MASHED CODE MAGAZINE

June 2012 Issue

FEATURES

PLUS! Interview with David McKinnon & Local User Group Listings

If you've been following the magazine, you know that the first issues were published in conjunction with the CodeMash conference. Many of you asked for better than a once-per-year publishing schedule; we agreed and this is the first response—the first stand-alone issue of *Mashed Code Magazine*.

With this issue we bring change. We are now focused on the talented and burgeoning software development communities throughout Ohio and Michigan. We'll demonstrate the acumen of local software developers—that's you, your colleagues, your conference friends and user group buddies—by featuring articles from some of them each issue. In this issue you'll see what two Ohio technologists offer with Carin Meier's *The Further Adventures of Princess Chloe in Clojureland* (featuring original illustrations by the talented, and local, Sarah Allgire) and Guy Royse's *The Prototypal Wizard*. In future issues, we hope to share the code and technologies *you* are working on too.

In every issue we'll also point out the people who are working hard to enable your development prowess. Each issue will also feature a piece about a local organization, like Detroit DevDays this time, that is positively contributing to our readers. If you are running a similar organization that organizes events for developers or you run an active user group, please contact us to be featured in a future issue.

The technology being used in Ohio and Michigan isn't always the same as what developers are using around the country. So for a broader scope, we'll also bring in authors of national acclaim each issue. This issue we have Seth Juarez's *C#, Math and You* and Peter Bell's *Data Modeling with MongoDB*.

I sincerely hope you enjoy the magazine. It's something we toil on only because we are, ourselves, part of the local software development community and care to help keep it so special. In fact, to maximize the impact, we are dedicated to always providing this magazine free to all. There is no cost for electronic copies of the magazine and all articles and images in the magazine are published under a Creative Commons license.

This is the first iteration of the magazine, which means we have a minimum viable product (thanks Peter) that we will improve with every new issue. If you have any suggestions for improvement or enhancement, please email them to mashedcodemag@gmail.com, comment on our blog at www.mashedcodemagazine.com or tweet to us @mashedcodemag. Thanks for reading.

Nick Watts
Creator & Curator

Table of Contents

Magazine Staff
Nick Watts
Creator & Curator

Sara Smith
Graphic Designer
Special thanks to
Jason Gurik

Visit mashedcodemagazine.com for information on advertising with us, submitting articles and future issues.

A New Era of Developer Days for Detroit—Interview with David McKinnon

By Onorio Catenacci

If you are a member of southeast Michigan's software development community, you know, or know of David McKinnon. You've probably seen him rushing around, always in the background, at one of the many events he produces. Curious, I decided to interview him and bring David and DetroitDevDays, his latest project, to the forefront.

David McKinnon

Born and raised on Detroit's east side, David has worked as a programmer and software developer in Michigan for twenty years. Last year, after twelve years in the corporate world at Compuware, he left to start anew as an Android app Developer for local start-up Detroit Labs.

In 2008, he took the helm of the then moribund Detroit Java User Group. Within that first year the group's membership grew 300 percent. It was while organizing the Detroit JUG that his passion for building and empowering the Detroit area's software developer community was born.

For the past year he has been focusing on a project called DetroitDevDays. In that capacity David has produced a full roster of events including MobiDevDay—a developer conference focused on mobile software development, CloudDevDay—focused on Cloud technologies, CloudCamp Detroit and, for three years running, he has produced the highly successful polyglot developer conference 1DevDay Detroit.

Onorio: So, in 50 words or less, tell me what Detroit Dev Days is?

David: DetroitDevDays produces conferences, workshops, boot camps and events for the software developer community. The mission has always been to grow, empower, educate, organize, and champion the Detroit area's software developer community. I've always said our goal "is to build a software developer community in the Detroit area that is regarded as the best in the world." This tends to raise a few eyebrows but, if you are going to climb a mountain, you set your sights on the summit.

Onorio: When you think about building a community of software developers, what are the biggest challenges you feel that we face?

David: Something I've been thinking a lot about are the large number of developers and programmers that for one reason or another become 9 to 5-ers. I speak to so many devs that have a ton of responsibilities outside of work. The last thing on their mind is going to a monthly user group meeting or to a conference or paying to take a class. Any passion they may have once had for their craft, now their job, is long gone. I've been trying to think of ways to draw these devs back into the community. Many have professional and life experience that can only benefit our community.

I've also had more than one woman say to me, "I would go to this or that user group meeting, but I would be the only woman there". We need to find a way to make this less of a male oriented profession and community. Again, this can only help build a stronger community.

The Detroit area faces its own challenges. Far too many software developers left when the economy tanked. My dad used to say, "When the country sneezes, Detroit gets a cold." Well the country caught pneumonia.

We need to figure out how to get developers to move to our area. I think things like the Valley to Detroit movement may help. I hope the Michigan Software Development Summit will help bring some attention to it as well.

Onorio: You said "Any passion they may have once had for their craft, now their job, is long gone." How do we help people to find that passion again when it is gone?

David: I think that is where conferences and events can help a lot. Craig Maloney of the Lococast podcast wrote a nice post about the 2011 1DevDay Detroit developer conference. It was titled "1DevDay Detroit: The cure for the

For a schedule of events and other information, visit http://detroitdevdays.com

9-5 developer". A conference like 1DevDay Detroit or the conference you are producing, the Great Lakes Functional Programming Conference, can recharge a developer's batteries. Bring developers together, let them share knowledge and explore together and the passion will return.

It is getting potential attendees to see that and then trying to make the event as accessible and affordable as possible, that can be a real problem. I've compounded my own problem, I'm afraid, by challenging the dev community to venture to downtown Detroit for events. Too many people hear they will get mugged or something like that if they come down here. Once they get here however, it is all good.

Onorio: You recently announced plans for a multi-day software developer event. Is that the Michigan Software Development Summit? Can you share any additional details about this and other upcoming events? What can we look forward to?

David: The plan for DetroitDevDays is to have several smaller (about 150 people max) conferences that will be more focused on learning and also networking events, throughout the year. I want to have workshops or boot camps on specific topics throughout the year as well, but again at most 100 people. This year's MobiDevDay Detroit is coming up on May 19th and our first all day workshop, Scala Koans with Dianne Marsh and Bruce Eckel will be on May 31st.

With The Michigan Software Development Summit, I want to produce an event that will bring together developers, leaders and businesses to talk about growing the Software Development Industry in Michigan. I'd like to incorporate the 1DevDay Detroit developer conference into this event as well.

If it's done right, I think we can attract attendees not just from Michigan but from across the Midwest and possibly from across the United States.

Onorio: Which victories in the process of helping to build a community of software developers in Southeast Michigan are you most proud of?

David: It seems like a year or so ago a bit of a boom started in regards to user groups forming and several local or regional conferences being organized. I like to think some of this can be attributed to the success of the Detroit Java User Group and DevDays.

I hope I've helped to change some of the perceptions about Detroit and our tech community. Not just the perceptions of folks outside of Michigan but the perceptions of Michiganders as well. I'm afraid [that] years of urban mythology and negative media coverage is a hard thing to combat.

Personally, I'm most proud when I bump into someone at a restaurant or on the street and they say "hey, aren't you the guy that puts on all those dev days?" When I acknowledge it, they always say "Thank you for doing all that." I think I'm most proud of that.

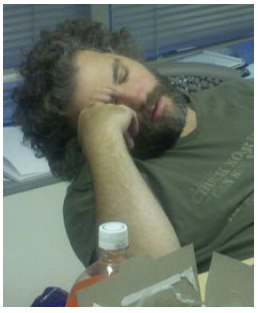
Onorio: Do you feel like you're making steady progress toward the top of the mountain? Is the climb getting any easier as you learn more about producing conferences and events?

David: When it comes to organizing events or conferences, I think I've got that down.
Will Detroit be viewed as a great software development hub? I think we have a ways to go still, but we will get there. Believe me, we will get there.

Onorio is also the founder and organizer of the Great Lakes Functional Programmer Conference. Ask Onorio about his Capuchines at your own risk. http://glfpc.org/

Local Authors

Carin Meier started off as a professional ballet dancer, studied Physics in college, and has been developing software for both the enterprise and as an entrepreneur for the past 15 years. She comes from a solid Java background but has discovered a passion for the power and elegance of the dynamic languages of Ruby and Clojure. She brings fun, enthusiasm, and hot tea to all her projects and especially enjoys participating in the developer and Open Source communities.



Guy Royse works for Pillar Technology in Columbus, Ohio as an agile coach and software engineer. He has programmed in numerous languages—many of them semi-colon delimited—but has more recently been working with Ruby and JavaScript. He is also the chief organizer for the Columbus JavaScript User Group and is active in the local development community.

In his personal life, Guy is a hard-boiled geek interested in role-playing games, science fiction, and technology. He also has a slightly less geeky interest in history and linguistics. In his spare time he volunteers as Cubmaster for his kids' Cub Scout Pack.

Onorio Catenacci is a software developer and currently leads the FSharp Special Interest Group in conjunction with the Michigan Great Lakes Area .Net User Group. Onorio has been developing software professionally since 1987 in several different languages and a few different paradigms. One of Onorio's most memorable jobs was supporting an application written in the Excel 4.0 macro language: "It was like trying to write War and Peace with a fifth grade vocabulary." Ask Onorio about his Capuchines at your own risk.

Sarah Allgire is a graphic designer and illustrator originally from Bucyrus, Ohio. She graduated from the Columbus College of Art and Design in 2009 with a bachelor's degree in Illustration and a minor in Writing. She currently lives in Columbus, Ohio with her husband Brad, 9-month old son Rome and Hank the cat. www.sarahallgire.com

The Prototypal Wizard

By Guy Royse

Like many a programmer, I spent much of my pasty youth indoors engaged in geekery of the highest order. I'm talking about Dungeons & Dragons. As a young man who was learning to program—arguably also geekery of the highest order—I had this persistent fantasy of automating the whole gaming experience. I wanted to take the two things I loved and put them together. I wanted to code D&D.

As an adult I have since gotten closer to realizing that adolescent fantasy. My colleague George Walters II (@walterg2) and I have created a workshop called *Putting the D&D in TDD*. It's a one-day workshop where George and I teach Test Driven Development (TDD) using what we call the *Evercraft Kata* (https://github.com/walterg2/EverCraft-Kata) which we also developed mostly in an effort to realize that adolescent fantasy.

