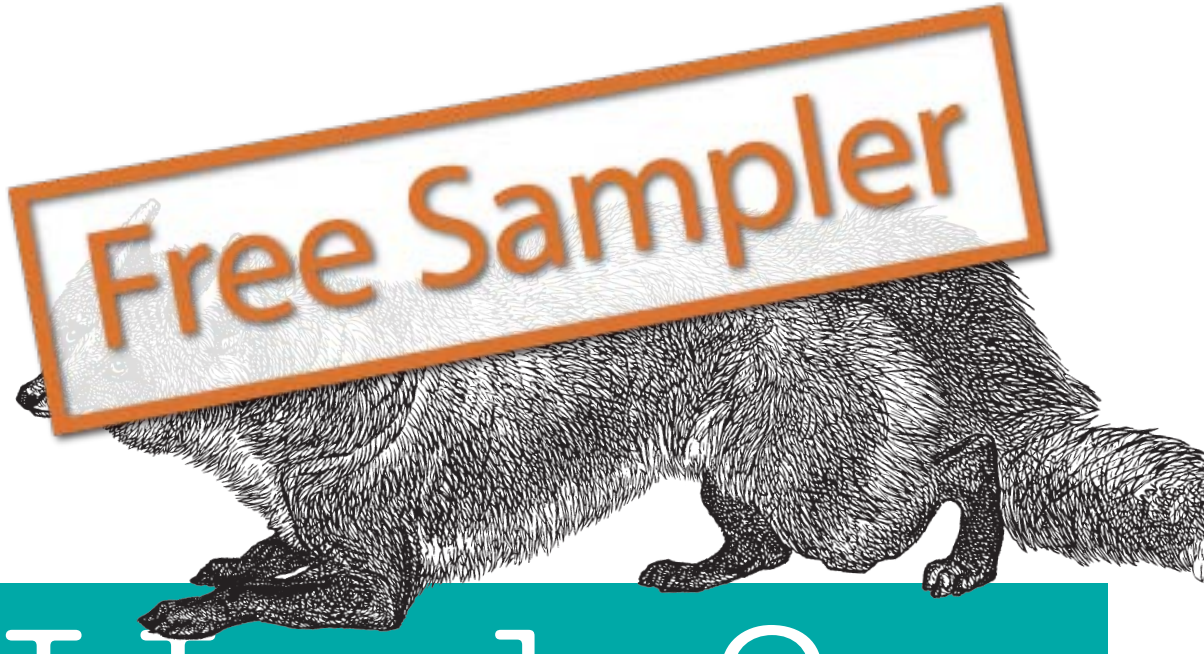
O'REILLY®




# Hack & HHVM

PROGRAMMING PRODUCTIVITY WITHOUT BREAKING THINGS

Owen Yamauchi

O'REILLY®

# Hack and HHVM

How can you take advantage of the HipHop Virtual Machine (HHVM) and the Hack programming language, two new technologies that Facebook developed to run their web servers? With this practical guide, Owen Yamauchi—a member of Facebook's core Hack and HHVM teams—shows you how to get started with these battle-tested open source tools.

You'll explore static typechecking and several other features that separate Hack from its PHP origins, and learn how to set up, configure, deploy, and monitor HHVM. Ideal for developers with basic PHP knowledge or experience with other languages, this book also demonstrates how these tools can be used with existing PHP codebases and new projects alike.

- Learn how Hack provides static typechecking while retaining PHP's flexible, rapid development capability
- Write typesafe code with Hack's generics feature
- Explore HHVM, a just-in-time compilation runtime engine with full PHP compatibility
- Dive into Hack collections, asynchronous functions, and the XHP extension for PHP
- Understand Hack's design rationale, including why it omits some PHP features
- Use Hack for multitasking, and for generating HTML securely
- Learn tools for working with Hack code, including PHP-to-Hack migration

**Owen Yamauchi** is a software engineer at Facebook, where he works on the Hack and HHVM teams. Before joining the company in 2009, he worked as a software engineer at Apple and served as an intern at VMware.

"Hack is remarkable not only for the elegance and power of its type system and concurrency model, but because it provides existing PHP applications a thoughtful, iterative migration strategy that can be executed at scale. Yamauchi's survey of the language and its runtime is clear, expert, and essential. Highly recommended."

—Ori Livneh
Principal Performance Engineer,
Wikimedia Foundation

PHP

US $34.99 CAN $40.99
ISBN: 978-1-491-92087-9


Twitter: @oreillymedia
facebook.com/oreilly

**Hack and HHVM**
by Owen Yamauchi



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editor:** Allyson MacDonald
**Production Editor:** Melanie Yarbrough
**Copyeditor:** Rachel Head
**Proofreader:** Jasmine Kwityn
**Indexer:** Ellen Troutman-Zaig
**Interior Designer:** David Futato
**Cover Designer:** Ellie Volkhausen
**Illustrator:** Rebecca Demarest

September 2015: First Edition

**Revision History for the First Edition**
2015-09-02: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491920879* for release details.



978-1-491-92087-9

[LSI]

# Table of Contents

CHAPTER 1

# Typechecking

The typechecker is the flagship feature of Hack. It analyzes Hack programs statically (i.e., without running them) and checks for many different kinds of errors, which prevents bugs at an early stage of development and makes code easier to read and understand. To enhance the typechecker's ability to do this, Hack allows programmers to explicitly annotate the types of some values in their programs: function parameters, function return types, and properties. The typechecker will infer the rest.

The choice between statically typed languages and dynamically typed languages is endlessly debated among programmers. It's often presented as a choice between the robustness of static typing and the flexibility of dynamic typing. The philosophy of Hack rejects this as a false dichotomy. Hack retains the flexible, rapid-development character of PHP, a dynamically typed language, while adding a layer of robust, sophisticated typechecking.

In this chapter, we'll see why you should use the typechecker, how to use it, and how to write type annotations for it.

## Why Use the Typechecker?

The argument in favor of Hack typechecking sounds similar to the argument often used in favor of statically typed languages. The typechecker is able to look for mistakes without running the program, so it can catch problems even with codepaths that aren't run during testing. Because it doesn't need to run the program, it catches problems earlier in development, which saves development time. Static analysis capability makes refactoring easier, as it can ensure that there are no breakages at module boundaries.

In the classic debate, the disadvantage that supposedly accompanies these features is a drag on development speed. Before you can run your program, you have to wait for it

to compile, and depending on the language and the size of the program, that can take a long time. You also have to write out types everywhere, making your code more verbose and harder to change.

These downsides aren't present in Hack, for two reasons. First, the typechecker is designed for instant feedback, even when working in very large codebases. It uses a client/server model: the typechecking server runs in the background and monitors the filesystem for changes. When you edit a file, the server updates its in-memory analysis of your codebase. By the time you're ready to run your code, the analysis is already done; the client simply queries the server and displays results almost instantaneously. It can easily be integrated into text editors and IDEs, giving you feedback in real time.

Second, Hack type annotations are designed to be gradual. You can use as many or as few as you want. Type-annotated code can interoperate seamlessly with non-annotated Hack code and with PHP code. In addition, you don't annotate local variables; the typechecker infers their types from their surroundings.

## Setting Up the Typechecker

Before we look at the syntax and semantics of Hack type annotations, we'll get the typechecker set up.

The first thing you need is an *.hhconfig* file. As well as holding project-wide configuration settings, this file serves to mark the top-level directory of your codebase, so the typechecker knows which files to include in its analysis.

For now, we don't need any configuration; our *.hhconfig* file can just be empty. So, navigate to the top-level directory of your project, and do this:

```
$ touch .hhconfig
$ hh_client
```

Running `hh_client` first checks for a running `hh_server` process. If there isn't one, the client will start one, so you should never have to start one yourself. The server will find the *.hhconfig* file and analyze every Hack file it finds in the directory containing that file and all directories below it.

