



CodeRealm

Meet the Cast

Standard Edition

Spark & Anvil

Copyright & License

© 2026 Spark & Anvil (501(c)(3) public charity). Chapter text and illustrations licensed under CC BY-NC-SA 4.0. App software © Spark & Anvil — all rights reserved. Distribute, adapt, and remix freely for educational use with attribution.

This book collects 6 chapter books from the CodeRealm cast — each character embodies a different curricular primitive; together they teach the full subject.

Methodology: story-based learning, where the characters embody the ideas being taught, with a trauma-informed, emotionally-safe approach.

Spark & Anvil is a 501(c)(3) public charity. All apps free forever; no ads; no tracking; no in-app purchases.

spark-and-anvil.org

This is Volume 1 of 2 of the *CodeRealm — Meet the Cast* series; each volume is a complete standalone book you can read on its own.

For everyone who learns by hearing a story first.

Contents

Copyright & License	
Contents	
Introduction	
Stash	
Fork	
Trek	
Module	
Glitch	
Order	
About Spark & Anvil	
More chapter books from Spark & Anvil	
Methodology	
License	
Keep exploring online	

Introduction

The CodeRealm cast was authored to embody the curriculum, not decorate around it. Each of the 6 characters you'll meet in this book teaches a specific primitive — a particular tactic, a particular technique, a particular way of seeing. Together they form an ensemble: the cast IS the curriculum.

Read in any order. Each chapter stands alone.

Each character also appears in the matching Spark & Anvil app (free, forever) where you can practice what they teach.

— *The editors at Spark & Anvil*

Stash

*VARIABLE / STORAGE — *the labeled box that holds a value until you call for it.**



Stash wasn't a fuzzy animal or a talking robot. Stash was a *box* — a small painted wooden box that sat on the classroom workbench, with a label on its side and a little value visible inside. The label read **NAME = X**. You could open the box to put a value in, or open it to take a value out. That was the whole of Stash.

"This is Stash," Loop said, patting the box. She always spoke on Stash's behalf, because Stash never pretended to be a person. "Stash is a *variable* — a labeled box. And I want to be honest with you right away, the way Stash would if it could talk: *I am not a mind. I do not think, I do not decide. I hold a value until you call for it. I am a box.*" A few kids who'd come in nervous — sure that "coding variables" would be something huge and mysterious — sat up a little.



Loop lifted the lid. "Watch. I'll *put a value in*." She set a card marked **5** inside and closed it. On the little screen behind her, it showed: `x = 5`. "That's called *assignment* — putting the value 5 into a box labeled 'x'." Then she opened the box and held up the 5 for everyone to see. "And this is *reference* — asking for whatever's in the box right now: `print(x)` — and out comes 5." She smiled. "A variable has two parts, and only two: a **name** (the label on the side) and a **value** (whatever's inside). That's it. Put in, take out."



"Now the part that surprises people," Loop said. She opened the box, took out the 5, and dropped in a card marked **10**. The screen updated: `x = 10`. "Same box. Same label — still 'x.' But brand-new contents. That's *reassignment* — you can empty the box and refill it whenever you like. The *label* stays put; the *value* can change as often as you want." She held up different cards. "And lots of kinds of things can go in the box, not just numbers — a bit of text like `"hello"`, or a whole list of things — depending on the language's rules. The box doesn't care much what you keep in it. It just *holds* it, steady, until you come back for it."



"Two more things, and they're about being *kind to future-you*," Loop said, tapping the label. "First — *names matter*. A good label tells you what's in the box: `score`, `playerName`, `currentLevel`. A lazy label like `x` or `temp` or `data` leaves you guessing later what you even stored." She pointed around the room to imaginary boxes. "Second — *scope*. Some boxes sit out where the *whole* program can reach them; others live only inside one little corner — one function's workbench — and vanish when that job is done. Knowing *where* a box can be reached keeps you from hunting for a value that isn't there." She closed Stash gently. "But under all of it — Stash is still just: box, label, value. Put in, take out, change the contents."



A student who'd been chewing her lip finally asked the thing she'd been dreading: "Is this the part that's really hard? Variables?"

Loop shook her head and opened Stash one more time, showing the plain card inside. "Not hard. Box. Label. Value. Put-in, take-out, change. *That's all a variable is.*" And Stash sat there on the bench, quiet and steady, waiting for the next value — not magical, not a mind, just a reliable place to keep something safe until you called for it.

And the kids who'd come in nervous felt it then — the worry loosening into something calm, and even a little brave. The thing that had seemed mysterious and out of reach had turned out to be as friendly and solid as a box you could hold in your own two hands. That feeling — *oh... I can hold onto this. I've got a steady place to keep what matters until I need it* — is the warm, quiet thing Stash leaves behind. Chase that steady feeling. Everything else in CodeRealm is built on it.

