

PragPub

The Second Iteration

SPECIAL ISSUE: TEACHING KIDS TO CODE

IN THIS ISSUE

- * Why you should be teaching kids to code
- * The Anti-Cosby Approach
- * The Hour of Code
- * David Bock on giving back
- * Constraints and Freedom
- * A Coding Competition
- * Coding Unplugged
- * The Selfish Teacher

Contents

FEATURES



The Anti-Cosby Approach to Teaching Kids Programming 2

by Chris Strom

Bill Cosby was wrong. Here's how to teach kids to program.



The Hour of Code 8

by David Bock

David reflects on a week teaching kids to code, and asks why we can't make every week Computer Science Education Week.



Giving Back 12

by David Bock

In this latest in our series on teaching kids to code, David shares his experience and invites you to join in.



Constraints and Freedom 15

by Jimmy Thrasher

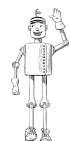
It was Igor Stravinsky who said, "The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit." The same insight also applies when teaching kids to program.



Lego League: Lessons Learned 19

by Seb Rose

Seb Rose shares lessons learned from coaching and judging a coding competition.



Coding Unplugged 23

by Fahmida Y. Rashid

Most child-oriented programming initiatives target kids in the 9–16-year-old range. But their kids were just five and eight. They needed to get creative.

DEPARTMENTS

Editorial	1
by Michael Swaine	
Why teach kids to code?	
The Selfish Teacher	26
by Michael Swaine	
How teaching a kid to code could be a wonderful thing to do—for yourself.	
Resources	27
Tools for teaching kids to code.	

Except where otherwise indicated, entire contents copyright © 2014 The Pragmatic Programmers.

You may not sell this magazine or its content, nor extract and use more than a paragraph of content in some other publication without our permission.

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail webmaster@swaine.com. The editor is Michael Swaine (michael@pragprog.com). Visit us at <http://pragprog.com> for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Editorial

Teaching Kids to Code

by Michael Swaine



Welcome to this special issue of *PragPub* on teaching kids to code. This issue is entirely free. It is also sort of an experiment. Every month we will try to add to the issue. Right now, it contains:

- “The Anti-Cosby Approach to Teaching Kids” by Chris Strom
- “The Hour of Code” and “Giving Back” by David Bock
- “Constraints and Freedom” by Jimmy Thrasher
- “Lego League: Lessons Learned” by Seb Rose
- “Coding Unplugged” by Fahmida Y. Rashid
- “The Selfish Teacher” by your editor

Each month we will try to add an article. The issue will remain free as it grows, and will be available in pdf, mobi, and epub formats. If you would like to be alerted when the next update occurs, just [just email us](#) ^[UI].

Yeah, but why teach kids to code? It’s not a silly question. We’ve convinced ourselves, based on copious evidence, that kids pick up all things technological more quickly than adults do. We all know toddlers who seem to grok the user interfaces of phones and tablets and game machines almost intuitively, and if you’re over thirty you probably have the strong impression that people younger than you have a much more intimate connection with their phones than you.

So is all this necessary? Won’t kids just pick it up on their own?

Well, they might. But I’m assuming that if you’re reading this, you’re a programmer. And so you know that there is good coding and there is bad coding. You’ve learned that lesson, and it didn’t just come to you by osmosis.

Besides, it’s really not true that all the kids who could benefit by learning to code will pick it up on their own. One case in point: The dearth of women in tech jobs is not just a result of sexism in hiring. It goes all the way back to grade school, where girls typically don’t even perceive learning programming as an option for them. Just getting a kid to think about programming as something they could do is a positive thing.

This is one of those places where an individual can really make a difference. You have knowledge that kids can benefit from. You know that programming is problem solving, and you have a toolbox filled with problem-solving tools. Spending a few weekends or evenings teaching a roomful of kids to code could have more impact than all the code you’ll ever write. It is quite literally an investment in the future.

Besides, the kids will probably teach you a few things, too.

The Anti-Cosby Approach to Teaching Kids Programming

Spinning Donuts

by Chris Strom

Start with dessert and save the green beans for later.

Agree? Disagree? Have related experiences to share? Don't be shy. Weigh in at [the forum](#) [U1].



“The Cos” was wrong.

Yeah, I said it. With all due respect to the great Bill Cosby and his beloved Cliff Huxtable character, he got it wrong when teaching his TV kids. And I think we suffer from the same problem when teaching our kids to program.

In yet another classic episode, daughter Vanessa became intensely interested in singing. Naturally, her parents sent her to a vocal coach. Just as naturally, Vanessa was bored stiff by the basic vocal exercises with which the vocal coach started.

And here is where The Cos went wrong. He explained, as only he could, why the fundamentals were so important. Why Vanessa could never be a great singer without a really strong foundation. Since it was TV, Vanessa saw the wisdom in her father’s words and agreed to keep at it.

Look, that might work in the world of television, but in the real world she never would have gone back—or only would have gone back a few times before her initial overwhelming excitement was extinguished by the boredom of the fundamentals.

Killing the Excitement of Programming

Let’s talk about programming. Programming is hard. There are so many fundamentals of programming that one could spend every single day for 5-plus years working to master them and still feel like there are more unknown than known fundamentals.

But we do it anyway. We program because we once typed some code and made something really cool. We might not have understood everything, but we created something that could shape our reality. And we want to keep doing just that. Along the way, we learn fundamentals and we may even work very hard to perfect them. But none of us ever got excited about the fundamentals and then created something amazing. So why teach kids like this?

I am not a gamer. I am much happier tinkering with obscure new protocols, hipster programming languages, and the latest web frameworks. But when I was a kid, I started programming by copying games from books and typing them into the computer. I would play the games a little, but I had much more fun coding the games and then exploring what changes I could make without breaking things.

Maybe there are kids out there who really enjoy laboriously working through individual fundamentals. But I am pretty sure that optimizing teaching for play rather than for fundamentals leads to more learning—and eventually to more

understanding of the fundamentals. So how do you optimize teaching for play? It takes some effort, but it is very doable.

A 3D Example

Consider the following HTML and JavaScript that creates a Three.js donut on a web page:

```
...
<body></body>
<script src="http://gamingJS.com/Three.js"></script>
<script src="http://gamingJS.com/ChromeFixes.js"></script>
<script>
  // This is where stuff in our game will happen:
  var scene = new THREE.Scene();
  // This is what sees the stuff:
  var aspect_ratio = window.innerWidth / window.innerHeight;
  var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
  camera.position.z = 400;
  scene.add(camera);
  // This will draw what the camera sees onto the screen:
  var renderer = new THREE.CanvasRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);
  var shape = new THREE.TorusGeometry(100, 25, 8, 25);
  var cover = new THREE.MeshNormalMaterial();
  var donut = new THREE.Mesh(shape, cover);
  scene.add(donut);
  // Now, show what the camera sees on the screen:
  renderer.render(scene, camera);
</script>
...
```

That is 20-ish lines of HTML, JavaScript code, comments, and whitespace—and nearly that number of concepts that kids (or anyone new to Three.js) would need to learn. To name a few, there are the HTML tags, the script tags with src attributes, the script tags that wrap all of the Three.js code, and there is even some DOM coding to measure the window and add the Three.js renderer to the DOM. Then there are the 3D concepts like scenes, cameras, renderers, geometries, materials, and meshes. There are fundamental programming concepts like numbers, strings, objects, methods, and comments. Finally, there are files, web servers, and HTTP.