A Hack file is one whose contents start with `<?hh`.[1] This is an adaptation of PHP's "opening tag" syntax. After the `<?hh` at the beginning (possibly supplemented by a *mode*, as described in "Typechecker Modes" on page 14), the rest of the file is Hack

1 A Hack file is also allowed to start with a shebang line like `#!/usr/bin/hhvm`, but the `<?hh` must be the next non-blank line.

code. Unlike in PHP, the closing tag `?>` is not valid in Hack; you can't use Hack with PHP's templating-language syntax.

Filename extensions are irrelevant: it's fine to name Hack files with the extension *.php*, although *.hh* is also conventional.

Once the typechecking server is started, if you have no Hack files in your project (i.e., all of your code is inside `<?php` tags instead of `<?hh`), running `hh_client` should simply print `No errors!`. This is because the typechecker only looks at Hack files; it doesn't do anything with PHP files.

## Autoload Everything

One key assumption that the typechecker makes is that your project is set up so that any class, function, or constant in your codebase can be used from anywhere else in the codebase. It makes no attempt to analyze `include` or `require` statements to make sure that the right files have been `included` or `required` by the time their contents are used. Instead, it assumes that you have autoloading set up.

This both sidesteps a difficult static analysis problem and reflects modern best practice. "Autoload everything" is the approach taken by Composer, a popular package manager for PHP and Hack. Note that autoloading isn't mandatory—you can write your code using `require` and `include`, and the typechecker won't complain—but it's strongly recommended, because the typechecker won't protect you from missing `require` or `include` statements.

PHP provides autoloading for classes, and HHVM supports this, through both `__autoload()` and `spl_autoload_register()`. HHVM provides an additional feature that allows autoloading for functions and constants in both PHP and Hack, plus autoloading for type aliases (see "Type Aliases" on page 60) in Hack only. See "Enhanced Autoloading" on page 73 for full details on the HHVM-specific API.

## Reading Error Messages

The typechecker's error messages are designed to be both detailed and easy to understand. Here's some example code with an error:

```
<?hh
function main() {
  $a = 10;
  $a[] = 20;
}
```

We'll put this in a file called *test.hh* and run the typechecker:

```
$ hh_client
/home/oyamauchi/test.hh:4:3,6: an int does not allow array append (Typing[4006])
  /home/oyamauchi/test.hh:3:8,9: You might want to check this out
```

Each line shows the full path to the file with the error, followed by the line number and the column numbers where the erroneous code starts and ends. The first error message line explains what the actual problem is—"an `int` does not allow array append"—and gives a number that uniquely identifies this error message (see "Silencing Typechecker Errors" on page 80 to find out how this is used). The line and column numbers are pointing to the code `$a[]`.

The next line of the error message is indented, to show that it's not a separate error but is elaborating on the previous line. It explains why the typechecker thinks `$a` is an `int`: it's pointing to the code `10`, which gets assigned to `$a`.

# Type Annotation Syntax

This section explains the syntax for the three places where you can put type annotations. We haven't seen the full range of type annotations that Hack supports yet—that will be covered in "Hack's Type System" on page 6—but for now, all you need to know is that `int` and `string` are valid type annotations.

The three places where you can put type annotations are on function return types, function parameters, and properties.

## Function Return Types

The syntax for function return types is the simplest. After the closing parenthesis of a function's parameter list, add a colon and a type name. You can do this with functions and methods, as well as body-less method declarations in interfaces and abstract classes. For example:

```
function returns_an_int(): int {
  // ...
}
function returns_a_string(): string {
  // ...
}
```

Whitespace is allowed between the closing parenthesis and the colon. It's common to put a newline between them in function signatures that are too long to fit on one line.

Closures can also have their return types annotated:

```
$add_one = function ($x): int { return $x + 1; };
$add_n = function ($x): int use ($n) { return $x + $n; };
```

This syntax is compatible with the return typehint syntax that will be released in PHP 7, except for the case of closures with lists of captured variables. In PHP 7, the return typehint goes after the list of captures, but in Hack, it goes after the list of parameters.

## Function Parameters

Annotating function parameters uses exactly the same syntax as PHP uses for parameter typehints—just put the type name before the parameter name:

```
function f(int $start, string $thing) {
  // ...
}
```

Default arguments are supported as usual, but of course the default value must satisfy the type annotation. In regular PHP, there is a special allowance for a default value of `null` for a typehinted parameter, so that this is valid:

```
function f(SomeClass $obj = null) {
  // ...
}
```

This is *not* valid in Hack—it conflates the concept of an optional argument with that of a required argument that allows a placeholder value. In Hack, you can express the latter by making the parameter type nullable (see "Hack's Type System" on page 6).

### Parameters Versus Arguments

These terms are often used interchangeably in casual talk among programmers, but they aren't the same thing. The difference between them is the same as the difference between variables and values. Parameters are variables, and arguments are the values that get assigned to parameters when a function is called. Consider this code:

```
function add_one($x) {
  return $x + 1;
}

echo add_one(10);
```

`$x` is a parameter of the function `add_one()`. `10` is an argument that gets assigned to the parameter `$x`.

We say that a function *has* parameters, but it's also correct to say that it *takes* arguments, because you pass arguments to a function when you call it.

#### Variadic functions

A variadic function is one that can take a variable number of arguments. In PHP, all functions are implicitly variadic; passing a function more arguments than it has parameters doesn't result in an error, and any function can access all arguments that were passed to it using the built-in functions `func_get_args()`, `func_get_arg()`, and `func_num_args()`.

In Hack, by contrast, passing excess arguments to a function is an error, unless the function is explicitly declared as variadic. The Hack syntax for making a function variadic is to put `...` as the last argument in the function signature. Within such a function, you can access the arguments with `func_get_args()`, `func_get_arg()`, and `func_num_args()`, the same way as in PHP:

```
function log_error(string $format, ...) {
  $varargs = func_get_args();
  // ...
}
```

The variadic arguments are allowed to be of any type. The first argument to `log_error()` must be a string, but the subsequent arguments can be of any type and the typechecker will accept it.

## Properties

In the declaration of a property (either static or non-static), the type annotation goes immediately before the property name:

```
class C {
  public static int $logging_level = 2;
  private string $name;
}
```

Initial values are supported (like `2` for `$logging_level` in the example), and the initial value must satisfy the type annotation.

Initialization of properties with type annotations actually has several more rules, to avoid situations where code can access a property that hasn't been initialized. See "Property Initialization" on page 20 for details.

# Hack's Type System

Hack provides a multitude of powerful ways to describe types. It builds on PHP's basic type system of booleans, integers, strings, arrays, etc., and adds many new ways to combine them or make them more expressive:

*Primitive types*

These are the same as PHP's primitive types: `bool`, `int`, `float`, `string`, `array`, and `resource`. All these are valid Hack type annotations.

In PHP, there are additional names for these types: `boolean`, `integer`, `real`, and `double`. These are *not* valid in Hack. The six mentioned above are the only acceptable primitive types in Hack.

There are two other types that express a simple combination of primitive types: `num`, which is either an integer or a float; and `arraykey`, which is either an integer or a string.

*Object types*
: The name of any class or interface—built-in or non-built-in—can be used in a type annotation.

*Enums*
: Enums are described more fully in Chapter 3. For our purposes here, it's enough to know that an enum gives a name to a set of constants. The name of an enum can be used as a type annotation; the only values that satisfy that annotation are the constants that are members of the enum.

*Tuples*
: Tuples are a way to bundle together a fixed number of values of possibly different types. The most common use for tuples is to return multiple values from a function.