The *Evercraft Kata* teaches TDD by practice. In it, you code the domain model for the eponymous game from Blizzards of the Coast. There are a raft of requirements for the game including rules around basic combat, classes, races, and equipment. Completing the kata takes many hours and few complete it during the workshop.

OK. This is fun and all. But what's this article about? Is Guy just plugging his workshop and taking a trip down memory lane? Well, yes on both points but there is a larger point to all this.

You see, this is actually an article about JavaScript—the coolest new, old language out there. It's been around for about 10 years. For the longest time it was the social outcast of programming languages. It sat in the browser all alone, feeling neglected, while all the cool programming languages were back on the server hanging out and doing things like parsing XML and handling database requests.

Somewhere along the way people started wanting to write JavaScript more and Java and its ilk less. And now, much like that adolescent geeks we were, JavaScript has come into its own. Those other languages are now fat and old. All they know how to do is parse XML and read databases. But JavaScript is making the web come alive. It's new and fresh. And those old languages are relegated to the role of serving up data for JavaScript.

So what's the cause of all this buzz? Part of it is related to what JavaScript can do. It makes web applications more interesting and responsive, certainly, but so did applets and ActiveX. So do Flash and Silverlight. But I think the underlying hotness of JavaScript is something more than just its capabilities.

Everyone loves JavaScript because it's the first real functional programming language they have encountered. When all you have done is Java, a functional programming language is really different. When all you have done is C#, JavaScript is a really approachable functional programming language. The syntax of JavaScript is comfortable and familiar compared against something like Clojure. But, once you dive just below the surface, you realize that JavaScript is really different.

OK. JavaScript is cool. Got that. And there's that *Putting the D&D in TDD* thing that you plugged. Where are you going with all of this?

Well, one of the really weird things about JavaScript is prototypal inheritance. Prototypal inheritance is object-based inheritance over class-based inheritance. If you are kind of green to JavaScript, your head probably just exploded. I know mine did when I was introduced to this idea. So, to really understand something that entails head-exploding risks, you need a good solid project to work though. Something like, oh, I dunno, automating a role-playing game?

So that's what I did. I did the first two iterations of the Evercraft Kata in JavaScript and learned quite a bit. I have attempted to distill that knowledge down into an article that explains prototypal inheritance using a subset of the stories in the Evercraft Kata as examples.

Lesson 1: What's in a Name?

The first requirement of the *Evercraft Kata* is:

As a character
I want to have a name
So that I can be distinguished from other characters'

This is, perhaps, one of the easiest requirements in the world to implement—add a property to an object—which makes it a great place to start. The simple object-based solution might suggest you do this.

```
var character = {
  characterName : 'Alice'
};

character.characterName = 'Bob';
```

That works. We have a character. A character has a name, but it misses that "distinguished from other characters" part of the requirement. How can we have multiple characters? And we know D&D, er, Evercraft, is all about combat so we'll need more than one character, if for no other reason, to kill them.

So, we refactor and arrive at this code.

```
var alice = {
  characterName : 'Alice'
};

var bob = {
  characterName : 'Bob'
};
```

Now the character is gone and we need a character! What should we do? How can we have instances if we don't have classes?

Well, this is where `Object.create()` comes in. `Object.create()` allows you to create an object that has a prototype which it inherits from. The prototype itself is an object, not a class, as there are no classes in JavaScript.
Still confused? OK. Maybe I need to start by explaining what objects really are.

What's an Object?

Objects in JavaScript are really just glorified hashes or maps or dictionaries depending on whether you like Ruby or Java or C# or whatever. They are merely a list of name-value pairs. Nothing more. Like all hashes, there is a key and a value. They key can be any valid string. This can make for some strange looking properties if, for example, your key is "3.14" or "true" or "function". While any valid string is valid for a property name on an object, not all are created equally. The language parser for JavaScript makes some of these easier to access than others.

To access most properties you simply use the `object.property` syntax that we've been using since C structures. But this only works for strings that match certain naming conventions in JavaScript (starts with a letter, `$`, or `_`; contains only letters, numbers, and _; etc.). To access values properties, JavaScript provides an array-style accessor written `object['property']`.

For example consider accessing the following JavaScript object:

```
var o = {
  foo : 'foo',
  3.14 : 3.14,
  true : true,
  function : function() {}
};
```

The `foo` property does not require anything special to access it. Simply use it as a property in the old familiar way. `3.14` and `true` require that you use `[]` to access them. `[]` expects a string, which the literals `3.14` and `true` are coerced into, but allows you to pass in a quoted string as well. `function` requires quotes as it is a reserved word and cannot be easily coerced to a string by JavaScript like the boolean and the numeric.

The following code shows all the way to access these properties.

```
o.foo;
o['foo']
o[3.14];
o['3.14']
o[true];
o['true'];
o['function'];
```

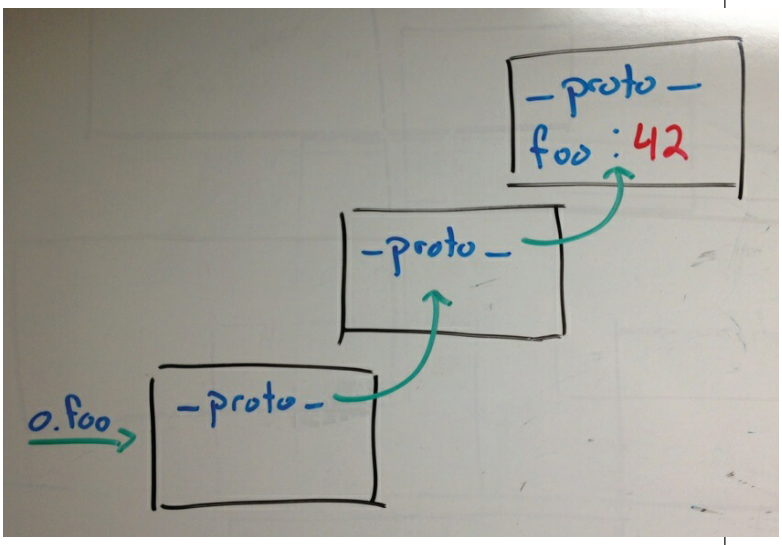
Weird, I know. And, honestly, there isn't a lot of value in defining properties on objects in such an obtuse fashion.

I do it simply to illustrate the point that objects are nothing more than hashes. Any valid string is a key.

But what if you try to access a property in an object and it isn't found? What does JavaScript do then? This is where it prototypes come into the picture.

All JavaScript objects have an intrinsic property called `__proto__`. This special property is not accessible to the developer but is nevertheless there[1] . It contains another object which is the prototype of the current object. Remember the prototype? I mentioned it at the end of the last section. This is how JavaScript does inheritance.

It's simple really. When you try to access a property on an object and it isn't found, then JavaScript looks in the prototype via the `__proto__` property and returns that instead. And it turns out that the prototype has a `__proto__` of its own which means that if object's prototype doesn't have the property then it will check in the object's prototype's prototype. And then in the object's prototype's prototype's prototype. Hard to visualize? OK. Just look at the picture.



Put more simply, if an object doesn't have a particular property, it will ask Dad. If Dad doesn't have it then he will ask his Dad. And if *he* doesn't have it he'll ask his Dad. Lather. Rinse. Repeat. Until presumably we arrive at the property of someone who doesn't have a Dad.

But remember, this is all in the context of objects and objects are just hashes. So, in essence we have a series of chained hashes that we walk up until there are no more hashes to inspect. This chaining of objects via prototypes is imaginatively named the prototype chain and it's a key concept in understanding how to have inheritance without classes.

Using Prototypes

So back to our example. What we really need is a prototypal character that characters can point to for most of their implementation. Right now a character just has a name, which has no default value mentioned in the story. This can be accomplished by defining character thusly.

```
var Character = {};
```

Pretty boring, huh?

Two things to note here. First, we have no implementation and so this is really no better than just defining Alice and Bob like we did in the beginning. That's fair, but we'll be adding stuff soon enough and the requirements did say that we need a character. And, this is just an example for educational purposes. So, stop being so pedantic.

Second, I have capitalized the name of this object. This is because of convention. JavaScript objects that are intended to be inherited from but not used directly are commonly defined with a starting capital letter. String, Object, and Array come to mind as some examples. Character will be used in a similar fashion.

We have a character but how do we set it as the prototype of Alice and Bob? That used to be hard and confusing in JavaScript but now it's the easiest thing in the world to do. Simply call `Object.create()`. This function's purpose in life is to create a new object and set it's prototype to whatever is passed into it. We'll get into how it works in detail in a little bit. For now, we can just use it.

So, using the power of `Object.create()`, we can now create our specific characters and assign them names like this.

```
var alice = Object.create(Character);
alice.characterName = 'Alice';

var bob = Object.create(Character);
bob.characterName = 'Bob';
```

1 In reality, all browsers except Internet Explorer will allow access to the __proto__ property. However, it is not part of the ECMAScript language definitions.

Now we have two distinct characters with two distinct names.

```
alice.characterName == bob.characterName;  // false
```

But what happens if we don't set the `characterName` property? As of right now, it will ask the prototype object (that's Character), not find it, and return `undefined`. This is fine, as the requirements do not say that we need a default name. Alignment on the other hand, is a different story.

Lesson 2: If You Choose Not to Decided You Still Have Made a Choice

The next story states that characters have an alignment.

As a character I want to have an alignment of Good, Neutral, or Evil and I want my default alignment to be Neutral so that I always have something to guide my actions

So, alignment is not optional. Pick a side. If you're not with us, you're against us. A character has an alignment and if the character does not specify that alignment, then said character is Neutral.

So the initial cut of this requirement's implementation could look something like this.

```
var alice = Object.create(Character);
alice.alignment = 'Good';

alice.alignment == 'Good';  // true
```

Alignment is set. Alignment has a value. This works just like name. In fact, it is just like name. Just another property on Alice.

On to the next requirement, defaulting alignment to Neutral. Right now the default is undefined. This shouldn't be terribly surprising as we have not actually defined a default. Defining that default is where we will start to modify the prototype. Consider the following.

```
var Character = {
  alignment : 'Neutral'
};
```

The Character object now has a property named `alignment`. Now when we create another object with Character as the prototype we will see that it has an alignment of Neutral.

```
bob = Object.create(Character);
bob.alignment == 'Neutral';  // true
```

What's happening here is simple. Bob doesn't have an alignment so JavaScript checks Bob's prototype (i.e. Character) and finds the *alignment* property and then proceeds to return it.

