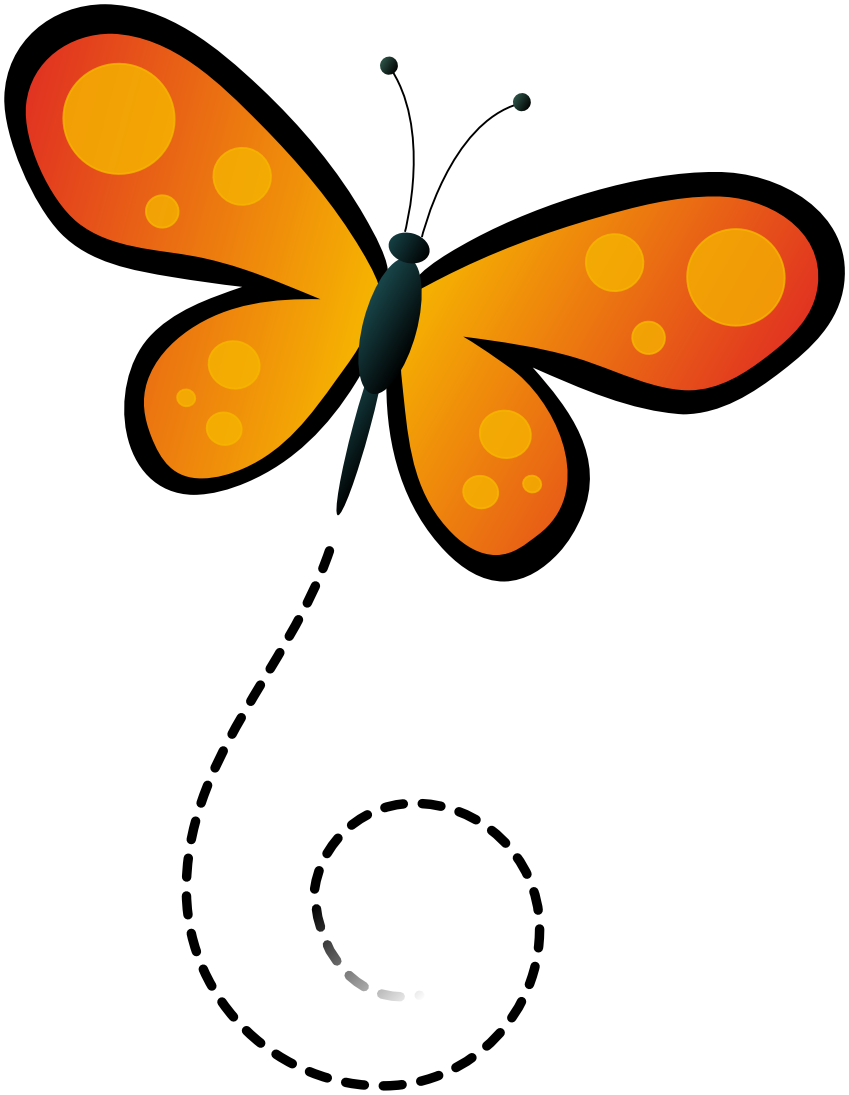

Using Perl 6

Jonathan S. Duff, Moritz Lenz, Carl Masäk,
Patrick R. Michaud & Jonathan Worthington

Credits

Text Jonathan S. Duff, Moritz Lenz, Carl Mäsak, Patrick R. Michaud, Jonathan Worthington.

Front cover Sebastian Riedel (`http://kraih.com`)

Fonts Adobe Minion® Pro, Adobe Myriad® Pro, B&H Luxi Mono

LaTeX-Layout Nikolai Prokoschenko, Luc St-Louis et al. Special thanks to Konrad Mühler for his LaTeX tips collection (`http://www.kfiles.de/latex.php`)

Editor chromatic

Further contributions Carlin Bingham, Tim Bunce, Nuno Carvalho, Will Coleda, Patrick Donelan, James E. Keenan, Alex Elsayed, Jason Felds, Matt Follett, Solomon Foster, Piotr Fusik, Bruce Gray, Prakash Kailasa, Matt Kraai, Philipp Marek, molecules, Kevin Polulak, Hongwen Qiu, Dean Serenevy, Aaron Sherman, snarkyboojum, Tadeusz Sośnierz, Gabor Szabo, Ralf Valerien, Przemysław Wesołek,

Contents

1

Preface

Perl 6 is a language specification for which multiple compilers and interpreters exist in various stages of completeness. These implementations have in turn influenced the design of the language by highlighting misfeatures, contradictions, or features of difficult implementation and little benefit. This process of iteration has produced a more cohesive and consistent language specification.

Perl 6 is versatile, intuitive, and flexible. It embraces several paradigms like procedural, object oriented, and functional programming, and offers powerful tools for parsing text.

This book is a work-in-progress. Even releases will contain some amount of `TODO` comments prior to the printing of the book. We've left them in because they might serve as useful cues for the reader as well as for us authors about what remains to be done. Even so, we pray the reader's indulgence and understanding.

1.1 Audience

This book is primarily for people who want to learn Perl 6. It is a series of tutorials, not a comprehensive reference. We expect some experience in another programming language, though no prior knowledge of Perl is necessary. After working through this book, you should have a solid grasp of the basics of Perl 6 sufficient to solve your own problems with Perl 6.

1.2 Format of this book

Each chapter opens with a reasonably complete example that illustrates the topic of the chapter. We intend these examples to demonstrate how to use the features, techniques, and idioms explained in the chapter in real programs. Our goal is to convey the "flavor" of writing Perl 6 programs so that the reader may proceed to write their own native Perl 6 programs rather than programs that resemble some other language[1].

1.3 Relationship between Perl 6 and Perl 5

Perl 6 is the newest member of the family of languages known as Perl. It represents a major break in syntactic and semantic compatibility from Perl 5, thus the increase from 5 to 6. However, this does not mean that Perl 5 is going away. In fact, quite the opposite. Both Perl 5 and Perl 6 have active developer communities which mold the languages. Perl 5 developers try to extend the language in various ways while keeping backwards compatibility with past versions of Perl. Perl 6 developers extend the language by adding new syntactic and semantic features that enable more power and expressiveness without the restriction of backward compatibility with Perl 5 or earlier versions.

Some might ask, "Why call it Perl if it's a different language?" Perl is more than the vagaries of syntax. Perl is philosophy (there's more than one way to do it; easy things should be easy, and hard things possible); Perl is custom (comprehensive testing, idioms); Perl is architectural edifice (the Comprehensive Perl Archive Network); and Perl is community (perl5-porters, perl6-language). Both Perl 5 and Perl 6 share these attributes to varying

[1] Some programmers can still write Fortran in any language, however :)

degrees. As well, Perl is syncretic. Just as Perl borrows good ideas from other languages, so Perl 5 and Perl 6 share features.

1.4 Perl 6 implementations

Perl 6 is a specification. Any implementation that passes the official test suite can call itself "Perl 6". Several implementations exist at various levels of maturity. All of the examples in this book will run with the *Rakudo Perl 6* Compiler, but they are in no way specific to Rakudo–any sufficiently advanced Perl 6 implementation can run them. Good luck and–as the Perl 6 community often says–have fun!

1.5 Installing Rakudo

For complete instructions for downloading and installing Rakudo, see `http://www.rakudo.org/how-to-get-rakudo`. Source code releases are available from `http://github.com/rakudo/rakudo/downloads`. A binary release for windows is available from `http://sourceforge.net/projects/parrotwin32/files/`.

1.6 Executing programs

To run a Perl 6 program with Rakudo, include the installation directory in your system `PATH` variable and issue a command like:

```
$ perl6 hello.pl
```

If you invoke the Rakudo compiler without an explicit script to run, it enters a small interactive mode that allows the execution of Perl 6 statements from the command line.

1.7 Getting involved

If you are inspired by the contents of this book and want to contribute to the Perl 6 community, there are some resources available to you:

World Wide Web

: The Perl 6 homepage at `http://perl6.org/` links to many useful resources.

IRC

: The channel `#perl6` on `irc.freenode.net` discusses all things Perl 6.

Mailing lists

: If you need programming help with Perl 6, send an email to `perl6-users@perl.org`.

: For issues regarding the Perl 6 language specification, contact `perl6-language@perl.org`. For issues regarding Perl 6 compilers, send email to `perl6-compiler@perl.org`.

2

The Basics

Perl originated as a programming language intended to gather and summarize information from text files. It's still strong in text processing, but Perl 5 is also a powerful general-purpose programming language. Perl 6 is even better.

Suppose that you host a table tennis tournament. The referees tell you the results of each game in the format `Player1 Player2 | 3:2`, which means that `Player1` won against `Player2` by 3 to 2 sets. You need a script that sums up how many matches and sets each player has won to determine the overall winner.

The input data (stored in a file called "scores") looks like this:

```
Beth Ana Charlie Dave
Ana Dave | 3:0
Charlie Beth | 3:1
Ana Beth | 2:3
Dave Charlie | 3:0
Ana Charlie | 3:1
Beth Dave | 0:3
```

The first line is the list of players. Every subsequent line records a result of a match.

Here's one way to solve that problem in Perl 6:

```
use v6;

my $file  = open 'scores';
my @names = $file.get.words;

my %matches;
my %sets;

for $file.lines -> $line {
    my ($pairing, $result) = $line.split(' | ');
    my ($p1, $p2)          = $pairing.words;
    my ($r1, $r2)          = $result.split(':');

    %sets{$p1} += $r1;
    %sets{$p2} += $r2;

    if $r1 > $r2 {
        %matches{$p1}++;
    } else {
        %matches{$p2}++;
    }
}

my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;

for @sorted -> $n {
    say "$n has won %matches{$n} matches and %sets{$n} sets";
}
```

This produces the output:

```
Ana has won 2 matches and 8 sets
Dave has won 2 matches and 6 sets
Charlie has won 1 matches and 4 sets
Beth has won 1 matches and 4 sets
```

Every Perl 6 program should begin with `use v6;`. This line tells the compiler which version of Perl the program expects. Should you accidentally run the file with Perl 5, you'll get a helpful error message.

A Perl 6 program consists of zero or more statements. A *statement* ends with a semicolon or a curly bracket at the end of a line:

```
my $file = open 'scores';
```

`my` declares a lexical variable. Lexical variables are visible only in the current block from the point of declaration to the end of the block. If there's no enclosing block, it's visible throughout the remainder of the file. A block is any part of the code enclosed between curly braces { }.

A variable name begins with a *sigil*, which is a non-alpha-numeric symbol such as `$`, `@`, `%`, or `&`–or occasionally the double colon `::`. Sigils indicate the structural interface for the variable, such as whether it should be treated as a single value, a compound value, a subroutine, etc. After the sigil comes an *identifier*, which may consist of letters, digits and the underscore. Between letters you can also use a dash `-` or an apostrophe `'`, so `isn't` and `double-click` are valid identifiers.

The `$` sigil indicates a *scalar* variable, which indicates that the variable stores a single value.

The built-in function `open` opens a file, here named *scores*, and returns a *file handle*–an object representing that file. The equality sign `=` *assigns* that file handle to the variable on the left, which means that `$file` now stores the file handle.

`'scores'` is a *string literal*. A string is a piece of text, and a string literal is a string which appears directly in the program. In this line, it's the argument provided to `open`.

```
my @names = $file.get.words;
```

The right-hand side calls a *method* –a named group of behavior– named `get` on the file handle stored in `$file`. The `get` method reads and returns one line from the file, removing the line ending. `words` is also a method, called on the string returned from `get`. `words` decomposes its *invocant*–the string on which it operates–into a list of words, which here means strings without whitespace. It turns the single string `'Beth Ana Charlie Dave'` into the list of strings `'Beth'`, `'Ana'`, `'Charlie'`, `'Dave'`.

Finally, this list gets stored in the array `@names`. The `@` sigil marks the declared variable as an `Array`. Arrays store ordered lists.

```
my %matches;
my %sets;
```

These two lines of code declare two hashes. The `%` sigil marks each variable as a `Hash`. A `Hash` is an unordered collection of key-value pairs. Other programming languages call that a *hash table*, *dictionary*, or *map*. You can query a hash table for the value that corresponds to a certain `$key` with `%hash{$key}`[1].

In the score counting program, `%matches` stores the number of matches each player has won. `%sets` stores the number of sets each player has won.

Sigils indicate the default access method for a variable. Variables with the `@` sigil are accessed positionally; variables with the `%` sigil are accessed by string key. The `$` sigil, however, indicates a general container that can hold anything and be accessed in any manner. A scalar can even contain a compound object like an `Array` or a `Hash`; the `$` sigil signifies that it should be treated as a single value, even in a context that expects multiple values (as with an `Array` or `Hash`).

```
for $file.lines -> $line {
    ...
}
```

`for` produces a loop that runs the *block* delimited by curly brackets once for each item of the list, setting the variable `$line` to the current value of each iteration. `$file.lines` produces a list of the lines read from the file *scores*, starting with the line where the previous calls to `$file.get` left off, and going all the way to the end of the file.

During the first iteration, `$line` will contain the string `Ana Dave | 3:0`; during the second, `Charlie Beth | 3:1`, and so on.

```
my ($pairing, $result) = $line.split(' | ');
```

`my` can declare multiple variables simultaneously. The right-hand side of the assignment is a call to a method named `split`, passing along the string `' | '` as an argument.

`split` decomposes its invocant into a list of strings, so that joining the list items with the separator `' | '` produces the original string.

[1] Unlike Perl 5, in Perl 6 the sigil does not change when accessing an array or hash with `[ ]` or `{ }`. This is called *sigil invariance*.

`$pairing` gets the first item of the returned list, and `$result` the second.

After processing the first line, `$pairing` will hold the string `Ana Dave` and `$result` `3:0`.

The next two lines follow the same pattern:

```
my ($p1, $p2) = $pairing.words;
my ($r1, $r2) = $result.split(':');
```

The first extracts and stores the names of the two players in the variables `$p1` and `$p2`. The second extracts the results for each player and stores them in `$r1` and `$r2`.

After processing the first line of the file, the variables contain the values:

Table 2.1: Contents of Variables

Variable	Contents
`$line`	`'Ana Dave | 3:0'`
`$pairing`	`'Ana Dave'`
`$result`	`'3:0'`
`$p1`	`'Ana'`
`$p2`	`'Dave'`
`$r1`	`'3'`
`$r2`	`'0'`

The program then counts the number of sets each player has won:

```
%sets{$p1} += $r1;
%sets{$p2} += $r2;
```

This is a shortcut for:

```
%sets{$p1} = %sets{$p1} + $r1;
%sets{$p2} = %sets{$p2} + $r2;
```

`+= $r1` means *increase the value in the variable on the left by $r1*. In the first iteration `%sets{$p1}` is not yet set, so it defaults to a special value called `Any`. The addition and incrementing operators treat `Any` as a number with the value of zero; the strings get automatically converted to numbers, as addition is a numeric operation.

Before these two lines execute, `%sets` is empty. Adding to an entry that is not in the hash yet will cause that entry to spring into existence just-in-time, with a value starting at zero. (This is *autovivification*). After these two lines have run for the first time, `%sets` contains `'Ana' => 3, 'Dave' => 0`. (The fat arrow `=>` separates key and value in a `Pair`.)

```
1  if $r1 > $r2 {
2      %matches{$p1}++;
3  } else {
4      %matches{$p2}++;
5  }
```

If `$r1` has a larger value than `$r2`, `%matches{$p1}` increments by one. If `$r1` is not larger than `$r2`, `%matches{$p2}` increments. Just as in the case of `+=`, if either hash value did not exist previously, it is autovivified by the increment operation.