  The syntax for tuple type annotations is simply a parenthesis-enclosed, comma-separated list of types (which may be any of the other types in this list, except `void`). The syntax for creating a tuple is identical to the `array()` syntax for creating arrays, except that the keyword `array` is replaced by `tuple`, and keys are not allowed.

  For example, this function returns a tuple containing an integer and a float:

```
function find_max_and_index(array<float> $nums): (int, float) {
  $max = -INF;
  $max_index = -1;
  foreach ($nums as $index => $num) {
    if ($num > $max) {
      $max = $num;
      $max_index = $index;
    }
  }

  return tuple($max_index, $max);
}
```

  Tuples behave like a restricted version of arrays. You can't change a tuple's set of keys: that is, you can't add or remove elements. You can change the values in a tuple, as long as you don't change their type. You can read from a tuple with array-indexing syntax, but it's more common to unpack them with list assignment instead of reading individual elements.

  Under the hood, tuples really are arrays: if you pass a tuple to `is_array()`, it will return `true`.

`mixed`
: `mixed` means any value that can possibly exist in a Hack program, including `null`.

`void`
: `void` is only valid as a function return type, and it means that the function returns nothing. (In PHP, a function that "returns nothing" actually has a return value of `null`, but in Hack, it's an error to use the return value of a function returning `void`.)

  `void` is included within `mixed`. That is, it's legal for a function with return type `mixed` to return nothing.

`this`
: `this` is only valid as a method return type—it's not a valid return type for a bare function. It signifies that the method returns an object of the same class as the object that the method was called on.

  The purpose of this annotation is to allow chained method calls on classes that have subclasses. Chained method calls are a useful trick. They look like this:

```
$random = $rng->setSeed(1234)->generate();
```

  To allow for this, the class in question has to return `$this` from methods that have no logical return value, like this:

```
class RNG {
  private int $seed = 0;

  public function setSeed(int $seed): RNG {
    $this->seed = $seed;
    return $this;
  }

  // ...
}
```

  In this example, if `RNG` has no subclasses, you can use `RNG` as the return type annotation of `setSeed()`, and there will be no problems. The trouble begins if `RNG` has subclasses.

  The typechecker will report an error in the following example. Because the return type of `setSeed()` is `RNG`, it thinks that the call `$rng->setSeed(1234)` returns a `RNG`, and calling `generateSpecial()` on a `RNG` object is invalid; that method is only defined in the subclass. The more specific type of `$rng` (which the typechecker knows is a `SpecialRNG`) has been lost:

```
class SpecialRNG extends RNG {
  public function generateSpecial(): int {
    // ...
```

```
  }
}

function main(): void {
  $rng = new SpecialRNG();
  $special = $rng->setSeed(1234)->generateSpecial();
}
```

The `this` return type annotation solves this problem:

```
class RNG {
  private int $seed = 0;

  public function setSeed(int $seed): this {
    $this->seed = $seed;
    return $this;
  }

  // ...
}
```

Now, when the typechecker is calculating the type returned from the call `$rng->setSeed(1234)`, the `this` annotation tells it to preserve the specific type of the expression to the left of the arrow. That way, the chained call to `generateSpecial()` is valid.

Static methods can also have the `this` return type, and in that case, it signifies that they return an object of the same class that the method was called on—that is, the class whose name is returned from `get_called_class()`. The way to satisfy this type annotation is to return `new static()`:

```
class ParentClass {
  // This is needed to reassure the typechecker that 'new static()'
  // is valid
  final protected function __construct() {}

  public static function newInstance(): this {
    return new static();
  }
}

class ChildClass extends ParentClass {
}

function main(): void {
  ParentClass::newInstance();  // Returns a ParentClass instance
  ChildClass::newInstance();   // Returns a ChildClass instance
}
```

*Type aliases*
: Described fully in "Type Aliases" on page 60, type aliases are a way to give a new name to an existing type. You can use the new name as a type annotation.

*Shapes*
: Shapes, described in "Array Shapes" on page 64, are a special kind of type alias, and their names can also be used as type annotations.

*Nullable types*
: All types except `void` and `mixed` can be made nullable by prefixing them with a question mark. A type annotation of `?int` indicates a value that can be an integer or `null`. `mixed` can't be made nullable because it already includes `null`.

*Callable types*
: Although PHP allows `callable` as a parameter typehint, Hack does not. Instead, Hack offers a much more powerful syntax that allows you to specify not only that a value is callable, but what types it takes as arguments and what type it returns.

  The syntax is the keyword `function`, followed by a parenthesis-enclosed list of parameter types, followed by a colon and a return type, with all of that enclosed in parentheses. This mirrors the syntax of type annotations for functions; it is essentially a function signature without a name and without names for the parameters. In this example, `$callback` is a function taking an integer and a string, and returning a string:

```
function do_some_work(array $items,
                      (function(int, string): string) $callback): array {
  foreach ($items as $index => $value) {
    $string_result = $callback($index, $value);
    // ...
  }
}
```

  There are four kinds of callable values that satisfy callable type annotations: closures, functions, instance methods, and static methods. Let's take a look at how to express them:

- Closures simply work as is:

```
function do_some_work((function(int): void) $callback): void {
  // ...
}

function main(): void {
  do_some_work(function (int $x): void { /* ... */ });
}
```

- To use a named function as a callable value, you have to pass the name through the special function `fun()`:

```
function do_some_work((function(int): void) $callback): void {
  // ...
}

function f(int $x): void {
  // ...
}

function main(): void {
  do_some_work(fun('f'));
}
```

The argument to `fun()` must be a single-quoted string literal. The typechecker will look up that function to determine its parameter types and return type, and treat `fun()` as if it returns a callable value of the right type.

- To use an instance method as a callable value, you have to pass the object and the method name through the special function `inst_meth()`. This is similar to `fun()` in that the typechecker will look up the named method and treat `inst_meth()` as if it returns a callable value of the right type. Again, the method name must be a single-quoted string literal:

```
function do_some_work((function(int): void) $callback): void {
  // ...
}

class C {
  public function do_work(int $x): void {
    // ...
  }
}

function main(): void {}
  $c = new C();
  do_some_work(inst_meth($c, 'do_work'));
}
```

- Using static methods is very similar: pass the class name and method name through the special function `class_meth()`. The method name must be a single-quoted string literal. The class name can be either a single-quoted string literal, or the Hack-specific construct `::class` appended to an unquoted class name:

```
function do_some_work((function(int): void): $callback): void {
  // ...
}

class C {
  public static function prognosticate(int $x): void {
    // ...
```

```
  }
}

function main(): void {
  do_some_work(class_meth(C::class, 'prognosticate'));

  // Equivalent:
  do_some_work(class_meth('C', 'prognosticate'));
}
```

At runtime, `ClassName::class` simply evaluates to `'ClassName'`.

There's another way to create a callable value that calls instance methods, which is `meth_caller()`. It creates a callable value that calls a specific method on objects you pass to it. You pass it a class name and a method name (there is a restriction that the method must have no parameters, but this will be lifted in a future version):

```
class C {
  function speak(): void {
    echo "hi!";
  }
}

function main(): void {
  $caller = meth_caller(C::class, 'speak');
  $obj = new C();
  $caller($obj);  // Equivalent to calling $obj->speak();
}
```

This is in contrast to `inst_meth()`, which bundles together a specific object and a method to call on it. `meth_caller()` is especially useful with utility functions like `array_map()` and `array_filter()`:

```
class User {
  public function getName(): string {
    // ...
  }
}

function all_names(array<User> $users): string {
  $names = array_map($users, meth_caller(User::class, 'getName'));
  return implode(', ', $names);
}
```

There is one kind of value that is callable in PHP, but isn't recognized as such by the Hack typechecker: objects with an `__invoke()` method. This may change in the future.