At its heart, this is all the prototypal inheritance is. Just dictionaries and parents. It's not even that hard of a concept, although it is admittedly a bit different than what you might be used to if you grew up in object-oriented languages. So why is it so confusing? Well, that's really the fault of the `new` operator.

What's new is Old

Back in the day there was this thing called the `new` operator. Perhaps you've heard of it? It was the way we used to create objects in JavaScript. And it worked just fine (and still does) but I don't recommend it as it is misleading. Here's how it works using what we have developed so far as an example. First, we create a function that is to serve as our object constructor. Convention dictates that it start with a capital letter. Using character this would look something like this.

```
var Character = function() {};
```

Next, we want to define some of the construction logic. this represents the instance being constructed, so we'll add a property to it.

```
var Character = function() {
  this.alignment = 'Neutral';
};
```

Finally we create some new characters and update some of their properties.

```
var alice = new Character();
alice.name = 'Alice';
alice.alignment = 'Good';

var bob = new Character();
bob.name = 'Bob';
```

This code looks attractive in many ways. It seems to be a way of instantiating a class, which appeals to the object-oriented programmer in us all. It also seems very straight forward—even if we are using a function as a class. The pragmatist in you is probably calling me a purist right about now.

But I'm not. Because, behind the scenes there are no classes and no instantiating. Just obfuscated prototypes. All this class business is a lie. Code that lies is evil.

So why the lie? Why the evil? Well, like so much of the evil in the world, it started with Lawful good intentions. You see, JavaScript was developed and written as a functional language (as opposed to an object-oriented language or a procedural language) and, at the time, it was thought that new might make it feel a little more object-oriented and thus be more approachable to the C++ and Java programmers of the day. It was intended to make prototypal inheritance feel like classical inheritance. It was intended to protect you from reality.

Clearly it failed as the code above does not look much like object-oriented programming. Who calls `new` on a function? What if I call the function without `new`? And how is inheritance going to work? This style of programming fails because it does not accurately reflect what is actually happening.

Of course, this code could easily meet the requirements of the *Evercraft Kata* and there is nothing inherently *incorrect* with this code. However, there are some subtle differences in how using new creates the object that can change the behavior of you code.

If you want to write code like this then I recommend you do some long, hard Googling so that you really understand what you're doing. And, don't forget, this style of coding is what made prototypal inheritance so hard to understand in the first place.

Enough of the past, on to the next lesson.

Lesson 3: Using Your Abilities

The next story of the *Evercraft Kata* relates to a character's abilities.

As a character
I want to have the six abilities of Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma
and I want them to have a default score of 10
and I want them to have a modifier based on the score
So that I am not identical to other characters except in name

For brevity, here is an implementation of Ability.

```
var Ability = {
  score : 10,
  modifier : function() {
    return Math.round((this.score - 11) / 2);
  }
};

var strength = Object.create(Ability);
strength.score = 12;
```

This implementation is very similar to Character. We have a score with a default in the prototype object. This default is overridden by assignment. Not terribly different from the implementation of alignment for Character.

We also have in the prototype a function, which we haven't seen before. But it is easy enough to see how it works given what we have learned about prototypes. The function simply uses the score from the current object and performs some basic math to calculate a modifier.

Now, it might be tempting to think that `this` in the `modifier` function points to `Ability`. After all, the function is defined in the `Ability` object and `this` does point to the current object. Wouldn't that make `Ability` the current object?

In this case, no. JavaScript is smart enough to know we are walking the prototype chain and so sets `this` to the value of the object we came in on. In this case, that is `strength`.

At this point, we have defined an ability without much fanfare. Now, let's add them to a character, using just Strength to keep things simple, and see what happens.

Bulking Up

Adding and ability to a character seems straightforward enough. It should be just like adding alignment. The only difference is that we are assigning an object as the default instead of a string.

```
var Character = {
  alignment : 'Neutral',
  strength : Object.create(Ability)
};

var alice = Object.create(Character);
alice.strength.score == 10;  // true

var bob = Object.create(Character);
bob.strength.score == 10;  // true
```

Looks good. Now each character has a strength and it defaults to 10. But Bob has been lifting weights and now his Strength is a 14. Good for you Bob although you should probably work some cardio into your exercise routine as well! Regardless, we update his score to reflect his new, buffer status.

```
bob.strength.score = 14;
bob.strength.score == 14;  // true
```

Great! Bob has a Strength of 14. Let's check on Alice.

```
alice.strength.score == 10;  // false
```

Oops! Something is wrong here. How much do you want to bet that Alice has a Strength of 14 now? Let's check really quick.

```
alice.strength.score == 14;  // true
```

Looks like Alice has been living the dream. She's gotten stronger without having to workout. Somehow Alice and Bob have ended up sharing the same strength score. The problem here is that Alice and Bob are sharing the same strength object, the score just comes along for the ride. We need something a little smarter than `Object.create()` to get the job done here.

And Object Beget Character and Character Beget Alice

The following function is not part of standard JavaScript. But neither was `Object.create()` originally. And they were both created by Douglas Crockford so it does have a seal of approval of sorts. And it's handy—which is arguably much more important. It's called `Object.beget()` and here's how it's implemented.

```
Object.beget = function() {
  return Object.create(this);
};
```

By adding this function to `Object`, which is the root of every prototype chain, every object in JavaScript gets this function. This is, in essence, a lazy man's `Object.create()`. Now we don't have to pass in all the pesky prototype objects. Instead, we call a method on those pesky prototype objects themselves and get the object we want. Niftiness!

With `Object.beget()` I can now create a character like this:

```
var alice = Character.beget();
```

and an ability like this.

```
var strength = Ability.beget();
```

OK. So, this is slightly cleaner syntax but there's a part of you saying "What's the big deal?" Well, while I wouldn't discount cleaner syntax, the real coolness occurs when you override this function by redefining it somewhere lower in the prototype chain. Somewhere like—I dunno—in Character.

```
var Character = {
  beget : function() {
    var that = Object.create(this);
    that.strength = Ability.beget();
    return that;
  },
 alignment : 'Neutral',
  strength : Object.beget(Ability)
};
```

In this override we essentially do a bit of construction. We need to make sure that every object that has Character as a prototype has its own `strength`. By calling `Ability.`

`beget()` we can accomplish that. Now each begotten character has their own begotten strength object. And now that everyone has their own strength objects then everyone also has their own scores (or the scores in the Ability object if they don't set them).

In fact, since each begotten object gets their own Ability begotten strength, we really don't need to assign strength in Character anymore. It'll never get called. So let's remove it.

```
var Character = {
  beget : function() {
    var that = Object.create(this);
    that.strength = Ability.beget();
    return that;
  },
  alignment : 'Neutral'
};
```

Now we can create characters and set their score without impact. Bob can pump iron and, unfortunately for her, Alice won't see the benefits of it—except perhaps on the field of battle!

```
var bob = Character.beget();
var alice = Character.beget();

bob.strength.score = 14;
bob.strength.score == 14;  // true
alice.strength.score == 10;  // true
```

Here we have the completed code if you want to try it out.

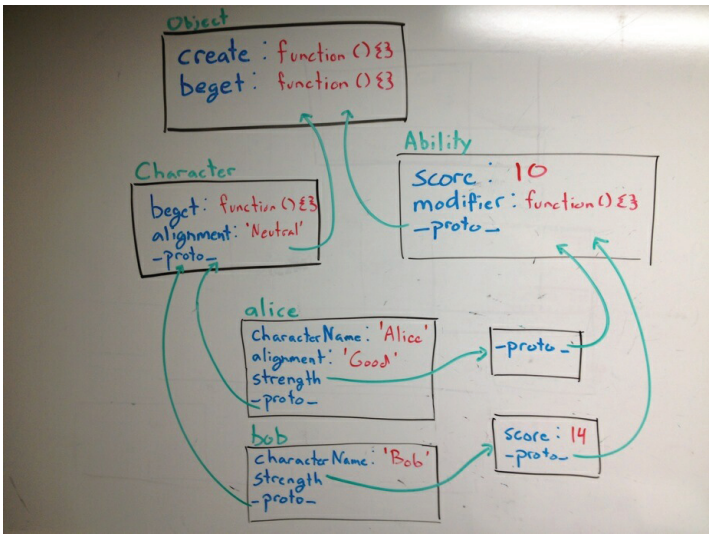
```
Object.beget = function() {
  return Object.create(this);
};

var Ability = {
  score : 10,
  modifier : function() {
    return Math.round((this.score - 11) / 2);
  }
};

var Character = {
  beget : function() {
    var that = Object.create(this);
    that.strength = Ability.beget();
    return that;
  },
  alignment : 'Neutral'
};

var alice = Character.beget();
alice.characterName = 'Alice';
alice.alignment = 'Good';

var bob = Character.beget();
bob.characterName = 'Bob';
bob.strength.score = 14;
```



The results of this code might be hard to visualize so I have provided an awesome, detailed, and painstakingly rendered image of the final object graph after all of these changes.

What's Next?

So what's next? Nothing, really. That was as far as I got in the *Evercraft Kata* that exposed any interesting prototypal goodness that I wanted to share. We learned that objects are really just lists of name/value pairs and that prototypes are just lists that objects will look in to help resolve properties. We also looked at how to create objects with prototypes using `Object.create()`. We also wrote an `Object.beget()` function that was a smidgeon cleaner than `Object.create()` but, more importantly, enabled us to write constructors for objects that were a bit more complex.

Exploring these patterns in the context of the *Evercraft Kata* was a very useful exercise for me. I learned what I think are some spiffy techniques that I have tucked away for future use. So what's next for you? Well, I expect you to go out and start playing around with some of these techniques too. Try them out in your applications, use them in your exercises, and implement them in your pet projects. But most importantly, let me know how they worked for you.

THE FURTHER ADVENTURES OF PRINCESS CHLOE in Clojureland

a fairy tale programmed by

Carin Meier

illustrated by Sarah Allgire

Once upon a time, there was a beautiful kingdom on the JVM called **Clojureland**. It was ruled by the good and kind Princess Chloe and her adorable puppy, Sparkles. She was loved by her people and there was peace and prosperity throughout the land.

```
(def royal-puppy "Sparkles")
 royal-puppy ;=> "Sparkles"
```

Here is an example of lovely Clojure code. We define a var named royal-puppy to be the string "Sparkles". royal-puppy now evaluates to the string, which is in fact a java.lang.String.