`$thing++` is short for `$thing += 1` or `$thing = $thing + 1`, with the small exception that the return value of the expression is `$thing` *before* the increment, not the incremented value. As in many other programming languages, you can use `++` as a prefix. Then it returns the incremented value; `my $x = 1; say ++$x` prints `2`.

```
1  my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

This line consists of three individually simple steps. An array's `sort` method returns a sorted version of the array's contents. However, the default sort on an array sorts by its contents. To print player names in winner-first order, the code must sort the array by the *scores* of the players, not their names. The `sort` method's argument is a *block* used to transform the array elements (the names of players) to the data by which to sort. The array items are passed in through the *topic variable* `$_`.

You have seen blocks before: both the `for` loop `-> $line { ... }` and the `if` statement worked on blocks. A block is a self-contained piece of Perl 6 code with an optional signature (the `-> $line` part). See *sec:signatures* on page 30 for more information.

The simplest way to sort the players by score would be `@names.sort({ %matches{$_} })`, which sorts by number of matches won. However Ana and Dave have both won two matches. That simple sort doesn't account for the number of sets won, which is the secondary criterion to decide who has won the tournament.

When two array items have the same value, `sort` leaves them in the same order as it found them. Computer scientists call this a *stable sort*. The program takes advantage of

this property of Perl 6's `sort` to achieve the goal by sorting twice: first by the number of sets won (the secondary criterion), then by the number of matches won.

After the first sorting step, the names are in the order `Beth Charlie Dave Ana`. After the second sorting step, it's still the same, because no one has won fewer matches but more sets than someone else. Such a situation is entirely possible, especially at larger tournaments.

`sort` sorts in ascending order, from smallest to largest. This is the opposite of the desired order. Therefore, the code calls the `.reverse` method on the result of the second sort, and stores the final list in `@sorted`.

```
1  for @sorted -> $n {
2      say "$n has won %matches{$n} matches and %sets{$n} sets";
3  }
```

To print out the players and their scores, the code loops over `@sorted`, setting `$n` to the name of each player in turn. Read this code as "For each element of sorted, set `$n` to the element, then execute the contents of the following block." `say` prints its arguments to the standard output (the screen, normally), followed by a newline. (Use `print` if you don't want the newline at the end.)

When you run the program, you'll see that `say` doesn't print the contents of that string verbatim. In place of `$n` it prints the contents of the variable `$n`– the names of players stored in `$n`. This automatic substitution of code with its contents is *interpolation*. This interpolation happens only in strings delimited by double quotes `"..."`. Single quoted strings `'...'` do not interpolate:

```
1  my $names = 'things';
2  say 'Do not call me $names'; # Do not call me $names
3  say "Do not call me $names"; # Do not call me things
```

Double quoted strings in Perl 6 can interpolate variables with the `$` sigil as well as blocks of code in curly braces. Since any arbitrary Perl code can appear within curly braces, `Array`s and `Hash`es may be interpolated by placing them within curly braces.

Arrays within curly braces are interpolated with a single space character between each item. Hashes within curly braces are interpolated as a series of lines. Each line will contain a key, followed by a tab character, then the value associated with that key, and finally a newline.

TODO: explain <. . .> quote-words

```
say "Math: { 1 + 2 }"                  # Math: 3
my @people = <Luke Matthew Mark>;
say "The synoptics are: {@people}"     # The synoptics are: Luke Matthew Mark

say "{%sets}";                         # From the table tennis tournament

# Charlie 4
# Dave    6
# Ana     8
# Beth    4
```

When array and hash variables appear directly in a double-quoted string (and not inside curly brackets), they are only interpolated if their name is followed by a postcircumfix – a bracketing pair that follows a statement. It's also ok to have a method call between the variable name and the postcircumfix.

```
my @flavours = <vanilla peach>;

say "we have @flavours";           # we have @flavours
say "we have @flavours[0]";        # we have vanilla
# so-called "Zen slice"
say "we have @flavours[]";         # we have vanilla peach

# method calls ending in postcircumfix
say "we have @flavours.sort()";    # we have peach vanilla

# chained method calls:
say "we have @flavours.sort.join(', ')";
                                 # we have peach, vanilla
```

2.1 Exercises

1. The input format of the example program is redundant: the first line containing the name of all players is not necessary, because you can find out which players participated in the tournament by looking at their names in the subsequent rows.

How can you change the program if the first input line is omitted? Hint: `%hash.keys` returns a list of all keys stored in `%hash`.

Answer: Remove the line `my @names = $file.get.words;`, and change:

```
my @sorted = @names.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

... into:

```
my @sorted = %sets.keys.sort({ %sets{$_} }).sort({ %matches{$_} }).reverse;
```

2. Instead of removing the redundancy, you can also use it to warn if a player appears that wasn't mentioned in the first line, for example due to a typo. How would you modify your program to achieve that?

Answer: Introduce another hash with the names of the legitimate players as keys, and look in this hash when the name of a player is read:

```
...
my @names = $file.get.split(' ');
my %legitimate-players;
for @names -> $n {
    %legitimate-players{$n} = 1;
}

...

for $file.lines -> $line {
    my ($pairing, $result) = $line.split(' | ');
    my ($p1, $p2)          = $pairing.split(' ');
    for $p1, $p2 -> $p {
        if !%legitimate-players{$p} {
            say "Warning: '$p' is not on our list!";
        }
    }

    ...
}
```

3

Operators

Operators are very short names for often used routines. They have a special calling syntax, and can manipulate each other.

Consider the table tennis example from the previous chapter. Suppose you want to plot the number of sets that each player won in a tournament. This example makes a very simple text output by printing `X` characters to represent horizontal bars:

```
use v6;

my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;

my $screen-width     = 30;

my $label-area-width = 1 + [max] @scores».key».chars;
my $max-score        = [max] @scores».value;
my $unit             = ($screen-width - $label-area-width) / $max-score;
my $format           = '%- ' ~ $label-area-width ~ "s%s\n";

for @scores {
    printf $format, .key, 'X' x ($unit * .value);
}
```

This program produces the output:

```
Ana     XXXXXXXXXXXXXXXXXXXXX
Dave    XXXXXXXXXXXXXXX
Charlie XXXXXXXXXX
Beth    XXXXXXXXXX
```

The line:

```
1  my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

... already contains three different operators: `=`, `=>`, and `,`.

The `=` operator is the *assignment operator*–it takes the values from the right-hand side, and stores them in the variable on the left-hand side, here `@scores`.

> Like other languages that have adopted a similar syntax to C, Perl 6 allows for a shorthand way to write certain assignments. You can express any assignment of the form `$var = $var` *op* `EXPR` as `$var` *op*`= EXPR`. For example, `~` (tilde) is the string concatenation operator; to append some text to the end of a string, you can write `$string ~= "text"`, which is equivalent to `$string = $string ~"text"`.

The `=>` operator (the *fat arrow*) constructs `Pair` objects. A Pair stores a key and a value; the key is on the left-hand side of the `=>` operator, the value on the right. This operator also has a special feature: it causes the parser to interpret any bare identifier on the left-hand side as a string. You could also write the example line as:

```
1  my @scores = Ana => 8, Dave => 6, Charlie => 4, Beth => 4;
```

Finally, the `,` operator constructs a `Parcel`, which is a sequence of objects. In this case the objects are pairs.

All of the three operators above are *infix* operators, which means they stand between two *terms*. A term can be a literal like `8` or `'Dave'`, or a combination of other terms and operators.

The previous chapter already used other types of operators. It contained the statement `%matches{$p1}++;`. The *postcircumfix* operator `{...}` occurs after (*post*) a term, and con-

sists of two symbols (an opening and a closing curly bracket) which enclose (*circumfix*) another term. After this postcircumfix operator is an ordinary *postfix* operator with name ++, which increments the value it qualifies. You may not use whitespace between a term and its postfix or postcircumfix operators.

Another operator type is the *prefix* operator, which occurs before a term. An example is the - operator, which negates the following numeric value, as in `my $x = -4`.

The - operator can also mean subtraction, so `say 5 - 4` will print a `1`. To distinguish the prefix operator - from the infix operator -, the Perl 6 parser always keeps track of whether it expects an infix operator or a term. A term can have zero or more prefix operators, so you can actually write `say 4 + -5`. After the + (an infix operator), the compiler expects a term, so as to interpret the - as a prefix operator to the term `5`.

The next line containing new features is:

```
1  my $label-area-width = 1 + [max] @scores».key».chars;
```

It begins harmlessly with a variable declaration `my $label-area-width` and an assignment. Next comes a simple numeric addition, `1 + ...`. The right side of the + operator is more complicated.

The infix `max` operator returns the greater of two values, so `2 max 3` returns 3. Square brackets around an infix operator cause Perl to apply the operator to a list element by element. `[max] 1, 5, 3, 7` is the same as `1 max 5 max 3 max 7` and evaluates to `7`. You might know this as as *reduce* from other programming languages.

Likewise, you can write `[+]` to get the sum of a list of values, `[*]` for the product, and `[<=]` to check if a list is ordered by ascending values.

Next comes the expression `@scores».key».chars`. Just like `@variable.method` calls a method on `@variable`, `@array».method` calls a method for each item in `@array` and returns the list of the return values.

» is a *hyper operator*. It is also a Unicode character. If your operating system[1] or text editor[2] does not make it easy to write it you can also write it using two angle brackets (>>).

[1] Ubuntu 10.4: In System/Preferences/Keyboard/Layouts/Options/Compose Key position select one of the keys to be the "Compose" key. Then press Compose-key and the "greater than" key twice.

[2] Emacs users can type C-x 8 Enter 00bb Enter, vim users can type `C-k >>`.

`@scores».key` is a list of all the keys of the pair objects in `@scores`, and `@scores».key».chars` is a list of the length of all keys in `@scores`.

The expression `[max] @scores».key».chars` gives the largest of these values. It is the same as:

```
@scores[0].key.chars
    max @scores[1].key.chars
    max @scores[2].key.chars
    max ...
```

These circumfix square brackets are the *reduction meta operator*, which transforms the enclosed infix operator into an operator that expects a list (a *listop*), and carries out the operation between each two consecutive list items.

For plotting the names of the players and bar charts, the program needs to know how much space to allocate for the player names. Adding 1 to it leaves space for a single blank space between the name of the longest player and the left edge of the bars.

The program next determines the maximum score:

```
my $max-score       = [max] @scores».value;
```

The drawing area has the width `$screen-width - $label-area-width`, so for each score, it should print:

```
my $unit = ($screen-width - $label-area-width) / $max-score;
```

... amount of `X` characters. This expression uses the infix operators `-` and `/` for numerical calculations.

Now all the necessary informations are in place, and the chart can print:

```
my $format = '%- ' ~ $label-area-width ~ "s%s\n";
for @scores {
    printf $format, .key, 'X' x ($unit * .value);
}
```

These lines loop over the items in `@scores`, binding them to the special variable `$_` one at a time. For each item, the program uses the `printf` built-in function to print both the

name of the player and a bar. This function is similar to `printf` in C and Perl 5. It takes a format string, which specifies how to print the following parameters. If `$label-area-width` is 8, the format string is `"%- 8s%s\n"`, which means a string (`'s'`) filled to 8 spaces (`' 8'`) and left-justified (`'-'`), followed by another string and a newline.

The first string is the name of the player and the second is the bar.

The infix `x` operator, or *repetition operator*, generates this bar. It takes a string on the left-hand side and a number on the right-hand side, and sticks the strings together as many times as the number specifies, so `'ab' x 3` returns the string `'ababab'`. `.value` returns the value of the current pair, `($unit * .value)` multiplies that values with `$unit`, and `'X' x ($unit * .value)` returns as that many `X` characters.

3.1 A Word on Precedence

The explanations of this example have one implication which is not entirely obvious. In the line:

```
1  my @scores = 'Ana' => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

... the right-hand side of the assignment produces a list (because of the `,` operator) that is made of pairs (because of `=>`), and the result is then assigned to the array variable. You could think of other ways for Perl 6 to interpret this program. Perl 5 will interpret it as:

```
1  (my @scores = 'Ana') => 8, 'Dave' => 6, 'Charlie' => 4, 'Beth' => 4;
```

... so that `@scores` will contain only one item. The rest of the expression is merely a list of constants evaluated, then discarded.

Precedence rules govern how a parser will parse this line. Perl 6's precedence rules state that the infix `=>` operator binds its arguments more tightly than the infix `,` operator, which in turn binds more tightly than the `=` assignment operator.

> There are actually two assignment operators with different precedence. When the right-hand side is a scalar, the *item assignment operator* with tight precedence is used, otherwise the loose-precedence *list assignment operator* is used. This allows the two expressions `$a = 1, $b = 2` and `@a = 1, 2` to both mean something

sensible: assignment to two variables in a list, and assignment of a two-item list to a single variable.

Perl 6's precedence rules allow you to express many common operations naturally, without thinking about precedence at all. If you want to force a different parsing precedence, surround an expression with parentheses, so that this new group has the tightest possible precedence:

```
say 5 - 7 / 2;      # 5 - 3.5  = 1.5
say (5 - 7) / 2;    # (-2) / 2 =  -1
```

But beware that parenthesis directly after an identifier – without any whitespace between – are always parsed as the argument list, so

```
say(5 - 7) / 2;     # -2
```

prints out only the result of `5 - 7`.

3.2 Comparisons and Smart Matching

There are several ways to compare objects in Perl. You can test for value equivalence using the `===` infix operator. For immutable objects[3], this is an ordinary value comparison. `"hello" === "hello"` is true because both strings are immutable and have the same value.

For mutable objects, `===` compares their identities. Two objects only share the same identity if, in fact, they are the same object. Even if the two arrays `@a` and `@b` *contain* the same values, if their containers are two separate array objects, they will have different identities and will *not* be equivalent when compared with `===`:

```
my @a = 1, 2, 3;
my @b = 1, 2, 3;

```

[3] Objects whose values *can not* be changed; literal values. For instance, the literal `7` will always and forever be just a `7`.

Table 3.1: Precedence table

Examples	Name
	(tightest precedence)
`(), 42.5`	term
`42.rand`	method calls and postcircumfixes
`$x++`	autoincrement and autodecrement
`$x**2`	exponentiation operator
`?$x, !$x`	boolean prefix
`+$x, ~$x`	prefix context operators
`2*3, 7/5`	multiplicative infix operators
`1+2, 7-5`	additive infix operators
`$a x 3`	replication operators
`$x ~".\n"`	string concatenation
`1&2`	junctive AND
`1|2`	junctive OR
`abs $x`	named unary prefix
`$x cmp 3`	non-chaining binary operators
`$x == 3`	chaining binary operators
`$x && $y`	tight AND infix
`$x || $y`	tight OR infix
`$x > 0 ?? 1 !! -1`	conditional operator
`$x = 1`	item assignment
`not $x`	loose unary prefix
`1, 2`	comma
`1, 2 Z @a`	list infix
`@a = 1, 2`	list prefix, list assignment
`$x and say "Yes"`	loose AND infix
`$x or die "No"`	loose OR infix
`;`	statement terminator
	(loosest precedence)

```
4  say @a  === @a;    # Bool::True
5  say @a  === @b;    # Bool::False
6
7  # these use identity for value
8
9  say  3  ===  3     # Bool::True
```

```
say 'a' === 'a';   # Bool::True

my $a = 'a';
say  $a === 'a';   # Bool::True
```

The `eqv` operator returns `True` only if two objects are of the same type *and* the same structure. With `@a` and `@b` as defined in the previous example, `@a eqv @b` is true because `@a` and `@b` contain the same values each. On the other hand `'2' eqv 2` returns `False`, because one argument is a string, the other an integer and so they are not of the same type.

3.2.1 Numeric Comparisons

You can ask if two objects have the same numeric value with the == infix operator. If one of the objects is not numeric, Perl will do its best to make it numeric before doing the comparison. If there is no good way to convert an object to a number, Perl will use the default of `0`.

```
say 1 == 1.0;      # Bool::True
say 1 == '1';      # Bool::True
say 1 == '2';      # Bool::False
say 3 == '3b';     # fails
```

The operators <, <=, >=, and > compare the relative size of numbers. `!=` returns `True` if the two objects differ in their numerical value.

When you use an array or list as a number, it evaluates to the number of items in that list.

```
my @colors = <red blue green>;

if @colors == 3 {
    say "It's true, @colors contains 3 items";
}
```

3.2.2 String Comparisons

Just like == converts its arguments to numbers before comparing, `eq` as an infix operator compares for string equality, converting its arguments to strings as necessary:

```
if $greeting eq 'hello' {
    say 'welcome';
}
```

Other operators compare strings lexicographically.