*Generics*

Also known as *parameterized types*, generics allow a single piece of code to work with multiple different types in a way that is still verifiably typesafe. The simplest example is that instead of simply specifying that a value is an array, you can specify that it's an array of strings, or an array of objects of class `Person`, and so on.

Generics are an extremely powerful tool, and there's quite a bit to learn about them. They're fully described in Chapter 2.

For this chapter, though, it's enough to understand the syntax for generic arrays. It consists of the keyword `array` followed by either one or two types inside angle brackets. If there's just one type inside the angle brackets, that is the type of the *values* in the array, and the keys are assumed to be of type `int`. If there are two types, the first one is the type of the keys, and the second one is the type of the values. So, for example, `array<bool>` signifies an array with integer keys mapping to booleans, and `array<string, int>` signifies an array with string keys mapping to integers. The types inside the angle brackets are called *type parameters*.

One very important thing to note is that in Hack, you can't create any values that you can't create in PHP. The underlying bits are all the same between PHP and Hack; Hack's type system just gives you ways to express interesting unions and subsets of the possible values.

More concretely, consider this code:

```
function main(): void {
  f(10, 10);
}

function f(mixed $m, int $i): void {
  // ...
}
```

Within the body of `f()`, we say that `$m` is of type `mixed` and `$i` is of type `int`, even though they're storing exactly the same bits.

Or consider this:

```
function main(): void {
  $callable = function(string $s): ?int { /* ... */ };
}
```

Although we say that `$callable` is of type `(function(string): ?int)`, under the hood, it's still just an object, like any other closure. It's not a magical "function pointer" value that is only possible in Hack, or anything like that.

In general, saying that some expression "is of type X" is a statement about what the *typechecker* knows, not about what the *runtime* knows.

## Typechecker Modes

The Hack typechecker has three different modes: strict, partial, and decl. These modes are set on a file-by-file basis, and files in different modes can interoperate seamlessly. Each file declares, in a double-slash comment on its first line, which mode the typechecker should use on it. For example:

```
<?hh // strict
```

If there is no comment on the first line (i.e., the first line is just `<?hh`), then partial mode is used.

There are several differences between the modes, and we'll see many of them as we look at the typechecker's features. Here's the general idea of each mode:

*Strict mode:* `<?hh // strict`

The most important feature of strict mode is that all named functions (and methods) must have their return types and all parameter types annotated, and all properties must have type annotations. In other words, anywhere there can be a type annotation, there must be one, with a few exceptions:

- Closures don't need their parameter types or return types annotated.
- Constructors and destructors don't need return type annotations—it doesn't make sense for them to return anything.

There are three major restrictions in strict mode:

- Using any named entity[2] that isn't defined in a Hack file is an error. This means that strict-mode code can't call into PHP code. Note that strict-mode code *can* call into partial-mode or decl-mode Hack code.
- Most code at the top level of a file results in an error. The `require` family of statements[3] is allowed, as are statements that define named entities.[4]
- Using reference assignment (e.g., `$a = &$b`), or defining a function or method that returns by reference or takes arguments by reference, results in an error.

There are a few smaller differences, too; we'll cover those as we get to them.

To take full advantage of the typechecker, you should aim to have as much of your code in strict mode as possible. Strict-mode Hack is a sound type system. That means that if 100% of your code is in strict mode, it should be impossible to

2 A named entity is a function, class, interface, constant, trait, enum, or type alias.

3 `require`, `include`, `require_once`, and `include_once`.

4 Defining constants with `const` syntax is allowed, but doing so with `define()` is not allowed.

incur a type error at runtime. This is a very powerful guarantee, and the closer you can get to achieving it, the better.

*Partial mode:* `<?hh`
: Partial mode relaxes the restrictions of strict mode. It does all the typechecking it can, but it doesn't require type annotations. In addition:

- If you use functions and classes that the typechecker doesn't see in a Hack file, there's no error. The typechecker leniently assumes that the missing entity is defined in a PHP file. See "Calling into PHP" on page 17 for details.
- Top-level code is allowed, but not typechecked. To minimize the amount of unchecked code you have, ideally you should wrap all your top-level code in a function and have your only top-level statement be a call to that function. That is, instead of this:

```
<?hh

set_up_autoloading();
do_logging();
$c = find_controller();
$c->go();
```

  Do this:

```
<?hh

function main() {
  set_up_autoloading();
  do_logging();
  $c = find_controller();
  $c->go();
}

main();
```

  Even better, put the definition of `main()` in a strict-mode file.

- References are allowed, but the typechecker essentially pretends they don't exist and doesn't try to model their behavior. In this example, after the last line the typechecker still thinks `$a` is an integer, even though it is really a string:

```
$a = 10;
$b = &$a;
$b = 'not an int';
```

  Put simply, you can use references in partial mode, but they break type safety, so it's best to avoid them.

  Even in a project written in Hack from the ground up, there are uses for partial mode. In any script or web app, there has to be some amount of top-level

code to serve as an entry point, so you'll always have at least one partial-mode file. You'll also need partial mode for access to superglobals like `$_GET`, `$_POST`, and `$argv`; we'll learn more about that in "Using Superglobals" on page 18.

*Decl mode:* `<?hh // decl`

In decl mode, code is not typechecked. All the typechecker does is read and index the signatures of functions and classes defined in the file. (There can still be errors in decl mode, for things like invalid type annotation syntax.)

The purpose of decl mode is to be a transition aid when migrating an existing PHP codebase to Hack: it provides a stepping stone between PHP and the other Hack modes. Changing a PHP file into decl-mode Hack is generally a very easy step, and has significant benefits over leaving the file as PHP. First, typechecking around calls to PHP code is very loose (see "Calling into PHP" on page 17), but calls to decl-mode Hack can be typechecked much more rigorously. Second, strict-mode Hack can't call into PHP at all, but it can call into decl-mode Hack.

If you're writing a new codebase that is 100% Hack from the beginning, you shouldn't use decl mode at all.

## Code Without Annotations

There's one type that I didn't mention in the list earlier. It's the type signified by the absence of an annotation. For example, it's the type of `$x` inside this function:

```
function f($x) {
}
```

This type doesn't have a name that you can write in code. Among the Hack team, it's referred to as "any."

The typechecker treats this type specially. It can never be involved in a type error. Every value that can possibly exist in a Hack program satisfies this type "annotation," so you can pass anything at all to the function `f()` in this example without a type error. In the other direction, a value of this type satisfies every possible type annotation, so within `f()`, you can do anything at all with `$x` without a type error.

This may sound similar to `mixed`, but there is a very important difference. Every possible value satisfies `mixed`, but a value of type `mixed` does *not* satisfy every possible type annotation. If you want to pass a value of type `mixed` to a function that expects an `int`, for example, you must either make sure it's an integer (see "Refining Mixed Types to Primitives" on page 32) or cast it.

Values of the "any" type work the same way in all Hack modes. In strict mode, you can't *write* code without annotations, but you can *call into* code without annotations,

defined in partial or decl mode. Another way to phrase the "everything that can be annotated must be annotated" restriction of strict mode is: code in strict mode may use values of this special type, but it's not allowed to produce them.

## Calling into PHP

In partial and decl modes, if you use a named entity that the typechecker doesn't see defined in any Hack file, there will be no error. (In strict mode, there will be an "unbound name" error.) This may seem like a strangely loose behavior, but its purpose is rooted in Hack's easy migration path from PHP. This allows code in Hack files to use code in PHP files: to call functions, to use constants, and to instantiate and extend classes. You are on your own in cases like this—remember, the typechecker makes no attempt at all to analyze PHP files, not even to see what functions they define.

