to understand problems in their code.

```
# BAD INTRO: uses params keys that don't make it explicit where the values come from
@product = Product.find_by(id: params[:id])
@product.update(name: params[:name])

# GOOD INTRO: uses params keys that make it explicit where the values come from
@product = Product.find_by(id: params[:url_segment_id])
@product.update(name: params[:form_input_name])
```

Note that this takes some dedication - to teach code using these explicit variable names, you would also need to use the same explicit names in the routes file and html form to match. As cumbersome as this sounds, this approach eliminates the abstraction in the params hash and helps students make the appropriate connections. Once they grasp the concept, feel free to demo code using more typical conventions!

Learners need to see each step.

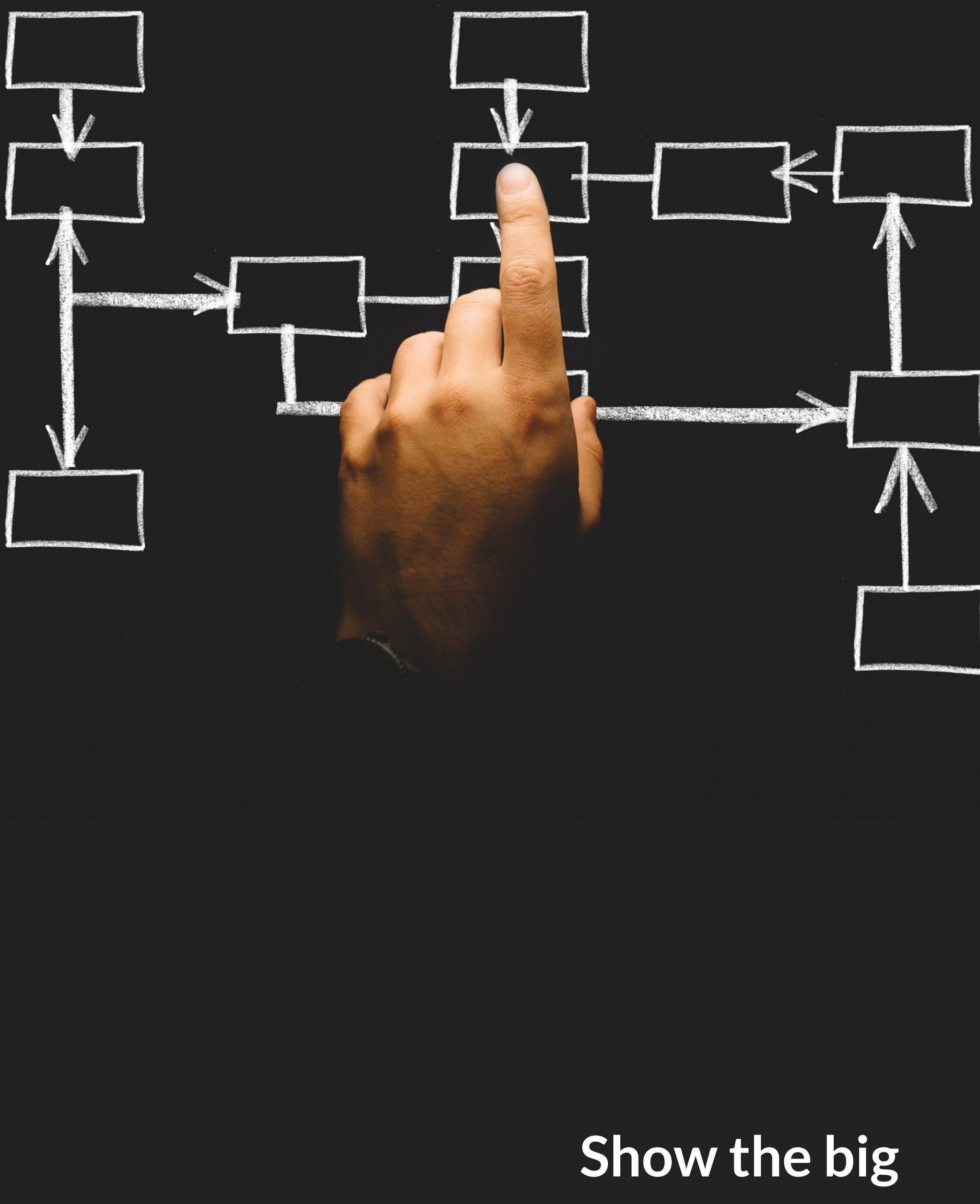
Demo one thing per line of code

It’s very easy to underestimate the number of things happening in a single line of code when you’re not a beginner.

```
# BAD INTRO: shows multiple things in single line of code
@products = Category.find_by(name: params[:name]).products

# GOOD INTRO: shows each thing on a separate line
selected_name = params[:url_query_name]
selected_category = Category.find_by(name: selected_name)
@products = selected_category.products
```

Not only does this help with the teachability of the code, but it also helps students debug their code when things go wrong. It’s easier to see which part of the code has the problem when it’s not all on the same line.



Show the big picture first.

Demo a single long method before using shorter methods

It's easier to understand a method at first when everything is written in one long method, despite the fact that code with multiple short methods is easier for an advanced developer to read and maintain. Breaking it up into smaller sub-methods increases cognitive load on students. This is because their eyes have to jump back and forth from method to method.

```ruby
# BAD INTRO: uses multiple short methods to perform some task
# (easier to maintain but harder to read)
def print_owing
  outstanding = calculate_outstanding
  print_details(outstanding)
end

def calculate_outstanding
  outstanding = 0.0
  @orders.each do |order|
    outstanding += order.amount
  end
  outstanding
end

def print_details(amount)
  puts "name: #{@name}"
  puts "amount: #{amount}"
end

# GOOD INTRO: uses single long method to perform some task
# (harder to maintain but easier to read)
def print_owing
  outstanding = 0.0
  @orders.each do |order|