Read it? Now hear Stash's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/stash>

Fork

*CONDITIONAL / BRANCHING — *chooses a path based on what's true right now.**



Fork wasn't a fluffy squirrel or a talking badger. Fork was, well, a *fork* — not the dinner kind, but a small painted wooden sign, the sort you'd see on a tiny path. Above it floated a little diamond shape with the word **IF** inside, and two arrows pointed away from the diamond: one left, one right.

"Alright, team! Meet Fork!" Loop clapped her hands — Loop always sounded excited. "Fork helps us make choices. Not *Fork's* choices, mind you. *Our* choices. Or really, the *program's* choices." I looked at Fork. It just sat there, not deciding anything. "See the diamond?" Loop asked. "It holds a question — a *condition*. If the answer is YES, we go one way. If it's NO, we go the other."



"Let's say we want a super-sparkle from the Sparkle Collector," Loop said, patting the big clunky contraption that usually just coughed out regular sparkles. "The machine's picky. It only gives a super-sparkle if you already have *exactly five* regular sparkles." She pointed at Fork's diamond. "So the question in here is: *Do we have five sparkles?*"

She traced the left arrow. "If the answer is YES — this way. The program tells the machine: super-sparkle, please. Hooray!" Then the right arrow. "If NO — that way. Go find more sparkles. Boo."

"So Fork doesn't *decide*?" I asked. "It just shows the paths?"

"Exactly! Fork is a *branch-point*, not a decision-maker. The *condition* makes the decision. Condition true, branch one way. Condition false, branch the other."



"Conditions are like yes-or-no questions," Loop went on. "They're either TRUE or FALSE. No 'maybe' allowed in code — it's super strict. 'Is the sky blue?' TRUE. 'Are pigs flying right now?' FALSE — unless you know something I don't." I giggled. "So a condition is just a fact we check. We can ask if two things are the *same* with two equals signs: `==`. Is `sparkles == 5`? We ask if they're *not* the same with `!=`. Is `machine != broken`?" She was on a roll. "We can ask if one thing's *bigger* or *smaller* — `>` and `<`. Is `your_age > 9`? Is `shoe_size < 12`? Every one of these gives back a TRUE or a FALSE. That's all a condition ever does."



"What if we need to ask two questions at once?" I wondered.

"Great thinking! We *combine* conditions with **AND** and **OR**." Loop nodded at the machine. "Maybe it only works if you have five sparkles **AND** the machine is switched on. For an **AND**, *both* must be true — one false, and you branch the other way. For an **OR**, only *one* needs to be true: 'five sparkles **OR** it's your birthday' gets you the super-sparkle either way." She grinned. "Verity, over in DiscreteQuest, taught us all about **AND** and **OR** — she says they're the building blocks of good thinking. And here's the wild part: the true path can hold *another* Fork inside it. That's called *nesting* — a choice that leads to another choice, like a tree of paths. If seven days each need a different path, we could stack 'if... else if... else if...' — but there's a tidier way called a **match** or **switch** statement, a neat little menu you pick from. Loop *loves* tidy code."



"Does Fork know the answer ahead of time?" I asked. "Like, whether I have five sparkles before I even get there?"

"Nope — and that's the clever part," Loop said, tapping the diamond. "The condition is checked *right when you need it*. Not decided in advance. It depends on what's true *right now*. Different inputs, different paths — that's the whole magic of branching. So remember: Fork is the *conditional*. If a thing is true, go this way; if not, go that way. Fork's a branch-point, not a decision-maker. It's not hard — true, this way; false, that way."

I looked at Fork's two arrows one more time, and something in my chest went quiet and easy. All those choices that used to make me feel wobbly and stuck — they were just a question with a yes and a no. I felt lighter, like I could stand at a fork in the path now and not be scared of it. Just check the condition. Then go.

Read it? Now hear Fork's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/fork>

