The Lox Language 3

> *What nicer thing can you do for somebody than make them breakfast?*
>
> — Anthony Bourdain

We'll spend the rest of this book illuminating every dark and sundry corner of the Lox language, but it seems cruel to have you immediately start grinding out code for the interpreter without at least a glimpse of what we're going to end up with.

At the same time, I don't want to drag you through reams of language lawyering and specification-ese before you get to touch your text editor. So this will be a gentle, friendly introduction to Lox. It will leave out a lot of details and edge cases. We've got plenty of time for those later.

A tutorial isn't very fun if you can't try the code out yourself. Alas, you don't have a Lox interpreter yet, since you haven't built one!

Fear not. You can use mine.

Hello, Lox 3.1

Here's your very first taste of Lox:

```
// Your first Lox program!
print "Hello, world!";
```

Your first taste of Lox, the language, that is. I don't know if you've ever had the cured, cold-smoked salmon before. If not, give it a try too.

As that // line comment and the trailing semicolon imply, Lox's syntax is a member of the C family. (There are no parentheses around the string because `print` is a built-in statement, and not a library function.)

Now, I won't claim that C has a *great* syntax. If we wanted something elegant, we'd probably mimic Pascal or Smalltalk. If we wanted to go full Scandinavian-furniture-minimalism, we'd do a Scheme. Those all have their virtues.

What C-like syntax has instead is something you'll often find more valuable in a language: *familiarity*. I know you are already comfortable with that style because the two languages we'll be using to *implement* Lox—Java and C—also inherit it.

I'm surely biased, but I think Lox's syntax is pretty clean. C's most egregious grammar problems are around types. Dennis Ritchie

Using a similar syntax for Lox gives you one less thing to learn.

had this idea called "declaration reflects use", where variable declarations mirror the operations you would have to perform on the variable to get to a value of the base type. Clever idea, but I don't think it worked out great in practice.

Lox doesn't have static types, so we avoid that.

A High-Level Language 3.2

While this book ended up bigger than I was hoping, it's still not big enough to fit a huge language like Java in it. In order to fit two complete implementations of Lox in these pages, Lox itself has to be pretty compact.

When I think of languages that are small but useful, what comes to mind are high-level "scripting" languages like JavaScript, Scheme, and Lua. Of those three, Lox looks most like JavaScript, mainly because most C-syntax languages do. As we'll learn later, Lox's approach to scoping hews closely to Scheme. The C flavor of Lox we'll build in Part III is heavily indebted to Lua's clean, efficient implementation.

Lox shares two other aspects with those three languages:

Now that JavaScript has taken over the world and is used to build ginormous applications, it's hard to think of it as a "little scripting language". But Brendan Eich hacked the first JS interpreter into Netscape Navigator in *ten days* to make buttons animate on web pages. JavaScript has grown up since then, but it was once a cute little language.

Because Eich slapped JS together with roughly the same raw materials and time as an episode of MacGyver, it has some weird semantic corners where the duct tape and paper clips show through. Things like variable hoisting, dynamically bound `this`, holes in arrays, and implicit conversions.

I had the luxury of taking my time on Lox, so it should be a little cleaner.

Dynamic typing 3.2.1

Lox is dynamically typed. Variables can store values of any type, and a single variable can even store values of different types at different times. If you try to perform an operation on values of the wrong type—say, dividing a number by a string—then the error is detected and reported at runtime.

There are plenty of reasons to like static types, but they don't outweigh the pragmatic reasons to pick dynamic types for Lox. A static type system is a ton of work to learn and implement. Skipping it gives you a simpler language and a shorter book. We'll get our interpreter up and executing bits of code sooner if we defer our type checking to runtime.

After all, the two languages we'll be using to *implement* Lox are both statically typed.

Automatic memory management 3.2.2

High-level languages exist to eliminate error-prone, low-level drudgery, and what could be more tedious than manually managing the allocation and freeing of storage? No one rises and greets the morning sun with, "I can't wait to figure out the correct place to call `free()` for every byte of memory I allocate today!"