Most people had nearly forgotten the battle with the evil wizard that was fought only a few years earlier. That was when Princess Chloe had rescued the kingdom from the clutches of the evil wizard's Infinite Headed Hydra. After his defeat, no one had heard anything from the wizard. Some people thought he had moved away to a distant JVM land and would never bother anyone again. But evil rarely lies dormant for long and the wizard was no exception. He was watching and waiting for his opportunity for revenge.

The evil wizard struck.

This time he had gone too far. He dog-napped Princess Chloe's favorite puppy, Sparkles! He carried the poor puppy off to a tall tower in a remote corner of Clojureland, where he imprisoned her in a java.lang.String of XML.

He made his awful demand clear...

Turn over the kingdom or kiss the puppy goodbye.

P.S. Don't even think of trying a rescue. I have booby trapped all the methods on java.lang.String, and of course, the class is final.

MUHAHAHAHA!!!!!

- The Evil Wizard

```
(def trapped-puppy <evilxml><level354><level355>Sparkles</level355></level354></evilxml>")
trapped-puppy
;=> "<evilxml><level354><level355>Sparkles</level355></level354></evilxml>">
(class trapped-puppy) ;=> java.lang.String
```

Princess Chloe was distraught. She needed to find a way to rescue her poor puppy. But what could she do? All the methods on the trapped-puppy string were booby-trapped. Since the java.lang.String class was final, she couldn't hope to add any new methods ... or could she?

There might just be a way.

She rushed to the royal library and opened the Imperial Clojure Docs. There it was! Protocols. Clojure protocols might allow her to add a rescue method the java.lang.String class, effectively creating a backdoor, and save her kingdom and her pup. "Let's see", she thought. "First, I need a function that will use a regex to extract a string encased by that XML mess."

```
(re-find #"needle" "haystackneedlehaystack") ;=> "needle"
```

Here we are using the re-find function to give us the regex match for "needle" in the "haystackneedlehaystack string. This will be handy later to get Sparkles out of all that XML.

Now, all she needed was a backdoor in the java.lang.String class. Something like this:

```
(rescue trapped-puppy) ;=> Error: Unable to resolve rescue
```

Hmm.... Doesn't work yet. There is no method rescue. We need to define one that will work on a String with a protocol.

She got to work and started defining the protocol that would rescue Sparkles....

```
(defprotocol Rescue
(rescue [this] "Save the puppy"))
```

We are defining a protocol named Rescue. It is polymorphic based on what the type of "this" is. It won't do anything right now though, because we don't have anything implementing it yet.

```
(extend-type java.lang.String
Rescue (rescue [this] (re-find #"Sparkles" this)))
```

We extend java.lang.String to implement our protocol. When we call rescue on a java.lang.String it will run the regex to extract "Sparkles" from the XML string.

Now, for the moment of truth. Would it work?

```
(rescue trapped-puppy) ;=> "Sparkles"
```

Hurray!

Sparkles is rescued and the kingdom is saved! The Evil Wizard shouted "Curses Princess Chloe! Foiled again!" and disappeared. The entire kingdom rejoiced and threw a great celebration tea party in which Sparkles wagged her tail the entire time.

The End

Guest Authors

Peter Bell is Senior VP of Engineering and a Senior Fellow at General Assembly - a campus for entrepreneurship, technology and design. He presents regularly at conferences including DLD conference, ooPSLA, QCon, Strange Loop, RubyNation, SpringOne2GX, Code Generation, Practical Product Lines, the British Computer Society Software Practices Advancement conference, DevNexus, WebManiacs, UberConf, the Rich Web Experience and the No Fluff Just Stuff Enterprise Java tour.

He has been published in IEEE Software, Dr. Dobbs, IBM developerWorks, Information Week, Methods & Tools, Mashed Code Magazine, NFJS the Magazine and GroovyMag. He's currently writing a book on managing software development for Pearson.

Seth Juarez has a Master's Degree in Computer Science where his field of research was Artificial Intelligence, specifically in the realm of Machine Learning. He is a Technical Evangelist for DevExpress where he specializes in data analysis and shaping in conjunction with their reporting toolset. When he is not working in that area, he devotes his time to an open source Machine Learning Library specifically for .NET that is intended to simplify the use of popular supervised and unsupervised learning models.

Data Modeling Using MongoDB

By Peter Bell

There are plenty of articles that provide a basic introduction to MongoDB. In this article I want to dig a little deeper. We'll start by looking at what MongoDB is and why you'd use it, but then we'll dig more deeply into data modeling which is the topic that people often get into trouble with when building systems using MongoDB.

The Basics

MongoDB (from "humongous") is a schemaless, document oriented, NoSQL data store. It is comprised of collections of documents which–as a first approximation–could be thought of as tables of records in a relational database. So for example, you might have a user collection with one document for each user in your system. Here's the fundamental things to know about MongoDB.

NoSQL data store

MongoDB is a NoSQL data store, which is simply one that doesn't use relational algebra or implement Structured Query Language for accessing data. There are a wide range of NoSQL data stores which generally fall into four categories: key-value stores like redis or voldemort, column data stores like Cassandra, graph databases like Neo4j, or document databases like CouchDB or MongoDB.

Document oriented data store

MongoDB is a document oriented database. Like a key-value store, there are collections of values that can be accessed by a unique key, so if you wanted to get a user's shopping cart based on their user ID, you could do that using MongoDB. However, unlike simple key-value stores, the values (documents in the case of MongoDB) are easily searchable, so you can easily find all of the User documents where their home address is in Ohio or where their last name starts with the letter "c".

Schemaless - strengths and weaknesses

MongoDB is schemaless because any given document within a collection can have an arbitrary set of (nestable) key-value pairs. So if you have some documents containing users which have a first name and a last name, but now you want to start storing a middle initial as well, instead of having to apply a schema migration to the collection, you just start including a middle initial key-value pair in your new (or existing) documents. This flexibility solves a number of problems and creates a few new ones.

Being schemaless, you never need to worry about the time it will take to apply schema migrations. Sometimes you will still have to apply data migrations as your understanding of your domain changes, but you won't have to deal with maintenance windows just for changing the structure of data that your database can persist because you can support any structure without making schema changes. This can be a boon when you have large data sets as in such cases schema migrations can take a while to run. It is also useful in the early days of a project as you don't have to keep writing migrations to update the structure of your database as your understanding of the domain evolves–allowing for more rapid prototyping.

It's also a particularly good fit for semi-structured data. Imagine implementing an e-commerce application for Walmart. Think of all of the attributes of the various products sold–from dresses to camping equipment to kids toys. To create a rich, normalized representation of such data that could be easily queried would be a huge undertaking. With MongoDB, you could just have a collection for products and each product could have its own attributes–from playing time for a DVD to size and color for clothing.

The downside of a schemaless data store is analogous to the difference between static and dynamic typing in programming languages. In statically typed languages, the compiler will catch a range of type based problems automatically. Similarly, if you have absolute requirements for your data such as a last name is required and we should never store a middle initial, you can express and enforce those constraints at the database level with a relational database. In a dynamically typed language, you can accidentally get unexpected outcomes because of mistakes in the runtime types of your variables, so often it makes sense to have more unit tests to both document

and enforce those constraints at test time. Similarly, with a schemaless database, you now have a responsibility for a wider range of data integrity issues in your application or test layer as the database doesn't provide any documentation or enforcement of types for your data.

Why MongoDB?

Unlike key-value stores, which are primarily used to solve scaling problems, MongoDB works for both small and large projects. For example, Startup Giraffe is a Rails consultancy in New York that will build Minimum Viable Products for startups in just a few weeks. They use MongoDB for almost all of the projects that they build. It is easy to install, configure and launch in production and the lack of schema makes it very easy to prototype with, while still scaling much better than a relational database for larger projects.

In addition to the lack of schemas, the key strengths of MongoDB are powerful queries when compared to many other NoSQL data stores, strong location based search capabilities and easy scalability. Let's look at each one of those in detail.

Powerful Queries

MongoDB allows for rich queries which are almost as expressive as those for a relational database (but without support for joins). If you have a database called db with a users collection, you can return all of the users with the first name of "John" simply with the following command in the MongoDB shell:

```
db.users.find({'first_name': 'John'})
```

If you only wanted the email address for users named John, you could query:

```
db.users.find({'first_name': 'John'}, {'email': 1});
```

If you wanted every field except for the social security number for users named John, you could retrieve those with:

```
db.users.find({'first_name': 'John'}, {'ssn': 0});
```

In the previous two example, we were including the email address with {'email': 1} and excluding the SSN with {'ssn': 0}. In MongoDB, you can choose to either explicitly include or exclude the fields you want. For example, {'first_name': 1, 'last_name': 1} will return first and last names for users. {'ssn': 0} will return all fields except for the social security number. Actually, there is one exception to this rule. By default you always get the _id, so if you just want names without id's, you'd specify `{'first_name': 1, 'last_name': 1, _id: 0}`.

It's also easy to order the returned records. So if we wanted all of the users names John ordered by their last name we could simple ask for:

```
db.users.find({'first_name': 'John'}).sort({last_name: 1});
```

And if we wanted to page through those results 10 records at a time and were on the third page we could get that using:

```
db.users.find({'first_name': 'John'}).sort({last_name:
1}).skip(20).limit(10);
```

Mongo doesn't provide the joins that you'd get with a relational database, but compared to most other NoSQL data stores the ability to run ad-hoc queries is both powerful and relatively easy to learn.

Positioning

10gen (the company behind MongoDB) uses the following diagram to summarize the positioning of the database:

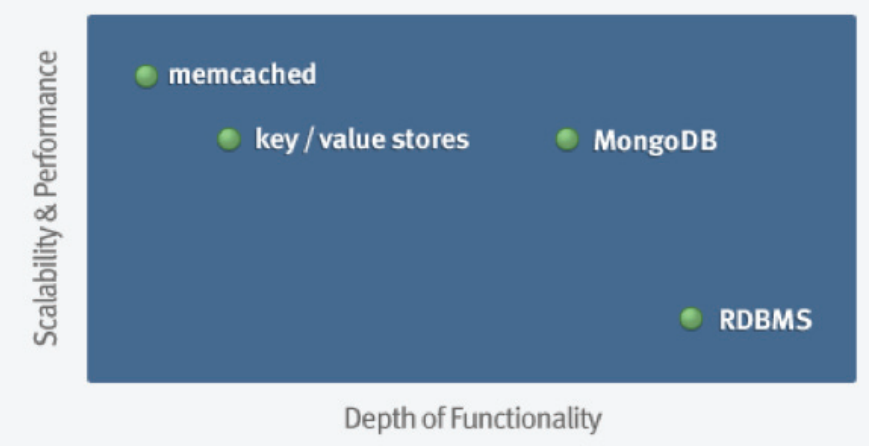


Figure 1: Positioning of MongoDB in the NoSQL market

Basically it is designed to provide much (but not all) of the power of the ad-hoc queries provided by a relational database, providing in return a much more scalable and performant solution that performs comparably to disk based key-value stores.