You can also make this an error in partial mode with a configuration option. The option is called `assume_php` (as in: "assume missing entities are defined in PHP"), and it's turned on by default. You can turn it off by adding this line to your *.hhconfig* file and restarting the typechecker server with the command `hh_client restart`:

```
assume_php = false
```

If you're just starting to migrate a large PHP codebase to Hack, it will be easier if you leave `assume_php` on. Later on, as more of the codebase becomes Hack, it's a good idea to turn it off, to get the benefit of stricter checking. If you're starting a new Hack codebase, you should turn it off (i.e., set `assume_php = false`) from the very beginning.

The use of unknown functions and classes hamstrings the typechecker somewhat, as it has to make generous assumptions around them:

- Calls to unknown functions are typechecked as if they could take any number of arguments of any type, and had no return type annotation.
- Unknown constants are assumed to be of the special "any" type—as if they were the result of calling a function with no return type annotation.
- Instantiating an unknown class results in a value that is known to be an object. Any method call on an object like this is valid, and is typechecked like a call to an unknown function. Any property access on an object like this is valid too, and returns a value of the special "any" type.
- A Hack class that has *any unknown ancestor*, or uses any unknown trait, or has any ancestor that uses an unknown trait, is very similar to an unknown class. A single unknown trait or class will cripple the typechecker in the entire hierarchy it's part of. Calling any unknown method on such a class is valid, and so is accessing any unknown property.

However, if the typechecker can resolve a method call or property access to a method or property defined in Hack (even in decl mode), it will typecheck the call or access appropriately. For example:

```
class C extends SomeClassNotDefinedInHack {
  public int $known_property;

  public function known_method(string $s) {
    // ...
  }
}

function main(): void {
  $c = new C();
  $c->unknown_method();  // No error
  $c->known_method(12);  // Error: int not compatible with string

  $c->unknown_property->func();  // No error
  $c->known_property->func();    // Error: can't call method on an int
}
```

# Rules

The rules enforced by the typechecker are largely quite straightforward, and its error messages are designed to explain problems clearly and suggest solutions. There are a few cases that are more subtle, though, and this section explains them.

## Using Superglobals

Superglobals are global variables that are available in every scope, without the need for a `global` statement. There are nine of them, special-cased by the runtime:

- `$GLOBALS`
- `$_SERVER`
- `$_GET`
- `$_POST`
- `$_FILES`
- `$_COOKIE`
- `$_SESSION`
- `$_REQUEST`

- `$_ENV`

Hack's strict mode doesn't support superglobals; if you try to use one, the typechecker will say the variable is undefined. However, to write nontrivial web apps and scripts, you'll need to use them.

The simplest thing you can do is to write accessor functions in a partial-mode file, and call them from strict-mode files:

```
function get_params(): array {
  return $_GET;
}

function env_vars(): array {
  return $_ENV;
}

// ...
```

That approach doesn't contribute any type safety to your codebase, though, and it's easy to do better. With HTTP `GET` and `POST` parameters especially, you often know the type of the value you expect, and you can use this knowledge to get more strongly typed code:

```
function string_param(string $key): ?string {
  if (!array_key_exists($_GET, $key)) {
    return null;
  }
  $value = $_GET[$key];
  return is_string($value) ? $value : null;
}

// Alternative, stronger version: throw if wrong type
function string_param(string $key): ?string {
  if (!array_key_exists($_GET, $key)) {
    return null;
  }
  $value = $_GET[$key];
  invariant(is_string($value), 'GET param must be a string');
  return $value;
}
```

We'll see the `invariant()` function in more detail in "Refining Types" on page 29. For now, it's enough to know that it throws an exception if its first argument is `false`.

You can write similar accessors for other superglobals, and for other value types.

## Types of Overriding Methods

Inheritance is one of the more complex interactions between pieces of code in Hack. The complexity arises from the action-at-a-distance phenomenon that inheritance

creates. For example, if you have an object that has been type-annotated as `SomeClass` and you call a method on it, you could enter a method in any class that descends from `SomeClass`. The call still has to be typesafe, though, which means there have to be rules around the types of methods that override other methods.

In an overriding method, parameter types must be exactly the same as in the overridden method. This is mainly due to a behavior inherited from PHP. In PHP, any method that is overriding an abstract method, or a method declared in an interface, must match the overridden method's parameter types exactly. This is likely to change in future versions of Hack, to instead allow overriding methods' parameter types to be more general.

Return types, on the other hand, do not have to be the same when overriding. An overriding method may have a *more specific* return type than the overridden method. For example:

```
class ParentClass {
  public function generate(): num {
    // ...
  }
}

class ChildClass extends ParentClass {
  public function generate(): int {  // OK
    // ...
  }
}
```

Despite the changed return type, polymorphic callsites are still typesafe:

```
function f(ParentClass $obj) {
  $number = $obj->generate();
  // Even if $obj is a ChildClass instance, generate() still returns a num,
  // because ChildClass::generate() returns an int, and all ints are nums.
}
```

Overriding with a more general return type isn't valid—for example, if `ChildClass`'s version of `generate()` were declared to return `mixed`, the typechecker would report an error.

## Property Initialization

To maintain type safety, the typechecker enforces rules about how type-annotated properties are initialized, in both strict and partial modes. The overarching aim is to ensure that no property is ever read from before it is initialized to a value of the right type.

For static properties, the rule is simple: any non-nullable static property is required to have an initial value. Nullable properties without an explicit initial value are implicitly initialized to `null`.

Non-static properties have a more complex set of rules. The typechecker has to make sure that it's not possible to instantiate an object with an uninitialized non-nullable property. To that end, any non-nullable non-static property without an initial value must be initialized in the class's constructors:

```
class Person {
  private string $name;
  private ?string $address;

  public function __construct(string $name) {
    $this->name = $name;
  }
}
```

This code will pass the typechecker: the property `$name` is properly initialized, and `$address` is nullable so doesn't need to be initialized.

The typechecker will make sure that all possible codepaths through the constructor result in all properties being initialized. For this code:

```
class Person {
  private string $name;

  public function __construct(string $name, bool $skip_name) {
    if (!$skip_name) {
      $this->name = $name;
    }
  }
}
```

the typechecker will report this error:

```
/home/oyamauchi/test.php:5:19,29: The class member name is not always properly
initialized
Make sure you systematically set $this->name when the method __construct is
called
Alternatively, you can define the type as optional (?...)
 (NastCheck[3015])
```

Another component of the typechecker's enforcement of this rule is that you aren't allowed to call public or protected methods from within the constructor until after all properties are initialized. For this code:

```
class C {
  private string $name;

  public function __construct(string $name) {
    $this->doSomething();
```

```
    $this->name = $name;
  }

  protected function doSomething(): void {
    // ...
  }
}
```

the typechecker will raise this error (you would, however, be allowed to call `$this->doSomething()` *after* the assignment to `$this->name`):

```
/home/oyamauchi/test.php:6:14,18: Until the initialization of $this is over,
you can only call private methods
The initialization is not over because $this->name can still potentially be
null (NastCheck[3004])
```

You *are* allowed to call private methods in that situation, but any private methods you call will be checked to make sure they don't access potentially uninitialized properties. Non-private methods can't be checked in this way, because they may be overridden in subclasses, so it's invalid to call them in this situation. For the following code:

```
class C {
  private string $name;

  public function __construct(string $name) {
    $this->dumpInfo();
    $this->name = $name;
  }

  private function dumpInfo(): void {
    var_dump($this->name);
  }
}
```

the typechecker will raise this error (again, however, you would be allowed to call `$this->dumpInfo()` after assigning to `$this->name`):

```
/home/oyamauchi/test.php:11:21,24: Read access to $this->name before
initialization (Typing[4083])
```