    outstanding += order.amount
  end
  puts "name: #{@name}"
  puts "amount: #{outstanding}"
end
```

It’s easier for a beginner to understand the second code with a single method, despite the fact that the first code with multiple methods is easier for an advanced developer to read and maintain. Again, once the structure of the method is understood, it is acceptable to break it up into smaller methods. But don’t jump ahead until you are sure students understand the overall structure of the method.

Test early, test often.

Test to see if your code is working in small chunks

Instead of showing the final product of some code, show the evolution of the code in small testable chunks. This will encourage students to write their code in a similar way.

```ruby
# OKAY INTRO: shows two concepts at a time
# (getting values from params and saving something to the database)
Recipe.create(title: params[:title], ingredients: params[:ingredients])

# BETTER INTRO: shows one concept at a time
# (saving something to the database)
Recipe.create(title: "Ice cubes", ingredients: "Water")
```

In the above example, the first code is demoing two separate concepts - getting values from params **and** saving something to the database. The second example is demoing the single concept of saving something to the database. After you prove this works, then replace the hard-coded values with the params. This helps students see the evolution of the code in testable pieces.

Don't always
demo things the
right way.

Demo code the wrong way and ask what went wrong

Often, teachers demo code without making any mistakes to show how things work. Then students try doing the same, but spend most of their time writing broken code and trying to fix it. Instead of demoing perfectly working code all the time, it is beneficial to demo code that uses common student misconceptions, watch it break, then ask the students how to fix it.

```
# OKAY INTRO: shows code correctly
def update
  @tag = Tag.find_by(id: params[:id])
  @tag.update(name: params[:name])
  redirect_to "/tags/#{@tag.id}"
end

# BETTER INTRO: shows code incorrectly (watch it crash and ask students to help debug)
def update
  @tag = Tag.find_by(id: params[:id])
  @tag.update(name: params[:name])
  redirect_to "/tags/:id"
end
```

This is a common mistake that beginners make working with Ruby on Rails (it's easy to think that the wildcard url syntax in the routes file works in other places). Watch the code break, read the error message, and show the process of debugging the code. This helps quash misunderstandings before they begin.