And all of this is necessary to make a donut:



If we took the Cosby approach to teaching kids all of that, we would have on our hands a very effective means of demotivating just about anyone who ever got excited in the first place. [And we'd never get any donuts.—ed] Maybe that was the real point of Cliff Huxtable's approach—to verify that Vanessa would be really, *really* dedicated to singing before he had to commit significant time and money to the effort.

But perhaps the alternative can be more effective.

Filtering Teaching Concepts to the Bare Minimum

Personally, I want to know all about renderers, the DOM, and object methods. Kids just want to make the donut spin. And make it chocolate. Ooh and shiny with a cool shadow!

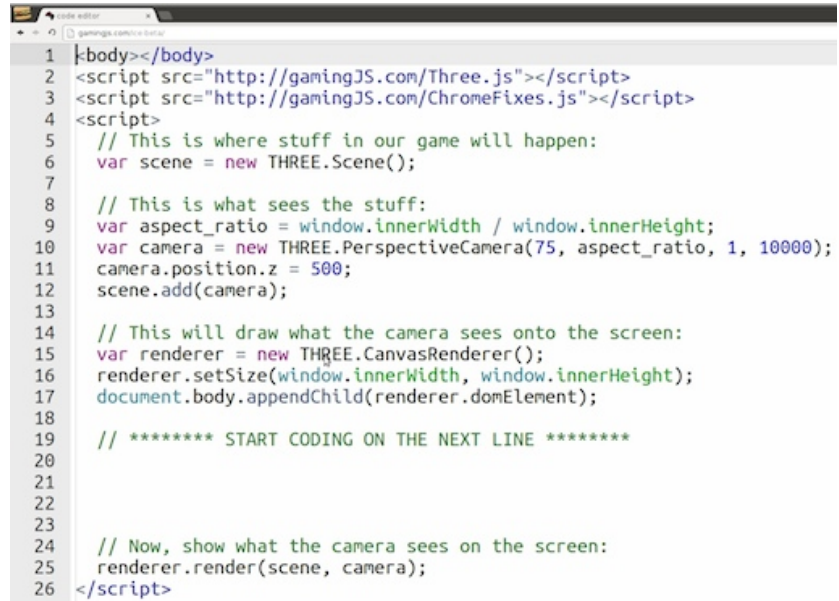
But how are they going to go about doing that if they have to type in all that code *and* understand the 20-plus concepts in it? There is just no way that is going to work. What *does* work is identifying a single concept and optimizing teaching around that single concept. Oh, and let's make a fun concept. Like donuts. Everyone loves donuts.

To make a donut—or anything in 3D—you need two things: a shape and something to cover that shape. The combination of the shape and cover is a “mesh.” This is a single core concept that kids can understand and start coding. Thanks to the excellent Three.js library, it is four lines of code:

```
...  
var shape = new THREE.TorusGeometry(100, 25, 8, 25);  
var cover = new THREE.MeshNormalMaterial();  
var donut = new THREE.Mesh(shape, cover);  
scene.add(donut);  
...
```

Create a shape. Create a cover for that shape. Together they make a mesh. Add the mesh to the scene. All of these concepts kids can understand. There are fundamentals to be learned in there (objects, methods, variables), but those can come later.

By itself, those four lines won't work. The other 20-plus lines of code from the original example are needed to set up the 3D scene and to draw it. But if we optimize for learning the single concept of a mesh, we can see a way to enable kids to focus only on the donut mesh—by starting them with a code template that contains the other 20-plus lines:

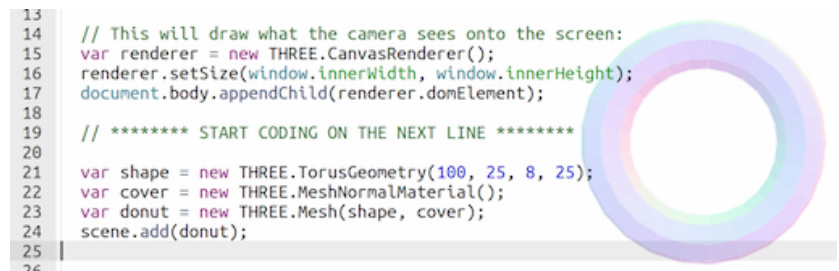


```
1 <body></body>
2 <script src="http://gamingJS.com/Three.js"></script>
3 <script src="http://gamingJS.com/ChromeFixes.js"></script>
4 <script>
5   // This is where stuff in our game will happen:
6   var scene = new THREE.Scene();
7
8   // This is what sees the stuff:
9   var aspect_ratio = window.innerWidth / window.innerHeight;
10  var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
11  camera.position.z = 500;
12  scene.add(camera);
13
14  // This will draw what the camera sees onto the screen:
15  var renderer = new THREE.CanvasRenderer();
16  renderer.setSize(window.innerWidth, window.innerHeight);
17  document.body.appendChild(renderer.domElement);
18
19  // ***** START CODING ON THE NEXT LINE *****
20
21
22
23
24  // Now, show what the camera sees on the screen:
25  renderer.render(scene, camera);
26 </script>
```


The new programmer can then start coding on line 20, right below the very enticing START CODING line. The other stuff is there—for exploration and to make our donut work—but the focus is now clearly on the new code that we type.

It is still possible to eliminate more concepts in an effort to focus on the single concept of the 3D mesh. Since this is a web page, it still needs a web server or file system so that a browser can load it. To eliminate that complication, enter the [ICE Code Editor](#) [U2].

The ICE Code Editor is a *very* simple, in-browser editor. One of the nicer features is that it has code templates such as “3D starter project” that start kids off with reasonable defaults. More importantly to kids, the code entered is rendered right in the browser. Immediately after entering those four lines of code, the ICE editor displays the results:



```
13
14 // This will draw what the camera sees onto the screen:
15 var renderer = new THREE.CanvasRenderer();
16 renderer.setSize(window.innerWidth, window.innerHeight);
17 document.body.appendChild(renderer.domElement);
18
19 // ***** START CODING ON THE NEXT LINE *****
20
21 var shape = new THREE.TorusGeometry(100, 25, 8, 25);
22 var cover = new THREE.MeshNormalMaterial();
23 var donut = new THREE.Mesh(shape, cover);
24 scene.add(donut);
25
26
```



There is something visceral about typing a few lines and creating something like that. It taps into what we still love about coding as adults: the ability to control computers to shape the world that we see.

Best of all, this approach invites play. What do those numbers in `THREE.TorusGeometry()` mean? Kids can play with them to find out. What do the numbers in the template mean? Again, kids can explore by changing them and watching the ICE Code Editor display the results.

Eventual Fundamentals

Teaching a single concept at a time makes learning easier. Especially with code that produces visual results, it makes it much easier to map these concepts into previous real-world experience.

At some point fundamentals will come. At some point kids will wonder what the `var` means and why some code has a `new` while other code does not. At some point they will want to know about the renderer because that will let them explore other, more sophisticated renderers. Heck, they will probably even want to know about the DOM (and we adults should try to protect them from that as long as possible!).