Trek

*LOOP / ITERATION — *keeps going around until the work is done.**

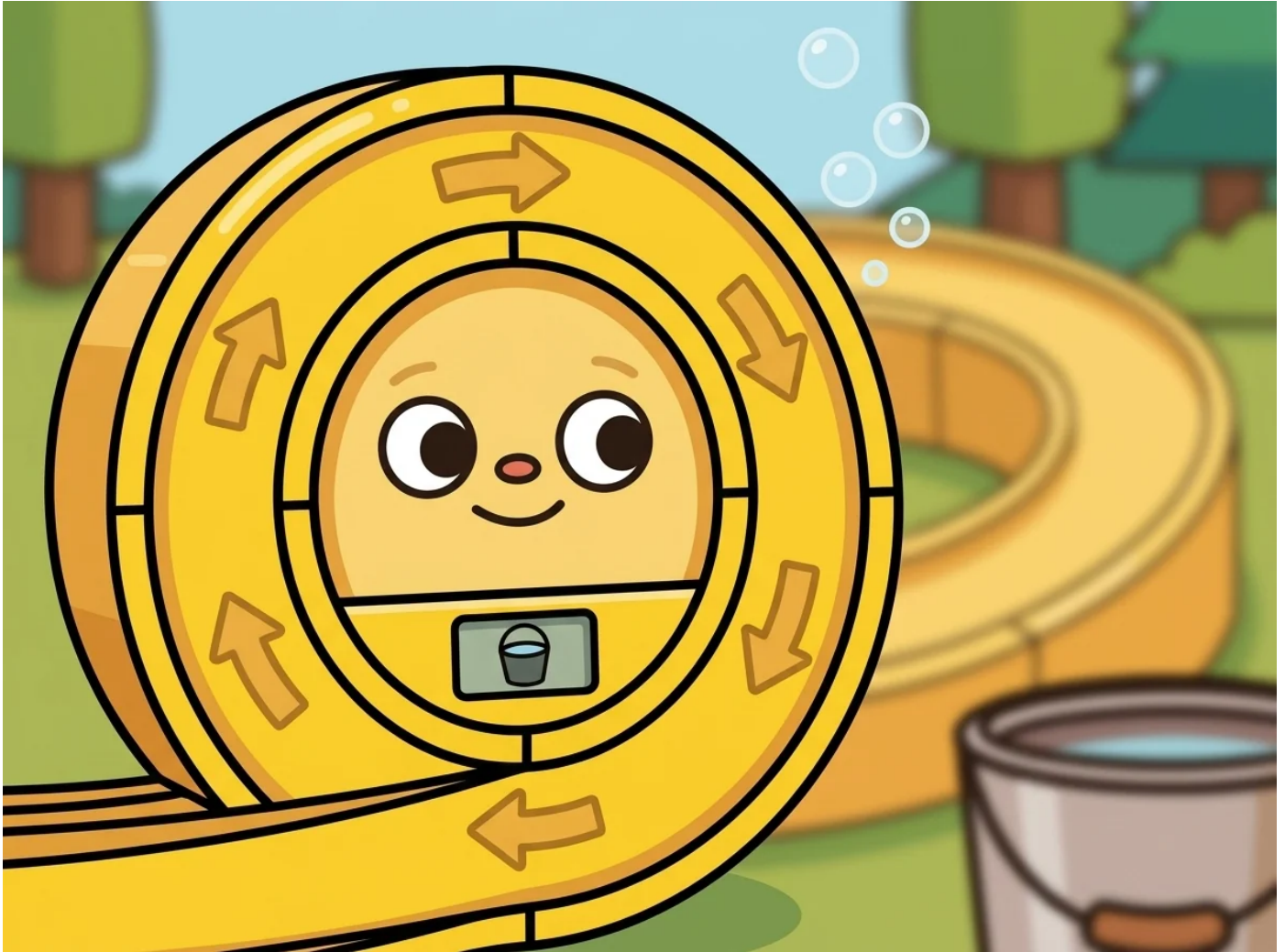


Trek wasn't a furry creature or a squishy blob. Trek was a small round track, painted bright yellow, with little arrows pointing all the way around to show the way to go, and a tiny screen on its front that showed numbers.

Trek's whole job was to *loop* — to do a thing again and again until it was done. It was the master of repetition, the champion of doing-it-over. Trek showed how a program runs the same block of steps over and over, which is called **looping**, or **iteration**.



One sunny morning Pip — a zippy little Variable, always changing, always moving — zoomed into the plaza holding a big empty bucket. "Trek! We need water balloons! Lots of them, for the big water fight!" Trek rolled to the Water Spout, held a blue balloon under it until — *POP* — it was full, tied it off, dropped it in the bucket, and its counter flashed **1**. "Again!" said Pip. Trek filled another — **2** — and another — **3, 4, 5** — until Pip suddenly gasped, "Wait — how many do we *need*?" Trek paused, arrows glowing faintly. "Oh! Exactly *ten*!" Trek's eye brightened: a **stopping condition**. It went back to work — **6, 7, 8, 9, 10** — and when the counter hit ten, Trek stopped cold. The arrows faded. "Perfect!" Pip cheered. (If Pip had never said *ten*, Trek would've kept going, and going, and going, and buried CodeRealm in balloons *forever* — that's an **infinite loop**, and it's why you *always* need a stopping condition.)