Table 3.2: Operators and Comparisons

Number Comparison	String Comparison	Stands for
==	eq	equals
!=	ne	not equal
!==	!eq	not equal
<	lt	less than
<=	le	less or equal
>	gt	greater than
>=	ge	greater or equal

For example, `'a' lt 'b'` is true, and likewise `'a' lt 'aa'`.

`!=` is really just a convenience for `!==`, which in turn is really the `!` meta operator added to the infix == operator. An equivalent explanation applies to `ne` and `!eq`.

Three-way Comparison

The three-way comparison operators take two operands, and return `Order::Increase` if the left is smaller, `Order::Same` when both are equal, and `Order::Decrease` if the right operand is smaller[4]. For numbers, the comparison operator is `<=>` and for strings it's `leg` (from *l*esser, *e*qual, *g*reater). The infix `cmp` operator is a type sensitive three-way comparison operator which compares numbers like `<=>`, strings like `leg`, and (for example) pairs first by key, and then by value if the keys are identical:

[4] `Order::Increase`, `Order::Same`, and `Order::Decrease` are enumerations (enums); see *sec:subtypes* on page 89

```
say 10   <=> 5;       # +1
say 10   leg 5;       # because '1' lt '5'
say 'ab' leg 'a';     # +1, lexicographic comparison
```

A typical use case for three-way comparison is sorting. The `.sort` method in lists can take a block or function that takes two values, compares them, and returns a value less than, equal to or, greater than 0. The sort method then orders the values according to that return value:

```
say ~<abstract Concrete>.sort;
# output: Concrete abstract

say ~<abstract Concrete>.sort:
        -> $a, $b { uc($a) leg uc($b) };
# output: abstract Concrete
```

The default comparison is case sensitive; by comparing not the values, but their upper case variant, this example sorts case insensitively.

3.2.3 Smart Matching

The various comparison operators all coerce their arguments to certain types before comparing them. This is useful if you wish to be very specific about what kind of comparison you want and are unsure of the types of the values you are comparing. Perl 6 provides another operator that allows you to perform comparisons that Do The Right Thing with ~~, the smart match operator.

```
if $pints-drunk ~~ 8 {
    say "Go home, you've had enough!";
}

if $country ~~ 'Sweden' {
    say "Meatballs with lingonberries and potato moose, please."
}

unless $group-size ~~ 2..4 {
    say "You must have between 2 and 4 people to book this tour.";
}
```

The smart match operator always decides what kind of comparison to do based upon the type of the value on the right hand side. In the previous examples, the comparisons are numeric, string, and range, respectively. While this chapter has demonstrated the numeric and string comparison operators –`==` and `eq`–there is no operator for comparing ranges. This is part of the power of smart matching: more complex types can define interesting and useful ways to compare themselves to other things.

> Smart match works by calling the `ACCEPTS` method on the operand on the right hand side and passing it the operand on the left hand side as an argument. Thus `$answer ~~ 42` actually desugars to a method call like `42.ACCEPTS($answer)`. The upshot of this is that–after reading the chapter on writing classes and methods–you too will be able to write things that can smart-match just by implementing an `ACCEPTS` method to do the right thing.

4

Subroutines and Signatures

A *subroutine* is a piece of code that performs a specific task. It may operate on provided data (*arguments*) and may produce results (*return values*). The *signature* of a subroutine is a description of any arguments it takes and any return values it produces.

Chapter 2 demonstrated simple subroutines. In a sense, the operators described in chapter 3 are also subroutines that Perl 6 parses in interesting ways. However, they only scratch the surface of what's possible.

4.1 Declaring a Subroutine

A subroutine declaration consists of several parts. First, the subroutine declarator `sub` indicates that you are starting a subroutine declaration. Next comes an optional name and an optional signature. The body of the subroutine follows as a block of code enclosed in curly braces. This is what will execute every time the subroutine is called.

For example, in:

```
sub panic() {
    say "Oh no! Something has gone most terribly wrong!";
}
```

... the name of the sub is `panic`. Its signature is empty. Its body consists of a single `say` statement.

By default, subroutines are lexically scoped, just like any variable declared with `my`. This means that a subroutine may only be called within the scope in which it was declared. Use the scope declarator `our` to make the subroutine available within the current package, and install a lexical alias in the scope where you want to use it:

```
{
    our sub eat() {
        say "om nom nom";
    }

    sub drink() {
        say "glug glug";
    }
}
our &eat; # makes the package-scoped sub eat
          # available in this lexical scope

eat();    # om nom nom
drink();  # fails, can't drink outside of the block
```

`our` also makes subroutines visible from the outside of a package or module:

```
module EatAndDrink {
     our sub eat() {
         say "om nom nom";
     }

     sub drink() {
         say "glug glug";
     }
}
EatAndDrink::eat();    # om nom nom
EatAndDrink::drink();  # fails, not declared with "our"
```

You may also export a subroutine to make it available to another scope.

```
# in file Math/Trivial.pm
# TODO: find a better example
# TODO: explain modules, search paths
module Math::Trivial {
    sub double($x) is export {
        return 2 * $x;
    }
}
```

then in a different program or module you can write

```
use Math::Trivial; # imports sub double
say double(21);    # 21 is only half the truth
```

Perl 6 subroutines are objects. You can pass them around and store them in data structures just as you can do with any other piece of data. Programming language designers often call these *first-class subroutines*; they are as fundamental to the language as hashes or arrays are.

First-class subroutines can help you solve complex problems. For example, to make a little ASCII art dancing figure, you could build up a hash where the keys are names of the dance moves, and the values are anonymous subroutines. Assume that users should be able to enter a list of moves (perhaps on a dance pad or other exotic input device). How can you maintain a variable list of custom behaviors, allow user input, and restrict that input to a safe set of behaviors?

TODO this example doesn't seem like a good one for first-class subs.

```
my %moves =
    hands-over-head => sub { say '/o\ '   },
    bird-arms       => sub { say '|/o\| ' },
    left            => sub { say '>o '    },
    right           => sub { say 'o< '    },
    arms-up         => sub { say '\o/ '   };

my @awesome-dance = <arms-up bird-arms right hands-over-head>;

for @awesome-dance -> $move {
    %moves{$move}.();
}

```

```
# outputs:
#               \o/
#               |/o\|
#               o<
#               /o\
```

From the output of this program, you can observe that doing the YMCA dance in ASCII art looks just as bad as in real life.

4.2 Adding Signatures

A subroutine signature performs two tasks. First, it declares the arguments which callers may or must pass to the subroutine. Second, it declares the variables in the subroutine to which the arguments are bound. These variables are called *parameters*[1] Perl 6 signatures go further; they allow you to constrain the type, value, and "definedness" of its arguments and match against and extract parts of complex data structures. Additionally, they also allow you to explicitly specify the return type of a subroutine.

4.2.1 The Basics

In its simplest form, a signature is a comma separated list of variable names to which to bind incoming arguments.

```
sub order-beer($type, $pints) {
    say ($pints == 1 ?? 'A pint' !! "$pints pints") ~ " of $type, please."
}

order-beer('Hobgoblin', 1);    # A pint of Hobgoblin, please.
order-beer('Zlatý Bažant', 3); # 3 pints of Zlatý Bažant, please.
```

The use of the term *bound* instead of *assigned* is significant here. In Perl 6, the variables in a subroutine's signature are read-only references to the passed arguments by default. This means that you **can not** modify them from inside the subroutine.

[1] In this book, we will use the traditional convention of using the term *parameters* when referring to the variables in a subroutine's signature and *arguments* when referring to the values actually passed when a subroutine is called. Though, for the most part, you can consider these terms interchangeable.

If read-only binding is too limiting, you can relax this restriction by applying the `is rw` ("rw" being short for *read/write*) trait to a parameter. This allows you to modify the argument inside the subroutine. Use caution as this **will** alter the original object passed. If you attempt to pass a literal value, a constant, or some other type of immutable object[2] to a parameter that has the `is rw` trait, the binding will fail at the time of the call and throw an exception:

```
sub make-it-more-so($it is rw) {
    $it ~= substr($it, $it.chars - 1) x 5;
}

my $happy = "yay!";
make-it-more-so($happy);
say $happy;                # yay!!!!!!
make-it-more-so("uh-oh");  # Fails; can't modify a constant
```

If, instead, you want a local copy of the argument to work with inside the subroutine - leaving the caller's variable untouched - use the `is copy` trait:

```
sub say-it-one-higher($it is copy) {
    $it++;
    say $it;
}

my $unanswer = 41;
say-it-one-higher($unanswer);  # 42
say-it-one-higher(41);         # 42
```

In other programming languages, such as C/C++ and Scheme, this evaluation strategy is known as *pass-by-value*. When using the `is copy` trait, only the local copy is assigned; any arguments passed to the subroutine remain unchanged in the caller's scope.

The extra verbosity of marking parameters as mutable or immutable may seem excessive at first, but as an average user, it's unlikely that you'll need to use these traits often.

[2] An *immutable* object is an object whose state cannot be changed after it has been created. By contrast, a *mutable* object may be changed after being created.

4.2.2 Passing Arrays, Hashes and Code

A variable's sigil indicates its intended use. In a signature, a variable's sigil also acts as a constraint on the type of argument that can be passed. For example, the `@` sigil checks that the object passed does the `Positional` role (a role which includes types like `Array` and `List`). Failing to pass something that matches this constraint will cause the call to fail:

```
sub shout-them(@words) {
    for @words -> $w {
        print uc("$w ");
    }
}

my @last_words = <do not want>;

shout-them(@last_words);  # DO NOT WANT
shout-them('help');       # Fails; a string is not Positional
```

Similarly, the `%` sigil implies that the caller must pass an object that does the `Associative` role; that is, something which allows indexing through `<...>` or `{...}`. The `&` sigil requires the caller to pass an object that does the `Callable` role such as an anonymous subroutine. In that case, you may also call the callable parameter without the `&` sigil:

```
sub do-it-lots(&it, $how-many-times) {
    for 1..$how-many-times {
        it();
    }
}

do-it-lots(sub { say "Eating a stroopwafel" }, 10);
```

A scalar uses the `$` sigil and implies no constraints. Anything may bind to it, even if it could also bind to an object with one of the other sigils.

4.2.3 Interpolating Arrays and Hashes

Sometimes you want to fill positional arguments from an array. Instead of writing `eat(@food[0], @food[1], @food[2], ...)` and so on, you can *flatten* them into the argument list by prepending it with the vertical bar or "pipe" character (`|`): `eat(|@food)`.

Likewise, you can interpolate hashes into named arguments:

```
sub order-shrimps($count, :$from) {
    say "I'd like $count pieces of shrimp from the $from, please";
}

my %user-preferences = from => 'Northern Sea';

order-shrimps(3, |%user-preferences);
# I'd like 3 pieces of shrimp from the Northern Sea, please
```

4.2.4 Optional Parameters

Some parameters may have sensible default values, or may not be required for the subroutine to operate; they merely add some extra, optional, configurability. In this case, it is possible to mark the parameter as optional. Those calling the subroutine can then choose whether or not to supply an argument.

To make a parameter optional, either assign a default value to the parameter in the signature:

```
sub order-steak($how = 'medium') {
    say "I'd like a steak, $how";
}

order-steak();
order-steak('well done');
```

or append a question mark to the parameter's name:

```
sub order-burger($type, $side?) {
   say "I'd like a $type burger" ~
       ( defined($side) ?? " with a side of $side" !! "" );
```

```
}

order-burger("triple bacon", "deep fried onion rings");
```

If no argument is passed, an undefined value will be bound to the parameter. As demonstrated, the `defined(...)` function can be used to check if there is a value or not.

4.2.5 Required Parameters

Positional parameters are always required by default. However, you can explicitly specify that a parameter is required by appending a `!` to it:

```
sub order-drink($size, $flavor!) {
    say "$size $flavor, coming right up!";
}

order-drink('Large', 'Mountain Dew');    # OK
order-drink('Small');                    # Error
```

Required parameters must always appear at the beginning of a subroutine's parameter list.

4.2.6 Named Arguments and Parameters

When a subroutine has many parameters, it can become difficult for the caller to remember in what order they should pass the arguments. In this case, it is often easier to pass the arguments by name. When you do so, the order in which they appear does not matter:

```
sub order-beer($type, $pints) {
    say ($pints == 1 ?? 'A pint' !! "$pints pints") ~ " of $type, please."
}

order-beer(type => 'Hobgoblin', pints => 1);
# A pint of Hobgoblin, please.

order-beer(pints => 3, type => 'Zlatý Bažant');
# 3 pints of Zlatý Bažant, please.
```

You may also specify that a parameter may only ever be passed an argument by name (meaning that it is not allowed to pass it by position). To do this, precede the name of the parameter with a colon:

```
sub order-shrimps($count, :$from = 'Northern Sea') {
    say "I'd like $count pieces of shrimp from the $from, please";
}

order-shrimps(6);                       # takes 'Northern Sea'
order-shrimps(4, from => 'Atlantic Ocean');
order-shrimps(22, 'Mediterranean Sea'); # not allowed, :$from is named only
```

Unlike positional parameters, named parameters are optional by default. Append a `!` to make a named parameter mandatory.

```
sub design-ice-cream-mixture($base = 'Vanilla', :$name!) {
    say "Creating a new recipe named $name!"
}

design-ice-cream-mixture(name => 'Plain');
design-ice-cream-mixture(base => 'Strawberry chip'); # error, missing $name
```

Renaming Parameters

Since it is possible to pass arguments to parameters by name, the parameter names should be considered as part of a subroutine's public API. Choose them carefully! Sometimes it may be convenient to expose a parameter with one name while binding to a variable of a different name:

```
sub announce-time(:dinner($supper) = '8pm') {
    say "We eat dinner at $supper";
}

announce-time(dinner => '9pm');      # We eat dinner at 9pm
```

Parameters can also have multiple names. If some of your users are British and others are Americans, you might write:

```
sub paint-rectangle(
        :$x      =   0,
```

```
        :$y      =   0,
        :$width  = 100,
        :$height =  50,
        :color(:colour($c))) {

    # print a piece of SVG that represents a rectangle
    say qq[<rect x="$x" y="$y" width="$width" height="$height"
                  style="fill: $c" />]
}

# both calls work the same
paint-rectangle :color<Blue>;
paint-rectangle :colour<Blue>;

# of course you can still fill the other options
paint-rectangle :width(30), :height(10), :colour<Blue>;
```

Alternative Named Argument Syntaxes

Named arguments are actually `Pair`s (of keys and values). There are multiple ways to write `Pair`s. The difference between the approaches is primarily one of clarity, as each alternative provides a different quoting mechanism. These three calls all mean the same thing:

```
announce-time(dinner => '9pm');
announce-time(:dinner('9pm'));
announce-time(:dinner<9pm>);
```

If you're passing a boolean value, you may omit the value portion of the pair:

```
toggle-blender( :enabled); # enables  the blender
toggle-blender(:!enabled); # disables the blender
```

A named argument of the form `:name` with no value has an implicit value of `Bool::True`. The negated form of this, `:!name`, has an implicit value of `Bool::False`.

If you use a variable to create a pair, you can reuse the variable name as the key of the pair.

```
my $dinner = '9pm';
announce-dinner :$dinner;  # same as dinner => $dinner;
```

pairForms on page 37 lists possible `Pair` forms and their meanings.

Table 4.1: C<Pair> forms and their meanings

Shorthand	Long form	Description
`:allowed`	`allowed => Bool::True`	Boolean flag
`:!allowed`	`allowed => Bool::False`	Boolean flag
`:bev<tea coffee>`	`bev => ('tea', 'coffee')`	List
`:times[1, 3]`	`times => [1, 3]`	Array
`:opts{ a => 2 }`	`opts => { a => 2 }`	Hash
`:$var`	`var => $var`	Scalar variable
`:@var`	`var => @var`	Array variable
`:%var`	`var => %var`	Hash variable
`:&var`	`vaf => &var`	Callable/ Subroutine variable

You can use any of these forms in any context where you can use a `Pair` object. For example, when populating a hash:

```
# TODO: better example
my $black = 12;
my %color-popularities = :$black, :blue(8),
                         red => 18, :white<0>;
# same as
# my %color-popularities =
#       black => 12,
#       blue  => 8,
#       red   => 18,
#       white => 0;
```

Finally, to pass an existing `Pair` object to a subroutine by position, not name, either put it in parentheses (like `(:$thing)`), or use the `=>` operator with a quoted string on the left-hand side: `"thing" => $thing`.