But you know what? The next thing most kids will want to do is move the donut around. Personally, I like spinning it:

```
...
var clock = new THREE.Clock();
function animate() {
  requestAnimationFrame(animate);
  var t = clock.getElapsedTime();
  donut.rotation.set(t, 2*t, 0);
  renderer.render(scene, camera);
}
animate();
...
```

That is a fair bit of code, but there are only two simple concepts. The first is that we need to spin the donut by a small amount with every clock tick. The second is that the scene needs to be redrawn after the donut is rotated by that small amount. Why we use a clock, a function, and that `requestAnimationFrame()` thing are all concepts that can come later.

But, for now, who cares? It's a spinning donut:

```
5 // This is where stuff in our game will happen:
6 var scene = new THREE.Scene();
7
8 // This is what sees the stuff:
9 var aspect_ratio = window.innerWidth / window.innerHeight;
10 var camera = new THREE.PerspectiveCamera(75, aspect_ratio, 1, 10000);
11 camera.position.z = 250;
12 scene.add(camera);
13
14 // This will draw what the camera sees onto the screen:
15 var renderer = new THREE.CanvasRenderer();
16 renderer.setSize(window.innerWidth, window.innerHeight);
17 document.body.appendChild(renderer.domElement);
18
19 var shape = new THREE.TorusGeometry(100, 25, 8, 25);
20 var cover = new THREE.MeshNormalMaterial();
21 var donut = new THREE.Mesh(shape, cover);
22 scene.add(donut);
23
24 var clock = new THREE.Clock();
25 function animate() {
26   requestAnimationFrame(animate);
27   var t = clock.getElapsedTime();
28   donut.rotation.set(t, 2*t, 0);
29   renderer.render(scene, camera);
30 }
31 animate();
32
```



And even without learning the fundamentals, kids have already learned a ton about programming. Hopefully they have learned enough—and have had enough fun doing so—that they will seek out those fundamentals on their own. Even if they don't, they learned something and had some fun along the way.

What Comes Next?

This approach to teaching programming is not easy. It takes a lot of work to identify engaging core concepts and present them in a way that encourages play and eventually raises questions that lead to fundamental understanding.

I would like to think that we did this well in [3D Game Programming for Kids](#) [U3]. I know that the Khan Academy does an exceptional job of doing this in their online CS examples (complete with follow-along video). Regardless of the effort required of us, it is worth it. We are thinking of the kids.



About the Author

Chris Strom is a relentless public learner, with more than 1,500 blog posts serving as research notes for his writing. His books include *Patterns in Polymer*, [3D Game Programming for Kids](#) [U4], [Dart for Hipsters](#) [U5], [The SPDY Book](#) [U6], and *Recipes with Backbone*. He has more than ten years of experience programming in Perl, Ruby, JavaScript, and whatever his current obsession happens to be. Chris lives in Baltimore, Maryland, with his wife, four kids, and a goldfish named Martin Tanner.

External resources referenced in this article:

- [U1] <http://forums.pragprog.com/forums/134/topics/12676>
- [U2] <http://gamingjs.com/ice>
- [U3] <http://pragprog.com/book/csjava/3d-game-programming-for-kids>
- [U4] <http://pragprog.com/book/csjava/3d-game-programming-for-kids>
- [U5] <http://pragprog.com/book/csdart/dart-for-hipsters>
- [U6] <http://pragprog.com/book/csspdy/the-spdy-book>

The Hour of Code

Reflections on Computer Science Education Week

by David Bock

David reflects on a week teaching kids to code, and asks why we can't make every week Computer Science Education Week.



December 9th through the 15th was Computer Science Education Week in American public schools. Andy Hunt recently had the opportunity to catch up over dinner with David Bock, who had volunteered the entire week at a local elementary school.

Andy: Dave, so how did you get involved in Computer Science Education Week?

David: It started for me sometime in October. I had seen an article about Computer Science Education Week online, and sent an email to my kid's principal saying that if they'd pull something together in their computer lab, I'd be willing to volunteer for the week to teach kids.

Andy: So the school already has a computer lab?

David: Yeah, it has about 30 PCs in it with educational software, but they are all locked down, so it's hard to do something *interesting* with them.

Andy: Does the school teach any kind of programming?

David: Like most of the U.S., my county's school system doesn't teach anything like that. I've previously commented that they basically teach "20th Century office worker skills."

Andy: That's unfortunate.

David: Yeah, but while the curriculum might not be there from the top down, I've found that, at my school anyway, the administration, teachers, and kids are eager to make something happen.

Andy: So what happened after the email to the principal?

David: A few emails bounced around about logistics, but nothing solid came of it. Then, just about 2 weeks before the date, I got a rough schedule with teachers, classes, and times of the day. I suddenly felt in over my head.

Andy: How did you decide what you were going to teach them?

David: Luckily there is a bunch of material in support of Computer Science Education Week, including the Hour of Code. That's a [website](#)^[U1] with several one-hour lessons in computer science—with stuff you can do with kindergartners and graph paper, all the way up to high school level.

Andy: Sounds great! So what did you teach the kids?

David: The lesson plan that was heavily promoted for that week involved a few videos and something akin to turtle graphics, but using characters from Angry Birds and Plants vs. Zombies. You basically have to move a character

around with commands like **move forward** and **turn right**, and get them to their destination. There are 20 problems, and we had a great time with them.

Andy: And do you think this was a good introduction?

David: At first I had my doubts. The programming is done in a language called Blockly, where you take pre-written snippets of code and snap them together like legos. There is literally a *snap* sound when you connect the lines. To my cynical self, it seemed too cartoonish, without enough real “stuff” behind it.

Andy: What changed your mind?

David: I did it with my own kids first. I was amazed at how the choice of those two games kept them engaged. Also, I realized the problems they had to solve with the code were legitimately computer logic, but the way Blockly snaps together, the kids don’t have to worry about any language syntax.

Andy: Can you give an example of the “computer logic” they had to code?

David: Sure... in fact, I have 2 examples. In the first seven or so problems, the students have to move an angry bird to the same square a pig is in, using nothing but **move forward**, **turn left**, and **turn right**. The solutions are hard coded for that particular puzzle. But then, they have to start messing around with loops and if tests about their position in the maze. In one of the first puzzles to use loops, they have to repeat **move forward**, **move forward**, **turn right** three times in order to make the bird move around the 3 sides of a square. At first, the kids want to hard code the solution, and they would get frustrated when they couldn’t drag the **repeat 3 times** block to the trash can. I would explain the problem and then ask “What steps can we do 1... 2... 3... times (drawing my finger along the three sides of the square) to get the bird to the pig?”

At this point you could see the light go on behind their eyes as they’d exclaim “oooohhhhhh!” I joked with the other teacher that “that’s the sound of ignorance leaving the body.”

Another student said to me, “This is fun! It’s hard, but it’s easy at the same time.”

Andy: Cool. You said you had a second example?

David: Yeah—the last problem is a doozy, but the puzzle starts with most of the solution in place... the kids just have to debug it and add a few lines. The solution involves a **while** loop, and three **if-then-else** tests along with the commands to move or rotate.

Andy: That sounds pretty complex for an hour intro.

David: Well, the kids actually had less than the full hour, but I’ll talk about that in a minute. Only about ten percent of the kids made it through all 20 problems, but that didn’t dampen their enthusiasm since they could finish it at home. We’d walk through that final solution on the whiteboard/projector at the end of the lab time so everyone could see it. After they understood the solution, I’d point out, “But that isn’t just a solution for this one maze.... We could put that zombie in *any* maze and he’d be able to follow the path to the solution.” You could see in their eyes when they made the connection that this was more than just **more forward**, **turn left** instructions.