There are two main techniques for managing memory: **reference counting** and **tracing garbage collection** (usually just called **garbage collection** or **GC**). Ref counters are much simpler to implement—I think that's why Perl, PHP, and Python all started out using them. But, over time, the limitations of ref counting become too troublesome. All of those languages eventually ended up adding a full tracing GC, or at least enough of one to clean up object cycles.

Tracing garbage collection has a fearsome reputation. It *is* a little harrowing working at the level of raw memory. Debugging a GC can sometimes leave you seeing hex dumps in your dreams. But, remember, this book is about dispelling magic and slaying those monsters, so we *are* going to write our own garbage collector. I think you'll find the algorithm is quite simple and a lot of fun to implement.

In practice, ref counting and tracing are more ends of a continuum than opposing sides. Most ref counting systems end up doing some tracing to handle cycles, and the write barriers of a generational collector look a bit like retain calls if you squint.

For lots more on this, see "A Unified Theory of Garbage Collection" (PDF).

Data Types 3.3

In Lox's little universe, the atoms that make up all matter are the built-in data types. There are only a few:

- **Booleans.** You can't code without logic and you can't logic without Boolean values. "True" and "false", the yin and yang of software. Unlike some ancient languages that repurpose an existing type to represent truth and falsehood, Lox has a dedicated Boolean type. We may be roughing it on this expedition, but we aren't *savages*.

  There are two Boolean values, obviously, and a literal for each one.

  ```
  true;  // Not false.
  false; // Not *not* false.
  ```

Boolean variables are the only data type in Lox named after a person, George Boole, which is why "Boolean" is capitalized. He died in 1864, nearly a century before digital computers turned his algebra into electricity. I wonder what he'd think to see his name all over billions of lines of Java code.

- **Numbers.** Lox has only one kind of number: double-precision floating point. Since floating-point numbers can also represent a wide range of integers, that covers a lot of territory, while keeping things simple.

  Full-featured languages have lots of syntax for numbers—hexadecimal, scientific notation, octal, all sorts of fun stuff. We'll settle for basic integer and decimal literals.

  ```
  1234;  // An integer.
  12.34; // A decimal number.
  ```

- **Strings.** We've already seen one string literal in the first example. Like most languages, they are enclosed in double quotes.

```
"I am a string";
"";    // The empty string.
"123"; // This is a string, not a number.
```

As we'll see when we get to implementing them, there is quite a lot of complexity hiding in that innocuous sequence of characters.

Even that word "character" is a trickster. Is it ASCII? Unicode? A code point or a "grapheme cluster"? How are characters encoded? Is each character a fixed size, or can they vary?

- **Nil.** There's one last built-in value who's never invited to the party but always seems to show up. It represents "no value". It's called "null" in many other languages. In Lox we spell it `nil`. (When we get to implementing it, that will help distinguish when we're talking about Lox's `nil` versus Java or C's `null`.)

  There are good arguments for not having a null value in a language since null pointer errors are the scourge of our industry. If we were doing a statically typed language, it would be worth trying to ban it. In a dynamically typed one, though, eliminating it is often more annoying than having it.

Expressions 3.4

If built-in data types and their literals are atoms, then **expressions** must be the molecules. Most of these will be familiar.

Arithmetic 3.4.1

Lox features the basic arithmetic operators you know and love from C and other languages:

```
add + me;
subtract - me;
multiply * me;
divide / me;
```

The subexpressions on either side of the operator are **operands**. Because there are *two* of them, these are called **binary** operators. (It has nothing to do with the ones-and-zeroes use of "binary".) Because the operator is fixed *in* the middle of the operands, these are also called **infix** operators (as opposed to **prefix** operators where the operator comes before the operands, and **postfix** where it comes after).