Programmers spend more time debugging errors than writing code. It's, therefore, unrealistic to demo perfectly working code 100% of the time - you miss out on opportunities to show the debugging process. Try to demo things the wrong way (based on common student mistakes) on occasion to talk through the debugging process.

Stick to one direction at a time.

Don't introduce multiple syntaxes

As an instructor, you will be tempted to show students multiple ways of doing the same thing. In Ruby, you can write single-line blocks using curly brackets instead of writing out the "do" and the "end":

```ruby
# BAD INTRO: The short syntax
numbers.each { |number| puts number }

# GOOD INTRO: The long syntax
numbers.each do |number|
  puts number
end
```

Of course you want to teach someone the short syntax, it's way more fun! But you have to ask yourself: Is it worth teaching an alternative syntax given the cognitive load hit? What exactly does this syntax offer? If it doesn't offer any new functionality for a programmer who's just learning, then it may not be worth teaching.

This isn't to say you should never teach it. You should only teach an alternative syntax once you feel comfortable that the student has mastered the originally taught syntax.

Show, don't tell.

Don’t describe something when you can demo it instead

Students have short attention spans in general, and studies show that it’s far shorter for passive learning compared to active learning. Instead of talking about how something works, it’s much easier for them to focus and grasp the concept when they see it in action.

Distraction comes in many forms.