Properties declared in abstract classes are exempt from these rules. However, concrete child classes will be required to initialize their ancestors' uninitialized properties. For this code:

```
abstract class Abstr {
  protected string $name;
}
class C extends Abstr {
}
```

the typechecker reports this error:

```
/home/oyamauchi/test.php:5:7,7: The class member name is not always properly
initialized
```

```
Make sure you systematically set $this->name when the method __construct is
called
Alternatively, you can define the type as optional (?...)
 (NastCheck[3015])
```

Lastly, for simple cases like the examples in this section, where the property is simply initialized with a parameter of the constructor, you should use constructor parameter promotion (see "Constructor Parameter Promotion" on page 68). It cuts down on boilerplate code, and you don't have to think about property initialization issues:

```
class C {
  public function __construct(private string $name) { }
}
```

## Typed Variadic Arguments

As we saw earlier, Hack has syntax to declare that a function is variadic:

```
function log_error(string $format, ...) {
  $args = func_get_args();
  // ...
}
```

PHP 5.6 introduced a different variadic function syntax, which has two features beyond Hack's—it packs variadic arguments into an array automatically, and it allows a typehint on the variadic parameter:

```
function sum(SomeClass ...$args) {
  // $args is an array of SomeClass objects
}
```

This syntax also exists in Hack. The typechecker supports the syntax, and typechecks calls to such functions correctly. HHVM supports the syntax too, but only *without* the type annotation. HHVM doesn't support checking the types of the variadic arguments, so it will raise a fatal error if it encounters a type annotation on a variadic parameter, to avoid giving the impression that the annotation is having an effect.

This creates a conflict. In strict mode, the Hack typechecker won't allow a parameter without a type annotation—even a variadic parameter—but HHVM won't run code that has an annotated variadic parameter.

There are two possible solutions to the conflict:

- Omit the annotation, and use partial mode.
- Omit the annotation, use strict mode, and add an `HH_FIXME[4033]` comment (see "Silencing Typechecker Errors" on page 80). This is the preferred solution, as strict mode should always be preferred over partial mode when possible.

## Types for Generators

There are three interfaces you can use when adding return type annotations to generators: `Iterator`, `KeyedIterator`, and `Generator`. All three are generic. We won't cover generics in full until Chapter 2, but we'll see some basics here.

Use the first two when you don't expect to call `send()` on the generator. Use `Iterator` when you're only yielding a value, and `KeyedIterator` when you're yielding a key as well:

```
function yields_value_only(): Iterator<int> {
  yield 1;
  yield 2;
}

function yields_key_and_value(): KeyedIterator<int, string> {
  yield 1 => 'one';
  yield 2 => 'two';
}
```

The return type annotation `Iterator<int>` means that the generator is yielding values of type `int`, and no keys. The annotation `KeyedIterator<int, string>` means that the generator is yielding keys of type `int` and values of type `string`. This is similar to array types, which we've already seen; for example, `array<int, string>` means an array whose keys are integers and whose values are strings.

If you will be calling `send()` on the generator, use the annotation `Generator`:

```
function has_send_called(): Generator<int, string, User> {
  // Empty yield to get first User
  $user = yield 0 => '';
  // $user is of type ?User

  while ($user !== null) {
    $id = $user->getID();
    $name = $user->getName();
    $user = yield $id => $name;
  }
}

function main(array<User> $users): void {
  $generator = has_send_called();
  $generator->next();

  foreach ($users as $user) {
    $generator->send($user);
    var_dump($generator->key());
    var_dump($generator->current());
  }
}
```

The return type annotation `Generator<int, string, User>` means that the generator yields `int` keys and `string` values, and expects values of type `User` to be passed to its `send()` method.

Note that the value resulting from the `yield` is not of type `User`, but rather `?User`. This is because it's always possible for the caller of the generator to call `next()` instead of `send()`, which makes the corresponding `yield` evaluate to `null`. You have to check that value against `null` before calling methods on it; see "Refining Nullable Types to Non-Nullable" on page 30 for details.

## Fallthrough in switch Statements

There's a common mistake in `switch` statements of having one case that unintentionally falls through to the next. Hack adds a rule that catches this mistake—it's an error to have a case that falls through to the next case, unless the first one is empty:

```
switch ($day) {
  case 'sun':
    echo 'Sunday';  // Error
  case 'sat':
    echo 'Weekend';
    break;
  default:
    echo 'Weekday';
}

switch ($day) {
  case 'sun':  // OK: this case falls through, but is empty
  case 'sat':
    echo 'Weekend';
    break;
  default:
    echo 'Weekday';
}
```

If the fallthrough is intentional, put the comment `// FALLTHROUGH` as the last line of the falling-through case:

```
switch ($day) {
  case 'sun':
    echo 'Sunday';
    // FALLTHROUGH
  case 'sat':
    echo 'Weekend';
    break;
  default:
    echo 'Weekday';
}
```

This requires action on the part of the programmer, which greatly reduces the chances that the fallthrough is an oversight.

# Type Inference

Type inference is central to Hack's approach to static typechecking. Like in PHP, local variables are not declared with types. However, being able to typecheck operations on locals is crucial to getting a useful amount of coverage.

Hack closes the gap with type inference. The typechecker starts with a small set of known types, from annotations and from literals, and then follows them through operators and function calls, deducing and checking types for everything downstream.

The way Hack's type inference works isn't always obvious at first glance. Let's take a look at the details.

## Variables Don't Have Types

In most statically typed languages, a local variable is given a type when it comes into existence, and the variable can only hold values of that type for its entire lifetime. This example code could be C++ or Java, and in either case, there is a type error—because `x` was declared as an `int`, it can never hold values that aren't integers:

```
int x = 10;
x = "a string";  // Error
```

This is not the case in Hack. Like in PHP, local variables are not declared in Hack. You create a local variable simply by assigning a value to it. You can assign a new value to any local variable, regardless of what type of value the variable already holds:

```
$x = 10;
$x = "a string";  // OK
```

The key difference is that in Hack, local variables don't have types. Local variables hold values, which have types.

At each point in the program, the typechecker knows what type of value each variable holds *at that point*. If it sees a new value assigned to a variable, it will update its knowledge of what type of value that variable holds.

## Unresolved Types

The fact that variables don't have types means that the typechecker needs a way to deal with code like the following:

```
if (some_condition()) {
  $x = 10;
} else {
```

```
  $x = 'ten';
}
```

This pattern is not uncommon in PHP code, and it's legal in Hack. The question, then, is: after the end of the conditional, what does the typechecker think the type of `$x` is?

The answer is that it uses an *unresolved type*. This is a construct that the typechecker uses to remember every type that `$x` *could* have. In this case, it remembers that `$x` could be an integer, or it could be a string.

After the conditional, you can do anything with `$x` that you could do with an integer *and* with a string, and you can't do anything that would be invalid for *either* an integer *or* a string. For example:

```
if (some_condition()) {
  $x = 10;
} else {
  $x = 'ten';
}

echo $x;            // OK: you can echo ints and strings
echo $x + 20;       // Error: can't use + on a string
echo $x->method();  // Error: can't call a method on an int or a string
```

Most importantly, `$x` will satisfy any type annotation that includes both integers and strings—like `arraykey` and `mixed`—and it won't satisfy anything else:

```
function takes_mixed(mixed $y): void {
}

function takes_int(int $y): void {
}

function main(): void {
  if (some_condition()) {
    $x = 10;
  } else {
    $x = 'ten';
  }

  takes_int($x);    // Error: $x may be a string
  takes_mixed($x);  // OK
}
```

This situation also commonly arises with class and interface hierarchies:

```
interface I {
}
class One implements I {
  public function method(): int {
    // ...
  }
```

```
}
class Two implements I {
  public function method(): string {
    // ...
  }
}

function main(): I {
  if (some_condition()) {
    $obj = new One();
  } else {
    $obj = new Two();
  }

  $int_or_string = $obj->method();  // OK

  return $obj;  // OK
}
```

Here, the call `$obj->method()` is valid, because both classes `One` and `Two` have a method with the right name and the right number of parameters. The type returned from the call is itself an unresolved type consisting of both possibilities: `int` or `string`.