Order of Parameters

When both positional and named parameters are present in the same signature, all the positional parameters need to come before the named parameters.

```
sub mix(@ingredients, :$name)    { ... }    # OK
sub notmix(:$name, @ingredients) { ... }    # Error
```

Required positional parameters need to come before optional positional parameters. However, named parameters have no such restriction.

```
sub copy-machine($amount, $size = 'A4', :$color!, :$quality) { ... }    # OK
sub fax-machine($amount = 1, $number)                        { ... }    # Error
```

4.2.7 Slurpy Parameters

Sometimes, you may wish to allow a subroutine to receive any number of arguments, and collect them all together into an array. In order to do this, add an array parameter to the signature, placing the *slurpy* prefix (`*`) before it.

```
sub shout-them(*@words) {
    for @words -> $w {
        print uc("$w ");
    }
}

# now you can pass items
shout-them('go');           # GO
shout-them('go', 'home');   # GO HOME
```

In addition to collecting all of the values, a slurpy parameter will also flatten any arrays that it collects, so that you end up with a single, flat list. Therefore:

```
my @words = ('go', 'home');
shout-them(@words);
```

Will result in the `*@words` parameter having two string elements, not just a single array element.

You may choose to capture some arguments into positional parameters and leave the rest to be captured by a slurpy array parameter. In this case, the slurpy should come last. Similarly, *%hash slurps all the remaining unbound named arguments into a hash.

Slurpy arrays and hashes allow you to pass all positional and named arguments to another routine:

```
sub debug-wrapper(&code, *@positional, *%named) {
    warn "Calling '&code.name()' with arguments "
         ~ "@positional.perl(), %named.perl()\n";
    code(|@positional, |%named);
    warn "... back from '&code.name()'\n";
}

debug-wrapper(&order-shrimps, 4, from => 'Atlantic Ocean');
```

4.3 Returning Results

Subroutines can also return values. The ASCII art dancing example from earlier in this chapter is simpler when each subroutine returns a new string:

```
my %moves =
   hands-over-head => sub { return '/o\ '   },
   bird-arms       => sub { return '|/o\| ' },
   left            => sub { return '>o '    },
   right           => sub { return 'o< '    },
   arms-up         => sub { return '\o/ '   };

my @awesome-dance = <arms-up bird-arms right hands-over-head>;

for @awesome-dance -> $move {
    print %moves{$move}.();
}

print "\n";
```

A subroutine can also return multiple values:

```
sub menu {
    if rand < 0.5 {
        return ('fish', 'white wine')
    } else {
        return ('steak', 'red wine');
    }
}

my ($food, $beverage) = menu();
```

If you exclude the `return` statement, then the value produced by the last statement run inside the subroutine will be returned. This means that the previous example may be simplified to:

```
sub menu {
    if rand < 0.5 {
        'fish', 'white wine'
    } else {
        'steak', 'red wine';
    }
}

my ($food, $beverage) = menu();
```

Be wary of relying on this: when the flow of control within a subroutine is sufficiently complex, adding an explicit `return` will clarify the code. As a general rule, only the simplest subroutines benefit from implicit `return`.

`return` has the additional effect of immediately exiting the subroutine:

```
sub create-world(*%characteristics) {
    my $world = World.new(%characteristics);
    return $world if %characteristics<temporary>;

    save-world($world);
}
```

... and you'd better not misplace your new `$world` if it's temporary, as it's the only one you're going to get.

4.4 Return Types

Like most other modern programming languages, Perl 6 gives you the option of explicitly specifying the return type of a subroutine. This allows you to restrict the data type of the value returned from a subroutine. This is done using the `returns` trait:

```
sub double-up($i) returns Int {
    return $i * 2;
}

my Int $ultimate-answer = double-up(21);    # 42
```

Using the `returns` trait is, of course, always optional. However, it may allow the compiler to perform certain optimizations depending on which Perl 6 implementation you are using.

4.5 Working With Types

Many subroutines cannot meaningfully work with arbitrary parameters, but require that the parameters support certain methods or have other properties. In these cases, it makes sense to restrict the types of parameters, such that attempts to pass incorrect values as arguments will cause Perl to raise an error at the time of calling the subroutine, or even at the compile time, if the compiler is smart enough to catch that[3].

4.5.1 Basic Types

The easiest way to restrict the possible values that a subroutine accepts is by writing a type name before a parameter. For example, a subroutine that performs numeric calculations on its parameters could require that its arguments are of the type `Numeric`:

```
sub mean(Numeric $a, Numeric $b) {
    return ($a + $b) / 2;
}

```

[3] at the time of writing, Rakudo recognizes type mismatches for literals and explicitly typed variables

```
say mean 2.5, 1.5;
say mean 'some', 'strings';
```

This produces the output:

```
2
Nominal type check failed for parameter '$a';
    expected Numeric but got Str instead
```

The *nominal* type is an actual type name, here `Numeric`, as opposed to a type plus additional constraints, which are discussed in the next section.

If multiple parameters have type constraints, each argument must fulfill the type constraint of the parameter to which it binds.

4.5.2 Adding Constraints

Sometimes a type name is insufficient to describe the requirements for an argument. In this case, you may add an additional *constraint* to the parameter with a `where` block:

```
sub circle-radius-from-area(Real $area where { $area >= 0 }) {
    ($area / pi).sqrt
}

say circle-radius-from-area(3);    # OK
say circle-radius-from-area(-3);   # Error
```

Because the calculation is meaningful only for non-negative area values, the parameter includes a constraint which returns `True` for non-negative values. If this constraint returns a false value, the type check will fail when something calls this subroutine[4].

The block after the `where` is optional; Perl performs the check by smart matching the argument against whatever follows the `where`. For example, it is possible to accept arguments in a certain range by writing:

[4] note that the nominal type is now `Real` instead of `Numeric`, because not all `Numeric` values can be compared for positiveness

```
sub set-volume(Numeric $volume where 0..11) {
    say "Turning it up to $volume";
}
```

Or one could constrain arguments to those that exist as keys of a hash:

```
my %in-stock = 'Staropramen' => 8, 'Mori' => 5, 'La Trappe' => 9;

sub order-beer(Str $name where %in-stock) {
    say "Here's your $name";
    %in-stock{$name}--;
    if %in-stock{$name} == 0 {
        say "OH NO! That was the last $name, folks! :'(";
        %in-stock.delete($name);
    }
}
```

4.6 Abstract and Concrete Parameters

TODO, XXX This section needs to be verified as accurate because even though it's correct according to the spec, Rakudo seems to allow type objects with :U. Also, add examples for each adverb.

One of the Perl5-isms that Perl 6 eliminates is the need to verify the "definedness" of a subroutine's arguments.

For example, the following Perl 5 code:

```
sub foo {
    my $arg = shift;
    die "Argument is undefined" unless defined $arg;

    # Do something
}
```

can now be written as:

```
sub foo(Int:D $arg) {
    # Do something
}
```

Notice the little smiley face attached to the parameter's type: `:D`. This adverb indicates that the given argument must be bound to a concrete object; if not, a runtime exception will be thrown. That's why it's so happy!

By contrast, the `:U` adverb can be used to indicate that the parameter requires an undefined or *abstract* object.

Additionally, the `:_` adverb allows either defined or undefined values. In practice, using `:_` is a bit redundant.

Lastly, the `:T` adverb can be used to indicate that the parameter may only be given as a type object. For example:

```
sub say-foobar(Int:T $arg) {
    say 'FOOBAR!';
}

say-foobar(Int);
# FOOBAR!
```

4.7 Captures

Signatures are not just syntax; instead, they are first-class objects that hold a list of `Parameter` objects. Likewise, there is a data structure that holds a collection of arguments, named a `Capture`. Just as you rarely think of a signature as a whole–instead focusing on individual parameters–you rarely have to think about captures. However, in some cases it is useful to do so, and therefore Perl allows you to manipulate captures directly.

Captures have both positional and named parts which act like lists and hashes. The list-like part contains the positional arguments and the hash-like parts contains the named arguments.

4.7.1 Creating And Using A Capture

Whenever you write a sub-routine call, you are implicitly creating a `Capture`. However, it is immediately consumed by the call. Sometimes you may wish to make a `Capture`, store it and then later apply a subroutine–or multiple subroutines– to the set of arguments it contains. To do this, use the `\(...)` syntax.

```
my @tasks = \(39, 3, action => { say $^a + $^b }),
            \(6, 7, action => { say $^a * $^b });
```

Here, the `@tasks` array will end up containing two `Captures`, each of which contains two positional arguments and one named argument.

It doesn't matter where in the capture the named arguments appear, because they are passed by name, not by position.

Like arrays and hashes, a `Capture` can be flattened into an argument list using `|`:

```
sub act($left, $right, :$action) {
    $action($left, $right);
}

for @tasks -> $task-args {
    act(|$task-args);
}
```

However, in this case it is specifying the full set of arguments for the call, including both named and positional arguments.

Unlike signatures, captures work like references. Any variable mentioned in a capture exists in the capture as a *reference* to the variable. Thus `rw` parameters still work with captures involved.

```
my $value     = 7;
my $to-change = \($value);

sub double($x is rw) {
    $x *= 2;
}

```

```
sub triple($x is rw) {
    $x *= 3;
}

triple(|$to-change);
double(|$to-change);

say $value; # 42
```

Perl types with both positional and named parts also show up in various other situations. For example, regex matches have both positional and named matches–`Match` objects themselves are a type of capture. It's also possible to conceive of an XML node type that is a type of capture, with named attributes and positional children. Binding this node to a function could use the appropriate parameter syntax to work with various children and attributes.

4.7.2 Captures In Signatures

All calls build a capture on the caller side and unpack it according to the signature on the callee side[5]. It is also possible to write a signature that binds the capture itself into a variable. This is especially useful for writing routines that delegate to other routines with the same arguments.

```
sub visit-czechoslovakia(|$plan) {
    warn "Sorry, this country has been deprecated.";
    visit-slovakia(|$plan);
    visit-czech-republic(|$plan);
}
```

The benefit of using this over a signature like `:(*@pos, *%named)` is that these both enforce some context on the arguments, which may be premature. For example, if the caller passes two arrays, they would flatten into `@pos`. This means that the two nested arrays could not be recovered at the point of delegation. A capture preserves the two array arguments, so that the final callee's signature may determine how to bind them.

[5] An optimizing Perl 6 compiler may, of course, be able to optimize away part or all of this process, depending on what it knows at compilation time.

4.8 Unpacking

Sometimes you need to work with only part of an array or a hash. You can do that with ordinary slicing access, or you can use signature binding:

```
sub first-is-largest(@a) {
    my $first = @a.shift;
    # TODO: either explain junctions, or find a
    # concise way to write without them
    return $first >= all(@a);
}

# same thing:
sub first-is-largest(@a) {
    my :($first, *@rest) := \(|@a)
    return $first >= all(@rest);
}
```

The signature binding approach might seem clumsy, but when you use it in the main signature of a subroutine, you get tremendous power:

```
sub first-is-largest([$first, *@rest]) {
    return $first >= all(@rest);
}
```

The brackets in the signature tell the compiler to expect a list-like argument. Instead of binding to an array parameter, it instead *unpacks* its arguments into several parameters–in this case, a scalar for the first element and an array for the rest. This *subsignature* also acts as a constraint on the array parameter: the signature binding will fail unless the list in the capture contains at least one item.

Likewise you can unpack a hash by using `%(...)` instead of square brackets, but you must access named parameters instead of positional.

```
sub create-world(%(:$temporary, *%characteristics)) {
    my $world = World.new(%characteristics);
    return $world if $temporary;

    save-world($world);
}
```

TODO: come up with a good example # maybe steal something from http://jnthn.net/papers/2010-yapc-eu-signatures.pdf

TODO: generic object unpacking

4.9 Currying

Consider a module that provided the example from the "Optional Parameters" section:

```
sub order-burger( $type, $side? ) { ... };
```

If you used `order-burger` repeatedly, but often with a side of french fries, you might wish that the author had also provided a `order-burger-and-fries` sub. You could easily write it yourself:

```
sub order-burger-and-fries ( $type ) {
    order-burger( $type, side => 'french fries' );
}
```

If your personal order is always vegetarian, you might instead wish for a `order-the-usual` sub. This is less concise to write, due to the optional second parameter:

```
sub order-the-usual ( $side? ) {
    if ( $side.defined ) {
        order-burger( 'veggie', $side );
    }
    else {
        order-burger( 'veggie' );
    }
}
```

Currying gives you a shortcut for these exact cases; it creates a new sub from an existing sub, with parameters already filled in. In Perl 6, curry with the `.assuming` method:

```
&order-the-usual        := &order-burger.assuming( 'veggie' );
&order-burger-and-fries := &order-burger.assuming( side => 'french fries' );
```

The new sub is like any other sub, and works with all the various parameter-passing schemes already described.

```
order-the-usual( 'salsa' );
order-the-usual( side => 'broccoli' );

order-burger-and-fries( 'plain' );
order-burger-and-fries( :type<<double-beef>> );
```

4.10 Introspection

Subroutines and their signatures are objects like any other. Besides calling them, you can learn things about them, including the details of their parameters:

```
sub logarithm(Numeric $x, Numeric :$base = exp(1)) {
    log($x) / log($base);
}

my @params = &logarithm.signature.params;
say @params.elems, ' parameters';

for @params {
    say "Name:       ", .name;
    say "  Type:     ", .type;
    say "  named?    ", .named    ?? 'yes' !! 'no';
    say "  slurpy?   ", .slurpy   ?? 'yes' !! 'no';
    say "  optional? ", .optional ?? 'yes' !! 'no';
}
```

```
2 parameters
Name:       $x
  Type:     Numeric()
  named?    no
  slurpy?   no
  optional? no
Name:       $base
  Type:     Numeric()
  named?    yes
  slurpy?   no
  optional? yes
```

The `&` sigil followed by a subroutine name gets the object representing that subroutine. `&logarithm.signature` returns the signature associated with the subroutine, and calling `.params` on the signature returns a list of `Parameter` objects. Each of these objects describes one parameter in detail.

Table 4.2: Methods in the Parameter class

```
# stolen straight from S06, adapted a bit
```

method	description
name	The name of the lexical variable to bind to, if any
type	The nominal type
constraints	Any further type constraints
readonly	True if the parameter has `is readonly` trait
rw	True if the parameter has `is rw` trait
copy	True if the parameter has `is copy` trait
named	True if the parameter is to be passed by name
named_names	List of names a named parameter can be passed as
slurpy	True if the parameter is slurpy
optional	True if the parameter is optional
default	A closure returning the default value
signature	A nested signature to bind the argument against

```
# TODO: talk about &signature.cando once that's implemented
```

Signature introspection allows you to build interfaces that can obtain and then pass the right data to a subroutine. For example, you could build a web form generator that knew how to get input from a user, validate it, and then call a routine with it based upon the information obtained through introspection. A similar approach might generate a command line interface along with some basic usage instructions.

Beyond this, traits (`traits`) allow you to associate extra data with parameters. This metadata can go far beyond that which subroutines, signatures, and parameters normally provide.

4.11 The `MAIN` Subroutine

Frequent users of the UNIX shells might have noticed a symmetry between postional and named arguments to routines on the one hand, and argument and options on the command line on the other hand.