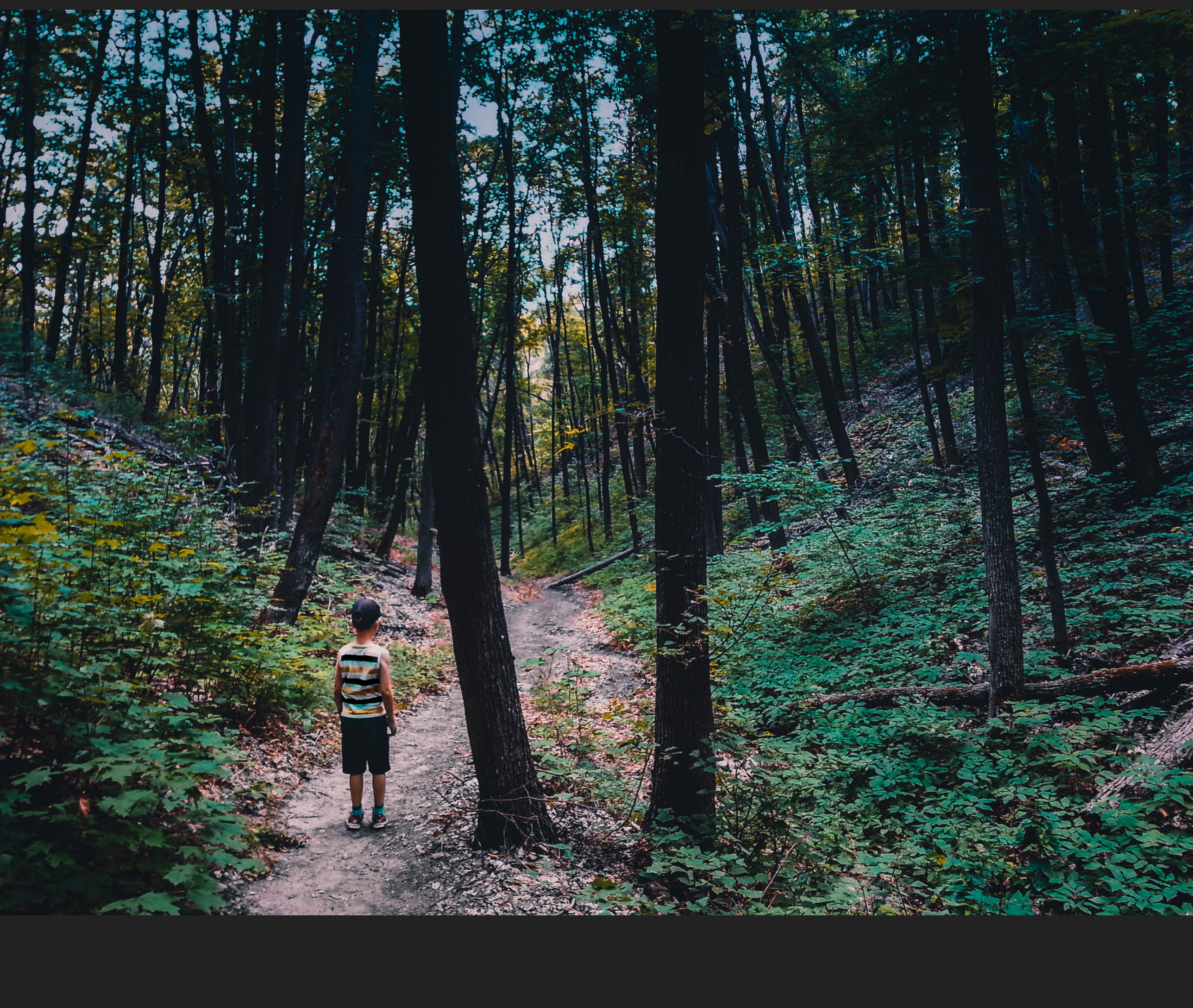
Don’t get into tangents

Any tangent away from the main concept being taught adds unnecessary cognitive load to the student. It’s human nature to want to share knowledge, but good classroom instruction requires restraint on the part of the instructor in this way.

A less effective instructor can’t resist sharing all their knowledge, even if students aren’t ready to learn that particular concept. And even if they are, if this other concept doesn't help clarify the topic at hand, this additional info dilutes their understanding of the main topic.

The instructor understands that this is "just an interesting tangent." But to a student, everything an instructor says has equal importance, so they divide their mental effort evenly among each concept the instructor mentions - even the unimportant ones.

A master instructor has restraint and knows when it is the appropriate time to introduce a particular idea.

