www.phparch.com

November 2018
Volume 17 - Issue 11

Generics and Project Success

Free Sample Article

The Case for Generics in PHP

How to Knock Down Any Project in Ten Steps

Maintaining Laravel Applications

Getting Started With Php? Let's Start the Right Way!

ALSO INSIDE

The Dev Lead Trenches: The Talk

The Workshop: Producing Packages, Part Two

Education Station: The Day the Internet Died

Community Corner: Leveling up

Security Corner: Five Risks to Look for In a Code Review

finally{}: The Seven Deadly Sins of Programming: Sloth

AUTOMATTIC

Building a new web

Join us!

Learn more and apply at
automattic.com/jobs

WordPress.com Jetpack WooCommerce Akismet

Longreads Simplenote Gravatar VIP

Square Developer

Take online payments with Square.

Checkout our our PHP SDK and integrations with Magento, WooCommerce, Drupal Cart, and more.

square.com/developer

The Case for Generics in PHP

Chris Holland

In 2016, Ben Scholzen and Rasmus Schultz published their *PHP RFC: Generic Types and Functions*[1], aka *The Generics RFC.*

Having worked with generics in other languages, I was very grateful and thrilled to come across this RFC, as I could immediately see the tremendous benefits this would bring to the PHP ecosystem.

With this said, the benefits of generics can be difficult to understand without having worked with them.

To elucidate their merit, we will look at how they might fit within the evolution of PHP's type system to put us in a position to write more robust software:

"How can I signal that my method will return a Collection of User objects? And why would I want to do that?"

Generics In Other Languages

Generics were introduced in Java J2SE 5.0[2] in 2004. An example would look like this:

```
List<String> v = new ArrayList<String>();
v.add("test");
// compilation-time type error
Integer i = v.get(0);
```

Generics were introduced in C# 2.0[3] in 2005.

With this said, generic programming[4] can be traced as far back as 1973 with the ML programming language.

1. Languages supporting generics include Ada, C#, Delphi, Eiffel, F#, Java, Rust, Swift, TypeScript, and Visual Basic .NET.
2. Languages supporting parametric polymorphism include ML, Scala, Haskell, and Julia.
3. C++ and D support templates.

PHP'S Type System

Consider the example in Listing 1 in PHP 5.6.

> *Defining type system is no easy task. For example, Ruby describes its type system as being "dynamic" and "loosely typed," but they don't allow you to put return types on methods, or types on method arguments, which is really like not having a type system at all.*

> *What are currently referred to in PHP as type hints are more than that: they're contracts enforced by the compiler; it's just that they are optional. But if you do use them, they are enforced in a helpful way. The very bare minimum requirement for even claiming to have a type system is to be able to define types on method arguments and method returns, which we have in PHP 7.*

A type system allows us to enforce the correctness of a program, in terms of specifying the accepted inputs and outputs.

From the above example, the following will fail:

```
$registration = new UserRegistrationService(new Duck());
// Fails. I passed a Duck when it expected a UserRepository
// PHP will clearly signal to me that I passed a Duck
// to something that expected a UserRepository
```

Why does this matter? If I were to remove `UserRepository` from the `UserRegistrationService` constructor's method signature, I would not know I did something wrong until later when invoking the `createUser()` method. It would then try to call some method named `saveUser` on a `Duck`, and this method may or may not exist, but we won't know that until the code is executed.

Listing 1

```
1. <?php
2.
3. public class UserRegistrationService
4. {
5.     private $userRepo;
6.
7.     public function __construct (UserRepository $userRepo)
8.     {
9.         $this->userRepo = $userRepo;
10.    }
11.
12.    public function createUser($firstName, $lastName)
13.    {
14.        $newUser = new User($firstName, $lastName);
15.        return $this->userRepo->saveUser($newUser);
16.    }
17. }
```

1 *PHP RFC: Generic Types and Functions:* https://wiki.php.net/rfc/generics

2 *Java J2SE 5.0:* https://phpa.me/wikip-java-generics

3 *C# 2.0:* https://phpa.me/microsoft-c-generics

4 *generic programming:* https://phpa.me/wikip-generic-programming

If a system is going to fail, it is more helpful for the system to fail earlier than later. Leveraging a language's type system puts us in a position to do just that. Before we run our code, static analyzers or even our code editors could highlight potential errors.

Similar advantages can be derived from method return types and class property types.

The evolution of PHP's type system could be summarized as follows:

1. PHP 5 gave us optional types for classes, interfaces, and callables on method arguments.
2. PHP 7 introduced scalar types on method arguments and gave us optional method return types.
3. PHP 7.4 is currently slated to support optional types on class properties

It is important to note, as of this writing, PHP's type system has always been optional, and will likely continue to be. This reduces compatibility issues while promoting adoption. You can gradually add types to function signatures and then, hopefully, remove a lot of boilerplate code from your functions which checks argument types.

With PHP 7.4, the earlier example might look like Listing 2.