"I could've filled them one by one," Pip mused, looking at the full bucket. Trek just blinked. Filling ten by hand takes a while — and what about a *hundred*? A *thousand*? Trek's way was better: it repeated the *same* steps automatically. No need to write out *fill balloon, tie balloon, drop in bucket* ten separate times — you just say *do it ten times*. That's the power of a loop: it saves you from endless copy-paste. And that little counter on Trek's screen — that was its **counter variable**, keeping track of how many times it had gone around, like a scoreboard for the work.



Just then a green balloon rolled by, and Pip gasped: "That one's got a hole!" Trek reached for it, spotted the leak, and nudged it aside with a wheel — *skipping* the bad one and moving straight to the next good balloon. No point filling a leaky balloon. A little later Pip called, "Now we need a *red* one — just one!" Trek filled balloons — blue, yellow, green, no red — then found a bright red one, filled it, set it in a special spot, and *stopped*. It didn't need any more. "Sometimes," Trek's arrows seemed to say, "you skip a step, and sometimes you stop early." Those two moves have names: **continue** (skip this one, keep looping) and **break** (stop the loop right now). They let you steer the flow instead of blindly repeating.



"New plan!" Pip announced. "*Five buckets* of ten balloons each!" Trek's eye widened — that was a loop *inside* a loop, a **nested loop**. It would loop ten times to fill one bucket, and loop five times to do that for five buckets, its screen now reading *Bucket 1 of 5 · Balloon 1 of 10*. Fill, tie, place — ten times — bucket one done, set aside. Then bucket two. It took a while, but soon five full buckets stood ready: fifty balloons. "Amazing!" Pip gasped. Trek just blinked — it was all still *repeating*, only with more layers. A slinky figure named Coil slithered past — Coil was a master of *recursion*, doing things by calling *itself* over and over, a twistier cousin of the loop. "Keep on looping, Trek!" Coil waved. Loops and recursion: cousins, both about doing a thing again until the job is done.

And Pip flopped down beside the fifty finished balloons, worn out but happy, a warm glow of pride spreading through them. The job that had felt *huge* and impossible at the start — fifty balloons, one at a time? never! — now felt light, and *done*. That easy, satisfied, big-repeating-job-finally-finished feeling — the deep exhale when the work is complete and you didn't have to grind through it by hand — that was the best part of the whole day. That's what a good loop gives you: it carries the repeating so *you* get to feel the finish.

Read it? Now hear Trek's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/trek>