One kid said, “So computers aren’t smart... they are just dumb, really, really fast.”

Andy: So you think they learned something valuable about programming?

David: I watched all these kids model a problem while thinking about it from the bird’s position—they were modeling state—“Where is the bird *now*?” I watched them count the number of blocks their character would have to move, and sometimes forget that turning didn’t actually move one square in that direction (an off-by-one error), then I watched them run their program and correct it based on what happened... that’s debugging.

When a particular student would get frustrated, I’d just reload the page and tell them to “reload their head” too... start over, and I’d explain “there are days I spend debugging a problem only to realize I had a typo in another part of the program. It’s not important that I get it right, it’s only important that I *eventually* get it right.” And I saw this really touch the kids... At times it seemed like I might have been the first adult that shared struggling on a problem with them.

Andy: So what else did you do with the kids besides the exercises?

David: Well, I love an audience, and I wanted to start with a connection to the kids. So I started by showing them a picture of me at 10 years old with my TRS-80 model 1 in the background, and I’d show them a book of computer games I used to type in.

Andy: Wow... how’d that go over?

David: They looked at me with disbelief that I’d spend hours typing a game just to have my older brother beat me in a few minutes. But I explained that I learned to program by typing in those games, because I learned to change them so I could beat my brother. I then told them that they “were going to learn more about programming in the next hour than I knew at their age,” which is true, and also motivated them.

After the exercises I wrapped up by showing them a program I wrote that plays Connect 4. After the computer would easily trounce the entire class playing collectively, I’d open the code and show them some of the *if-then-else* statements they had just seen, as well as the few hundred lines of code that actually plays the game. I showed them it was just a more complex example of what they were just doing.

Finally, I’d show them the cover to your book [Learn to Program with Minecraft Plugins](#)^[U2]... Many of them already played Minecraft, and I thought that might be another hook to keep them interested.

Andy: Hah! Thanks for the plug! How many kids did you do this with?

David: Originally our plan was to do it with eight classes of 4th and 5th graders, with about 25 kids in each class. After I went through the plans with the Principal and the Technology Resource Teacher, we decided to do it with the 3rd graders as well. In total it was about 300 kids.

Andy: So what’s next?

David: Well, I’m in a pretty small town, and I’ve already seen some of the kids out in public. One kid introduced me to his mom in the grocery store, excitedly

proclaiming, “Mom! It’s the computer guy! It’s the computer guy!” I guess there are worse things to be known for....

Seriously though, I volunteer at my kid’s school about once a month, so I’ll see these kids again. I’m hoping to find kids I really motivated, and maybe create an after-school club through the PTA next spring.

Andy: How can other parents get involved like this?

David: Take a look at the [Hour of Code website](#)^[U3], and get familiar with some of the examples. Then contact a principal at your local school, show them the links, and volunteer your time. I bet they will be receptive... Computer Science Education week got a lot of positive press, and I see no reason why we have to limit it to just one week.



About the Authors

David Bock is a Principal Consultant at CodeSherpas, a software engineering services consultancy he founded in 2007. He spends his days writing Ruby, Java, and Clojure, and consulting with teams and managers to help them improve their software development practices. He spends his evenings at technical community user groups in the Washington D.C. Metropolitan region, and his weekends attending and speaking at conferences.

In past technical lives, Mr. Bock has been the editor for O’Reilly’s *OnJava* newsletter, served as a founder and President of the Northern Virginia Java and Ruby user groups, and served on the JCP as a member of JSR 270, defining the contents of the Java 6 platform. He also served as the Technical Director of the Federal Domain at FGM Inc. and was the Chief Architect of the Export Control Program for the U.S. Department of State, Nonproliferation and Disarmament Fund.

You can find Dave on Twitter, LinkedIn, Github, CoderWall, Stack Overflow, and most other social networks as “bokmann”.

Send the author your [feedback](#)^[U4] or discuss the article in the [magazine forum](#)^[U5].

External resources referenced in this article:

- [U1] <http://csedweek.org>
- [U2] <http://pragprog.com/book/ahmine/learn-to-program-with-minecraft-plugins>
- [U3] <http://csedweek.org>
- [U4] <mailto:michael@pragprog.com?subject= Kids>
- [U5] <http://forums.pragprog.com/forums/134>

Giving Back

How You Can Pass on Your Expertise to Kids

by David Bock

Teaching kids to code is one of those powerful levers by which you can move the world. In this latest in our series on teaching kids to code, David shares his experience and invites you to join in.



At the beginning of this year I shared in *PragPub* my story of volunteering at my kids' school, teaching over 300 elementary-age kids the "hour of code" curriculum from Code.org during Computer Science Education Week. Since then, my involvement has increased. I've learned a lot about Computer Science in our school system, both locally and nationally, and learned about several organizations that are striving to influence it for the better.

Last April my school's principal sent me an email about the TEALS (Technology Education And Literacy in Schools) program and its involvement in our county. TEALS is a national program that organizes volunteering professionals from the tech industry and places them in high schools across the country to work with math and computer science teachers, bringing real industry experience to the classroom and driving increased enrollment. She suggested I go to an informational meeting about the program.

At this organizational meeting I met other software engineer parents and heard about their involvement in TEALS, how the program works, and met several kids who had been in classes with TEALS volunteers. TEALS is a grassroots employee-driven program that recruits, mentors, and places high-tech professionals who are passionate about digital literacy and computer-science education into high-school classes as part-time teachers in a team-teaching model. TEALS is about 6 years old, and is still relatively small; it's in about 70 schools with a volunteer network of 280 people.

At this meeting, something clicked for me. For a long time I've been involved in continuing education for software engineering; I've helped to organize user groups and conferences, I've presented in front of countless audiences, written articles, mentored people, and so on. Over the past few years, through my own children, I had been getting involved at the other end of the spectrum, helping kids read, do math problems, etc. But then I thought back to my own high school career...

In 1985 I was a junior in high school. I saw a flyer in the hallway for a computer club that was being organized by Honeywell Federal Information Systems. I joined, and met a small group (maybe 10 other high schoolers), and several adults.

We had no agenda. We listened to the adults tell war stories. We occasionally wrote code. I remember planning out on a whiteboard how we'd organize data structure for a chess playing program, and we talked about nerdy stuff. This is where I learned about the book *Gödel, Escher, Bach*, which was a major influence on my world view at that age. One day I remember they took us into the data center to show us a hard drive — a washing-machine-sized metal box — that had broken and flung one of the platters out so hard that it flew 15

feet like a frisbee and lodged itself into a wall. “It’s a good thing no one was standing here...,” I remember one guy saying.

Perhaps the most important thing I learned there was that *being a nerd* wasn’t a high school affliction ... I was ahead of the pack! I saw what a real, functional office environment looked like, and I saw these things that I had an interest in were going to be valuable out in the real world.

Suddenly I realized... this... THIS is where I can give back.

I applied on their website as a volunteer that day. The entire application took about 10 minutes, asked a few questions, and asked me to upload a recent résumé.