Location Based Queries
FourSquare uses MongoDB and has contributed some highly performant code that they use for all of their location based queries (return locations to check in near a given latitude and longitude). Because of this, MongoDB can be a great choice for applications that need to run a large number of location based queries.

Easy Scalability
MongoDB makes it easy to scale to multiple nodes. It has a range of features from covered indexes to counters to asynchronous updates which allow high performance from a single server. Replica sets and auto-sharding make it easy to scale horizontally to multiple nodes.

Popular Use Cases

Given the strengths above, the question then might be what are good use cases for MongoDB. Generally Mongo is a particularly good fit for the several topics that we'll discuss next: semi-structured data, document-like data, logging and mobile apps.

Semi-structured data
Whether it's a product catalog with a wide range of different products with varied product attributes or a content management system with a wide range of different content types, being schema-free, Mongo makes it very easy to persist such data without having to jump through the hoops that would be required to persist the information in a normalized, query-able way in a relational database.

Document-like data
Some data is just a really good fit for storing in a document database. For example, WordNik stores a wide range of information about common words in MongoDB, with one document per word storing everything from a definition to examples, lists and tweets containing the word. They migrated from MySQL because the structure of the data (a number of distinct types of information that need to be retrieved together) was a much better fit from a performance perspective in a document than using a number of joined tables in a relational database.

Logging
A number of companies are using MongoDB for logging. It supports counters for easily incrementing log information. It also allows for asynchronous writes which increase write speed by not requiring the application to wait for an acknowledgment that data has been saved. Finally, Mongo supports auto-sharding across multiple servers to scale write loads when they exceed the capacity of a single server, so Mongo scales extremely well for logging style applications.

Mobile apps
With FourSquare contributed geo-querying capabilities, MongoDB makes a great data store for server applications that need to return a lot of location based queries to mobile applications or websites.

When not to use MongoDB
MongoDB is not the right fit for all projects. It only supports document level transactions, so if you need ACID (Atomic, Consistent, Isolated and Durable) transactions across multiple documents, you'd be much better served using a relational data store. It also doesn't provide joins, so if you need the full power of a relational data model, MongoDB probably isn't a good fit.

Data Modeling in MongoDB

One of the first problems that most people run into with MongoDB is how to model their application data. Just because MongoDB doesn't have a schema doesn't mean that data modeling is unnecessary. The biggest data modeling issues in MongoDB typically revolve around what to embed.

The most common mistake that developers new to MongoDB make is that they embed too much. They embed items based on a single view without thinking through all of the implications. For example, one commonly touted example use-case for a document database is a blog post with comments. In that example, the argument is made that it would be much more efficient to retrieve a single document instead of having to join the post to the associated comments using a join in a relational database. This is true as far as it goes, but in practice such decisions needs to be more nuanced.

Beware embedding unbounded collections
If your blog isn't very popular and you consistently get just a handful of comments, embedding those comments within a post document might work fairly well. However in general you should be careful when embedding unbounded collections that could continue to grow. Imagine a posting which gets up-voted on hacker news and ends up with 300 comments. There are three problems presented by embedding all those comments with the post. First, every

time you need to save a new comment, because document operations are atomic, MongoDB will have to read the post and all of the comments from disk, add the new comment to the document and then re-write the entire document to disk. That is an incredibly inefficient way of persisting a single comment. Second, you will probably keep blowing through the padding that Mongo puts around the document so Mongo will have to keep moving the document to other locations on the hard drive every few comments. Third, if a user requests the page, you'll probably only want to display 10-20 comments at a time, but because document operations are atomic, for every page view you'll have to retrieve all of the comments along with the post from the database and then just display 10-20 of them.

A better use case for embedding might be a recipe site. A recipe might have a few ingredients, a few directions, a couple of photos and a few related recipes. This would potentially be a much better fit as the number of ingredients or directions for a single recipe are unlikely to continue to grow or to need pagination, and storing all of the information in a single document provides much more efficient retrieval than having to join a bunch of information from normalized relational tables in a SQL database.

Consider the cost of writes

I recently heard a story about a problematic MongoDB install. A company had heard that MongoDB was a good fit for logging (it is), so they decided to log information from their sensors to a MongoDB database. That in itself would probably be a good decision, but unfortunately they decided to create one document for each day's logs and appended to that over the course of the day. So early in the day, to write, say, 50 bytes of data, they only had to open a document that contained a few hundred bytes of data and save it. However, by late in the day, every 50 byte logging entry required the reading and writing of a file that was many megabytes in size, making the write load many thousands of times what it needed to be. Unsurprisingly, the server slowed down each day as the day progressed until they realized that the data model was causing the problem and moved to simple daily documents using incrementing counters and a separate document for every few logging entries.

Consider atomic operations

MongoDB provides transactional support, but transactions are at a document level. Because of that, if you need something to be transactional, it must be contained within a single document. So, for example, if you wanted to be able to store an order with a collection of order items within MongoDB safely, you'd almost certainly have to create a single document which had an embedded collection of the items within the order. Any other strategy would risk the possibility of the order being saved successfully but some or all of the order items being lost–resulting in an invalid database state.

Think about actual use cases

It is important to think of common usage patterns when trying to determine what to embed. For example, let's say we have blog posts with comments which include the first and last name of the commenter. Should we embed the first and last name in the comments or just reference the user and pull the information from a user collection? Well, assuming that users don't change their names very often it probably makes sense to denormalize by storing the users first and last names with every comment. The downside is that if a user ever changes their name you're going to have a substantial write penalty as you're going to have to update their name in every single comment they made in the system, but given the ratio of comment reads to name updates it's probably an acceptable trade off. As for embedding the comments within a post, if you consistently have a small number of comments it might be a reasonable trade off, but think through other requirements. For example, if you embed comments under posts and need to be able to display all of the comments for a given user across all posts, you'll have to use either a MapReduce or take advantage of the new aggregation engine which was added to MongoDB in 2.2.

0-embed

It's often hard to know exactly what your use cases will be for accessing or updating data when you're building a new application. Because of this, Mike Dirolf–an ex-10gen engineer and experienced MongoDB developer–generally recommends a "0-embed" strategy. With this strategy, you start off by planning on making every object its own first class collection in MongoDB. You can then look at anything which is trivially easy to embed (perhaps it makes sense

to embed order items within an order or directions within a recipe) and embed just those few items. By favoring separate collections and then only embedding over time if it makes sense, you can avoid designing yourself into a corner.

n+1 -> 2 queries

One challenge that comes up pretty quickly when you favor separate collections is the n+1 query problem. Let's say you want to display a collection of 20 comments, but you want to retrieve detailed user info–perhaps including something volatile like the total number of comments by the user–for each of the comments. If you implement this naively, you'll end up with one query to return the 20 comments and then 20 individual queries each returning a single users information. Generally 21 queries to display a single list of comments is a bad choice.

The most common way to solve this problem is to turn the n+1 query into 2 queries. You get the list of comments, build up an array of user_id's for the commenters and then do a single IN() query containing all of those user id's. In a relational database IN() queries can cause performance problems pretty quickly, but in MongoDB, they aren't problematic until you have thousands of id's in the query, so they are a perfect way to remove n+1 query problems[1]. We still have two queries instead of one, but that's way better than 21 queries.

Conclusion

MongoDB is a powerful database which is both easy to get started with and capable of scaling to very substantial loads. However it is important to really think through your data modeling strategy to ensure it's going to be a good fit for the kind of usage patterns your application is likely to encounter. Hopefully some of the hints within this article will help you to avoid the most common pitfalls that developers encounter when working on their first few data models in MongoDB.

C#, Math and You

By Seth Juarez

Introduction

My name is Seth and I like math. There I said it. Now that we have gotten that out of the way I will tell you about some of the foundational language constructs that are often overlooked in C#. I'll also detail a somewhat sordid journey I took while implementing a small subset of linear algebra in an effort to produce an open source library for machine learning[1] . During the process of writing (and rewriting) code I learned some pretty interesting things that significantly reduced friction in an API dealing with large sets of numbers. I also stumbled across some interesting language features that made difficult language concepts more compact. Overall, the experience was both cathartic and painful.

The Problem (and simple solution)

Machine learning uses a lot of linear algebra. Linear algebra uses matrices and vectors which are, at their most basic level, arrays. The challenge I was running into was the incessant use of for loops. My brain simply wanted to abstract this type of code and leave it in the bowels of some other class or file. So I immediately got to work on building a Vector class. This class was simply intended to encapsulate an array and add supplementary methods to work with the array. At the bowels of the class was the simple array. The first revision, shown in Figure 1, is not necessarily coding perfection but it was definitely an adequate start.

```
public class Vector
{
    private double[] _vector;
    public Vector() { }
    public Vector(int length) {
    _vector = new double[length];
    public Vector(double[] vector) {
    _vector = vector;
}
    public int Length {
  get { return _vector.Length; }
}
```

Figure 1: Vector class

This first approximation indeed looks like wasted space but the understanding was that there would be some additions.

Operator Overloading

The first desperately needed set of features to get implemented dealt primarily with API simplification. Firstly, there was no way to implement any of the items in the Vector. Secondly, general Vector creation was verbose and obtuse. Thirdly, complex vector operations needed simplification. These three elements led me to a (re-) discovery of several underused features in C#—the first of which was overloading. This section describes how to overload various C# operators.

Indexers

A first approximation to exposing individual elements could be a read only getter of the form shown in Figure 2.

```
public double[] Items { get { return _vector; } }
```

Figure 2: Getter that returns the internal array

This crude first approximation leads to verbose requests for vector items as accessing individual items first requires a call to the Items property. A simple and useful construct is the use of indexer overloading. Figure 3 shows what it looks like.

```
public double this[int i]
{
    get { return _vector[i]; }
    set { _vector[i] = value; }
}
```

Figure 3: Overloading an indexer

This simple overload allows virtual access to elements of the vector array in a concise manner. While this particular overload can be added to any class, care should be used in its implementation. There is no rule regarding the parameters that can be passed into the indexer nor

1 See http://machine.codeplex.com as well as a newer yet-to-be-released library http://numl.codeplex.com.

is there a restriction on the multiplicity of parameters. In other words, an implementation that doesn't take in a single integer and return elements at that index can prove particularly tricky for those using your API. Later on we will see an interesting (and correct) use of this type of overload when solving a mass update problem with Vectors.
With the nifty addition of an overloaded indexer, our new code for declaring and using Vectors is shown in Figure 4.

```
Vector v = new Vector(new double[] { 1, 2, 3, 4, 5, 6,
7 });
double first = v[0];
```

Figure 4: Declaring and accessing a single value in a Vector

The code in Figure 4 creates a new vector that has a length of 7. Our indexer overloading proved a useful find in eliminating access verbosity.