Module

*FUNCTION / ENCAPSULATION — *does one job well and can be called anywhere.**

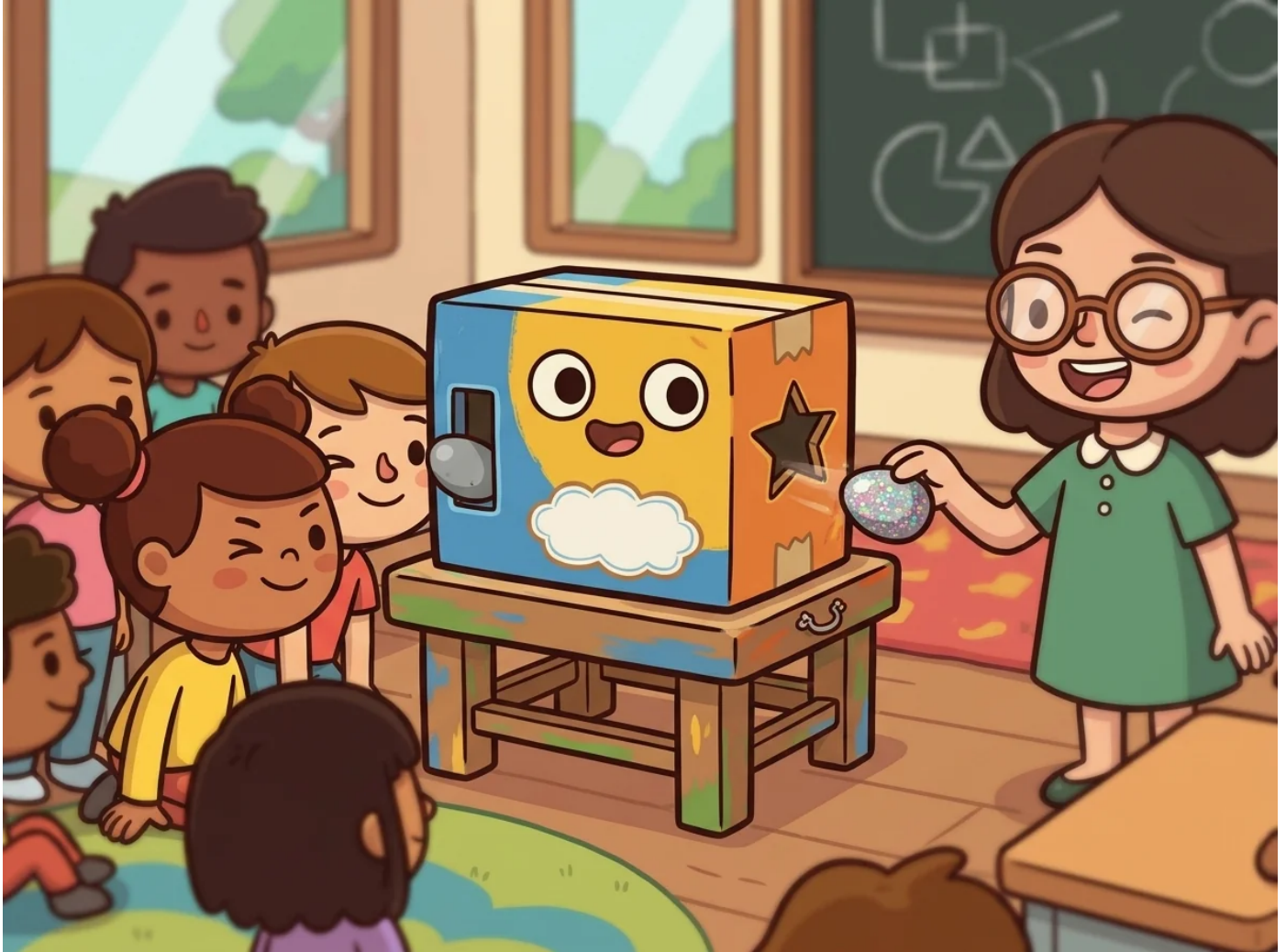


Module wasn't furry or feathery. Module was a box — a small painted box with a label on the front that told you exactly what it did. On its left side was a slot for *inputs*; on its right side, a slot where *outputs* came out. Loop always said it best: "Module does one thing. And it does it *well*."

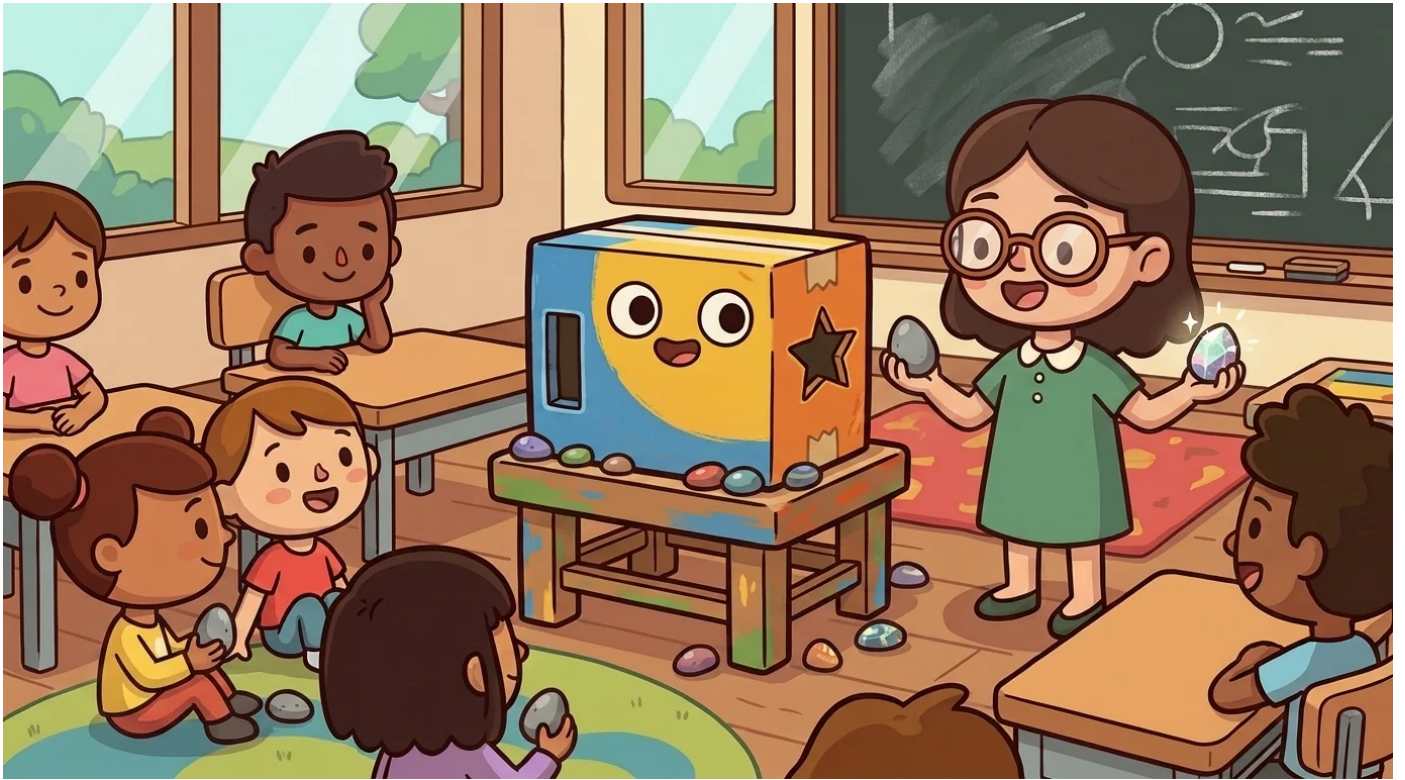
"Alright, class — today we meet Module!" Loop clapped her hands. Module sat on a little stand, its label reading *Add_Sparkle*. "See that? That's Module's name. It tells you its job." She held up a plain grey pebble. "And this is our *input*."