A few months later, in the middle of the summer, interviews were scheduled. I spent half an hour being interviewed by a panel of 5 people, including the teachers at both my local high schools. They had really done their homework preparing for the interview; I have to say. This made me feel that these teachers really take their jobs with these kids seriously.

A few weeks after that interview, I was sent my school assignment and found myself on an email list with my assigned teacher and several other TAs. We started conversations about what the classes and students are like, and other anecdotal prep work.

Last weekend I spent a Saturday in “training” with a few teachers, a few past TAs, and a representative from the TEALS program. We worked through several real classroom examples, familiarized ourselves with the tools and libraries used in the classroom, and role-played as student and TA, teaching each other the Socratic Method of teaching with leading questions so that the students get their own “aha!” moment, rather than just being told an answer. We also listened to a presentation by a representative from NCWIT, the National Center for Women & Information Technology, about the Aspirations program for school-age girls interested in technology.

Today I met with the teacher for a one-on-one. We talked about the several classes, the students, the computer club, and even about some of the extracurricular activities like coding competitions. I’m really excited about the stuff we have planned, and I hope in a few months I’ll be writing about a trophy or two the kids have earned!

I have one more appointment before I actually start in the classroom; I have to get fingerprinted and badged by the school system. As a regular visitor to the school, there will be a background check. Considering my kids are in this school system, I’m very supportive of that!

This year I’m planning to repeat my “hour of code” with my elementary school, but also this year I hope to spread it further throughout my county’s school system. The high school teacher and I discussed using the students from the Computer Club as volunteers to go to *other* elementary schools throughout the county and support teaching more 5th graders with the material from Code.org.

Visit www.TEALSk12.org [U1] to learn more. Volunteer at www.TEALSk12.org/apply [U2]. Visit www.ncwit.org/ [U3] for more information

on the National Center for Women & Information Technology, and www.aspirations.org^[U4] for the NCWIT Aspirations program.



About the Author

David Bock is a Principal Consultant at CodeSherpas, a software engineering services consultancy he founded in 2007. He spends his days writing Ruby, Java, and Clojure, and consulting with teams and managers to help them improve their software development practices. He spends his evenings at technical community user groups in the Washington D.C. Metropolitan region, and his weekends attending and speaking at conferences.

In past technical lives, Mr. Bock has been the editor for O'Reilly's *OnJava* newsletter, served as a founder and President of the Northern Virginia Java and Ruby user groups, and served on the JCP as a member of JSR 270, defining the contents of the Java 6 platform. He also served as the Technical Director of the Federal Domain at FGM Inc. and was the Chief Architect of the Export Control Program for the U.S. Department of State, Nonproliferation and Disarmament Fund.

You can find Dave on Twitter, LinkedIn, Github, CoderWall, Stack Overflow, and most other social networks as "bokmann".

External resources referenced in this article:

- [U1] <http://www.TEALSk12.org>
- [U2] <http://www.TEALSk12.org/apply>
- [U3] <http://www.ncwit.org/>
- [U4] <http://www.aspirations.org/>

Constraints and Freedom

"29638 Bytes Free"—What a Gift!

by Jimmy Thrasher

Freedom blossoms under constraints. So does that mean you should give your kid an obsolete computer to learn on? Maybe.

Do you agree that constraints are liberating? What's your experience? And how important is this to teaching kids to code? Weigh in at [the forum](#) [U1].



I gave my 10-year-old son a computer that was released in 1983. One of the first things you see when loading it is: "Copr. 1983 Microsoft," and soon after you see "29638 Bytes free." He tells me it's the best Christmas present he's ever received.

This may strike you as counterintuitive.

But those of us who were privileged enough to have similar devices growing up—Commodore 64, Atari 800, and the like—can completely relate.

What is it about these devices that encourages the kind of creativity that leads to hours of children hacking away at nothing in particular for the sheer joy of it? I have an idea about that, and it goes like this:

Constraints enable freedom and creativity.

The image I like to conjure up is that of a fence on a cliff. When you're on a cliff without a fence, you end up building the risk into your motions, keeping away from the edge, moving slowly, etc. If there's a good, solid fence on the cliff, you end up moving with more open movements, peering freely over the side, enjoying yourself. The constraint of the fence enables free movement.

The problem with constraints is that it's typically very difficult to impose them on yourself. If, for instance, you have a full-fledged computer with Internet access in front of you, you're a lot more likely to spend time getting distracted. At the other extreme, having a typewriter in front of you *can* be a way to stimulate creativity. You have nothing but your thoughts, whatever they're worth.

Children Are Not Grown-ups

Now, to children. For the purposes of my argument, the primary difference between grown-ups and children is that of competence, both self-perceived (how amazing do you think you are?) and actual (how amazing are you?). Of these, the most important is the self-perception, because it has a profound impact on the value of one's time.

Very young children are incompetent, in all the *best* senses of the word. They spend a large amount of time building up the basic skills necessary to live in the culture they've been dropped into and make mistakes as a matter of course. As their job, in fact. Toddlers speak "incorrectly" and don't care a whit what others think about them and, as a result, can zoom through the typically laborious process of learning language. Making mistakes is one of the best ways to learn, and they are much better at it than the typical grown-up.

Grown-ups, in contrast, tend to think of themselves as competent, and tend to limit themselves to those activities in which they think they are competent. Slowly but surely, they lose the ability to make mistakes, thus losing the ability to learn anything outside the well-trodden paths in any way approximating the joyful, slapdash energy of children.

Grown-ups also have a leaning toward pragmatism that dismisses large swaths of “useless” knowledge or projects from the playing field—thus triggering the terribly misguided question: “When will I ever use this?”

So, back to constraints and children. We have a group of people who don’t particularly care what they’re learning, so long as it’s fun, and we have the dynamic of constraints. What should we expect to result from such a combination?

A Study in Constraints

Lacking a database of research findings and thus at the risk of generalizing too hastily, let me tell you more about my son’s situation. I gave him a [TRS-80 Model 100](#) [U2] for Christmas. This machine is a study in constraints.

How do those constraints improve the opportunity for creativity? In short, they limit choice. Studies have shown that typically people move more slowly and are less happy with their decisions when faced with too many choices. Let’s take a look at a few specifics.

Speed and Size Constraints

The Model 100 runs a 2.4MHz Intel 80C85 processor and has 32K of RAM available to the user. No static storage to speak of built in.

These constraints have some benefits. For one, the scope of the projects my son takes on are limited by design. Rather than being faced with a world of possibilities he’s stuck with only solving problems that can be solved in a few kB by a processor that does relatively few things per second. No ray tracers or procedurally generated games for my son.

Display Constraints

The Model 100 has a monochrome 240x64 pixel LCD. It works out to about 31 pixels per inch (Apple’s Retina displays run at 326ppi). My son won’t be doing any fancy graphics here, so he’s limited to programs that make sense in a very low graphics context. If he were to write an Etch-a-Sketch program for the Model 100, it would feel like an accomplishment, whereas if he were to do so on a higher-res display it might feel like a waste of time.

Audio Constraints

The Model 100 has a single piezoelectric speaker. Any sound being made is typically of the square wave variety. My son is not going to waste his time recording himself saying silly things and listening to the output (which is what he did when he played with Scratch).

It is enough to make music, siren sounds, beep-boop level “future noise,” and the like. And he loves it.