The Need for Generics

So what's the deal with Generics? Let's start with another example.

```
class UserLookupService
{
   // ...
   public function getUsersByDepartmentName(
      string $departmentName
   ) {
       return $this->userRepo
                   ->getByDepartment($departmentName);
   }
}
```

As the method name indicates, we want to return a list of users. How would I signal this in the method signature? As of PHP 7, there isn't a way for me to natively signal "This method must return an array of user objects". I can do it with an annotation and hope my IDE will enforce this behavior throughout the system, see Listing 3.

Collections of Things

There are richer ways to express such collections beyond a primitive array. The Doctrine ORM provides us with an `ArrayCollection` class. It exposes a set of helpful methods to iterate through and manipulate members of a `Collection`.

Without generics, if I wished to enforce homogenous `ArrayCollections`, I might resort to a less-than-elegant hack as in Listing 4.

Listing 2

```
<?php

namespace Foo;

class UserRegistrationService
{
    // Class Property Type: Future PHP 7.4
    private UserRepository $userRepo;

    // Using Argument Type (class): Since PHP 5
    public function __construct(UserRepository $userRepo) {
        $this->userRepo = $userRepo;
    }

    // Uses Method Return Type: Since PHP 7
    public function createUser(string $firstName,
                               string $lastName): User {
        $newUser = new User($firstName, $lastName);
        return $this->userRepo->saveUser($newUser);
    }
}
```

Listing 3

```
class UserLookupService
{
    // ...
    /**
    * @return User[]  //<-- this is me using an annotation as a crutch.
    */
    public function getUsersByDepartmentName(string $departmentName)
    {
        return $this->userRepo
                    ->getByDepartment($departmentName);
    }
}
```

Listing 4

```
<?php

class UserArrayCollection extends ArrayCollection
{
    public function addUser(User $user) {
        parent::add($user);
    }

    public function nextUser(): User {
        return parent::next();
    }
}
```

I could also make a `DuckArrayCollection` (Listing 5).

Every time I wish to have an `ArrayCollection` of homogenous items, I have to make a new `ArrayCollection` class to enforce this contract. This would allow me to write the code in Listing 6.

Generics to the Rescue

My ultimate wish would be to signal that the method would return "an `ArrayCollection` of `User` objects" without creating a new child class for every type of object I might return. It's an `ArrayCollection` whose members are only allowed to be instances of the `User` class. Here's a possible evolution of the previous example.

```
public class UserLookupService
{
   public function getUsersByDepartmentName(
      string $departmentName ) : ArrayCollection[User]
   {
       return $this->userRepo
                   ->getByDepartment($departmentName);
   }
}
```

What I wish I could do is this:

```
class UserLookupService
{
   // ...
   public function getUsersByDepartmentName(
      string $departmentName) : User[] //<-- Not valid
   {
       return $this->userRepo
                   ->getByDepartment($departmentName);
   }
}
```

The brackets syntax is common in many languages to designate arrays, but this isn't what we're working with here. In the end, what we are trying to define is a composite type. It's an object of a given type—`ArrayCollection`—made of objects of another type—`User`.

The Generics RFC's proposed syntax for this would be:

```
ArrayCollection<User>
```

Applying Generics

The collections examples look tedious. Generics would allow us to remove this tedium, as in Listing 7.

The letter `T` acts as a placeholder for whichever type I need to bind my `GenericArrayCollection`, and this binding happens at instantiation.

```
$duckCollection = new GenericArrayCollection<Duck>();
```

From here on, the `add` method will only accept a `Duck`. And the `next` method is guaranteed to only ever return an instance of `Duck`.

Listing 5

```
1. <?php
2.
3. class DuckArrayCollection extends ArrayCollection
4. {
5.    public function addDuck(Duck $duck) {
6.       parent::add($duck);
7.    }
8.
9.    public function nextDuck(): Duck {
10.      return parent::next();
11.   }
12. }
```

Listing 6

```
1. // type-specific collection class for "User"
2. $users = new UserArrayCollection();
3.
4. // this is fine
5. $users->addUser(new User());
6.
7. // this breaks, as intended
8. $users->addUser(new Duck());
9.
10. // type-specific collection class for "Duck"
11. $ducks = new DuckArrayCollection();
12.
13. // this is fine
14. $ducks->addDuck(new Duck());
15.
16. // this breaks, as intended
17. $ducks->addDuck(new User());
```

Listing 7

```
1. <?php
2.
3.
//don't extend ArrayCollection, re-implement from scratch
4. class GenericArrayCollection<T>
5. {
6.    public function add(T $element) {
7.       $this->elements[] = $element;
8.       return true;
9.    }
10.
11.   public function next(): T {
12.      return next($this->elements);
13.   }
14. }
```

Based on the above examples, from this single `GenericArrayCollection` class, I can now write Listing 8.

With this generics syntax, our `UserLookupService` class could now look like Listing 9.