One arithmetic operator is actually *both* an infix and a prefix one. The `-` operator can also be used to negate a number.

```
-negateMe;
```

All of these operators work on numbers, and it's an error to pass any other types to them. The exception is the `+` operator—you can also pass it two strings to concatenate them.

There are some operators that have more than two operands and the operators are interleaved between them. The only one in wide usage is the "conditional" or "ternary" operator of C and friends:

```
condition ? thenArm : elseArm;
```

Some call these **mixfix** operators. A few languages let you define your own operators and control how they are positioned—their "fixity".

Comparison and equality 3.4.2

Moving along, we have a few more operators that always return a Boolean result. We can compare numbers (and only numbers), using Ye Olde Comparison Operators.

```
less < than;
lessThan <= orEqual;
greater > than;
greaterThan >= orEqual;
```

We can test two values of any kind for equality or inequality.

```
1 == 2;         // false.
"cat" != "dog"; // true.
```

Even different types.

```
314 == "pi"; // false.
```

Values of different types are *never* equivalent.

```
123 == "123"; // false.
```

I'm generally against implicit conversions.

Logical operators 3.4.3

The not operator, a prefix `!`, returns `false` if its operand is true, and vice versa.

```
!true;  // false.
!false; // true.
```

The other two logical operators really are control flow constructs in the guise of expressions. An `and` expression determines if two values are *both* true. It returns the left operand if it's false, or the right operand otherwise.

```
true and false; // false.
true and true;  // true.
```

And an `or` expression determines if *either* of two values (or both) are true. It returns the left operand if it is true and the right operand otherwise.

```
false or false; // false.
true or false;  // true.
```

The reason `and` and `or` are like control flow structures is that they **short-circuit**. Not only does `and` return the left operand if it is false, it doesn't even *evaluate* the right one in that case. Conversely (contrapositively?), if the left operand of an `or` is true, the right is skipped.

I used `and` and `or` for these instead of `&&` and `||` because Lox doesn't use `&` and `|` for bitwise operators. It felt weird to introduce the double-character forms without the single-character ones.

I also kind of like using words for these since they are really control flow structures and not simple operators.

Precedence and grouping 3.4.4

All of these operators have the same precedence and associativity that you'd expect coming from C. (When we get to parsing, we'll get *way* more precise about that.) In cases where the precedence isn't what you want, you can use `()` to group stuff.

```
var average = (min + max) / 2;
```

Since they aren't very technically interesting, I've cut the remainder of the typical operator menagerie out of our little language. No bitwise, shift, modulo, or conditional operators. I'm not grading you, but you will get bonus points in my heart if you augment your own implementation of Lox with them.

Those are the expression forms (except for a couple related to specific features that we'll get to later), so let's move up a level.

Statements 3.5

Now we're at statements. Where an expression's main job is to produce a *value*, a statement's job is to produce an *effect*. Since, by definition, statements don't evaluate to a value, to be useful they have to otherwise change the world in some way—usually modifying some state, reading input, or producing output.

You've seen a couple of kinds of statements already. The first one was:

```
print "Hello, world!";
```

A `print` statement evaluates a single expression and displays the result to the user. You've also seen some statements like:

Baking `print` into the language instead of just making it a core library function is a hack. But it's a *useful* hack for us: it means our in-progress interpreter can start producing output before we've implemented all of the machinery required to define functions, look them up by name, and call them.

```
"some expression";
```

An expression followed by a semicolon (`;`) promotes the expression to statement-hood. This is called (imaginatively enough), an **expression statement**.

If you want to pack a series of statements where a single one is expected, you can wrap them up in a **block**.

```
{
  print "One statement.";
  print "Two statements.";
}
```

Blocks also affect scoping, which leads us to the next section . . .

Variables 3.6

You declare variables using `var` statements. If you omit the initializer, the variable's value defaults to `nil`.