Transfer Constraints

The Model 100 has an RS-232C port, a built in 300-baud modem, and a cassette tape output port. In order to have any reasonable backup strategy, I had to dredge up a null modem, serial cable, buy a USB serial port, and on and on. (Any of you who remember doing this in the past may be having flashbacks. And yes, I had to break out the multitester to see if the cable I had was already a null modem or not.)

This means it's such a pain to back things up, that my son ends up getting a lot better at text entry and spitting out code.

Programming Language Constraints

Like any good early-80s personal computer, the Model 100 boots to a BASIC prompt. Copyright 1983, Microsoft (this is the last software Bill Gates has in production, incidentally). For the programmers out there, BASIC is a horrible programming language. Horrible. But so easy to write that it's the butt of many lazy jokes:

```
10 PRINT "LAME JOKE"  
20 GOTO 10
```

It may be a horrible programming language, but it's accessible. My son is one line away from emitting a horrible shriek. He's one line away from inverting all the text on the display. He's two lines away from lazy jokes like the one above.

It's also arcane. Interestingly, I think this is a plus. Most mature programmers realize their humanity and are humble enough to realize they must write simple, clear, straightforward code in order to have any hope of maintaining the software. I would submit this dynamic alone has stymied some of the efforts to get kids into programming.

At the heart of this is the failure to distinguish between the mere acquisition or exposure to an art, and the mastery of the art. Or the apprentice phase and the journeyman/master phase. Or, classically, the grammar and the rhetoric phase.

Think of it this way: when attempting to kindle a small child's interest in carpentry, you give her a saw, some scrap wood, some nails, and a hammer. Then you let her go. You don't spend hours lecturing on the elegance and strength of a good drawer joint, or even "measure twice, cut once."

Or think of it this way: when attempting to kindle a small child's interest in writing, you give him a pencil and paper and let him have at it. Then you read his results and praise the effort.

In both of these cases, the child is given the opportunity to make tons of mistakes and learn from them. This is what children do best.

In fact, not only is the general inelegance an asset in BASIC's case, but so is its arcanity. From a kid's perspective, nothing is as fun as having magic incantations that actually do something. How's this:

```
10 REM Play Beethoven's Fifth  
20 SOUND 5272, 12.5  
30 SOUND 5272, 12.5  
40 SOUND 5272, 12.5  
50 SOUND 6642, 50
```

Or this:

```
10 REM Disable the break key
20 POKE 128, 63056
```

Or this:

```
10 REM Load a BASIC file from serial
20 LOAD "COM:58N1e"
```

My son will eventually outgrow the constraints, and he will (hopefully) learn to appreciate elegance and simplicity, but for now, this is what's exciting him, and I think his case is probably not unique.

So what can we conclude from these observations?

First, I think, we need to maintain a distinction between the mastery of an art and the introduction to the art. This means that the languages a child learns may not be applicable in a professional context, and this is OK.

Second, we should embrace constrained environments, not only in programming, but also in any learning the child is doing.

Oh, and one thing more.

During his reign as Interesting Ruby Programmer, Why, the Lucky Stiff (A.K.A. _why) wrote an essay entitled "The Little Coder's Predicament." His primary focus is on the accessibility of the computing environment, rather than on constraints, but the two relate pretty closely. Typically, the more powerful an environment, the more complex it is to use. More specifically, the more complex it is to *start* using.

_why's thoughts led him to build Hackety Hack, a simple, constrained programming environment build to run on a typical PC. My thoughts lead me in a different direction.

I envision a device, built off an easy-to-buy, easy-to-use embedded environment like Raspberry Pi, combined with some sort of small, low-res LCD, a simple keyboard interface. I.e., I envision a modernized Model 100, updated to the point of maintainability, but not to the point of being wide open like a PC.

Let's create an intentionally constrained computing device. To grant our children the freedom to learn. And maybe to free the child in us.



About the Author

Jimmy Thrasher has been programming since he was a wee lad and has never tired of it. He loves to learn most anything, and has to fight the constant urge to acquire new hobbies. He is currently consulting in iOS and Ruby on Rails and serves on the board of a small private school in Mebane (meb'bin), NC with his wife and three children.

External resources referenced in this article:

[U1] <http://forums.pragprog.com/forums/134/topics/12543>

[U2] https://en.wikipedia.org/wiki/Model_100

Lego League: Lessons Learned

The Code Is not the Most Important Thing

by Seb Rose

Choose the right sort of problem, focus on strategies for solving the problem, and the code will come easily.



Some approaches to teaching kids to code work better than others.

I want to tell you a story. Two stories, actually.

A few years ago I found myself in one of those huge, soulless, air-conditioned conference halls, listening to one of those keynote speeches on a subject that had no discernible relevance to my work. But despite my preconceptions, I found myself surprisingly moved by the presentation. The speaker was inventor Dean Kamen, and he was talking about something called First Lego League.

I was inspired. Returning to the UK, I chased down our local FLL and registered. I coerced my children, aged ten and twelve, into participating. I put notices around the local junior schools, gathering another eight local offspring. I booked a room in a local hall one night a week for the rest of the year, bought the lumber needed to build the competition table, and waited for the kit to arrive.

The kit, right. I see I'm getting ahead of myself. I'd better back up and explain a bit about the FLL.

The First Lego League

The [First Lego League](#) [U1] is an annual, international competition for teams of 9-to 16-year-olds that involves programming Lego MindStorms to solve challenges, doing a piece of original research, and presenting the results of that research. You can read more about it at the website. It's interesting reading, really.

Anyway, about the kit. Each year FLL chooses a theme of global significance (in 2010 it was Biomedical Engineering) and devises a number of robotic challenges around it. And they send you this kit to get you started. The competition board is 8' x 4' and they provide a roll-out map to use as the base for your constructions, as well as a box of Lego bits and pages of instructions about how to put together the Lego challenges. Just building all of the challenges took our group 2 whole sessions.

Once the challenges are assembled and laid out on the board, it's time to start designing and programming your robot. The robot construction rules are strict—only Lego-manufactured material allowed, no more than 4 motors, no more than 1 of each sensor, no non-standard software environments—and you only get 2-1/2 minutes to complete as many challenges as you can. Whenever the robot returns to base, a small area at one corner of the board, you're allowed to touch it, modify its attachments, start a different program. At all other times it must be completely autonomous (NOT remote controlled) and any interference with it leads to penalty points.

So there I was, with ten 9- to 12-year-olds with no engineering or programming experience. Mercifully, the role of coach in FLL, they explain, is not to teach, but to enable. Good. So I printed out the challenges, loaded the Lego programming environment onto some school laptops, and asked them to pick a few challenges to have a go at. There were 17 to choose from, and no way to accomplish all of them in the short time available.

There *was* teaching involved, but once they had selected the challenges, it was pretty straightforward. We started with the engineering, because that was a more tractable problem, and you need to have an idea of the robot's physical characteristics before trying to control it with software. Then came the challenge of introducing them to software. The Lego programming environment is fully graphical—you drag action, sensor, conditional, and branching “blocks” around, linking them together to achieve the required behavior. We started with simple, dead-reckoning approaches—“turn both wheels for 8 revolutions at 5 revolutions per minute,” “turn left 90 degrees,” “lift arm 45 degrees”—and just that is enough to get a lot of the challenges done.