Implicit/Explicit Operators

We just solved access verbosity so what about declaration verbosity? There were several "a-ha" moments that led me to more .NET gems along the path to a truly concise methodology. The first of these realizations had to do with implicitly typed arrays and anonymous types. I had seen the ability to create anonymous types before and found that this extended to array declarations. Anonymous types are read-only mini-objects that can be declared as in Figure 5.

```
var things = new[]
{
    new { Name = "Sally", Age = 26 },
    new { Name = "Joe", Age = 16 },
    new { Name = "Ben", Age = 20 },
    new { Name = "Jody", Age = 18 },
    new { Name = "Sue", Age = 32 },
};

int age = things[0].Age;
```

Figure 5: Declaring an anonymous type

This is an implicitly typed array of anonymous objects that contain Name and Age as properties. Seeing this made me think that surely more simplistic things can be substituted for the anonymous objects – hence implicitly typed arrays. Figure 6 shows a new declaration that is much simpler.

```
var things = new[]{ 1, 2, 3, 4, 5, 6, 7 };
```

Figure 6: Implicitly typed array

For some reason, dear reader, I learned the former before I learned this simple array declaration. Having understood this, I set to work on simplifying the Vector declaration as in Figure 7.

```
Vector v = new Vector(new[] { 1, 2, 3, 4, 5, 6, 7 });
```

Figure 7: Simpler Vector declaration

This eliminated the double keyword but introduced a type problem. This implicitly typed array decided to make itself an implicitly typed array of int rather than double. To force things along I made the change in Figure 8 to satisfy the compiler but add to my failure in API conciseness.

```
Vector v = new Vector(new[] { 1d, 2, 3, 4, 5, 6, 7 });
```

Figure 8: Explicitly declaring the type for the values in the Vector

My sense has always been that things should be discoverable in any API. An API should always aim for conciseness and lead its users down a trodden path that leads to utilitarian felicity. At this point in the development of the API I thought: "Self, we should make a constructor that takes an array of ints and then cast them to doubles." As any good developer that argues with himself would say, the following retort ensued: "So you mean to make a constructor for decimals, single's and every other number type? Does that make things better?" Ultimately I would have to write a myriad of constructors to satisfy this line of thinking without really introducing any benefit. Each of these Vector declarations, with their associated improved constructor, would require the same 12 characters of cruft associated with declaring a new Vector, namely new Vector(). My sense was that there should be a more elegant way to combine implicitly typed arrays with the declaration of Vectors. I found my answer in a little known operator (at least to me) called implicit. First, the declaration is demonstrated in Figure 9.

```
public static implicit operator Vector(double[] array)
{
    return new Vector(array.ToArray());
}

public static implicit operator Vector(int[] array)
{
  return new Vector(arrray.Select(
  i => (double)i).ToArray());
}
```

Figure 9: Overloading the implicit operator

Let's focus on the first implicit operator by seeing what this looks like when it is used (Figure 10).

```
Vector v = new[] { 1d, 2, 3, 4, 5, 6, 7 };
double first = v[0];
```

Figure 10: Using the overloaded implicit operator

The implicit operator is designed to make an implicit conversion from the type passed into the operator to the return type of the operator. In other words, the '=' operation (which has a return type by the way) attempts to assign from one thing to another. In C#, the type of both operands of the assignment operator must be the same… unless there is an implicit conversion available. The assignment operator above is simply seeing a type mismatch and finding the appropriate implicit operator to use. In fact, if you removed the second implicit operator from an integer array to a Vector as well as removed the 'd' from the assignment, the compiler would complain. Adding the second implicit operator allows the user to declare Vectors implicitly without having to new up the Vectors themselves. In essence, readability and conciseness are improved. Some warnings need to be issued at this point. Initially, the first implicit operator (which has an interesting array. ToArray() call) passed the array by reference into the Vector constructor. This small oversight on my part could lead to unexpected data loss when using this implicit operator. Consider the Vector declarations in Figure 11.

```
double[] numbers = new double[]{ 1, 2, 3, 4, 5};
Vector bad = numbers;
int[] iNumbers = new int[] { 1, 2, 3, 4, 5 };
Vector bad2 = iNumbers;
```

Figure 11: Vector declarations

If the `ToArray()` call were missing in the first implicit operator in Figure 9, these two declarations would behave differently. Without a full copy of the array, subsequent changes to the numbers array would also affect the bad Vector – causing data loss[2] .

This brings up the next, and related, warning: pretend the programmer using your code does not know what or how you implemented the implicit operator. Since this is a language adjustment I made in order to improve code readability it should just work – and consistently. If it should not work, the compiler should point it out. Let's say I harbor ill will towards decimals and do not wish to allow that type of implicit conversion as would occur in Figure 12.

```
decimal[] numbers = new decimal[] { 1, 2, 3, 4, 5 };
Vector bad = numbers;
```

Figure 12: Implicit conversion for declaring a Vector of decimals

The code in Figure 12 should simply create a compile time error (which it does). Furthermore, throwing exceptions on these kinds of language adjustments seems incorrect. If I, as the programmer, explicitly force a cast on something, I should be aware that this might cause a run-time exception. If you feel that the implicit overload operator should throw an exception, consider using the explicit operator overload instead. Consider the cast used in Figure 13.

```
double d = 2;
Vector single = (Vector)d;
```

Figure 13: Casting a double to a Vector

In this case we wish to explicitly cast a double to a Vector. The declaration for this given in Figure 14 is familiar.

```
public static explicit operator Vector(double i)
{
    return new Vector(new[] { i });
}
```

Figure 14: Overloading the explicit operator

The main language difference between implicit and explicit casts is that one must invoke this operator with an explicit cast. In summary, if there is structural equivalence between two things and converting between the two can be done in both an exception-free and lossless way I would consider using the implicit operator overload. Otherwise, ask the programmer to force a cast by overloading the explicit operator.

Arithmetic

The next bit of operator overloading involves actual operators. While many may find a sprinkling of for loops to be desirable, my codebase was suffering from an overload

2 Thanks to Peter Richie for pointing this out (see: http://msmvps.com/Blogs/PeterRitchie/).

of desirability. Simple operations became verbose. Take, for instance, the code in Figure 15 where elements of two vectors are being added.

```
if (one.Length == two.Length)
{
    Vector three = new Vector(one.Length);
    for (int i = 0; i < one.Length; i++)
        three[i] = one[i] + two[i];
}
```

Figure 15: Adding elements of Vectors

This methodology is an unfortunate necessity of element-wise operations and one that was becoming rather tedious. Enter operator overloading—again—as in Figure 16.

```
public static Vector operator +(Vector one, Vector two)
{
    if (one.Length == two.Length)
    {
        Vector three = new Vector(one.Length);
        for (int i = 0; i < one.Length; i++)
            three[i] = one[i] + two[i];
        return three;
    }
    else {
    throw new InvalidOperationException(
    "Improper vector lengths");
}}
```

Figure 16: Overloading the + operator

This type of overload allows for a simpler and refined methodology for adding vectors (Figure 17).

```
Vector one = new[] { 1, 2, 3, 4, 5, 6, 7 };
Vector two = new[] { 1, 2, 3, 4, 5, 6, 7 };

Vector three = one + two;
```

Figure 17: Using the overloaded + operator on two Vectors

While code like this is not necessarily an episode of *Lifestyles of the Rich and Famous*, there is a sense of greater cohesion and freedom of expression when dealing with higher order types. The next logical question is then, what operators can be overloaded? Instead of listing them here, you can see an extensive list of what can and cannot overload in the MSDN documentation online.

One key challenge here is the understanding that ordering is important. This particular example did not suffer from an ordering problem but there might be other examples that will. A simple example is matrix multiplication. This particular operation is not commutative. In other words A times B is not the same as B times A. In fact, while one operation might be allowed, the other should cause an exception. This brings us to the second thing to keep in mind when writing any API of this nature: be explicit when throwing exceptions. It might even be to your advantage to create a set of exceptions specific to your library. Minimally you should be explicit in the message you pass to the exception you end up using. `InvalidOperationException` has certainly come in handy for me when throwing exceptions.

Functional Approaches

In an effort to be concise, I found help from thinking functionally. Of all the computer science classes I've taken, the one that stays with me is a class on Functional Programming. The professor allowed each student to pick a functional language of their choice and each week everyone picked a problem and solved it in a functional way. This approach to learning was extremely efficient. I found myself struggling with functional concepts until I had a realization: functions are things too. I will explain. Most programmers understand data types and how they should be used. No one will balk at a declaration of an int and its subsequent use. You can use an int in an expression, pass it around as a parameter and implicitly declare its scope by its placement in the code. The same is true for functions, which are known as delegates in .NET. Once a delegate type is declared, you can instantiate a new one, use it in an expression, pass it around as a parameter, and even implicitly declare its scope by placement. Once this understanding burned into my mind, other functional concepts became easier. Functional programming is about the composition of functions to create meaningful computation. Standard OO programming is about the composition of objects to create meaningful computation. Why not do both? There are several functional idioms you can use to tidy up the Vector API to introduce conciseness and reduce overall friction.

Operators Revisited

Earlier we took a dive into operator overloading. Given the list of binary operations that can be performed with Vectors, why not abstract these binary operations into a single

operation method? Consider the generic operation method in Figure 18.

```
public static Vector Operation(Vector one, Vector two,
    Func<double, double, double> op)
{
    if (one.Length == two.Length)
    {
        Vector three = new Vector(one.Length);
        for (int i = 0; i < one.Length; i++)
            three[i] = op(one[i], two[i]);
        return three;
    }
   else {
   throw new InvalidOperationException(
   "Improper vector lengths");
}
}
```

Figure 18: Abstracted Operation method

This method is almost exactly the same as our + operator overload in Figure 16, with the single exception being the delegate parameter (Func<double, double, double> op). Although a review of how delegates work in .NET should be a topic of a separate article, a brief synopsis of the Action and Func delegates should suffice. Delegates encapsulate functional units by first having a declaration of the delegate shape and subsequently a usage as shown in Figure 19.

```
delegate int Foo(int i);
...
Foo f = i => i * i;
```

Figure 19: Declaration and usage of a delegate

The first line Figure 19 declares the shape of the functional unit (i.e. its return type and parameters). The next line is a usage of the newly declared type in compressed lambda notation. The newly declared f variable can also be set to any method of the same type (i.e. return type and parameters). In this code, f simply becomes an operation that squares its parameter.