This is one of those cases where not having `nil` and forcing every variable to be initialized to some value would be more annoying than dealing with `nil` itself.

```
var imAVariable = "here is my value";
var iAmNil;
```

Once declared, you can, naturally, access and assign a variable using its name.

```
var breakfast = "bagels";
print breakfast; // "bagels".
breakfast = "beignets";
print breakfast; // "beignets".
```

Can you tell that I tend to work on this book in the morning before I've had anything to eat?

I won't get into the rules for variable scope here, because we're going to spend a surprising amount of time in later chapters mapping every square inch of the rules. In most cases, it works like you would expect coming from C or Java.

Control Flow 3.7

It's hard to write useful programs if you can't skip some code or execute some more than once. That means control flow. In addition to the logical operators we already covered, Lox lifts three statements straight from C.

We already have `and` and `or` for branching, and we *could* use recursion to repeat code, so that's theoretically sufficient. It would be pretty awkward to program that way in an imperative-styled language, though.

Scheme, on the other hand, has no built-in looping constructs. It *does* rely on recursion for repetition. Smalltalk has no built-in branching constructs, and relies on dynamic dispatch for selectively executing code.

An `if` statement executes one of two statements based on some condition.

```
if (condition) {
  print "yes";
} else {
  print "no";
}
```

A `while` loop executes the body repeatedly as long as the condition expression evaluates to true.

```
var a = 1;
while (a < 10) {
  print a;
  a = a + 1;
}
```

I left `do while` loops out of Lox because they aren't that common and wouldn't teach you anything that you won't already learn from `while`. Go ahead and add it to your implementation if it makes you happy. It's your party.

Finally, we have `for` loops.

```
for (var a = 1; a < 10; a = a + 1) {
  print a;
}
```

This loop does the same thing as the previous `while` loop. Most modern languages also have some sort of `for-in` or `foreach` loop for explicitly iterating over various sequence types. In a real language, that's nicer than the crude C-style `for` loop we got here. Lox keeps it basic.

This is a concession I made because of how the implementation is split across chapters. A `for-in` loop needs some sort of dynamic dispatch in the iterator protocol to handle different kinds of sequences, but we don't get that until after we're done with control flow. We could circle back and add `for-in` loops later, but I didn't think doing so would teach you anything super interesting.

Functions 3.8

A function call expression looks the same as it does in C.

```
makeBreakfast(bacon, eggs, toast);
```

You can also call a function without passing anything to it.

```
makeBreakfast();
```

Unlike in, say, Ruby, the parentheses are mandatory in this case. If you leave them off, the name doesn't *call* the function, it just refers to it.

A language isn't very fun if you can't define your own functions. In Lox, you do that with `fun`.

I've seen languages that use `fn`, `fun`, `func`, and `function`. I'm still hoping to discover a `funct`, `functi`, or `functio` somewhere.

```
fun printSum(a, b) {
  print a + b;
}
```

Now's a good time to clarify some terminology. Some people throw around "parameter" and "argument" like they are interchangeable and, to many, they are. We're going to spend a lot of time splitting the finest of downy hairs around semantics, so let's sharpen our words. From here on out:

- An **argument** is an actual value you pass to a function when you call it. So a function *call* has an *argument* list. Sometimes you hear **actual parameter** used for these.

- A **parameter** is a variable that holds the value of the argument inside the body of the function. Thus, a function *declaration* has a *parameter* list. Others call these **formal parameters** or simply **formals**.

The body of a function is always a block. Inside it, you can return a value using a `return` statement.

```
fun returnSum(a, b) {
  return a + b;
}
```

If execution reaches the end of the block without hitting a `return`, it implicitly returns `nil`.

Speaking of terminology, some statically typed languages like C make a distinction between *declaring* a function and *defining* it. A declaration binds the function's type to its name so that calls can be type-checked but does not provide a body. A definition declares the function and also fills in the body so that the function can be compiled.

Since Lox is dynamically typed, this distinction isn't meaningful. A function

Closures 3.8.1

declaration fully specifies the function including its body.