What the team noticed, though, was that this style of control wasn't very robust. It was susceptible to small variations in the robot's initial position. The state of the batteries affected the distance the robot travelled. Even with identical starting conditions, the end position of the robot could differ significantly. This is when I introduced them to the sensors.

The most useful sensor was the light sensor. They used this to detect color changes on the map and trigger a change in behavior of the robot. Different groups of children would work on different parts of the robot's route, and these would then be joined together by “blocks” switched by the sensors. This was particularly effective for the “pill dispenser” challenge, where the robot needed to push a panel to dispense 2 Lego “pills,” but leave the third “pill” in the dispenser. There were handy black markings on the map that could be used to count how far the robot had traveled, and hence how many “pills” had been dispensed.

Another useful sensor was the push button, which was useful to ensure the robot kept the right distance from obstacles. We never found a use for the ultrasonic sensor—it was just too unreliable. And the microphone would probably have been thought of as a remote control, so we never even tried that.

What interested me about the whole experience was that we rarely talked about programming. The team was always trying to solve a problem—sometimes the solution was a new attachment for the robot, sometimes it was a new behavior. They quickly picked up control and conditionals. Looping was harder, and I don't think they ever really got there—they were much happier copying and pasting sections of the program to get the number of iterations they wanted.

Second Story

The local group lost its funding the next year and FLL in Scotland lapsed until this year, when a local company, [Lamda Jam](#) ^[U2] brought it back. This year, as part of the local [British Computer Society](#) ^[U3] branch I was asked to judge the robot competition. There were 7 schools competing—two high schools and five junior schools.

One difference in this competition was that, although there was a lot of inventive engineering, there was no use of sensors at all. Even the best teams were controlling their robots using dead reckoning alone. One of the side effects of this was that scores for the same team, using the same programs, varied wildly. Each team played 3 heats, and the winning team's scores varied between 125 and 348.

So what did I learn from these two experiences in teaching kids to code?

Having seen FLL from both the participant and judging sides I'm confident that the FLL can be an excellent vehicle for getting children to learn to code. Because there are so many different angles to the challenge, it's easy for the team to do a little bit of everything without getting overwhelmed by any one aspect. The programming part of the challenge does introduce them to simple constructs, and with the right coach this can go a lot further. I can easily see a team building on its experience over a number of years, with some of the members eventually getting to be quite sophisticated practitioners.

But the counterintuitive insight was that it's not so much about the code.

A comment in a recent thread on the [accu-general](#) [U4] mailing list with the subject line "Programming for teenagers" captured nicely what I learned from my experiences:

"I think that finding an interesting and reasonable-sized project that can be expanded upon is more important than the choice of tools and environment. That is the hook to keep her interested and search for ways to learn more, and I also think the very first visible result must be reached quickly."

A good project, room to grow, and getting visible results quickly. Those are key. And one more thing: a good mentor. That, of course, would be you.

I encourage anyone with even the slightest interest in introducing children to programming to take a look at what FLL has to offer. You may end up doing more Lego than programming, but you'll be giving the children exactly the right role model—us. Because we're people who solve problems and get results quickly.



About the Author

Seb wrote his first commercial software in the early eighties on an Apple II. He went on to graduate from the University of Edinburgh with a 1st Class Joint Honours in Computer Science and Electronics in 1987. Since then he has had a varied career working with companies, both large and small, in roles that cover the complete technical spectrum.

Over the past 6 years, Seb has focused on helping teams adopt and refine their agile practices. He wrote internal training courses for IBM's Quality Software Engineering department (QSE) and went on to develop [his own courses](#) [U5] that he runs for clients throughout Europe. He speaks regularly at international conferences, specializing in topics such as Unit Testing, Test Driven Development (TDD), Behaviour Driven Development (BDD) and Acceptance Test Driven Development (ATDD). He is a popular blogger and a regular contributor to technical journals, and is a joint author of the forthcoming *The Cucumber-JVM Book* from the Pragmatic Bookshelf.

Send the author your [feedback](#) [U6] or discuss the article in the [magazine forum](#) [U7].

External resources referenced in this article:

- [U1] <http://www.firstlegoleague.org>
- [U2] <http://lambdajam.org>
- [U3] <http://bcs.org>
- [U4] <http://accu.org/mailman/listinfo/accu-general>

[U5] <http://claysnow.co.uk/training>
[U6] [mailto:michael@pragprog.com?subject=kids coding](mailto:michael@pragprog.com?subject=kids%20coding)
[U7] <http://forums.pragprog.com/forums/134>

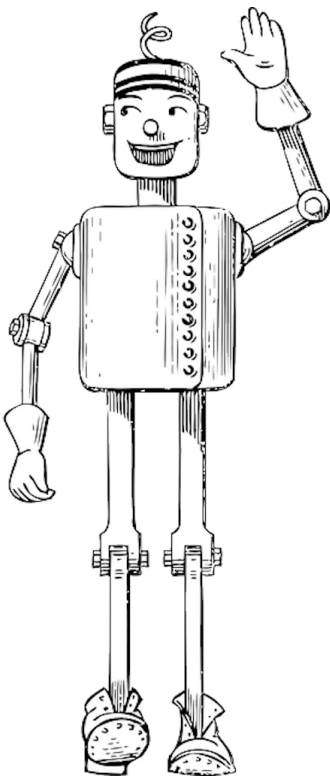
Coding Unplugged

A Case Study in Teaching Kids to Code

by Fahmida Y. Rashid

When you're first learning to code, a programming language can just get in the way.

Have you taught a child to code? What's your experience? Weigh in at [the forum](#) [U1].



As programmers ourselves, we were already on the “Let’s teach our kids to code!” bandwagon. The trick was finding ways to make programming fun for younger kids.

Our kids were just five and eight years old. But most programming initiatives that we were aware of targeted kids in the 9–16-year-old range. So we needed to get creative. What we found was that combining offline approaches with a graphical coding language worked best for this particular age range.

The older one likes to read and is curious about how things work. We probably could have handed him a copy of [3D Game Programming for Kids](#) [U2] and he would have gotten right to work on the examples. He wouldn’t have gotten far on his own, but at least he would have tried. The younger one, though, was just beginning to read, and while he was game to try anything his brother did, there was no way we could start going over if statements and variables. At least not right away.

The Computer Science Education Week’s (CSEW) initiative “[Hour of Code](#)” [U3] in December 2013 was a good start, but most of the focus was on computer literacy, particularly the ability to read and write code.

However, Code.org did have a few suggestions on “unplugged” coding, where kids learned the relationship between symbols and actions.

That drove our first step. We stepped away from the computer and introduced the elements of coding without getting tripped up over the actual language—we started with programming concepts. All we needed was a bunch of small objects, such as keys, pens—a snack, even.

The kids were perplexed.

“We need to use the computer!” they complained.

We made a game of it. We designated one child the “robot” and the other child the “programmer.” The programmer had to tell the robot what to do, such as picking up the keys from the floor, sitting on a chair, and opening the toy chest to pull out a toy.

After a few false starts, they figured out that you couldn’t just say “go straight,” but that you had to say “go straight 10 steps.” We talked about defining objects so that the robot knew what a chair was, giving commands in the right order, and how to fix mistakes in the instructions. When they started getting really elaborate with the tasks for the “robot,” we talked about chaining several commands into a single verb.

Just like that we were talking about if statements, functions, and debugging, while giggling at some of the silly tasks we came up with.