Loop dropped the pebble into Module's left slot. *Thunk*. A soft whirring came from inside, like a tiny happy bee — then *ping!* — and a pebble rolled out the right slot, no longer grey but shimmering with tiny rainbow sparkles. "Output!" Loop cheered. "A sparkly pebble!" She held up another plain pebble. "What'll happen?" "Sparkles!" someone shouted — and *thunk, whirr, ping*, out came another. "Exactly. Same input, same output, *every* time. And here's the cool part — you can use Module *anywhere*. At the park. On the moon! As long as it gets an input, it gives you an output." She mimed tucking it in a backpack. "Ready whenever you need it."



"Now look *closely*," Loop said. "Can you see inside?" Everyone squinted. "Nope!" "That's the magic! You don't need to know *how* Module adds sparkles — only *that* it does. The inside is hidden. It's Module's secret." She grinned. "And why is that good? Because if I swapped out Module's whole secret sparkle-machine for a better one, you'd never know — you'd still drop in a pebble and still get a sparkly pebble out." "It just *works*," a kid whispered. "Exactly. It just works. That's a good **function** — and keeping its insides tucked away where they can't trip you up? That's called **encapsulation**."



"Now imagine a *big* job," Loop said, waving her arms. "A whole galaxy of sparkly planets! You *could* try to do it all at once — but that'd be messy and hard. Instead, you break it down: use Module for *one* planet, then the next, then the next. Each small job makes the big job easier." She tapped the label. "And if you had a *thousand* pebbles to sparkle — would you build a thousand sparkle-machines?" "No way!" "Right — you build Module *once*, and *call* it a thousand times, or a million, just by its name: 'Hey Module — *Add_Sparkle* this!' No rewriting the steps." She picked up an already-sparkly pebble and dropped it *back* in — *thunk, whirr, ping* — out came one so bright it glowed. "See? One function can even use another function's output — or call another function from *inside* itself. That's how you build truly amazing things."



"One last thing — *names*," Loop said, tapping the label. "*Add_Sparkle* — see how it's a *doing*-word, like 'run' or 'jump'?" She wrote on the board: `calculateTotal()` — "this calculates a total" — and `isValid()` — "this asks, *is it valid*?" "A function's name should tell you what it *does*, or what question it answers. You wouldn't name Module just 'Pebble' — that only says what it works *on*, not its job." She smiled and held up her fingers. "So remember Module — a **function**. Inputs in, output out, one job, call it anywhere. Break big problems into small pieces — that's **modularity**. Build it once, use it many times."

And here's the feeling Loop most wanted you to have. That moment when a job feels *way* too big — a whole galaxy, a thousand pebbles — and your chest gets tight and you don't know where to even start. And then you remember Module: *just one small piece, done well, and I can reuse it*. Break the giant thing into little one-job boxes, and the tightness loosens, and the impossible galaxy becomes just *one planet at a time*. That calm, I-can-actually-do-this feeling when a huge job breaks into small pieces — that's the quiet reward of a good function. Chase that.

Read it? Now hear Module's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/module>

Glitch

*DEBUGGING + INSPECTION — *there's always a reason; bugs are findable without shame.**



Glitch wasn't an animal, and wasn't a person either. Glitch looked like a tiny magnifying glass that floated over a line of code, with a warm, soft light glowing under the glass — not a scary red alarm, just a gentle glow that said, *come look at this*. Glitch showed you where a problem was. It never, ever said *you broke it*.

That mattered more than anything. Because movies and games love to make bugs seem scary — the programmer who "broke everything," the big embarrassing failure — and that fear makes cheeks go warm and shoulders go tight, and you can't think clearly when you're worried. Glitch existed to take that bad feeling away. Its whole gentle message was: *there's always a reason. Bugs are findable. Bugs are just information — chances to learn — not proof that you're bad at this*.