See, I told you `nil` would sneak in when we weren't looking.

Functions are *first class* in Lox, which just means they are real values that you can get a reference to, store in variables, pass around, etc. This works:

```
fun addPair(a, b) {
  return a + b;
}

fun identity(a) {
  return a;
}

print identity(addPair)(1, 2); // Prints "3".
```

Since function declarations are statements, you can declare local functions inside another function.

```
fun outerFunction() {
  fun localFunction() {
    print "I'm local!";
  }

  localFunction();
}
```

If you combine local functions, first-class functions, and block scope, you run into this interesting situation:

```
fun returnFunction() {
  var outside = "outside";

  fun inner() {
    print outside;
  }

  return inner;
}

var fn = returnFunction();
fn();
```

Here, `inner()` accesses a local variable declared outside of its body in the surrounding function. Is this kosher? Now that lots of languages have borrowed this feature from Lisp, you probably know the answer is yes.

For that to work, `inner()` has to "hold on" to references to any surrounding variables that it uses so that they stay around even after the outer function has returned. We call functions that do this **closures**. These days, the term is often used for *any* first-class function, though it's sort of a misnomer if the function doesn't happen to close over any variables.

Peter J. Landin coined the term "closure". Yes, he invented damn near half the terms in programming languages. Most of them came out of one incredible paper, "The Next 700 Programming Languages".

In order to implement these kind of functions, you need to create a data structure that bundles together the function's code and the surrounding variables it needs. He called this a "closure" because it *closes over* and holds on to the variables it needs.

As you can imagine, implementing these adds some complexity because we can no longer assume variable scope works strictly like a stack where local variables evaporate the moment the function returns. We're going to have a fun time learning how to make these work correctly and efficiently.

Classes 3.9

Since Lox has dynamic typing, lexical (roughly, "block") scope, and closures, it's about halfway to being a functional language. But as you'll see, it's *also* about halfway to being an object-oriented language. Both paradigms have a lot going for them, so I thought it was worth covering some of each.

Since classes have come under fire for not living up to their hype, let me first explain why I put them into Lox and this book. There are really two questions:

Why might any language want to be object oriented? 3.9.1

Now that object-oriented languages like Java have sold out and only play arena shows, it's not cool to like them anymore. Why would anyone make a *new* language with objects? Isn't that like releasing music on 8-track?

It is true that the "all inheritance all the time" binge of the '90s produced some monstrous class hierarchies, but **object-oriented programming** (**OOP**) is still pretty rad. Billions of lines of successful code have been written in OOP languages, shipping millions of apps to happy users. Likely a majority of working programmers today are using an object-oriented language. They can't all be *that* wrong.

In particular, for a dynamically typed language, objects are pretty handy. We need *some* way of defining compound data types to bundle blobs of stuff together.

If we can also hang methods off of those, then we avoid the need to prefix all of our functions with the name of the data type they operate on to avoid colliding with similar functions for different types. In, say, Racket, you end up having to name your functions like `hash-copy` (to copy a hash table) and `vector-copy` (to copy a vector) so that they don't step on each other. Methods are scoped to the object, so that problem goes away.

Why is Lox object oriented? 3.9.2

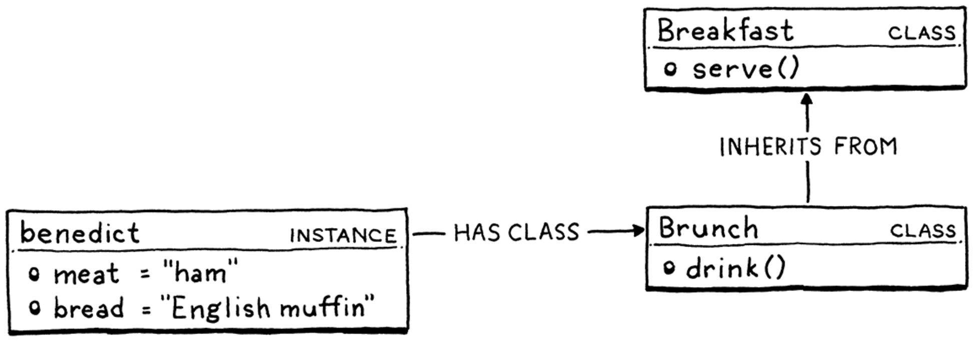
I could claim objects are groovy but still out of scope for the book. Most programming language books, especially ones that try to implement a whole language, leave objects out. To me, that means the topic isn't well covered. With such a widespread paradigm, that omission makes me sad.