The `return` statement is also valid, because both possibilities for `$obj` satisfy the return type annotation `I`.

We'll see unresolved types again when we discuss generics in "Unresolved Types, Revisited" on page 47.

## Inference Is Function-Local

A fundamental restriction of Hack's type inference is that when analyzing one function, it will never look at the body of another function or method. For example, suppose the following code is your entire codebase:

```
function f($str) {
  return 'Here is a string: ' . $str;
}

function main() {
  echo f('boo!');
}

main();
```

Two facts are clear to a human reader: that `$str` is always a string, and that `f()` always returns a string. However, the Hack typechecker will not infer these facts. While inferring types within `f()`, it will not go looking for callers of `f()` to find out what types of arguments they're passing. While inferring types within `main()`, it will

not go look at the body of `f()` to find out what type it returns. It will look at the signature of `f()` for a return type annotation, though, and find none, so it will treat `f()` as returning the special "any" type (see "Code Without Annotations" on page 16).

This restriction exists for performance reasons. Forcing inference in one function to stay within that function puts a strict upper bound on the amount of computation it takes to analyze one function, and by extension, an entire codebase. In computational-complexity terms, the type inference algorithm is superlinear in complexity, so it's important to give it many small inputs instead of one huge input, to keep the total running time manageable.

For large codebases—such as Facebook, the one Hack was originally designed for—this property is absolutely crucial. When the body of one function is changed (but not its signature), the typechecking server needs only to reanalyze that one function to bring its knowledge up to date, and it can do that almost instantaneously. When a function signature changes, the typechecking server reanalyzes that function and all of its callers, but not *their* callers, which puts a fairly low cap on the amount of work required.

There is one pseudoexception to this restriction: closures. Although a closure is technically a separate function from the one it's defined within, type inference on a function containing a closure is allowed to look inside the closure. Consider the following example:

```
$doubler = function ($x) { return $x + $x; };
var_dump($doubler(10));    // int(20)
var_dump($doubler(3.14));  // float(6.28)
```

Even though the closure has no annotations (which is valid even in strict mode), the typechecker can infer that the type of `$doubler(10)` is `int`—it analyzes the closure's body under the assumption that `$x` is an integer, and infers the return type because the addition operator applied to two integers results in an integer.[5] Similarly, it can infer that the type of `$doubler(3.14)` is `float`.

Incidentally, it's because type inference can look inside closures that strict mode allows closures to forgo type annotations.

## Refining Types

Suppose you have a value of type `?string`, and you want to pass it to a function that has a parameter of type `string`. How do you convert from one to the other? Or suppose you have an object that may or may not implement the interface `Polarizable`,

5 Except when it doesn't. See "Integer Arithmetic Overflow" on page 77.

and you want to call `polarize()` on it if it does. How can the typechecker know when the `polarize()` call is valid?

The task of establishing that a value of one type is also of another type is common in well-typed code. It may seem like a chore that you have to do to placate the typechecker, but this is really the key to how Hack catches mistakes early in development. This is how Hack prevents things like calling methods that don't exist, finding `null` in unexpected places, and other common annoyances of debugging a large PHP codebase.

You refine types using three constructs that the typechecker treats specially: null checks, type-querying built-in functions like `is_integer()`, and `instanceof`. When these constructs are used in control flow statements like loops and `if` statements, the type inference engine understands that this means types are different on different control flow paths.

## Refining Nullable Types to Non-Nullable

Null checks are used to refine nullable types into non-nullable types. This example passes the typechecker:

```
function takes_string(string $str) {
  // ...
}

function takes_nullable_string(?string $str) {
  if ($str !== null) {
    takes_string($str);
  }
  // ...
}
```

Inside the `if` block, the typechecker knows that `$str` is a non-nullable string, and thus that it can be passed to `takes_string()`. Note that null checks should use the identity comparison operators `===` and `!==` instead of equality comparison (`==` and `!=`) or conversion to a boolean; if you don't use identity comparison, the typechecker will issue an error.[6] The built-in function `is_null()` also works, as do ternary expressions:

```
function takes_nullable_string(?string $str) {
  takes_string($str === null ? "(null)" : $str);
  // ...
}
```

You can also use this style, where one branch of control flow is cut off:

6 This is because, for example, `null == "0"` is true, which makes the null check at least slightly nonsensical.

```
function processInfo(?string $info) {
  if ($info === null) {
    return;
  }
  takes_string($info);
}
```

The typechecker understands that the call to `takes_string()` will only be executed if `$info` is not `null`, because if it is `null`, the `if` block will be entered and the function will return. (If the `return` statement were a `throw` instead, the effect would be the same.)

Here's a slightly bigger example that demonstrates more complex control flow sensitivity:

```
function fetch_from_cache(): ?string {
  // ...
}

function do_expensive_computation(): string {
  // ...
}

function get_data(): string {
  $result = fetch_from_cache();
  if ($result === null) {
    $result = do_expensive_computation();
  }
  return $result;
}
```

At the point of the `return` statement, the typechecker knows that `$result` is a non-`null` string, so the return type annotation is satisfied. If the `if` block was entered, then a non-`null` string was assigned to `$result`; if the `if` block wasn't entered, then `$result` must have already been a non-`null` string.

Finally, Hack includes a special built-in function called `invariant()`, which you can use essentially to state facts to the typechecker. It takes two arguments—a boolean expression, and a string describing what's being asserted (for human readers' benefit):

```
function processInfo(?string $info) {
  invariant($info !== null, "I know it's never null somehow");
  takes_string($info);
}
```

At runtime, if the first argument to `invariant()` turns out to be `false`, an `InvariantException` will be thrown. The typechecker knows this and infers that in the code after the `invariant()` call, `$info` cannot be `null`, because otherwise an exception would have been thrown and execution wouldn't have reached that code.

## Refining Mixed Types to Primitives

For each primitive type, there is a built-in function to check whether a variable is of that type (e.g., `is_integer()`, `is_string()`, `is_array()`). The typechecker recognizes all of them specially, except for `is_object()`.[7] You'll often be using them on values of type `mixed`, or of a generic type.

The way you use these built-ins to give information to the typechecker is largely the same as the way you use null checks—the typechecker is control flow–sensitive, you can use `invariant()`, and so on. However, the type information these built-ins carry is more complex than just "`null` or not `null`," so there's a bit more detail in how inference works with them.

First, the typechecker doesn't remember negative information like "this value is *not* a string." For example:

```
function f(mixed $val) {
  if (!is_string($val)) {
    // $val is of type "mixed" here -- we don't remember it's not a string
  } else {
    // $val is of type "string" here
  }
}
```

In practice, this isn't much of a hindrance: there's little that could usefully be done with a value that we know is "anything but a string," other than refine its type further.