This symmetry can be exploited by declaring a subroutine called `MAIN`. It is called every time the script is run, and its signature counts as a specification for command line arguments.

```
# script roll-dice.pl
sub MAIN($count = 1, Numeric :$sides = 6, Bool :$sum) {
    my @numbers = (1..$sides).roll($count);
    say @numbers.join(' ');
    say "sum: ", [+] @numbers if $sum;
}
# TODO: explain ranges, .pick and [+]
# (or reference later chapters)
# note: [max] was already used in chapter 3, so [+] should
#       make sense to the reader already.
```

Calling this script without command line arguments leaves all three parameters at their default values.

This is how some invocations could look like [6]>

```
$ perl6 roll-dice.pl
4
$ perl6 roll-dice.pl 4
1 4 2 4
$ perl6 roll-dice.pl --sides=20 3
18 1 2
$ perl6 roll-dice.pl --sides=20 --sum 3
9 14 12
sum: 35
$ perl6 roll-dice.pl --unknown-option
Usage:
  roll-dice.pl [--sides=<Numeric>] [--sum] [<count>]
```

[6] <The use of random numbers through `roll` means that you will likely see different output if you try it.

Named arguments can be provided with the `--name=value` syntax common to many GNU tools, and positional arguments just by their value.

If an option does not require a parameter, the parameter in the MAIN signature needs to be marked as being of `Bool` type.

Usage of unknown options or too many or few arguments triggers an automatically generated usage message, which can be overridden with a custom `USAGE` sub:

```
sub MAIN(:$really-do-it!) { ... }
sub USAGE() {
    say "This script is dangerous, please read the documentation first";
}
```

TODO: maybe talk more about available options?

5

Classes and Objects

TODO: start with a much simpler bare-bones example!

The following program shows how a dependency handler might look in Perl 6. It showcases custom constructors, private and public attributes, methods and various aspects of signatures. It's not very much code, and yet the result is interesting and, at times, useful.

```
class Task {
    has      &!callback;
    has Task @!dependencies;
    has Bool $.done;

    method new(&callback, Task *@dependencies) {
        return self.bless(*, :&callback, :@dependencies);
    }

    method add-dependency(Task $dependency) {
        push @!dependencies, $dependency;
    }

    method perform() {
        unless $!done {
```

```
16              .perform() for @!dependencies;
17              &!callback();
18              $!done = True;
19          }
20      }
21  }
22
23  my $eat =
24      Task.new({ say 'eating dinner. NOM!' },
25          Task.new({ say 'making dinner' },
26              Task.new({ say 'buying food' },
27                  Task.new({ say 'making some money' }),
28                  Task.new({ say 'going to the store' })
29              ),
30              Task.new({ say 'cleaning kitchen' })
31          )
32      );
33
34  $eat.perform();
```

5.1 Starting with class

Perl 6, like many other languages, uses the `class` keyword to introduce a new class. The block that follows may contain arbitrary code, just as with any other block, but classes commonly contain state and behavior declarations. The example code includes attributes (state), introduced through the `has` keyword, and behaviors introduced through the `method` keyword.

Declaring a class creates a *type object* which, by default, is installed into the current package (just like a variable declared with `our` scope). This type object is an "empty instance" of the class. You've already seen these in previous chapters. For example, types such as `Int` and `Str` refer to the type object of one of the Perl 6 built- in classes. The example above uses the class name `Task` so that other code can refer to it later, such as to create class instances by calling the `new` method.

Type objects are *undefined*, in the sense that they return `False` if you call the `.defined` method on them. You can use this method to find out if a given object is a type object or not:

```
my $obj = Int;
if $obj.defined {
    say "Ordinary, defined object";
} else {
    say "Type object";
}
```

5.2 I can has state?

The first three lines inside the class block all declare attributes (called *fields* or *instance storage* in other languages). These are storage locations that every instance of a class will obtain. Just as a `my` variable can not be accessed from the outside of its declared scope, attributes are not accessible outside of the class. This *encapsulation* is one of the key principles of object oriented design.

The first declaration specifies instance storage for a callback – a bit of code to invoke in order to perform the task that an object represents:

```
has &!callback;
```

The `&` sigil indicates that this attribute represents something invocable. The `!` character is a *twigil*, or secondary sigil. A twigil forms part of the name of the variable. In this case, the `!` twigil emphasizes that this attribute is private to the class.

The second declaration also uses the private twigil:

```
has Task @!dependencies;
```

However, this attribute represents an array of items, so it requires the `@` sigil. These items each specify a task that must be completed before the present one can complete. Furthermore, the type declaration on this attribute indicates that the array may only hold instances of the `Task` class (or some subclass of it).

The third attribute represents the state of completion of a task:

```
has Bool $.done;
```

This scalar attribute (with the `$` sigil) has a type of `Bool`. Instead of the `!` twigil, this twigil is `.`. While Perl 6 does enforce encapsulation on attributes, it also saves you from writing accessor methods. Replacing the `!` with a `.` both declares the attribute `$!done` and an accessor method named `done`. It's as if you had written:

```
has Bool $!done;
method done() { return $!done }
```

Note that this is not like declaring a public attribute, as some languages allow; you really get *both* a private storage location and a method, without having to write the method by hand. You are free instead to write your own accessor method, if at some future point you need to do something more complex than return the value.

Note that using the `.` twigil has created a method that will provide with readonly access to the attribute. If instead the users of this object should be able to reset a task's completion state (perhaps to perform it again), you can change the attribute declaration:

```
has Bool $.done is rw;
```

The `is rw` trait causes the generated accessor method to return something external code can modify to change the value of the attribute.

5.3 Methods

While attributes give objects state, methods give objects behaviors. Ignore the `new` method temporarily; it's a special type of method. Consider the second method, `add-dependency`, which adds a new task to this task's dependency list.

```
method add-dependency(Task $dependency) {
    push @!dependencies, $dependency;
}
```

In many ways, this looks a lot like a `sub` declaration. However, there are two important differences. First, declaring this routine as a method adds it to the list of methods for the current class. Thus any instance of the `Task` class can call this method with the `.` method call operator. Second, a method places its invocant into the special variable `self`.

The method itself takes the passed parameter–which must be an instance of the `Task` class–and pushes it onto the invocant's `@!dependencies` attribute.

The second method contains the main logic of the dependency handler:

```
method perform() {
    unless $!done {
        .perform() for @!dependencies;
        &!callback();
        $!done = True;
    }
}
```

It takes no parameters, working instead with the object's attributes. First, it checks if the task has already completed by checking the `$!done` attribute. If so, there's nothing to do.

Otherwise, the method performs all of the task's dependencies, using the `for` construct to iterate over all of the items in the `@!dependencies` attribute. This iteration places each item–each a `Task` object–into the topic variable, `$_`. Using the `.` method call operator without specifying an explicit invocant uses the current topic as the invocant. Thus the iteration construct calls the `.perform()` method on every `Task` object in the `@!dependencies` attribute of the current invocant.

After all of the dependencies have completed, it's time to perform the current `Task`'s task by invoking the `&!callback` attribute directly; this is the purpose of the parentheses. Finally, the method sets the `$!done` attribute to `True`, so that subsequent invocations of `perform` on this object (if this `Task` is a dependency of another `Task`, for example) will not repeat the task.

5.4 Constructors

Perl 6 is rather more liberal than many languages in the area of constructors. A constructor is anything that returns an instance of the class. Furthermore, constructors are ordinary methods. You inherit a default constructor named `new` from the base class `Object`, but you are free to override `new`, as this example does:

```
method new(&callback, Task *@dependencies) {
    return self.bless(*, :&callback, :@dependencies);
}
```

The biggest difference between constructors in Perl 6 and constructors in languages such as C# and Java is that rather than setting up state on a somehow already magically created object, Perl 6 constructors actually create the object themselves. This easiest way to do this is by calling the `bless` method, also inherited from `Object`. The `bless` method expects a positional parameter–the so-called "candidate"–and a set of named parameters providing the initial values for each attribute.

The example's constructor turns positional arguments into named arguments, so that the class can provide a nice constructor for its users. The first parameter is the callback (the thing to do to execute the task). The rest of the parameters are dependent `Task` instances. The constructor captures these into the `@dependencies` slurpy array and passes them as named parameters to `bless` (note that `:&callback` uses the name of the variable–minus the sigil–as the name of the parameter).

5.5 Consuming our class

After creating a class, you can create instances of the class. Declaring a custom constructor provides a simple way of declaring tasks along with their dependencies. To create a single task with no dependencies, write:

```
1  my $eat = Task.new({ say 'eating dinner. NOM!' });
```

An earlier section explained that declaring the class `Task` installed a type object in the namespace. This type object is a kind of "empty instance" of the class, specifically an instance without any state. You can call methods on that instance, as long as they do not try to access any state; `new` is an example, as it creates a new object rather than modifying or accessing an existing object.

Unfortunately, dinner never magically happens. It has dependent tasks:

```
1  my $eat =
2      Task.new({ say 'eating dinner. NOM!' },
3          Task.new({ say 'making dinner' },
4              Task.new({ say 'buying food' },
5                  Task.new({ say 'making some money' }),
6                  Task.new({ say 'going to the store' })
7              ),
8              Task.new({ say 'cleaning kitchen' })
```

```
        )
    );
```

Notice how the custom constructor and sensible use of whitespace allows a layout which makes task dependencies clear.

Finally, the `perform` method call recursively calls the `perform` method on the various other dependencies in order, giving the output:

```
making some money
going to the store
buying food
cleaning kitchen
making dinner
eating dinner. NOM!
```

5.6 Inheritance

Object Oriented Programming provides the concept of inheritance as one of the mechanisms to allow for code reuse. Perl 6 supports the ability for one class to inherit from one or more classes. When a class inherits from another class that informs the method dispatcher to follow the inheritance chain to look for a method to dispatch. This happens both for standard methods defined via the method keyword and for methods generated through other means such as attribute accessors.

TODO: the example here is rather bad, and needs to be replaced (or much improved). See `https://github.com/perl6/book/issues/58` for discussion.

```
class Employee {
    has $.salary;

    method pay() {
        say "Here is \$$.salary";
    }

}

class Programmer is Employee {
    has @.known_languages is rw;
```

```
    has $.favorite_editor;

    method code_to_solve( $problem ) {
        say "Solving $problem using $.favorite_editor in "
        ~ $.known_languages[0] ~ '.';
    }
}
```

Now any object of type Programmer can make use of the methods and accessors defined in the Employee class as though they were from the Programmer class.

```
my $programmer = Programmer.new(
    salary => 100_000,
    known_languages => <Perl5 Perl6 Erlang C++>,
    favorite_edtor => 'vim'
);

$programmer.code_to_solve('halting problem');
$programmer.pay();
```

5.6.1 Overriding Inherited Methods

Of course, classes can override methods and attributes defined on ancestoral classes by defining their own. The example below demonstrates the `Baker` class overriding the `Cook`'s `cook` method.

```
class Cook is Employee {
    has @.utensils  is rw;
    has @.cookbooks is rw;

    method cook( $food ) {
        say "Cooking $food";
    }

    method clean_utensils {
        say "Cleaning $_" for @.utensils;
    }
}

class Baker is Cook {
```

```
15      method cook( $confection ) {
16          say "Baking a tasty $confection";
17      }
18  }
19
20  my $cook = Cook.new(
21      utensils => (<spoon ladel knife pan>),
22      cookbooks => ('The Joy of Cooking'),
23      salary => 40000);
24
25  $cook.cook( 'pizza' ); # Cooking pizza
26
27  my $baker = Baker.new(
28      utensils => ('self cleaning oven'),
29      cookbooks => ("The Baker's Apprentice"),
30      salary => 50000);
31
32  $baker.cook('brioche'); # Baking a tasty brioche
```

Because the dispatcher will see the `cook` method on `Baker` before it moves up to the parent class the `Baker`'s `cook` method will be called.

5.6.2 Multiple Inheritance

As mentioned before, a class can inherit from multiple classes. When a class inherits from multiple classes the dispatcher knows to look at both classes when looking up a method to search for. As a side note, Perl 6 uses the C3 algorithm to linearize the multiple inheritance hierarchies, which is a significant improvement over Perl 5's approach to handling multiple inheritance.

```
1   class GeekCook is Programmer is Cook {
2       method new( *%params ) {
3           %params<cookbooks> //= [];  # remove once rakudo fully supports autovivification
4           push( %params<cookbooks>, "Cooking for Geeks" );
5           return self.bless(*, |%params);
6       }
7   }
8
9   my $geek = GeekCook.new(
10      books           => ('Learning Perl 6'),
```

```
    utensils        => ('blingless pot', 'knife', 'calibrated oven'),
    favorite_editor => 'MacVim',
    known_languages => <Perl6>
);

$geek.cook('pizza');
$geek.code_to_solve('P =? NP');
```

Now all the methods made available by both the Programmer class and the Cook class are available from the GeekCook class.

While multiple inheritance is a useful concept to know and on occasion of use, it is important to understand that there are more useful OOP concepts. When reaching for multiple inheritance it is good practice to consider whether the design wouldn't be better realized by using roles, which are generally safer because they force the class author to explicitly resolve conflicting method names. For more information on roles see *sec:roles* on page 79.

5.7 Introspection

Introspection is the process of gathering information about some objects in your program, not by reading the source code, but by querying the object (or a controlling object) for some properties, like its type.

Given an object `$p`, and the class definitions from the previous sections, we can ask it a few questions:

```
if $o ~~ Employee { say "It's an employee" };
if $o ~~ GeekCook { say "It's a geeky cook" };
say $o.WHAT;
say $o.perl;
say $o.^methods(:local).join(', ');
```

The output can look like this:

```
It's an employee
Programmer()
Programmer.new(known_languages => ["Perl", "Python", "Pascal"],
```

```
        favorite_editor => "gvim", salary => "too small")
code_to_solve, known_languages, favorite_editor
```

The first two tests each smart-match against a class name. If the object is of that class, or of an inheriting class, it returns true. So the object in question is of class `Employee` or one that inherits from it, but not `GeekCook`.

The `.WHAT` method returns the type object associated with the object `$o`, which tells the exact type of `$o`: in this case `Programmer`.

`$o.perl` returns a string that can be executed as Perl code, and reproduces the original object `$o`. While this does not work perfectly in all cases[1], it is very useful for debugging simple objects.

Finally `$o.^methods(:local)` produces a list of methods that can be called on `$o`. The `:local` named argument limits the returned methods to those defined in the `Employee` class, and excludes the inherited methods.

The syntax of calling method with `.^` instead of a single dot means that it is actually a method call on the *meta class*, which is a class managing the properties of the `Employee` class - or any other class you are interested in. This meta class enables other ways of introspection too:

```
say $o.^attributes.join(', ');
say $o.^parents.join(', ');
```

Introspection is very useful for debugging, and for learning the language and new libraries. When a function or method returns an object you don't know about, finding its type with `.WHAT`, a construction recipe for it with `.perl` and so on you'll get a good idea what this return value is. With `.^methods` you can learn what you can do with it.

But there are other applications too: a routine that serializes objects to a bunch of bytes needs to know the attributes of that object, which it can find out via introspection.

[1] For example closures cannot easily be reproduced this way; if you don't know what a closure is don't worry. Also current implementations have problems with dumping cyclic data structures this way, but they are expected to be handlded correctly by `.perl` at some point.

5.8 Exercises

1. The method `add-dependency` in `Task` permits the creation of *cycles* in the dependency graph. That is, if you follow dependencies, you can eventually return to the original `Task`. Show how to create a graph with cycles and explain why the `perform` method of a `Task` whose dependencies contain a cycle would never terminate successfully.

Answer: You can create two tasks, and then "short-circuit" them with `add-dependency`:

```
my $a = Task.new({ say 'A' });
my $b = Task.new({ say 'B' }, $a);
$a.add-dependency($b);
```

The `perform` method will never terminate because the first thing the method does is to call all the `perform` methods of its dependencies. Because `$a` and `$b` are dependencies of each other, none of them would ever get around to calling their callbacks. The program will exhaust memory before it ever prints `'A'` or `'B'`.

2. Is there a way to detect the presence of a cycle during the course of a `perform` call? Is there a way to prevent cycles from ever forming through `add-dependency`?