Given how many of us spend all day *using* OOP languages, it seems like the world could use a little documentation on how to *make* one. As you'll see, it turns out to be pretty interesting. Not as hard as you might fear, but not as simple as you might presume, either.

Classes or prototypes 3.9.3

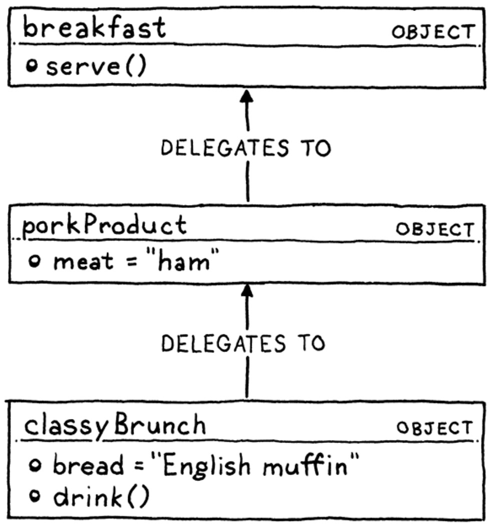
When it comes to objects, there are actually two approaches to them, classes and prototypes. Classes came first, and are more common thanks to C++, Java, C#, and friends. Prototypes were a virtually forgotten offshoot until JavaScript accidentally took over the world.

In class-based languages, there are two core concepts: instances and classes. Instances store the state for each object and have a reference to the instance's class. Classes contain the methods and inheritance chain. To call a method on an instance, there is always a level of indirection. You look up the instance's class and then you find the method *there*:



In a statically typed language like C++, method lookup typically happens at compile time based on the *static* type of the instance, giving you **static dispatch**. In contrast, **dynamic dispatch** looks up the class of the actual instance object at runtime. This is how virtual methods in statically typed languages and all methods in a dynamically typed language like Lox work.

Prototype-based languages merge these two concepts. There are only objects—no classes—and each individual object may contain state and methods. Objects can directly inherit from each other (or "delegate to" in prototypal lingo):



In practice the line between class-based and prototype-based languages blurs. JavaScript's "constructor function" notion pushes you pretty hard towards defining class-like objects. Meanwhile, class-based Ruby is perfectly happy to let you attach methods to individual instances.

This means that in some ways prototypal languages are more fundamental than classes. They are really neat to implement because they're *so* simple. Also, they can express lots of unusual patterns that classes steer you away from.

But I've looked at a *lot* of code written in prototypal languages—including some of my own devising. Do you know what people generally do with all of the power and flexibility of prototypes? ... They use them to reinvent classes.

I don't know *why* that is, but people naturally seem to prefer a class-based (Classic? Classy?) style. Prototypes *are* simpler in the language, but they seem to accomplish that only by pushing the complexity onto the user. So, for Lox, we'll save our users the trouble and bake classes right in.

Larry Wall, Perl's inventor/prophet calls this the "waterbed theory". Some complexity is essential and cannot be eliminated. If you push it down in one place, it swells up in another.

Prototypal languages don't so much *eliminate* the complexity of classes as they do make the *user* take that complexity by building their own class-like metaprogramming libraries.