Because these delegate shapes are so common, the .NET framework introduced the Action and Func typed delegates. The Action delegate represents a functional unit that returns void while the Func delegate returns a value. These were created with generics in order to provide a wide range of delegates without forcing developer to declare them.

In the example in Figure 18, the Func delegate takes two doubles and returns a double with the last generic type being the return type. In essence, we are making a method that takes two Vectors and creates a third by doing something that takes two doubles and returns the third. In essence we have effectively deferred that something to the caller. With this new generic operation over lists of doubles, we can overload the standard operators as done in Figure 20.

```
public static Vector operator +(Vector one, Vector two)
{
    return Operation(one, two, (o, t) => o + t);
}

public static Vector operator -(Vector one, Vector two)
{
    return Operation(one, two, (o, t) => o - t);
}

public static Vector operator *(Vector one, Vector two)
{
    return Operation(one, two, (o, t) => o * t);
}

public static Vector operator /(Vector one, Vector two)
{
    return Operation(one, two, (o, t) => o / t);
}
...
```

Figure 20: Overloading standard Vector operators with the generic Operation method

Achievement avoid-the-unnecessary-repetitive-use-of-for-loops unlocked.

Indexers Revisited

I spoke too soon (I realized). One of the nagging series of for loops I had to write revolved around setting individual Vector items based upon specific criteria. A simple example would be setting each Vector element with a value of 0 to -1. It turns out that using a functional approach, demonstrated in Figure 21, solves the issue famously.

```
public double this[Func<double, bool> criteria]
{
    set
    {
        for (int i = 0; i < Length; i++)
            if (criteria(this[i]))
                this[i] = value;
    }
}
```

Figure 21: Functional approach to reducing repetitive for loops

This seemingly crazy declaration allows the user of the Vector class to write code similar to Figure 22.

```
Vector one = new[] { 1, 0, 0, 4, 5, 0, 7 };
one[d => d == 0] = -1;
```

Figure 22: Using the functional approach to conditionally set element values in a new Vector

While the usefulness of this approach is debatable, it saved me a bunch of time when trying to conditionally set elements within the vector.

Lazy Evaluation

An interesting feature that I wanted to add to the Vector class was the notion of using the foreach loop over the items in the array. The simplest approach to take is implementing the IEnumerable<double> interface. This particular interface inherits from the standard IEnumerable interface and requires the overloading of two methods—one from each interface. I struggled over how to implement this. My first approach was to return the array itself. This methodology troubled me given that an iterator is supposed to return each item singularly. Consider a foreach loop whose primary job is to find the first element of the Vector that fits a certain criteria. Returning the entire array would be a useless exercise. This led me to another functional concept: lazy evaluation. This principle can be summarized as follows: compute only when necessary. In other words, if you don't need a value don't create/compute/retrieve/anything it. The .NET framework supports this principle through the use of a concept called co-routines using the yield keyword Figure 23 is an example.

```
public IEnumerator<double> GetEnumerator()
{
    for (int i = 0; i < Length; i++)
        yield return this[i];
}

IEnumerator IEnumerable.GetEnumerator()
{
    for (int i = 0; i < Length; i++)
        yield return this[i];
}
```

Figure 23: IEnumerable<double> interface

The first surprise for me was that this actually compiled! The second surprise happened when I set breakpoints on the yield statement and stepped through the code in Figure 24.

```
foreach (double item in three)
{
    Console.WriteLine(item);
    if (item > 10) break;
}
```

Figure 24: Using a foreach loop with the Vector class

The notion of co-routines becomes very clear when looking at how the GetEnumerator() calls literally yield execution to the caller for each item in the collection. It's almost as if the in keyword initiates the "pull" from the enumerator and the enumerator relinquishes the next available value. While this initially seems a bit useless, it becomes important when chaining together vectors.

Chaining

The last topic included in functional approaches takes lazy evaluation to the next level. With the understanding of how co-routines work, we can combine the notion of deferred computation with lambda expressions and add the concept of extension methods. I created a helper class to contain all of the static extension methods added to the Vector class. Extension methods let you add functionality to existing classes without resorting to inheritance or modification of the existing type. In this case I wanted to be able to chain operations on vectors in a lazy way. Consider the case where one would like to create a new vector from an existing one by performing operations on a subset of the items in the original vector. If these operations are expensive and the list can be short-circuited, it makes no sense to perform this complex operation on every element of the vector and then filter (or vice-versa). With this in mind I created the class in Figure 25.

```
public static class VectorHelpers
{
    public static IEnumerable<double> Filter(
        this IEnumerable<double> vector,
        Func<double, bool> op)
    {
        foreach (double item in vector)
            if (op(item))
                yield return item;
    }

    public static IEnumerable<double> Do(
        this IEnumerable<double> vector,
        Func<double, double> op)
    {
        foreach (double item in vector)
            yield return op(item);
    }
}
```

Figure 25: The VectorHelpers class

The first thing of note is the use of the keyword `this`. In essence, it instructs the compiler to allow the attachment of the method in question to the type following the `this` keyword. The first two methods (`Filter`[3] and `Do`) are similar in that they defer certain computation to the caller. The `Filter` method yields only the elements of the Vector that pass the boolean operation passed into the method. The `Do` method yields a computed value for each element in the Vector. The last method simply uses implicit casting (see explicit operator above) to cast an `IEnumerable` of `double` to a Vector. With these simple methods attached to a Vector[4] we can now construct complex chains that enable lazy computations of the individual elements of the same as shown in Figure 26.

```
Vector observations = new[] { 1, 0, 0, 4, 5, 0, 7 };
Vector output = observations
                .Filter(d => d > 0)
                .Do(d => Math.Sqrt(d))
                .Filter(d => d > 1)
                .Do(d => Math.Pow(d, 3.4))
                .ToArray();
```

Figure 26: Using method chaining

In Figure 26, all of the elements of the `observations` vector are subjected to a filter. Each element is tested and yielded to the next `Do` method which performs the square root operation. Notice that any element that *does not* fit the filter *will not* be passed to Do method and therefore *will not* be computed. Let's pretend for a second that the square root operation is very expensive. In this example we are strictly limiting the amount of square roots computed based upon an initial filter. These results are then filtered again and yielded up to the next `Do` method. The results are finally aggregated and materialized into an array which is implicitly converted to a Vector. The key thing to understand is what it means to "materialize" an `IEnumerable`. The `IEnumerable` only knows about what is next. Therefore, in order to actually get the list of things (rather than only what is next) there are certain operations which do the concrete task of forming a data structure to hold the collection.

These extension methods are found in the `System.Linq` namespace and include such methods as `ToArray()`, `ToList()`, `ToDictionary()`, etc. These methods serve as the catalyst for pulling the next value by implicitly calling the `MoveNext` method on the `Enumerator` that serves as its source and pooling the values into some type of list data structure. In the example in Figure 26, it is actually the `ToArray` method that asks its predecessor (the `Do` method that performs the power operation) for the next value. This `Do` method in turn asks the preceding `Filter` method for its next value in order to perform the power operation. This in turn asks the previous `Do` method etc. (you get the point). This "pull" style of retrieving the values as needed is the hallmark of lazy operations: only compute exactly what you have to compute.

Conclusion

Although this article focused on seldom used techniques and was principally mathematical in its implementation and approach, these concepts can be used in a variety of practical settings. You could envision an API over Customers that includes specialized indexing, implicit and explicit conversions, and maybe even Customer arithmetic (Customer A + Customer B?). Using functional approaches also has its benefit in standard APIs as it allows developers to defer execution by having functional units passed in to methods rather than explicitly having to imagine every possible implementation. I have always thought of this practice as the "poor-man's" approach to dependency injection. Lazy evaluation also provides a way to save those precious clock cycles for work that is actually needed. Finally the use of extension methods assists with method chaining that is certain to reduce API friction. These simple additions to an API will no doubt increase the joy your users will experience as they praise your sweet name for simplifying otherwise tedious tasks that would have had them working late into the night.

3 This is such a useful extension method that it is actually part of the Framework. See IEnumerable.Any for the actual Framework method: http://msdn.microsoft.com/en-us/library/bb534972.aspx

4 The methods are actually attached to IEnumerable<double> but since we added the interface to the Vector class it is correct to think of the Vector class as an IEnumerable of double.

Listing of Local User Groups

Local User Group Listing

We know that you are an active member of your local software development community and we want to help you to stay active. To help, we've compiled a list of pertinent user groups throughout Michigan and Ohio. If you're looking for a new user group to attend or speak at, you'll probably find it here. Check the user group's website for details on meeting times, dates and exact locations.

If you have information about a user group that is not listed here, or a correction to one that is, please email the pertinent information to mashedcodemag@gmail.com.

Other User Group Listings

Ohio is fortunate to have a couple of organizations that help technologists track user groups and tech events in the area. It's worth the time to check them out.

TechLife Ohio (@techlifecbus) - http://www.techlifeohio.com/
Cleveland Tech Events (@clevtechevents) - http://clevelandtechevents.com/

If you know of similar organizations (especially if you run it and especially in MI) let us know!