Answer: To detect the presence of a cycle during a `perform` call, keep track of which `Task`s have started; prevent a `Task` from starting twice before finishing:

```
augment class Task {
    has Bool $!started = False;

    method perform() {
        if $!started++ && !$!done {
            die "Cycle detected, aborting";
        }

        unless $!done {
            .perform() for @!dependencies;
            &!callback();
            $!done = True;
        }
    }
}
```

Another approach is to stop cycles from forming during `add-dependency` by checking whether there's already a dependency running in the other direction. (This is the only situation in which a cycle can occur.) This requires the addition of a helper method `depends-on`, which checks whether a task depends on another one, either directly or transitively. Note the use of » and `[||]` to write succinctly what would otherwise have involved looping over all the dependencies of the `Task`:

```
augment class Task {
    method depends-on(Task $some-task) {
        $some-task === any(@!dependencies)
        [||] @!dependencies».depends-on($some-task)
    }

    method add-dependency(Task $dependency) {
        if $dependency.depends-on(self) {
            warn 'Cannot add that task, since it would introduce a cycle.';
            return;
        }
        push @!dependencies, $dependency;
    }
}
```

3. How could `Task` objects execute their dependencies in parallel? (Think especially about how to avoid collisions in "diamond dependencies", where a `Task` has two different dependencies which in turn have the same dependency.)

Answer: Enabling parallelism is easy; change the line `.perform() for @!dependencies;` into `@!dependencies».perform()`. However, there may be race conditions in the case of diamond dependencies, wherein `Tasks A` starts `B` and `C` in parallel, and both start a copy of `D`, making `D` run twice. The solution to this is the same as with the cycle-detection in Question 2: introducing an attribute `$!started`. Note that it's impolite to die if a `Task` has started but not yet finished, because this time it might be due to parallelism rather than cycles:

```
augment class Task {
    has Bool $!started = False;

    method perform() {
        unless $!started++ {
            @!dependencies».perform();
            &!callback();
            $!done = True;
```

```
9                   }
10              }
11          }
```

6

Multis

Perl usually decides which function to call based on the name of the function or the contents of a function reference. This is simple to understand. Perl can also examine the contents of the arguments provided to decide which of several variants of a function–each with the same name–to call. In this case, the amount and types of the function's arguments help to distinguish between the variants of a function. This is called *multidispatch*, and the functions to which Perl can dispatch in this case are *multis*.

Javascript Object Notation (*JSON*) is a simple data exchange format often used for communicating with web services. It supports arrays, hashes, numbers, strings, boolean values, and `null`, the undefined value. Here you can find an implementation that turns Perl 6 data structures to JSON.

This snippet demonstrates how multis make the code simpler and more obvious.

The other way round, converting a JSON string to a Perl 6 data structure, is covered in the chapter *sec:grammars* on page 109.

> The code presented here is runnable. It is part of the libary `JSON::Tiny`, which is available from `http://github.com/moritz/json/`. It also includes tests and documentation.

```
multi to-json(Bool $d) { $d ?? 'true' !! 'false'; }
multi to-json(Real $d) { ~$d }
multi to-json(Str  $d) {
    '"'
    ~ $d.trans(['"',  '\\',   "\b", "\f", "\n", "\r", "\t"]
            => ['\"', '\\\\', '\b', '\f', '\n', '\r', '\t'])
    ~ '"'
}

multi to-json(Array $d) {
    return  '[ '
            ~ $d.values.map({ to-json($_) }).join(', ')
            ~ ' ]';
}

multi to-json(Hash  $d) {
    return '{ '
            ~ $d.pairs.map({ to-json(.key)
            ~ ' : '
            ~ to-json(.value) }).join(', ')
            ~ ' }';
}

multi to-json($d where {!defined $d}) { 'null' }

multi to-json($d) {
    die "Can't serialize an object of type " ~ $d.WHAT.perl
}
```

This code defines a single multi sub named `to-json`, which takes one argument and turns that into a string. `to-json` has many *candidates*; these subs all have the name `to-json` but differ in their signatures. Every candidate resembles:

```
multi to-json(Bool $data) { ... }
multi to-json(Real $data) { ... }
```

Which one is actually called depends on the type of the data passed to the subroutine. A call such as `to-json(Bool::True)` invokes the first candidate. Passing a numeric value of type `Real` instead invokes the second.

The candidate for handling `Real` is very simple; because JSON's and Perl 6's number formats coincide, the JSON converter can rely on Perl's conversion of these numbers to strings. The `Bool` candidate returns a literal string `'true'` or `'false'`.

The `Str` candidate does more work: it wraps its parameter in quotes and escapes literal characters that the JSON spec does not allow in strings–a tab character becomes `"\t"`, a newline `"\n"`, and so on.

The `to-json(Array $d)` candidate converts all elements of the array to JSON with recursive calls to `to-json`, joins them with commas, and surrounds them with square brackets. The recursive calls demonstrate a powerful truth of multidispatch: these calls do not necessarily recurse to the `Array` candidate, but dispatch to the appropriate candidate based on the types of *their* arguments.

The candidate that processes hashes turns them into the form `{ "key1" : "value1", "key2" : [ "second", "value" ] }`. It does this again by recursing into `to-json`.

6.1 Constraints

Candidates can specify more complex signatures:

```
multi to-json($d where {!defined $d}) { 'null' }
```

This candidate adds two new twists. It contains no type definition, in which case the type of the parameter defaults to `Any`, the root of the normal branch of the type hierarchy. More interestingly, the `where {!defined $d}` clause is a *constraint*, which defines a so-called *subset type*. This candidate will match only *some* values of the type `Any`–those where the value is undefined.

Whenever the compiler performs a type check on the parameter `$d`, it first checks the *nominal* type (here, `Any`). A nominal type is an actual class or role, as opposed to additional constraints in the form of code blocks.

If that check succeeds, it calls the code block. The entire type check can only succeed if the code block returns a true value.

The curly braces for the constraint can contain arbitrary code. You can abuse this to count how often a type check occurs:

```
my $counter = 0;

multi a(Int $x)  { }
multi a($x)      { }
multi a($x where { $counter++; True }) { }

a(3);
say $counter;       # says 0
a('str');
say $counter;       # says 2
```

This code defines three multis, one of which increases a counter whenever its `where` clause executes. Any Perl 6 compiler is free to optimize away type checks it knows will succeed. In the current Rakudo implementation, the second line with `say` will print a higher number than the first.

In the first call of `a(3)`, the nominal types alone already determine the best candidate match, so the where block never executes and the first `$counter` output is always `0`.

The output after the second call is at least `1`. The compiler has to execute the `where`-block at least once to check if the third candidate is the best match, but the specification does not require the *minimal* possible number of runs. This is illustrated in the second `$counter` output. The specific implementation used to run this test actually executes the `where`-block twice. Keep in mind that the number of times the subtype checks blocks execute is specific to any particular implementation of Perl 6.

> Avoid writing code like this in anything other than example code. Relying on the side effects of type checks produces unreliable code.

6.2 Narrowness

One candidate remains from the JSON example:

```
multi to-json($d) {
    die "Can't serialize an object of type " ~ $d.WHAT.perl
}
```

With no explicit type or constraint on the parameter `$d`, its type defaults to `Any`–and thus it matches any passed object. The body of this function complains that it doesn't know what to do with the argument. This works for the example, because JSON's specification covers only a few basic structures.

The declaration and intent may seem simple at first, but look closer. This final candidate matches not only objects for which there is no candidate defined, but it can match for *all* objects, including `Int`, `Bool`, `Num`. A call like `to-json(2)` has *two* matching candidates–`Int` and `Any`.

If you run that code, you'll discover that the `Int` candidate gets called. Because `Int` is a type that conforms to `Any`, it is a *narrower* match for an integer. Given two types `A` and `B`, where `A` conforms to `B` (`A ~~ B`, in Perl 6 code), an object which conforms to `A` does so more narrowly than to `B`. In the case of multi dispatch, the narrowest match always wins.

A successfully evaluated constraint makes a match narrower than a similar signature without a constraint. In the case of:

```
multi to-json($d) { ... }
multi to-json($d where {!defined $d}) { ... }
```

... an undefined value dispatches to the second candidate.

However, a matching constraint always contributes less to narrowness than a more specific match in the nominal type.

```
TODO: Better example

multi a(Any $x where { $x > 0 }) { 'Constraint'   }
multi a(Int $x)                  { 'Nominal type' }

say a(3), ' wins';       # says Nominal type wins
```

This restriction allows a clever compiler optimization: it can sort all candidates by narrowness once to find the candidate with the best matching signature by examining nominal type constraints. These are far cheaper to check than constraint checks. Constraint checking occurs next, and only if the constraint check of the narrowest candidate fails, other candidates are tried that are lass narrow by nominal type.

The `Int` type object both conforms to `Int`, but it is also an undefined value. If you pass it to the multi a, the second candidate, which is specific to `Int` wins, because nominal types are checked first.

6.3 Multiple arguments

Candidate signatures may contain any number of positional and named arguments, both explicit and slurpy. However only positional parameters contribute to the narrowness of a match:

```
class Rock     { }
class Paper    { }
class Scissors { }

multi wins(Scissors $, Paper    $) { +1 }
multi wins(Paper    $, Rock     $) { +1 }
multi wins(Rock     $, Scissors $) { +1 }
multi wins(::T      $, T        $) {  0 }
multi wins(         $,          $) { -1 }

sub play($a, $b) {
    given wins($a, $b) {
        when +1 { say 'Player One wins' }
        when  0 { say 'Draw'            }
        when -1 { say 'Player Two wins' }
    }
}

play(Scissors, Paper);
play(Paper,    Paper);
play(Rock,     Paper);
```

This example demonstrates how multiple dispatch can encapsulate all of the rules of a popular game. Both players independently select a symbol (rock, paper, or scissors). Scissors win against paper, paper wraps rock, and scissors can't cut rock, but go blunt trying. If both players select the same item, it's a draw.

The code creates a class for each possible symbol. For each combination of chosen symbols for which Player One wins there's a candidate of the form:

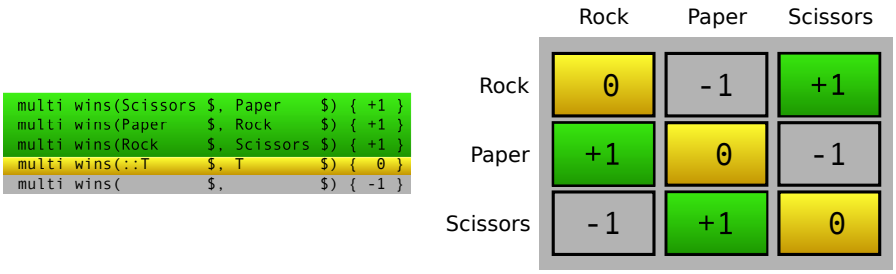

Figure 6.1: Who wins the *Rock, Paper, Scissors* game?

```
multi wins(Scissors $, Paper $) { +1 }
```

Because the bodies of the subs here do not use the parameters, there's no reason to force the programmer to name them; they're *anonymous parameters*. A single `$` in a signature identifies an anonymous scalar variable.

The fourth candidate, `multi wins(::T $, T $) { 0 }` uses `::T`, which is a *type capture* (similar to *generics* or *templates* in other programming languages). It binds the nominal type of the first argument to `T`, which can then act as a type constraint. If you pass a `Rock` as the first argument, `T` acts as an alias for `Rock` inside the rest of the signature and the body of the routine. The signature `(::T $, T $)` will bind only two objects of the same type, or where the second is a subtype of the first.

In this game, that fourth candidate matches only for two objects of the same type. The routine returns `0` to indicate a draw.

The final candidate is a fallback for the cases not covered yet–every case in which Player Two wins.

If the `(Scissors, Paper)` candidate matches the supplied argument list, it is two steps narrower than the `(Any, Any)` fallback, because both `Scissors` and `Paper` are direct subtypes of `Any`, so both contribute one step.

If the `(::T, T)` candidate matches, the type capture in the first parameter does not contribute any narrowness–it is not a constraint, after all. However `T` *is* a constraint for the second parameter which accounts for as many steps of narrowness as the number of inheritance steps between `T` and `Any`. Passing two `Rocks` means that `::T, T` is one step narrower than `Any, Any`. A possible candidate:

```
multi wins(Rock $, Rock $) {
    say "Two rocks? What is this, 20,000 years ago?"
}
```

... would win against `(::T, T)`.

6.4 Bindability checks

Traits can apply *implicit constraints*:

```
multi swap($a is rw, $b is rw) {
    ($a, $b) = ($b, $a);
}
```

This routine exchanges the contents of its two arguments. It must bind the two arguments as `rw`–both readable and writable. Calling the `swap` routine with an immutable value (for example a number literal) will fail.

The built-in function `substr` can not only extract parts of strings, but also modify them:

```
# substr(String, Start, Length)
say substr('Perl 5', 0, 4);         # prints Perl

my $p = 'Perl 5';
# substr(String, Start, Length, Substitution)
substr($p, 6, 1, '6');
# now $p contains the string Perl 6
```

You already know that the three-argument version and the four-argument version have different candidates: the latter binds its first argument as `rw`:

```
multi substr($str, $start = 0, $length = *) { ... }
multi substr($str is rw, $start, $length, $substitution) { ... }
```

This is also an example of candidates with different *arity* (number of expected arguments). This is seldom really necessary, because it is often a better alternative to make parameters optional. Cases where an arbitrary number of arguments are allowed are handled with slurpy parameters instead:

```
sub mean(*@values) {
    ([+] @values) / @values;
}
```

6.5 Nested Signatures in Multi-dispatch

Chapter *sec:subs* on page 27 showed how to use nested signatures to look deeper into data structures and extract parts of them. In the context of multiple dispatch, nested signatures take on a second task: they act as constraints to distinguish between the candidates. This means that it is possible to dispatch based upon the shape of a data structure. This brings Perl 6 a lot of the expressive power provided by pattern matching in various functional languages.

Some algorithms have very tidy and natural expressions with this feature, especially those which recurse to a simple base case. Consider quicksort. The base case is that of the empty list, which trivially sorts to the empty list. A Perl 6 version might be:

```
multi quicksort([]) { () }
```

The `[]` declares an empty nested signature for the first positional parameter. Additionally, it requires that the first positional parameter be an indexable item–anything that would match the `@` sigil. The signature will only match if the multi has a single parameter which is an empty list.

The other case is a list which contains at least one value–the pivot–and possibly other values to partition according to the pivot. The rest of quicksort is a couple of recursive calls to sort both partitions:

```
multi quicksort([$pivot, *@rest]) {
    my @before = @rest.grep({ $_ <= $pivot });
    my @after  = @rest.grep({ $_ >  $pivot });

    return quicksort(@before), $pivot, quicksort(@after);
}
```

6.6 Protos

You have two options to write multi subs: either you start every candidate with `multi sub ...` or `multi ...`, or you declare once and for all that the compiler shall view every sub of a given name as a multi candidate. Do the latter by installing a *proto* routine:

```
proto to-json($) { ... }       # literal ... here

# automatically a multi
sub to-json(Bool $d) { $d ?? 'true' !! 'false' }
```

Nearly all Perl 6 built-in functions and operators export a proto definition, which prevents accidental overriding of built-ins[1].

> To hide all candidates of a multi and replace them by another sub, declare it as `only sub YourSub`. At the time of writing, only Rakudo supports this..

6.7 Toying with the candidate list

Each multi dispatch builds a list of candidates, all of which satisfy the nominal type constraints. For a normal sub or method call, the dispatcher invokes the first candidate which passes any additional constraint checks.

A routine can choose to delegate its work to other candidates in that list. The `callsame` primitive calls the next candidate, passing along the arguments received. The `callwith` primitive calls the next candidate with different (and provided) arguments. After the called routine has done its work, the callee can continue its work.

If there's no further work to do, the routine can decide to hand control completely to the next candidate by calling `nextsame` or `nextwith`. The former reuses the argument list and the latter allows the use of a different argument list. This delegation is common in object destructors, where each subclass may perform some cleanup for its own particular data. After it finishes its work, it can delegate to its parent class meethod by calling `nextsame`.