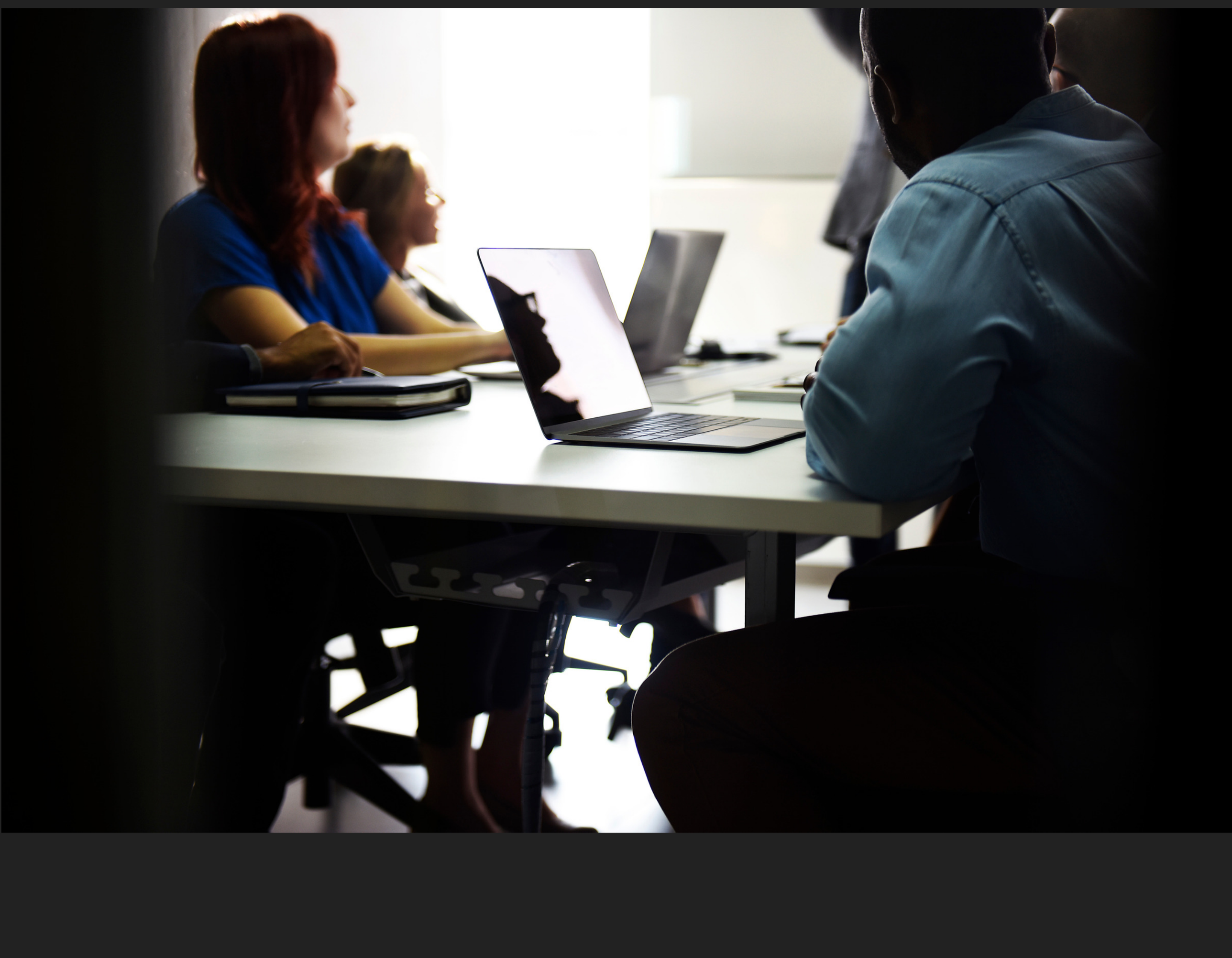


It's not always obvious when someone is lost.

Don’t assume that if no one asks a question, everyone understands

Teachers have the habit of asking “Does everyone understand?” as a way to check for understanding. When no one asks a question, it is often not due to understanding the material, but rather because they aren’t able to articulate what they don’t know.

Try posing specific questions involving specific scenarios to get a clearer picture of what students understand.



Don't let one person speak for everyone.