Michigan User Groups

Topic	User Group	Meeting Location	Website	Twitter
Agile	Agile Groupies		http://agilegroupies.groupsite.com/main/summary	
Agile	Grand Rapids Testers	Grand Rapids	http://groups.google.com/group/gr-testers	
Agile	Michigan Agile Enthusiasts Group		http://tech.groups.yahoo.com/group/Michigan_Agile_Enthusiasts_Membership	
Agile	MidMichigan Agile Group	East Lansing	http://www.meetup.com/Mid-Michigan-Agile-Group/	@mmagile
Apple	Grand Rapids Area Macintosh Users Group	Grand Rapids	http://www.gramug.org/index.php	
Cloud Computing	Greater Detroit Cloud Computing Users Group	Detroit	http://www.detroitcloudgroup.org/	@detroitcloud
Computers	Computer Club of Western Michigan University	Kalamazoo	http://yakko.cs.wmich.edu/	
Drupal	Grand Rapids Drupal User Group	Grand Rapids	http://groups.drupal.org/grand-rapids-mi	
FoxPro	Grand Rapids Area FoxPro User Group	Grand Rapids	http://www.grafug.com/	@grafug
iOS/Mac	CocoaHeads Ann Arbor	Ann Arbor	http://cocoaheads.org/us/AnnArborMichigan/index.html	
iOS/Mac	CocoaHeads Detroit	Birmingham	http://cocoaheads.org/us/DetroitMichigan/index.html	@CocoaHeadsDet
Java	Ann Arbor Java User Group	Ann Arbor	http://www.aajug.org	
Java	Detroit Java User Group	Troy	http://sites.google.com/site/detroitjug	@detroitJUG
Java	Grand Rapids Java User Group	Grand Rapids	http://www.gr-jug.org	
Java	Greater Lansing Java User Group	East Lansing	http://groups.google.com/group/greaterlansingjug	
Linux	Grand Rapids Linux Users Group	Grandville	http://wiki.grlug.org	
Linux	Greater Lansing Linux User Group	Lansing	http://www.gllug.org	@gllug

Linux	West Michigan Linux Users Group	Grand Rapids	http://www.wmlug.org/	
Microsoft .NET	Ann Arbor .NET User Group	Ann Arbor	http://www.aadnd.org	@aadnd
Microsoft .NET	Great Lakes Area .NET User Group	Southfield	http://www.migang.org	
Microsoft .NET	Greater Lansing User Group .NET	Flint	http://www.glugnet.org	
Microsoft .NET	Greater Lansing User Group .NET	East Lansing	http://www.glugnet.org	
Microsoft .NET	West Michigan .NET Developer Group	Grand Rapids	http://www.wmdotnet.org	
Microsoft Administration	West Michigan NT Users Group	Grand Rapids	http://www.wmntug.org/	
Microsoft Development	Microsoft Developers of Southwest Michigan	Kalamazoo	http://www.devmi.org	@mdsm
Mobile	Mobile Monday Detroit	Detroit	http://mobilemondaydetroit.org/	@MobileMondayDet
Python	Grand Rapids Python User Group	Grand Rapids	http://www.grpug.org	@grpug
Python	Michigan Python User Group	Ann Arbor	http://groups.google.com/group/michipug	
Ruby	Ann Arbor Ruby Brigade	Ann Arbor	http://groups.google.com/group/a2rb	@a2rb
Ruby	Lansing Ruby Users Group	Lansing	http://www.meetup.com/Lansing-Ruby-Users-Group/	@lansingruby
Ruby	Michigan Ruby Users Group	Grand Rapids	http://www.meetup.com/mi-ruby/	
Science/Technology	The Geek Group	Grand Rapids	http://www.thegeekgroup.org/	@thegeekgroup
Social Networking	DetroitNET		http://www.detroitnetworking.org	@detroitnet
Software Craftmanship	Software GR	Grand Rapids	http://www.softwaregr.org	@softwaregr
Software Development	Ann Arbor Computer Society	Ann Arbor	http://www.computersociety.org	
Software Development	Coffee House Coders, Ann Arbor	Ann Arbor	http://www.coffeehousecoders.org	@coffeehousecode
Software Development	Coffee House Coders, Detroit	Madison Heights	http://www.coffeehousecoders.org	@coffeehousecode
Software Development	Coffee House Coders, Downriver Area	Wyandotte	http://www.coffeehousecoders.org	@coffeehousecode
Technology	ConnecTech		http://www.connectech.org	@connectechD
Technology	West Michigan Small Business Technology User Group	Grand Rapids	http://wmsbtug.org/	
Technology	West Michigan Technology Association	Grand Rapids	http://www.wmta.biz/	
Unix & Linux	Michigan!/usr/group	Farmington Hills	http://www.mug.org	
Web Design	Refresh Detroit	Ann Arbor	http://www.refresh-detroit.org	@refreshdetroit
Web Development	Grand Rapids Web Developers Group	Grand Rapids	http://www.meetup.com/grwebdev/	@grwebdev

Ohio User Groups

Topic	User Group	Meeting Location	Website	Twitter
Acceptance Test Driven Development	Columbus ATDD Developers Group	Dublin	http://columbusatdd.wordpress.com/	@columbusatdd
Adobe Flex	Cincinnati Flex User Group	Mason	http://cincyflex.groups.adobe.com/	@cincyflex
Agile	Cincinnati Agile Round Table	Cincinnatti	http://www.agileroundtable.org	@agileroundtable
Agile	Cleveland Agile	Cleveland	http://www.meetup.com/ClevelandAgile/	#CLEAG
Application Lifecycle Management	Central Ohio Application Lifecycle Management Group	Columbus	http://www.coalmg.org	
Big Data	Big Data Analytics Special Interest Group	Dublin	http://www.meetup.com/Big-Data-Analytics-Special-Interest-Group-Columbus-OH/	
Business Process and Enterprise Integration	Mid-Ohio Connected Systems Developers Group	Columbus	http://www.mocsdug.org	
C# & VB.NET	Cleveland C#/VB.NET Special Interest Group	Independence	http://www.clevelanddotnet.info	@SamNasr
Clojure	Columbus Clojure User Group	Westerville	http://www.inclojure.com	@inclojure
Cloud Computing	Central Ohio Cloud Computing User Group	Columbus	http://www.meetup.com/coccug/	@coccug
Drupal	Central Ohio Drupal User Group	New Albany	http://www.meetup.com/Central-Ohio-Drupal/	
Dynamic Programming Languages	Dayton Dynamic Languages Special Interest Group	Dayton OH	http://www.dma.org/sigs.shtml#Dynamic	

FileMaker	Central Ohio FileMaker Users Group	Columbus	http://cofmug.org/	
Game Development	Central Ohio GameDev Group	Columbus	http://www.meetup.com/The-Cogg/	
Hadoop	Columbus Hadoop User Group	Columbus	http://www.meetup.com/Columbus-HUG/	
Information Technology	Computer Erie Bay Users Group	Sandusky	http://www.cebug.org	
iPhone Development	Columbus iPhone Developers User Group	Columbus	http://groups.google.com/group/cidug	@cidgu
Java	Central Ohio Java User Group	Dublin	http://www.cojug.org	@javajudd
Java	Cincinnati Java User Group	Mason	http://www.cinjug.org/	@cinjug
Java	Cleveland Java User Group	Independence	http://www.meetup.com/cleveland-java/	@javausers
JavaScript	Cincinnati JavaScript User Group	Cincinnati	http://blog.cincijs.com/	@cincijs
JavaScript	Columbus JavaScript User Group	Columbus	http://groups.google.com/group/cbusjs	@cbusjs
Microsoft .NET	.NET Special Interest Group	Independence	http://www.bennettadelson.com/ComingEvents.aspx	
Microsoft .NET	Central Ohio .Net Developers Group	Columbus	http://www.condg.org	@condg
Microsoft .NET	Central Ohio DotNetNuke User Group	Columbus	http://www.dnncmh.org/Home.aspx	@dnncmh
Microsoft .NET	Cincinnati .NET User Group	Mason	http://www.cinnug.org	
Microsoft .NET	Dayton .NET Developer Group	Dayton OH	http://www.daytondevgroup.net	
Microsoft .NET	Findlay Area .NET User Group	Findlay	http://www.fanug.org	@fanug
Microsoft .NET	Northwest Ohio .NET User Group	Toledo	http://www.nwnug.com	@nwnug
Microsoft SharePoint	Central Ohio Sharepoint User Group	Columbus	http://www.cospug.org	@buckeyespug
Microsoft SharePoint	Cleveland Sharepoint User Group	Cleveland	http://www.sharepointcleveland.com	@sharepointcle
Open Source Software	Java & Open Source Software User Group	Columbus	http://www.osc.edu/~bpowell/joss	
PHP	Columbus PHP Meetup	Columbus	http://www.meetup.com/phpphp/	@columbusphp
PHP	Ohio, Indiana, Northern Kentucky PHP Users Group	Cincinnati	http://oink-pug.org/	@oinkpug
Polyglot Programming	Columbus Polyglot Programmers Meetup Group	Columbus	http://www.meetup.com/The-Columbus-Polyglot-Programmers-Meetup-Group	
Python	Cleveland Area Python Interest Group	Cleveland	http://www.clepy.org	@clepy
Ruby	Cincinnati Ruby Brigade	Cincinnati	http://www.cincinnatirb.org	@cincinnatirb
Ruby	Cleveland Ruby Brigade	Cleveland	http://www.meetup.com/ClevelandRuby/	@clerb
Ruby	Columbus Ruby Brigade	Columbus	http://www.columbusrb.com	@columbusrb
Software Architecture	Cincinnati Software Architecture Group	Cincinnati	http://www.cinnug.org	@cinnug
Software Architecture	Columbus Architecture Group	Columbus	http://www.colarc.org	@colarc
Software Craftsmanship	Cincy Clean Coders	Mason	http://cincycleancoders.com	@cincycleancode
Software Craftsmanship	Columbus Software Craftsmanship	Columbus	http://groups.google.com/group/columbus-craftsmanship	
Software Craftsmanship	Hudson Software Craftsmanship Group	Hudson	http://hudsonsc.com/	@hudsonsc
Software Development	Bitslingers	Mason	http://www.cinnug.org	
Software Development	Cincinnati Programmers Guild	Mason	http://cincypg.org/	@cincypg
Software Development	Girl Develop It Columbus	Columbus	http://www.meetup.com/girldevelopitcbus/	@gdicbus
SQL	CincySQL	Mason	http://www.cincysql.org	
SQLServer	CBusPASS	Columbus	http://columbus.sqlpass.org/	@daveschutz
SQLServer	Ohio North SQL Server Users Group	Independence	http://ohionorth.sqlpass.org/Home.aspx	
Windows Phone Development	Central Ohio Windows Phone User Group	Columbus	http://cowpug.org/	#COWPUG
Windows PowerShell	Central Ohio PowerShell Users Group	Columbus	http://powershellgroup.org/central.oh	
Windows Presentation Foundation & Silverlight	Cleveland WPF User Group	Independence	http://www.clevelandwpf.info/	@SamNasr
WordPress	Columbus WordPress Meetup Group	Columbus	http://www.meetup.com/wp-columbus/	
Graph Database Dayton	Graph Databases	Dayton	http://www.meetup.com/graphdb-dayton/photos/7369342/109482982/	