[1] One of the very rare exceptions is the smart match operator `infix:<~~>` which is not easily overloadable. Instead it redispatches to overloadable multi methods.

6.8 Multiple MAIN subs

`MAIN` routines can also be multis. That way it is easy to add separate code for each action that the script can perform.

```
# file calc.pl
multi MAIN('add', $x, $y)  { say $x + $y }
multi MAIN('div', $x, $y)  { say $x / $y }
multi MAIN('mult', $x, $y) { say $x * $y }
```

Example calls to this script:

```
$ perl6 calc.pl add 3 5
8
$ perl6 calc.pl mult 3 5
15
$ perl6 calc.pl
Usage:
calc.pl add x y
or
calc.pl div x y
or
calc.pl mult x y
```

Here the multi dispatcher selects the routine based on the value of the first argument.

7

Roles

A *role* is a standalone, named, and reusable unit of behavior. You can compose a role into a class at compile time or add it to an individual object at runtime.

That's an abstract definition best explained by an example. This program demonstrates a simple and pluggable IRC bot framework which understands a few simple commands.

```
# XXX This is VERY preliminary code and needs filling out. But it
# does provide opportunities to discuss runtime mixins, compile time
# composition, requirements and a few other bits.

my regex nick { \w+ }
my regex join-line { ... <nick> ... }
my regex message-line { $<sender>=[...] $<message>=[...] }

class IRCBot {
    has $.bot-nick;
    method run($server) {
        ...
    }
}

```

```
16 role KarmaTracking {
17     has %!karma-scores;
18
19     multi method on-message($sender, $msg where /^karma <ws> <nick>/) {
20         if %!karma-scores{$<nick>} -> $karma {
21             return $<nick> ~ " has karma $karma";
22         }
23         else {
24             return $<nick> ~ " has neutral karma";
25         }
26     }
27
28     multi method on-message($sender, $msg where /<nick> '++'/) {
29         %!karma-scores{$<nick>}++;
30     }
31
32     multi method on-message($sender, $msg where /<nick> '--'/) {
33         %!karma-scores{$<nick>}--;
34     }
35 }
36
37 role Oping {
38     has @!whoz-op;
39
40     multi method on-join($nick) {
41         if $nick eq any(@!whoz-op) {
42             return "/mode +o $nick";
43         }
44     }
45
46     # I'm tempted to add another candidate here which checks any(@!whoz-op)
47     multi method on-message($sender, $msg where /^trust <ws> <nick>/) {
48         if $sender eq any(@!whoz-op) {
49             push @!whoz-op, $<nick>;
50             return "I now trust " ~ $<nick>;
51         }
52         else {
53             return "But $sender, I don't trust you";
54         }
55     }
56 }
57
58 role AnswerToAll {
```

```
59      method process($raw-in) {
60          if $raw-in ~~ /<on-join>/ {
61              self.*on-join($<nick>);
62          }
63          elsif $raw-in ~~ /<on-message>/ {
64              self.*on-message($<sender>, $<message>)
65          }
66      }
67  }
68
69  role AnswerIfTalkedTo {
70      method bot-nick() { ... }
71
72      method process($raw-in) {
73          if $raw-in ~~ /<on-join>/ {
74              self.*on-join($<nick>);
75          }
76          elsif $raw-in ~~ /<on-message>/ -> $msg {
77              my $my-nick = self.bot-nick();
78              if $msg<msg> ~~ /^ $my-nick ':'/ {
79                  self.*on-message($msg<sender>, $msg<message>)
80              }
81          }
82      }
83  }
84
85  my %pluggables =
86      karma => KarmaTracking,
87      op    => Oping;
88
89  role Plugins {
90      multi method on-message($self is rw: $sender, $msg where /^youdo <ws> (\w+)/) {
91          if %pluggables{$0} -> $plug-in {
92              $self does $plug-in;
93              return "Loaded $0";
94          }
95      }
96  }
97
98  class AdminBot    is IRCBot does KarmaTracking    does Oping       {}
99  class KarmaKeeper is IRCBot does KarmaTracking    does AnswerToAll {}
100 class NothingBot  is IRCBot does AnswerIfTalkedTo does Plugins     {}
```

You don't have to understand everything in this example yet. It's only important right now to notice that the classes `KarmaKeeper` and `NothingBot` share some behavior by inheriting from `IRCBot` and differentiate their behaviors by performing different roles.

7.1 What is a role?

A role is another type of package. Like classes, a role can contain methods (including named regexes) and attributes. However, a role cannot stand on its own; you cannot instantiate a role. To use a role, you must incorporate it into an object, class, or, as we'll see in later chapters, a grammar.

In other object systems, classes perform two tasks. They represent entities in the system, providing models from which to create instances. They also provide a mechanism for code re-use. These two tasks contradict each other to some degree. For optimal re-use, classes should be small, but in order to represent a complex entity with many behaviors, classes tend to grow large. Large projects written in such systems often have complex interactions and workarounds for classes which want to reuse code but do not want to take on additional unnecessary capabilities.

Perl 6 classes retain the responsibility for modeling and managing instances. Roles handle the task of code reuse. A role contains the methods and attributes required to provide a named, reusable unit of behavior. Building a class out of roles uses a safe mechanism called *flattening composition*. You may also apply a role to an individual object. Both of these design techniques appear in the example code.

Some roles–*parametric roles*–allow the use of specific customizations to change how they provide the features they provide. This helps Perl 6 provide generic programming, along the lines of generics in C# and Java, or templates in C++.

7.2 Compile Time Composition

Look at the `KarmaKeeper` class declaration. The body is empty; the class defines no attributes or methods of its own. The class inherits from `IRCBot`, using the `is` trait modifier–something familiar from earlier chapters–but it also uses the `does` trait modifier to compose two roles into the class.

The process of role composition is simple. Perl takes the attributes and methods defined in each role and copies them into the class. After composition, the class appears as if those attributes and methods had been declared in the class's declaration itself. This is part of the flattening property: after composing a role into the class, the roles in and of themselves are only important when querying the class to determine *if* it performs the role. Querying the methods of the `KarmaKeeper` class through introspection will report that the class has both a `process` method and an `on-message` multi method.

If this were all that roles provided, they'd have few advantages over inheritance or mixins. Roles get much more interesting in the case of a conflict. Consider the class definition:

```
class MyBot is IRCBot does AnswerToAll does AnswerIfTalkedTo {}
```

Both the `AnswerToAll` and `AnswerIfTalkedTo` roles provide a method named `process`. Even though they share a name, the methods perform semantically different–and conflicting–behaviors. The role composer will produce a compile-time error about this conflict, asking the programmer to provide a resolution.

Multiple inheritance and mixin mechanisms rarely provide this degree of conflict resolution. In those situations, the order of inheritance or mixin decides which method wins. All possible roles are equal in role composition.

What can you do if there is a conflict? In this case, it makes little sense to compose both of the roles into a class. The programmer here has made a mistake and should choose to compose only one role to provide the desired behavior. An alternative way to resolve a conflict is to write a method with the same name in the class body itself:

```
class MyBot is IRCBot does AnswerToAll does AnswerIfTalkedTo {
    method process($raw-in) {
        # Do something sensible here...
    }
}
```

If the role composer detects a method with the same name in the class body, it will then disregard all of the (possibly conflicting) ones from the roles. Put simply, methods in the class always supersede methods which a role may provide.

> What happens when a class performs a role but overrides all of its methods? That's okay too: declaring that a class performs a role does not require you to compose

in any behavior from the role. The role composer will verify that all of the role's requirements are satisfied once and only once, and from then on Perl's type system will consider all instances of the class as corresponding to the type implied by the role.

7.2.1 Multi-methods and composition

Sometimes it's okay to have multiple methods of the same name, provided they have different signatures such that the multidispatch mechanism can distinguish between them. Multi methods with the same name from different roles will not conflict with each other. Instead, the candidates from all of the roles will combine during role composition.

If the class provides a method of the same name that is also multi, then all methods defined in the role and the class will combine into a set of multi candidates. Otherwise, if the class has a method of the same name that is *not* declared as a multi, then the method in the class alone–as usual–will take precedence. This is the mechanism by which the `AdminBot` class can perform the appropriate `on-message` method provided by both the `KarmaTracking` and the `Oping` roles.

When a class composes multiple roles, an alternate declaration syntax - called an `also` declarator - may be more readable:

```
class KarmaKeeper is IRCBot {
    also does AnswerToAll;
    also does KarmaTracking;
    also does Oping;
}
```

The `also` declarator is primarily intended for roles but can also be useful when creating anonymous classes or roles:

```
my $karma = class {
    also does KarmaTracking;
    ...
}
```

7.2.2 Calling all candidates

The `process` methods of the roles `AnswerToAll` and `AnswerIfTalkedTo` use a modified syntax for calling methods:

```
1  self.*on-message($msg<sender>, $msg<message>)
```

The `.*` method calling syntax changes the semantics of the dispatch. Just as the `*` quantifier in regexes means "zero or more", the `.*` dispatch operator will call zero or more matching methods. If no `on-message` multi candidates match, the call will not produce an error. If more than one `on-message` multi candidate matches, Perl will call all of them, whether found by multiple dispatch, searching the inheritance hierarchy, or both.

There are two other variants. `.+` greedily calls all methods but dies unless it can call at least one method. `.?`, tries to call one method, but returns a `Failure` rather then throwing an exception. These dispatch forms may seem rare, but they're very useful for event driven programming. One-or-failure is very useful when dealing with per-object role application.

7.2.3 Expressing requirements

The role `AnswerIfTalkedTo` declares a stub for the method `bot-nick`, but never provides an implementation.

```
1  method bot-nick() { ... }
```

In the context of a role, this means that any class which composes this role must somehow provide a method named `bot-nick`. The class itself may provide it, another role must provide it, or a parent class must provide it. `IRCBot` does the latter; it `IRCBot` defines an attribute `$!bot-nick` along with an accessor method.

If you do not make explicit the methods on which your role depends, the role composer will not verify their existence at compilation time. Any missing methods will cause runtime errors (barring the use of something like `AUTOMETH`). As compile-time verification is an important feature of roles, it's best to mark your dependencies.

7.3 Runtime Application of Roles

Class declarations frozen at compilation time are often sufficient, but sometimes it's useful to add new behaviors to individual objects. Perl 6 allows you to do so by applying roles to individual objects at runtime.

The example in this chapter uses this to give bots new abilities during their lifetimes. The `Plugins` role is at the heart of this. The signature of the method `on-message` captures the invocant into a variable `$self` marked `rw`, which indicates that the invocant may be modified. Inside the method, that happens:

```
1 if %pluggables{$0} -> $plug-in {
2     $self does $plug-in;
3     return "Loaded $0";
4 }
```

Roles in Perl 6 are first-class entities, just like classes. You can pass roles around just like any other object. The `%pluggables` hash maps names of plug-ins to Role objects. The lookup inside `on-message` stores a Role in `$plug-in`. The `does` operator adds this role to `$self`–not the *class* of `$self`, but the instance itself. From this point on, `$self` now has all of the methods from the role, in addition to all of the ones that it had before. This does not affect any other instances of the same class; only this one instance has changed.

7.3.1 Differences from compile time composition

Runtime application differs from compile time composition in that methods in the applied role will automatically override any of the same name within the class of the object. It's as if you had written an anonymous subclass of the current class of the object that composed the role into it. This means that `.*` will find both those methods that mixed into the object from one or more roles along with any that already existed in the class.

If you wish to apply multiple roles at a time, list them all with `does`. This case behaves the same way as compile-time composition, in that the role composer will compose them all into the imaginary anonymous subclass. Any conflicts will occur at this point.

This gives a degree of safety, but it happens at runtime and is thus not as safe as compile time composition. For safety, perform your compositions at compile time. Instead of

applying multiple roles to an instance, compose them into a new role at compile time and apply that role to the instance.

7.3.2 The `but` operator

Runtime role application with `does` modifies an object in place: `$x does SomeRole` modifies the object stored in `$x`. Sometimes this modification is not what you want. In that case, use the `but` operator, which clones the object, performs the role composition with the clone, and returns the clone. The original object stays the same.

TODO: example

7.4 Parametric Roles

7.5 Roles and Types

8

Subtypes

```
enum Suit <spades hearts diamonds clubs>;
enum Rank (2, 3, 4, 5, 6, 7, 8, 9, 10,
           'jack', 'queen', 'king', 'ace');

class Card {
    has Suit $.suit;
    has Rank $.rank;

    method Str {
        $.rank.name ~ ' of ' ~ $.suit.name;
    }
}

subset PokerHand of List where { .elems == 5 && all(|$_) ~~ Card }

sub n-of-a-kind($n, @cards) {
    for @cards>>.rank.uniq -> $rank {
        return True if $n == grep $rank, @cards>>.rank;
    }
    return False;
}

```

```
23 subset Quad         of PokerHand where { n-of-a-kind(4, $_) }
24 subset ThreeOfAKind of PokerHand where { n-of-a-kind(3, $_) }
25 subset OnePair      of PokerHand where { n-of-a-kind(2, $_) }
26
27 subset FullHouse of PokerHand where OnePair & ThreeOfAKind;
28
29 subset Flush of PokerHand where -> @cards { [==] @cards>>.suit }
30
31 subset Straight of PokerHand where sub (@cards) {
32     my @sorted-cards = @cards.sort({ .rank });
33     my ($head, @tail) = @sorted-cards;
34     for @tail -> $card {
35         return False if $card.rank != $head.rank + 1;
36         $head = $card;
37     }
38     return True;
39 }
40
41 subset StraightFlush of Flush where Straight;
42
43 subset TwoPair of PokerHand where sub (@cards) {
44     my $pairs = 0;
45     for @cards>>.rank.uniq -> $rank {
46         ++$pairs if 2 == grep $rank, @cards>>.rank;
47     }
48     return $pairs == 2;
49 }
50
51 sub classify(PokerHand $_) {
52     when StraightFlush { 'straight flush',  8 }
53     when Quad          { 'four of a kind',  7 }
54     when FullHouse     { 'full house',      6 }
55     when Flush         { 'flush',           5 }
56     when Straight      { 'straight',        4 }
57     when ThreeOfAKind  { 'three of a kind', 3 }
58     when TwoPair       { 'two pair',        2 }
59     when OnePair       { 'one pair',        1 }
60     when *             { 'high cards',      0 }
61 }
62
63 my @deck = map -> $suit, $rank { Card.new(:$suit, :$rank) },
64                   (Suit.pick(*) X Rank.pick(*));
65
```

```
@deck .= pick(*);

my @hand1;
@hand1.push(@deck.shift()) for ^5;
my @hand2;
@hand2.push(@deck.shift()) for ^5;

say 'Hand 1: ', map { "\n  $_" }, @hand1>>.Str;
say 'Hand 2: ', map { "\n  $_" }, @hand2>>.Str;

my ($hand1-description, $hand1-value) = classify(@hand1);
my ($hand2-description, $hand2-value) = classify(@hand2);

say sprintf q[The first hand is a '%s' and the second one a '%s', so %s.],
    $hand1-description, $hand2-description,
    $hand1-value > $hand2-value
        ?? 'the first hand wins'
        !! $hand2-value > $hand1-value
            ?? 'the second hand wins'
            !! "the hands are of equal value"; # XXX: this is wrong
```

9

Pattern matching

Regular expressions are a computer science concept where simple patterns describe the format of text. Pattern matching is the process of applying these patterns to actual text to look for matches. Most modern regular expression facilities are more powerful than traditional regular expressions due to the influence of languages such as Perl, but the short-hand term `regex` has stuck and continues to mean "regular expression-like pattern matching". In Perl 6, although the specific syntax used to describe the patterns is different from PCRE[1] and POSIX[2], we continue to call them `regex`.