Second, the type-querying built-ins are the *only* way to refine types down to primitives. Even doing identity comparison against values of known type doesn't work:

```
function f(mixed $val) {
  if ($val === 'some string') {
    // $val is of type "mixed" here
    // Only is_string would tell the typechecker it's a string
  }
}
```

## Refining Object Types

Finally, the typechecker understands using `instanceof` to check if an object is an instance of a given class or interface. Like null checks and type-querying built-ins, the typechecker understands `instanceof` in conditional statements and in `invariant()`:

```
class ParentClass {
}
```

7 This is because `is_object()` returns `true` for resources. The lack of support for `is_object()` isn't a problem in practice, because you can't really do anything useful with an object without knowing its class.

```
class ChildClass extends ParentClass {
  public function doChildThings(): void {
    // ...
  }
}

function doThings(ParentClass $obj): void {
  if ($obj instanceof ChildClass) {
    $obj->doChildThings();  // OK
  }
}

function unconditionallyDoThings(ParentClass $obj): void {
  invariant($obj instanceof ChildClass, 'just trust me');
  $obj->doChildThings();  // OK
}
```

There are more details to cover here. Unlike null checks and the type-querying built-ins, `instanceof` deals with types that can overlap in complex ways, and the type-checker's ability to navigate them is slightly limited.

This example demonstrates the limitations—we have an abstract base class, with possibly many subclasses, some of which implement the built-in interface `Countable` and some of which don't:

```
abstract class BaseClass {
  abstract public function twist(): void;
}

class CountableSubclass extends BaseClass implements Countable {
  public function count(): int {
    // ...
  }
  public function twist(): void {
    // ...
  }
}

class NonCountableSubclass extends BaseClass {
  public function twist(): void {
    // ...
  }
}
```

Then we have a function that takes a `BaseClass`, calls `count()` on it if it's `Countable`, and then calls a method that `BaseClass` declares. This is a fairly common pattern in object-oriented codebases, albeit with interfaces other than `Countable`:

```
function twist_and_count(BaseClass $obj): void {
  if ($obj instanceof Countable) {
    echo 'Count: ' . $obj->count();
  }
```

```
  $obj->twist();
}
```

On the last line, there is a type error. This probably seems entirely unexpected, so let's go into detail about why.

The key to understanding the error is that when the typechecker sees an `instanceof` check, the information it derives from this is *exactly what the check says*, and it doesn't take inheritance hierarchies, interfaces, or anything else into account. It may even be the case that the condition is provably impossible to satisfy (e.g. if `Countable` were not implemented by `BaseClass` or any of its descendants), but the typechecker doesn't consider that.

At the beginning of the function, the typechecker thinks the type of `$obj` is `BaseClass`, because of the annotation. But then, within the `if` block, the typechecker thinks that the type of `$obj` is `Countable`—not a `BaseClass` instance that implements `Countable`; just `Countable`. It has *forgotten* that `$obj` is also a `BaseClass`.

Then we come to the part after the `if` block. Here, the type of `$obj` is an unresolved type (see "Unresolved Types" on page 26) consisting of either `BaseClass` or `Countable`. So when it sees `$obj->twist()`, it reports an error, because it thinks there are possible values of `$obj` for which the call isn't valid—ones that are `Countable` but not `BaseClass`. You, the human reader, know that this isn't possible, but the typechecker doesn't.

The workaround for this is to use a separate local variable for the `instanceof` check. This prevents the typechecker from losing type information about `$obj`, which is the root cause of the problem:

```
function twist_and_count(BaseClass $obj) {
  $obj_countable = $obj;
  if ($obj_countable instanceof Countable) {
    echo 'Count: ' . $obj_countable->count();
  }
  $obj->twist();
}
```

In all of the situations just described, the condition in the `if` statement or `invariant()` call must be just a single type query. Combining multiple type queries with logical operators like `||` isn't supported by the typechecker. For example, this is a type error:

```
class Parent {
}
class One extends Parent {
  public function go(): void {}
}
class Two extends Parent {
  public function go(): void {}
}

function f(Parent $obj): void {
  if ($obj instanceof One || $obj instanceof Two) {
    $obj->go();  // Error
  }
}
```

A good way to work around this is with interfaces. Create an interface that declares the `go()` method, make `One` and `Two` implement it, and check for that interface in `f()`.

## Inference on Properties

All our examples of inference so far have been on local variables. This is easy: the typechecker can be confident that it can see all reads and writes of local variables,[8] so it can make fairly strong guarantees when doing type inference on them.

Doing inference on properties is more difficult. The root of the problem is that, whereas local variables can't be modified from outside the function they're in, properties can. Consider this code, for example:

```
function increment_check_count(): void {
  // ...
}

function check_for_valid_characters(string $name): void {
  // ...
}

class C {
  private ?string $name;

  public function checkName(): void {
    if ($this->name !== null) {
```

8 As we've seen, the typechecker pretends that references don't exist; if you pass a local variable as a by-reference argument to a function, the typechecker assumes that it won't be changed.

```
      increment_check_count();
      check_for_valid_characters($this->name);
    }
  }
}
```

This code will *not* pass the typechecker. It will report an error:

```
/home/oyamauchi/test.php:16:34,44: Invalid argument (Typing[4110])
  /home/oyamauchi/test.php:6:37,42: This is a string
  /home/oyamauchi/test.php:11:11,17: It is incompatible with a nullable type
  /home/oyamauchi/test.php:15:7,29: All the local information about the member
  name has been invalidated during this call.
This is a limitation of the type-checker, use a local if that's the problem.
```

The error points to the call to `check_for_valid_characters()`. The error message gives a brief explanation of the problem. After the null check, the typechecker knows that `$this->name` is not `null`. However, the call to `increment_check_count()` forces the typechecker to *forget* that `$this->name` is not `null`, because that fact could be changed as a result of the call.

You, the programmer, might know that the value of `$this->name` won't change as a result of the call to `increment_check_count()`, but the typechecker can't find that out for itself—as we've seen, inference is function-local. The workaround for this is, as the error message says, to use a local variable. Copy the property into a local variable and use that instead:

```
public function checkName(): void {
  if ($this->name !== null) {
    $local_name = $this->name;
    Logger::log('checking name: ' . $local_name);
    check_for_valid_characters($local_name);
  }
}
```

You could also make the copy outside of the `if` block, and null-check the local instead. Either way, the typechecker can be sure that `$local_name` is not modified, and so it can remember its inferred non-nullable type.

## Enforcement of Type Annotations at Runtime

Even if the typechecker reports no errors in a Hack codebase, there may still be errors at runtime. The most obvious way for this to happen is through decl mode: because code in decl mode isn't typechecked, it can do things like call functions with the wrong types of arguments.

In future releases, HHVM's runtime typechecking will likely become much stricter, but for now it has only partial support for checking type annotations at runtime.

First of all, HHVM ignores property type annotations. You can assign anything you like to a type-annotated property, and HHVM won't complain.

Parameter type annotations behave just like PHP typehints: if they're violated, a catchable fatal error will be raised.[9] Return type annotations behave the same way.

You can make any parameter or return type annotation raise a warning instead of a catchable fatal error if violated, by putting an `@` before it. This is called a *soft* annotation. Soft annotations are meant solely as a transitional mechanism while adding new annotations to existing code (see "Inferring and Adding Type Annotations" on page 234). They shouldn't be used in new code, and existing hard annotations should certainly never be made soft.

In both parameter type annotations and return type annotations, some of the details of Hack type annotations are not enforced:

- Any annotation of a primitive type, object type, `num`, or `arraykey` is enforced exactly as is.
- The return type `void` is not enforced. That is, a function with return type `void` can return an actual value, and no error will occur at runtime.
- Callable type annotations are not enforced at all.
- Annotations of tuples and shapes are enforced as if they said only `array`. The inner types aren't checked.
- Annotations of enums are enforced as if they were the underlying type of the enum. At runtime, values will not be checked to make sure they're valid values of the enum.
- Generic type annotations are enforced without their type parameters. That is, an annotation of `array<string, MyClass>` is enforced as if it just said `array`. The inner types aren't checked.
- Nullable types are enforced correctly.

9 "Catchable fatal" may sound like an oxymoron. These errors do have odd behavior: the only way to "catch" them is with a user error handler, which you can set using the built-in function `set_error_handler()`.