Loop was trying to make her game character, Pip — a little pixelated frog — jump. She typed the code, pressed **Run**, and instead of leaping high, Pip just wiggled sadly on the ground. Loop sighed. "Oh, Pip. What did I do *wrong*?"

A tiny glow appeared, hovering gently over one line of code. Glitch. No red flash, no loud noise — just warm and curious. Loop's shoulders came down, and she smiled a little. "Thanks, Glitch. I forgot. There's always a reason." She looked at the screen, where a small message read: `TypeError: cannot multiply sequence by non-int of type 'float'`. "See — Glitch reminds us to *read the error message*. These are helpful hints — they tell you *where* the problem is and *what kind* it is. 'TypeError' means I used the wrong kind of data." She scrolled to the line Glitch hovered over: `jump_power = "5" * jump_strength`. "Aha! I put *quotes* around the five — that makes it *text*, not a number. You can't multiply text like that." She fixed it: `jump_power = 5 * jump_strength`.



She ran it again. Pip *still* wiggled — but the error was gone. "Okay, new problem — no error message this time. So what's next, Glitch?" The warm light pulsed. "Right! **Print debugging**. You make the computer tell you what's happening inside — like asking the code, *hey, what number are you holding right now?*" Loop added `print(jump_power)` and `print(jump_strength)`, ran it, and watched the console:

```
50
10.0
```

"Hmm. Jump power is 50, jump strength is 10 — that *seems* fine. But Pip still isn't jumping. So the trouble's somewhere later." She wasn't frustrated. She was *curious* now — the bug was just a clue she hadn't followed yet.



"Another trick is **stepping through with a debugger**," Loop told Glitch, "like pressing pause on a video game — you stop the code and watch every value, moving forward one tiny step at a time." She set a breakpoint and ran it, and the program froze mid-jump so she could see every variable. "Good for tricky bugs. But sometimes you just need to *talk it out*." She reached for a small green rubber duck. "Okay, Mr. Quackers. Pip the frog isn't jumping. My code sets `jump_power`, then updates Pip's position, then—" She stopped dead. "Wait a minute. I set Pip's `y` position to `0` **after** I add the jump power — so he lands back on the ground *immediately*!" She'd found it — that was **rubber-duck debugging**, where explaining code out loud lets your own brain hear the mistake. She swapped the two lines, put the position update *before* the jump, and ran it. Pip *leaped* — a perfect pixelated flip. "YES! Good job, Pip! Good job, Mr. Quackers!" Glitch's light pulsed happily.



"And when a bug's *really* hidden," Loop said, pulling out a notebook labeled *Bug Journal*, "there's **bisecting** — you comment out half your code and run it. Bug gone? It was in that half. Bug still there? It's in the other half. Keep cutting in half until you land right on it." She tapped the journal. "And I always write down what I tried and what I learned, so when the same kind of bug shows up again, I remember how I fixed it. A detective's notebook for code." She looked at Glitch fondly. "So — read the error, inspect your variables, print, step through, talk to a duck, bisect, keep a journal. Find it *gently*. Bugs are information, not failures."

And here's what Loop wanted every coder to feel: that first warm-cheeked, tight-shouldered *oh-no-I-broke-it* moment — and then Glitch's soft light arriving, and the whole feeling loosening, because a bug was never a verdict on *her*. It was just a reason, waiting to be found. She let out a slow breath, shoulders easy, and felt the quiet, steady calm of a person who knows that whatever's broken, *there's always a reason — and I can go look for it, gently*. That calm, she'd tell you, is the best part of the whole job.

Read it? Now hear Glitch's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/glitch>

Order

*SEQUENCE + SYNTAX — *order matters in code.**



Order was the last of the CodeRealm crew to arrive — the sixth member. Loop set it down carefully. It wasn't a fuzzy animal or a talking robot; Order was a small flat figure that looked like a recipe card folded just so, with bold numbers down its front — 1, 2, 3, 4 — each with a short line beside it. Order was a *list of steps*.

"This is Order," Loop announced, speaking for it as always, since Order itself was silent. The others gathered close — Stash the storage box, Fork the branching sign, Trek the looping track, Module the one-job box, Glitch the gentle magnifying glass — all of them clever *objects*, not people, not animals. "Order shows us two big things," Loop said. "First, *sequence*. Second, *syntax*."



Loop picked up a shiny toy rocket, in pieces. "We need to build this. Order will help." She pointed to line one. "*Step 1: attach the rocket body to the base.*" She clicked the body onto the base — perfect fit. "*Step 2: attach the fins.*" A kid named Leo grabbed a fin and tried to stick it on the nose cone. It wobbled, wouldn't stay. "Look at Order," Loop said gently. "Step 2 says *fins* — where do fins go?" Leo spotted the slots on the body and snapped them in. "*That's* sequence. Step 2 can't happen before Step 1." She held up a tiny engine. "*Step 3: insert the engine.*" Leo tried the top — it bounced off. "Wrong spot — the engine needs the body ready first." He found the hole at the bottom; it clicked. "Some steps *depend* on other steps finishing first. Do step 1, then 2, then 3. Top to bottom."