With the above example, we are now guaranteeing the method would always return a collection of `User` objects.

This would help IDEs enforce proper behavior in their static analysis while providing deeper auto-completion.

Without even using an IDE, PHP would throw helpful errors as soon as it parses the `getUsersByDepartmentName` method. Should its contents attempt to return anything but a collection of `User` objects, PHP would immediately tell us that what we're trying to do is incompatible with the method signature.

This is preferable to encountering an error upon system execution while running logic invoking the `getUsersByDepartmentName` method. It would break in unforeseen ways when faced with a stray `Duck` within what it otherwise expects to be a collection of `Users`.

Enforcing homogeneous collections tends to be the first and most popular use-case for generics, but they are applicable to unlimited use-cases, some of which include:

- HashMap<K, V> as an object-oriented abstraction-layer for PHP's associative arrays: https://phpa.me/javase8-hashmap
- object caching APIs
- worker queues

In these use-cases, systems will be made more robust from clearly signaling what types of objects are to be used as inputs and outputs, such that, a returned cached object might be contractually obligated to be a `User`, and not a `Duck`, for example.

Beyond those examples, we might organically come across a need to leverage generics whenever we find ourselves using `mixed` as the return type, or the argument type of a method.

Typically, our intention is a given instance of our class should only interact with objects of the same type. Our class might not care as to what that specific type might be, it just knows throwing mixed types at the same instance would result in disaster. Generics help re-enforce this expectation.

Listing 8

```
1. //generic-type collection class
2. $users = new GenericArrayCollection<User>();
3.
4. //this is fine
5. $users->add(new User());
6.
7. //this breaks, as intended
8. $users->add(new Duck());
9.
10. //$nextUser is guaranteed to be a User object.
11. $nextUser = $users->next();
12.
13. //generic-type collection class
14. $ducks = new GenericArrayCollection<Duck>();
15.
16. //this is fine
17. $ducks->add(new Duck());
18.
19. //this breaks, as intended
20. $ducks->add(new User());
21.
22. //$nextDuck is guaranteed to be a Duck object
23. $nextDuck = $ducks->next();
```

Listing 9

```
1. public class UserLookupService
2. {
3.     public function getUsersByDepartmentName(string $departmentName)
4.         : ArrayCollection<User>
5.     {
6.         return $this->userRepo
7.                     ->getByDepartment($departmentName);
8.     }
9. }
```

Is PHP Turning Into Java? (Or Language X)

As mentioned above, the concept of Generic Programming predates Java by decades, and so do type systems[5] in general.

Learning the strengths of other languages and adopting some of their more useful features will help keep PHP a competitive ecosystem while making itself more attractive to developers from other ecosystems.

For example, one of my first hires in our Austin office is a Software Engineer with little previous exposure to PHP, but had significant experience building systems with C#, using Test-Driven Development and applying best practices of object-oriented design. Working in PHP wasn't much of a hurdle, because in the end, it all came down to familiar constructs: interfaces, abstract classes, classes, and private/protected/public member variables and methods.

Generics are just one more construct leveraged by many software engineers across various ecosystems, who might be attracted to another language with a well-evolved type system.

5 *type systems:* https://en.wikipedia.org/wiki/Type_system

Tinkering with Generics

If you wish to try your hand at learning how to use Generics even before a proper syntax is introduced into PHP, you might consider tinkering with Daniel Labarge's experimental library to achieve roughly-similar behavior: https://github.com/artisansdk/generic

While the syntax and error signaling won't be as robust and powerful as natively-implemented generics, Labarge's framework should help illustrate generics behavior and some of its benefits.

> *Head over to GitHub to follow the current Generics RFC[6].*

In Conclusion

The Generics RFC may have been a bit ahead of its time to truly get the traction it deserved. Generic types only make sense in languages where a type system is pervasively used, and their type system is fully-featured.

With the adoption of typed properties, PHP 7.4's roadmap is about to get us there.

As PHP's type system evolves and matures, and its adoption increases in modern software systems, generics would provide a richer vocabulary to build more robust systems with reduced code-duplication.

6 *Generics RFC: https://phpa.me/php-generics-rfc*

Chris Holland leads a small Software Engineering Team at an HR company. Throughout a career spanning more than 20 years, Chris has held Sr. Engineering and Leadership roles for small and large successful publicly-traded companies such as EarthLink and Internet Brands, serving business models across Content, Commerce, Travel, and Finance on a wide variety of technology stacks including PHP/LAMP, Java/J2EE and C#/.Net, catering to audiences over 100 million monthly visitors. @chrisholland

Related Reading

- *PHP 7.3 is On Track!* by Damien Seguy. September 2018. https://phparch.com/magazine/2018-2/september/
- *Testing Strategy With the Help of Static Analysis* by Ondrej Mirtes. April 2018. https://phparch.com/magazine/2018-2/april/
- *Evolving PHP* by Chris Pitt. March 2018. https://phparch.com/magazine/2018-2/march/

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]