The next step was to channel the robot concepts into something a bit more structured, something that was a little closer to a programming language and less like spoken English. Cue Robot Turtles.

The board game [Robot Turtles](#) [U4] was originally launched on [Kickstarter](#) [U5] and is being released this summer from Thinkfun, a publisher of educational games.

The game, inspired by the Logo programming language, has kids use a deck of cards to instruct their turtle tokens to move across the board with the goal of collecting a jewel. The cards themselves are straightforward, with pictures for rotating the turtle left or right, going forward and backward. If you remember Logo from your own childhood, all this is familiar. The kids picked up the game mechanics easily, thanks to the “robot-and-programmer” game and the cute pictures on the cards. The game can become more complex by building obstacles the turtles have to knock down or break with a laser gun. It’s a lot of fun, and the best part is when the kids pull out the game on their own to play.

We were pretty confident in their grasp of basic programming concepts at this point, so we wanted to move back to the computer, to something closer to traditional programming. But we also wanted to make sure that we weren’t falling in the trap of coding just for the sake of learning to program. We wanted to take a “let’s make something” mindset.

Because learning to program wasn’t the end goal. Making something cool the kids could point to and say, “Look what I made!”—that was. And we wanted to make room to learn through trial and error.

We had that ourselves, as kids, when we spent hours writing simple programs in BASIC on the Commodore 64. There was something really absorbing about a screen full of “Hello, World!” printed in different colors. We had just learned to read, and we already knew PRINT, GOTO, and RUN.

So how were we to replicate that experience? Short of scouring eBay and buying a Commodore 64, how could we replicate some of the simple joy and fun in creating for these kids?

That’s when we discovered [Scratch](#) [U6].

Designed by MIT students and staff in 2003, Scratch is a web-based application that lets kids create games and animations using a graphical coding language. Each programming element is made up of colored bricks that they can drag from the tool palette into their workspace. Like Legos, each brick snaps together and slowly builds a program to animate the sprite on the screen. The feedback is immediate. You drop a “play sound” brick into the workspace, and the cat sprite on the screen makes a sound. Change the sound from a meow to a woof, and you now have a cat sounding like a dog. You snap a “move” brick inside a “forever” brick and that same cat moves across the screen in an infinite loop.

The game was originally designed for kids ages 8 to 16, but it didn’t take the five-year-old (who at this point had experience with the *logic* of if statements, if not the syntax) long to figure out how to make animations. We helped with reading and identifying the brick names at first, but after a few tries, it was easy to remember which colored brick represented which programming element.

The older one quickly branched out into making actual games. The site has a gallery where you can share your games and see what other people have done, making programming a bit more social.

It's really easy to get hooked. There is a lot of grumbling, "No, that's not what I wanted!" when the programs don't quite do what they want, but as of yet, no one has given up. Even the mistakes are fun to make.

That's an important lesson to learn: programming isn't always about the end result, but the steps taken to reach that goal. This means we, as parents, have had to learn to not "help" too much. We had to consciously make the decision not to offer suggestions, to point out which method would be better. We are learning to ask open-ended questions like "What does this do?" and "What do you want to do next?" and let the kids figure out what to try.

At the moment, we are happy with Scratch, although I am sure at some point we will start looking at CoffeeScript, Python, or even Ruby to teach "real" programming. We have big plans, like getting a Raspberry Pi or Arduino. We want to some day make our way over to the [HacKid conference](#)^[U7] in San Jose and [r00tz ASYLUM](#)^[U8] (formerly known as DefCon Kids) in Las Vegas. But there is plenty of time for that. For now, the fact that the kids are enjoying the process is enough.

"Look what I made!"

That's what we want to hear.



About the Author

Fahmida Y. Rashid wears many hats. A former developer, management consultant, network administrator, and product manager, she now spends most of her time writing. When she isn't editing books for Pragmatic Bookshelf, she is writing about information technology (usually security, but also networking), travel, or business. Her articles have appeared in *Forbes*, *CRN*, *eWEEK*, *PCMag*, *SecurityWeek*, and *Dark Reading*. In her spare time, she likes to read, travel, and study languages—both spoken and programming.

External resources referenced in this article:

- [U1] <http://forums.pragprog.com/forums/134/topics/12544>
- [U2] <http://pragprog.com/book/csjava/3d-game-programming-for-kids>
- [U3] <http://code.org>
- [U4] <http://www.thinkfun.com/robotturtles/>
- [U5] <https://www.kickstarter.com>
- [U6] <http://scratch.mit.edu/>
- [U7] <http://www.hackid.org/content/>
- [U8] <http://www.r00tz.org>

The Selfish Teacher

Teach a Kid to Code—for Your Own Good

by Michael Swaine

How teaching a kid to code could be a wonderful thing to do—for yourself.

Think the author is an idiot? Weigh in at [the forum](#) [U1].



If there are children in your life, you need to teach them to code.

It doesn't matter if they're your kids or your significant other's kids or your grandkids or your younger sibs or your nieces and nephews or the kids hanging out on your lawn. If they're in your life, you need to teach them to code.

It doesn't matter if they already know something about programming, or even if they know a lot about programming. They don't know what you know. You need to teach them that.

You need to do it for yourself, not for them. And not just kids. You need to teach everyone in your life to code. At least everyone who will stand for it. They may not enjoy it, any more than they enjoy it when you make your obscure jokes or quote extensively from science fiction novels they've never read, or any of your other annoying habits. But their comfort is not the point. You aren't doing it for them, you're doing it to make your life more awesome.

Programming is an isolating, lonely job. You dive deep into a place the people in your life know nothing about, are totally blind to, and when you come up for air and they ask you how your day was, you have no answer that they would understand. Your co-workers are different, but the people in that other part of your life that has nothing to do with programming, between you and them there is a wall.

Well, the problem is that you shouldn't have two disparate parts to your life. As Nick Floyd [eloquently explains](#) [U2] at the New Relic blog, trying to achieve some kind of work/life balance is a fool's game. What you want is work/life awesome.

"[S]omehow," he says, "we are convinced that we can't have job that comes from what we do in life." But that's wrong. The key insight, he says, is "[i]f you find something that you are genuinely excited about and you are fortunate enough to do it as a 'job,' how awesome would that be to share that passion with your family and friends?"

You only have one life. Anything that places arbitrary barriers between aspects of your life diminishes you. Anything that puts barriers between you and the people in your life makes your life more sterile. Anything that prevents you from sharing something that means a lot to you with the people who mean a lot to you weakens those relationships and means that you will get less from them.

But sharing your passion for coding with the people in your life helps them to see you more deeply. So you need to teach the kids in your life to code. You need to share what you do. And you need to do it for yourself. It will make you a happier and a richer person.

Resources

Tools for Teaching Kids to Code

As the issue grows, this list will grow.



Some resources you may find useful:

- [The Hour of Code](#) [U1]
- [The Hour of Code 2013](#) [U2]
- [First Lego League](#) [U3]
- [Lamda Jam](#) [U4]
- [British Computer Society](#) [U5]
- [Learn to Program with Minecraft Plugins](#) [U6]
- [3D Game Programming for Kids: Create Interactive Worlds with JavaScript](#) [U7]
- [Practical Programming \(2nd edition\): An Introduction to Computer Science Using Python 3](#) [U8]
- [Learn to Program \(2nd edition\)](#) [U9]