"Now — *syntax*," Loop said, pulling out a card like a recipe for Sparkle Slime:

```
Mix (water, glue, glitter);  
Stir for 3 minutes;  
Add (activator);
```

"Syntax is the *rules for how we write things* — like punctuation in a sentence." She poured water and glue. Leo peered at the card. "It says 'glitter' with a *comma* after 'glue.'" "Exactly — the comma tells the computer that water, glue, and glitter are three *separate* things to mix. Forget it, and it might think 'waterglue' is one thing." Leo added the glitter. "And see the *semicolon* at the end of the line? That tells the computer: *this instruction is done — move to the next.*"



"Now, 'Add (activator);'" Loop said. "What if I wrote 'Add activator' with no parentheses?" Glitch buzzed softly, a small exclamation point popping above it. "Glitch says that's a problem. The parentheses are special brackets — they tell the computer *activator* is the specific thing to add. Without them, it doesn't know *what* to add." She added the activator; the slime thickened and shimmered. "Computers are *very* strict about syntax — punctuation, spelling, even how you indent. They don't *guess* what you mean." She held up a tiny screen flashing `ERROR: Missing semicolon on line 1`. "See? A syntax error, and it *stops* — but it tells you exactly what's wrong. That's *helpful* — better to catch the mistake early than let the slime turn into a rock." "So Order's like a very picky recipe card," Leo said. "Exactly! Sequence *plus* syntax."



Loop gathered all six figures — Stash, Fork, Trek, Module, Glitch, Order. "All of us are here now, and we're *objects* — not humans, not animals. We show programming ideas without pretending to be people, because programming operations are *operations*, not personalities. That matters." She looked at Order. "Order is sequence and syntax: top-to-bottom execution, punctuation matters, steps encode dependencies. Not hard — order matters, read the steps."

Leo let out a slow breath and grinned. The rocket sat there, whole and steady, the slime shimmering beside it. He felt proud, and *settled* — like something that had been tangled and rushed and frustrated inside him, back when the fins wouldn't stay and nothing fit, had smoothed itself right out, one calm numbered step at a time. That feeling — when a jumble of instructions finally *clicks* into the right order and everything just *works* — that quiet, everything-in-its-place calm is the reward Order gives you. Slow down. Follow the steps. Feel it click.

Read it? Now hear Order's audio drama + go play — scan or visit:



<https://spark-and-anvil.org/cast/coderealm/order>

About Spark & Anvil

Spark & Anvil is a 501(c)(3) public charity. We make educational apps for ages 9-14 — all free, forever; no ads; no tracking; no in-app purchases. CodeRealm is one of 140+ apps in the portfolio.

More chapter books from Spark & Anvil

Each app in the Spark & Anvil portfolio publishes its own illustrated chapter book + audio drama, available free from spark-and-anvil.org/books. Highlights include:

- **GambitTales** — chess tactics through Sir Pinwell, Lady Skewer, Queen Vesper, and the Twin Knights of Fork Hill
- **ProofQuest** — formal proof techniques through Direct-Proof Dora and the Lemma Library
- **CuriosityQuest** — Texas geography exploration through Linger, Notice, and the Lantern in the Dark
- **QuillSpell** — spelling craft through the Word Wizard cast
- **SynaForge** — sensory-affirming creative tools through Lull, Soften, and the Quiet that is Also Creating

Methodology

This book uses story-based learning — the characters embody the ideas being taught — grounded in the work of Jerome Bruner (on narrative and cognition) and Sebastian Habgood (on weaving learning into the play itself), with a trauma-informed, emotionally-safe approach drawn from SAMHSA's trauma-informed care framework.

Trauma-informed design informed by Eggleston et al. (2025) and Stoltenburg et al. (2024).

License

© 2026 Spark & Anvil (501(c)(3) public charity). Chapter text and illustrations licensed under CC BY-NC-SA 4.0. App software © Spark & Anvil — all rights reserved. Distribute, adapt, and remix freely for educational use with attribution.

Cover art, chapter illustrations, and chapter text are reviewed for age-appropriate, trauma-aware content before publication. Audio drama transcripts available at spark-and-anvil.org/cast.

Keep exploring online

Play it now	Meet the cast	More free books
		
https://spark-and-anvil.org/play/coderealm	https://spark-and-anvil.org/cast/coderealm	https://spark-and-anvil.org/books

Scan a code with any phone camera, or type the address. Every app, story, and book is free, forever — no account, no tracking.