Classes in Lox 3.9.4

Enough rationale, let's see what we actually have. Classes encompass a constellation of features in most languages. For Lox, I've selected what I think are the brightest stars. You declare a class and its methods like so:

```
class Breakfast {
  cook() {
    print "Eggs a-fryin'!";
  }

  serve(who) {
    print "Enjoy your breakfast, " + who + ".";
  }
}
```

The body of a class contains its methods. They look like function declarations but without the `fun` keyword. When the class declaration is executed, Lox creates a class object and stores that in a variable named after the class. Just like functions, classes are first class in Lox.

They are still just as fun, though.

```
// Store it in variables.
var someVariable = Breakfast;

// Pass it to functions.
someFunction(Breakfast);
```

Next, we need a way to create instances. We could add some sort of `new` keyword, but to keep things simple, in Lox the class itself is a factory function for instances. Call a class like a function, and it produces a new instance of itself.

```
var breakfast = Breakfast();
print breakfast; // "Breakfast instance".
```

Instantiation and initialization 3.9.5

Classes that only have behavior aren't super useful. The idea behind object-oriented programming is encapsulating behavior *and state* together. To do that, you need fields. Lox, like other dynamically typed languages, lets you freely add properties onto objects.

```
breakfast.meat = "sausage";
breakfast.bread = "sourdough";
```

Assigning to a field creates it if it doesn't already exist.

If you want to access a field or method on the current object from within a method, you use good old `this`.

```
class Breakfast {
  serve(who) {
    print "Enjoy your " + this.meat + " and " +
        this.bread + ", " + who + ".";
  }

  // ...
}
```

Part of encapsulating data within an object is ensuring the object is in a valid state when it's created. To do that, you can define an initializer. If your class has a method named `init()`, it is called automatically when the object is constructed. Any parameters passed to the class are forwarded to its initializer.

```
class Breakfast {
  init(meat, bread) {
    this.meat = meat;
    this.bread = bread;
  }

  // ...
}

var baconAndToast = Breakfast("bacon", "toast");
baconAndToast.serve("Dear Reader");
// "Enjoy your bacon and toast, Dear Reader."
```

Inheritance 3.9.6

Every object-oriented language lets you not only define methods, but reuse them across multiple classes or objects. For that, Lox supports single inheritance. When you declare a class, you can specify a class that it inherits from using a less-than (`<`) operator.

```
class Brunch < Breakfast {
  drink() {
    print "How about a Bloody Mary?";
  }
}
```

Here, Brunch is the **derived class** or **subclass**, and Breakfast is the **base class** or **superclass**.

Every method defined in the superclass is also available to its subclasses.

```
var benedict = Brunch("ham", "English muffin");
benedict.serve("Noble Reader");
```

Even the `init()` method gets inherited. In practice, the subclass usually wants to define its own `init()` method too. But the original one also needs to be called so that the superclass can maintain its state. We need some way to call a

Why the `<` operator? I didn't feel like introducing a new keyword like `extends`. Lox doesn't use `:` for anything else so I didn't want to reserve that either. Instead, I took a page from Ruby and used `<`.

If you know any type theory, you'll notice it's not a *totally* arbitrary choice. Every instance of a subclass is an instance of its superclass too, but there may be instances of the superclass that are not instances of the subclass. That means, in the universe of objects, the set of subclass objects is smaller than the superclass's set, though type nerds method on our own *instance* without hitting our own *methods*.

As in Java, you use `super` for that.

```
class Brunch < Breakfast {
  init(meat, bread, drink) {
    super.init(meat, bread);
    this.drink = drink;
  }
}
```

That's about it for object orientation. I tried to keep the feature set minimal. The structure of the book did force one compromise. Lox is not a *pure* object-oriented language. In a true OOP language every object is an instance of a class, even primitive values like numbers and Booleans.

Because we don't implement classes until well after we start working with the built-in types, that would have been hard. So values of primitive types aren't real objects in the sense of being instances of classes. They don't have methods or properties. If I were trying to make Lox a real language for real users, I would fix that.

usually use `<:` for that relation.