Don’t assume that if 1 or 2 people answer every question, everyone understands

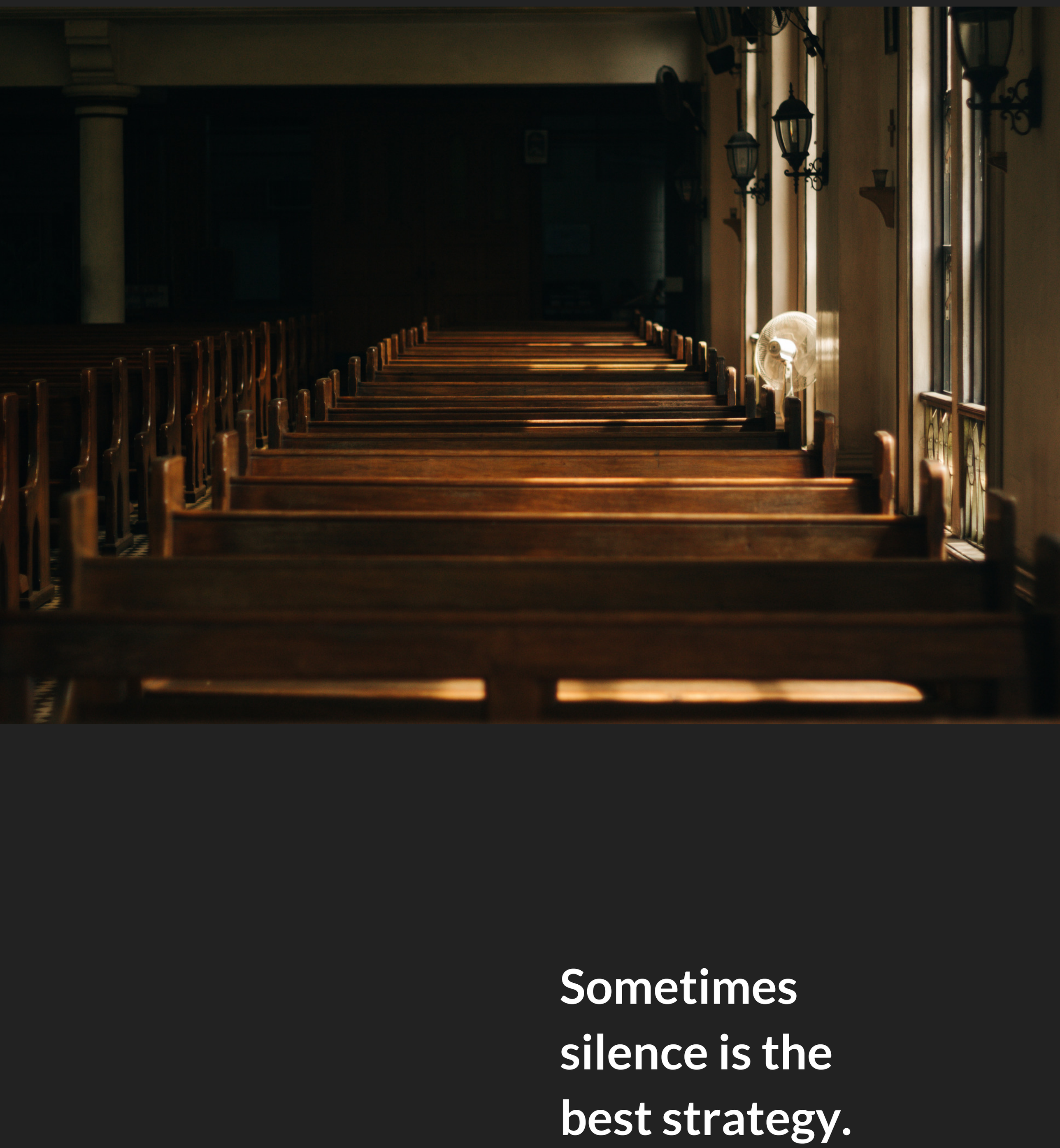
If 1 or 2 people answer every question, the only thing you know for sure is that 1 or 2 people understand the material. You must gather information about the understanding of *all* students. This doesn’t necessarily mean you have to call on specific students when asking questions - there are many good ways to check for understanding. Asking questions that only a few students answer isn’t one of them.

The best way to check for understanding is to assess students individually. However, that can be time consuming, so it's good to have some effective techniques to check for understanding as a group. Some possible strategies include conducting group polls, asking open-ended questions, and having students talk in pairs or groups.

Understanding is earned, not given.

Don’t give away the answer to a question immediately

The best answer to a question is with another question. Try to get the students to think through a problem by providing a guiding question that reinforces general problem solving strategies. Again, you should resist the natural desire to share your knowledge. Ideally, an instructor’s role is not to *give* knowledge but to *lead* students to discover that knowledge for themselves.



Sometimes
silence is the
best strategy.

Don’t try to answer all student questions

Some questions require students to understand several other concepts that have yet to be taught. Resist the temptation to answer every question - it’s okay to tell a student that the question will be revisited in the near future.

Conclusion

Teaching is one of the most important activities in the world. As an individual, becoming a better teacher helps you become a better learner, and becoming a better learner helps you become a better teacher. As a society, teaching helps better the population, enabling people to accomplish more than the sum of individual parts. This goes double for programming, which has a potential reach greater than any technology that came before it.

Today there are more resources for learning than ever before. Yet a large portion of those resources lack the fundamental understanding of what it means to learn and what it means to teach. We hope that you can apply the principles described in this book to improve your own teaching and make a difference in the world!