Lox is different from C++, Java, and C#, which do not inherit constructors, but similar to Smalltalk and Ruby, which do.

The Standard Library 3.10

We're almost done. That's the whole language, so all that's left is the "core" or "standard" library—the set of functionality that is implemented directly in the interpreter and that all user-defined behavior is built on top of.

This is the saddest part of Lox. Its standard library goes beyond minimalism and veers close to outright nihilism. For the sample code in the book, we only need to demonstrate that code is running and doing what it's supposed to do. For that, we already have the built-in `print` statement.

Later, when we start optimizing, we'll write some benchmarks and see how long it takes to execute code. That means we need to track time, so we'll define one built-in function, `clock()`, that returns the number of seconds since the program started.

And . . . that's it. I know, right? It's embarrassing.

If you wanted to turn Lox into an actual useful language, the very first thing you should do is flesh this out. String manipulation, trigonometric functions, file I/O, networking, heck, even *reading input from the user* would help. But we don't need any of that for this book, and adding it wouldn't teach you anything interesting, so I've left it out.

Don't worry, we'll have plenty of exciting stuff in the language itself to keep us busy.

CHALLENGES

1. Write some sample Lox programs and run them (you can use the implementations of Lox in my repository). Try to come up with edge case behavior I didn't specify here. Does it do what you expect? Why or why not?

2. This informal introduction leaves a *lot* unspecified. List several open questions you have about the language's syntax and semantics. What do you think the answers should be?

3. Lox is a pretty tiny language. What features do you think it is missing that would make it annoying to use for real programs? (Aside from the standard library, of course.)

DESIGN NOTE: EXPRESSIONS AND STATEMENTS

Lox has both expressions and statements. Some languages omit the latter. Instead, they treat declarations and control flow constructs as expressions too. These "everything is an expression" languages tend to have functional pedigrees and include most Lisps, SML, Haskell, Ruby, and CoffeeScript.

To do that, for each "statement-like" construct in the language, you need to decide what value it evaluates to. Some of those are easy:

- An `if` expression evaluates to the result of whichever branch is chosen. Likewise, a `switch` or other multi-way branch evaluates to whichever case is picked.

- A variable declaration evaluates to the value of the variable.

- A block evaluates to the result of the last expression in the sequence.

Some get a little stranger. What should a loop evaluate to? A `while` loop in CoffeeScript evaluates to an array containing each element that the body evaluated to. That can be handy, or a waste of memory if you don't need the array.

You also have to decide how these statement-like expressions compose with other expressions—you have to fit them into the grammar's precedence table. For example, Ruby allows:

```
puts 1 + if true then 2 else 3 end + 4
```

Is this what you'd expect? Is it what your *users* expect? How does this affect how you design the syntax for your "statements"? Note that Ruby has an explicit `end` to tell when the `if` expression is complete. Without it, the `+ 4` would likely be parsed as part of the `else` clause.

Turning every statement into an expression forces you to answer a few hairy questions like that. In return, you eliminate some redundancy. C has both blocks for sequencing statements, and the comma operator for sequencing expressions. It has both the `if` statement and the `?:` conditional operator. If everything was an expression in C, you could unify each of those.

Languages that do away with statements usually also feature **implicit returns**—a function automatically returns whatever value its body evaluates to without need for some explicit `return` syntax. For small functions and methods, this is really handy. In fact, many languages that do have statements have added syntax like `=>` to be able to define functions whose body is the result of evaluating a single expression.

But making *all* functions work that way can be a little strange. If you aren't careful, your function will leak a return value even if you only intend it to produce a side effect. In practice, though, users of these languages don't find it to be a problem.

For Lox, I gave it statements for prosaic reasons. I picked a C-like syntax for familiarity's sake, and trying to take the existing C statement syntax and interpret it like expressions gets weird pretty fast.

NEXT PART: "A TREE-WALK INTERPRETER" →