A common writing error is to duplicate a word by accident. It is hard to catch such errors by re-reading your own text, but Perl can do it for you using `regex`:

```
1 my $s = 'the quick brown fox jumped over the the lazy dog';
2
3 if $s ~~ m/ « (\w+) \W+ $0 » / {
4     say "Found '$0' twice in a row";
5 }
```

[1] **P**erl **C**ompatible **R**egular **E**xpressions
[2] **P**ortable **O**perating **S**ystem **I**nterface for Uni**x**. See IEEE standard 1003.1-2001

The simplest case of a regex is a constant string. Matching a string against that regex searches for that string:

```
if 'properly' ~~ m/ perl / {
    say "'properly' contains 'perl'";
}
```

The construct `m/ ... /` builds a regex. A regex on the right hand side of the ~~ smart match operator applies against the string on the left hand side. By default, whitespace inside the regex is irrelevant for the matching, so writing the regex as `m/ perl /`, `m/perl/` or `m/ p e rl/` all produce the exact same semantics–although the first way is probably the most readable.

Only word characters, digits, and the underscore cause an exact substring search. All other characters may have a special meaning. If you want to search for a comma, an asterisk, or another non-word character, you must quote or escape it[3]:

```
my $str = "I'm *very* happy";

# quoting
if $str ~~ m/ '*very*' /   { say '\o/' }

# escaping
if $str ~~ m/ \* very \* / { say '\o/' }
```

Searching for literal strings gets boring pretty quickly. Regex support special (also called *metasyntactic*) characters. The dot (`.`) matches a single, arbitrary character:

```
my @words = <spell superlative openly stuff>;

for @words -> $w {
    if $w ~~ m/ pe.l / {
        say "$w contains $/";
    } else {
        say "no match for $w";
    }
}
```

This prints:

[3] To search for a literal string–without using the pattern matching features of regex–consider using `index` or `rindex` instead.

```
spell contains pell
superlative contains perl
openly contains penl
no match for stuff
```

The dot matched an `l`, `r`, and `n`, but it will also match a space in the sentence *the spectrosco**pe l**acks resolution*–regexes ignore word boundaries by default.

The special variable `$/` stores the *match object*, which allows you inspect the matched text.

Suppose you want to solve a crossword puzzle. You have a word list and want to find words containing `pe`, then an arbitrary letter, and then an `l` (but not a space, as your puzzle has extra markers for those). The appropriate regex for that is `m/pe \w l/`. The `\w` control sequence stands for a "Word" character–a letter, digit, or an underscore. This chapter's example uses `\w` to build the definition of a "word".

Several other common control sequences each match a single character; you can find a list of those in *regexBackslash* on page 95.

Table 9.1: Backslash sequences and their meaning

Symbol	Description	Examples
`\w`	word character	l, ö, 3, _
`\d`	digit	0, 1
`\s`	whitespace	(tab), (blank), (newline)
`\t`	tabulator	(tab)
`\n`	newline	(newline)
`\h`	horizontal whitespace	(space), (tab)
`\v`	vertical whitespace	(newline), (vertical tab)

Invert the sense of each of these backslash sequences by uppercasing its letter: `\W` matches a character that's *not* a word character and `\N` matches a single character that's not a newline.

These matches extend beyond the ASCII range–`\d` matches Latin, Arabic-Indic, Devanagari and other digits, `\s` matches non-breaking whitespace, and so on. These *character classes* follow the Unicode definition of what is a letter, a number, and so on.

To define your own custom character classes, listing the appropriate characters inside nested angle and square brackets `<[ ... ]>`:

```
if $str ~~ / <[aeiou]> / {
    say "'$str' contains a vowel";
}

# negation with a -
if $str ~~ / <-[aeiou]> / {
    say "'$str' contains something that's not a vowel";
}
```

Rather than listing each character in the character class individually, you may specify a range of characters by placing the range operator `..` between the beginning and ending characters:

```
# match a, b, c, d, ..., y, z
if $str ~~ / <[a..z]> / {
    say "'$str' contains a lower case Latin letter";
}
```

You may add characters to or subtract characters from classes with the `+` and `-` operators:

```
if $str ~~ / <[a..z]+[0..9]> / {
    say "'$str' contains a letter or number";
}

if $str ~~ / <[a..z]-[aeiou]> / {
    say "'$str' contains a consonant";
}
```

The negated character class is a special application of this idea.

Inside a character class, non-word characters do not need to be escaped, and generally use their special meaning. So `/<[+.*]> /` matches a plus sign, a dot or an asterisk. The only exceptions to that are the backslash, square brackets and the dash `-`, which need to be escaped with a backslash:

```
my $str = 'A character [b] inside brackets';
if $str ~~ /'[' <-[ \[ \] ]> ']'/ ) {
```

```
    say "Found a non-bracket character inside square brackets';
}
```

A *quantifier* specifies how often something has to occur. A question mark `?` makes the preceding unit (be it a letter, a character class, or something more complicated) optional, meaning it can either be present either zero or one times. `m/ho u? se/` matches either `house` or `hose`. You can also write the regex as `m/hou?se/` without any spaces, and the `?` will still quantify only the `u`.

The asterisk `*` stands for zero or more occurrences, so `m/z\w*o/` can match `zo`, `zoo`, `zero` and so on. The plus `+` stands for one or more occurrences, `\w+` *usually* matches what you might consider a word (though only matches the first three characters from `isn't` because `'` isn't a word character).

The most general quantifier is `**`. When followed by a number, it matches that many times. When followed by a range, it can match any number of times that the range allows:

```
# match a date of the form 2009-10-24:
m/ \d**4 '-' \d\d '-' \d\d /

# match at least three 'a's in a row:
m/ a ** 3..* /
```

One can specify a separator with `%` after the quantifier:

```
1,2,3' ~~ / \d+ % ',' /
```

The separator is matched between two occurrences of the quantified regex. The separator can itself be a regex.

If a quantifier has several ways to match, Perl will choose the longest one. This is *greedy* matching. Appending a question mark to a quantifier makes it non-greedy[4]

For example, you can parse HTML very badly[5] with the code:

```
my $html = '<p>A paragraph</p> <p>And a second one</p>';

```

[4] The non-greedy general quantifier is `$thing **? $count`, so the question mark goes directly after the second asterisk.

[5] Using a proper stateful parser is always more accurate.

```
if $html ~~ m/ '<p>' .* '</p>' / {
    say 'Matches the complete string!';
}

if $html ~~ m/ '<p>' .*? '</p>' / {
    say 'Matches only <p>A paragraph</p>!';
}
```

To apply a modifier to more than just one character or character class, group items with square brackets:

```
my $ingredients = 'milk, flour, eggs and sugar';
# prints "milk, flour, eggs"
$ingredients ~~ m/ [\w+]+ % [\,\s*] / && say $/;
```

Here `\w+` matches a word, and `[\w+]+ % [\,\s*]` matches at least one word, where several words are separated by a comma and an arbitrary amount of whitespace.

Separate *alternations*–parts of a regex of which *any* can match–with vertical bars. One vertical bar between multiple parts of a regex means that the alternatives are tried in parallel and the longest matching alternative wins. Two bars make the regex engine try each alternative in order and the first matching alternative wins.

```
$string ~~ m/ \d**4 '-' \d\d '-' \d\d | 'today' | 'yesterday' /
```

9.1 Anchors

So far every regex could match anywhere within a string. Often it is useful to limit the match to the start or end of a string or line or to word boundaries. A single caret `^` anchors the regex to the start of the string and a dollar sign `$` to the end. `m/ ^a /` matches strings beginning with an `a`, and `m/ ^ a $ /` matches strings that consist only of an `a`.

Table 9.2: Regex anchors

Anchor	Meaning
^	start of string
$	end of string
^^	start of a line
$$	end of a line
<<	left word boundary
«	left word boundary
>>	right word boundary
»	right word boundary

9.2 Captures

Regex can be very useful for *extracting* information too. Surrounding part of a regex with round brackets (aka parentheses) `(...)` makes Perl *capture* the string it matches. The string matched by the first group of parentheses is available in `$/[0]`, the second in `$/[1]`, etc. `$/` acts as an array containing the captures from each parentheses group:

```
my $str = 'Germany was reunited on 1990-10-03, peacefully';

if $str ~~ m/ (\d**4) \- (\d\d) \- (\d\d) / {
    say 'Year:  ', $/[0];
    say 'Month: ', $/[1];
    say 'Day:   ', $/[2];
    # usage as an array:
    say $/.join('-');       # prints 1990-10-03
}
```

If you quantify a capture, the corresponding entry in the match object is a list of other match objects:

```
my $ingredients = 'eggs, milk, sugar and flour';

if $ingredients ~~ m/(\w+)% % [\,\s*] \s* 'and' \s* (\w+)/ {
    say 'list: ', $/[0].join(' | ');
    say 'end:  ', $/[1];
}
```

This prints:

```
list: eggs | milk | sugar
end:  flour
```

The first capture, `(\w+)`, was quantified, so `$/[0]` contains a list of words. The code calls `.join` to turn it into a string. Regardless of how many times the first capture matches (and how many elements are in `$/[0]`), the second capture is still available in `$/[1]`.

As a shortcut, `$/[0]` is also available under the name `$0`, `$/[1]` as `$1`, and so on. These aliases are also available inside the regex. This allows you to write a regex that detects that common error of duplicated words, just like the example at the beginning of this chapter:

```
1  my $s = 'the quick brown fox jumped over the the lazy dog';
2
3  if $s ~~ m/ « (\w+) \W+ $0 » / {
4      say "Found '$0' twice in a row";
5  }
```

The regex first anchors to a left word boundary with « so that it doesn't match partial duplication of words. Next, the regex captures a word (`(\w+)`), followed by at least one non-word character `\W+`. This implies a right word boundary, so there is no need to use an explicit boundary. Then it matches the previous capture followed by a right word boundary.

Without the first word boundary anchor, the regex would for example match *str**and and** beach* or *la**the the** table leg*. Without the last word boundary anchor it would also match ***the the**ory*.

9.3 Named regexes

You can declare regexes just like subroutines–and even name them. Suppose you found the example at the beginning of this chapter useful and want to make it available easily. Suppose also you want to extend it to handle contractions such as `doesn't` or `isn't`:

```
1  my regex word { \w+ [ \' \w+]? }
2  my regex dup  { « <word=&word> \W+ $<word> » }
```

```
3
4  if $s ~~ m/ <dup=&dup> / {
5      say "Found '{$<dup><word>}' twice in a row";
6  }
```

This code introduces a regex named `word`, which matches at least one word character, optionally followed by a single quote and some more word characters. Another regex called `dup` (short for *duplicate*) contains a word boundary anchor.

Within a regex, the syntax `<&word>` locates the regex `word` within the current lexical scope and matches against the regex. The `<name=&regex>` syntax creates a capture named `name`, which records what `&regex` matched in the match object.

In this example, `dup` calls the `word` regex, then matches at least one non-word character, and then matches the same string as previously matched by the regex `word`. It ends with another word boundary. The syntax for this *backreference* is a dollar sign followed by the name of the capture in angle brackets[6].

Within the `if` block, `$<dup>` is short for `$/{'dup'}`. It accesses the match object that the regex `dup` produced. `dup` also has a subrule called `word`. The match object produced from that call is accessible as `$<dup><word>`.

Named regexes make it easy to organize complex regexes by building them up from smaller pieces.

9.4 Modifiers

The previous example to match a list of words was:

```
1  m/(\w+)+ % [\,\s*] \s* 'and' \s* (\w+)/
```

This works, but the repeated "I don't care about whitespace" units are clumsy. The desire to allow whitespace *anywhere* in a string is common. Perl 6 regexes allow this through the use of the `:sigspace` modifier (shortened to `:s`):

[6] In grammars–see (`grammars`)–`<word>` looks up a regex named `word` in the current grammar and parent grammars, and creates a capture of the same name.

```
my $ingredients = 'eggs, milk, sugar and flour';

if $ingredients ~~ m/:s ( \w+ )%  % \,'and' (\w+)/ {
    say 'list: ', $/[0].join(' | ');
    say 'end:  ', $/[1];
}
```

This modifier allows optional whitespace in the text wherever there one or more whitespace characters appears in the pattern. It's even a bit cleverer than that: between two word characters whitespace is mandatory. The regex does *not* match the string `eggs, milk, sugarandflour`.

The `:ignorecase` or `:i` modifier makes the regex insensitive to upper and lower case, so `m/ :i perl /` matches `perl`, `PerL`, and `PERL` (though who names a programming language in all uppercase letters?)

9.5 Backtracking control

In the course of matching a regex against a string, the regex engine may reach a point where an alternation has matched a particular branch or a quantifier has greedily matched all it can, but the final portion of the regex fails to match. In this case, the regex engine backs up and attempts to match another alternative or matches one fewer character of the quantified portion to see if the overall regex succeeds. This process of failing and trying again is *backtracking*.

When matching `m/\w+ 'en'/` against the string `oxen`, the `\w+` group first matches the whole string because of the greediness of `+`, but then the `en` literal at the end can't match anything. `\w+` gives up one character to match `oxe`. `en` still can't match, so the `\w+` group again gives up one character and now matches `ox`. The `en` literal can now match the last two characters of the string, and the overall match succeeds.

While backtracking is often useful and convenient, it can also be slow and confusing. A colon `:` switches off backtracking for the previous quantifier or alternation. `m/ \w+: 'en'/` can never match any string, because the `\w+` always eats up all word characters and never releases them.

The `:ratchet` modifier disables backtracking for a whole regex, which is often desirable in a small regex called often from other regexes. The duplicate word search regex had to

anchor the regex to word boundaries, because \w+ would allow matching only part of a word. Disabling backtracking makes \w+ always match a full word:

```
# XXX: does actually match, because m/<&dup>/
# searches for a starting position where the
# whole regex matches. Find an example that
# doesn't match

my regex word { :ratchet \w+ [ \' \w+]? }
my regex dup  { <word=&word> \W+ $<word> }

# no match, doesn't match the 'and'
# in 'strand' without backtracking
'strand and beach' ~~ m/<&dup>/
```

The effect of :ratchet applies only to the regex in which it appears. The outer regex will still backtrack, so it can retry the regex word at a different staring position.

The regex { :ratchet ... } pattern is so common that it has its own shortcut: token { ... }. An idiomatic duplicate word searcher might be:

```
my token word { \w+ [ \' \w+]? }
my regex dup   { <word> \W+ $<word> }
```

A token with the :sigspace modifier is a rule:

```
# TODO: check if it works
my rule wordlist { <word>+ % \, 'and' <word> }
```

9.6 Substitutions

Regexes are also good for data manipulation. The subst method matches a regex against a string. When subst matches, it substitutes the matched portion of the string with its the second operand:

```
my $spacey = 'with    many  superfluous   spaces';

say $spacey.subst(rx/ \s+ /, ' ', :g);
# output: with many superfluous spaces
```

By default, `subst` performs a single match and stops. The `:g` modifier tells the substitution to work *globally* to replace every possible match.

Note the use of `rx/ ... /` rather than `m/ ... /` to construct the regex. The former constructs a regex object. The latter constructs the regex object *and* immediately matches it against the topic variable `$_`. Using `m/ ... /` in the call to `subst` would create a match object and pass it as the first argument, rather than the regex itself.

9.7 Other Regex Features

Sometimes you want to call other regexes, but don't want them to capture the matched text. When parsing a programming language you might discard whitespace characters and comments. You can achieve that by calling the regex as `<.otherrule>`.

If you use the `:sigspace` modifier, every continuous piece of whitespace calls the built-in rule `<.ws>`. This use of a rule rather than a character class allows you to define your own version of whitespace characters (see `grammars`).

Sometimes you just want to peek ahead to check if the next characters fulfill some properties without actually consuming them. This is common in substitutions. In normal English text, you always place a whitespace after a comma. If somebody forgets to add that whitespace, a regex can clean up after the lazy writer:

```
1 my $str = 'milk,flour,sugar and eggs';
2 say $str.subst(/',' <?before \w>/, ', ',  :g);
3 # output: milk, flour, sugar and eggs
```

The word character after the comma is not part of the match, because it is in a lookahead introduced by `<?before ... >`. The leading question mark indicates an *zero-width assertion*: a rule that never consumes characters from the matched string. You can turn any call to a subrule into a zero width assertion. The built-in token `<alpha>` matches an alphabetic character, so you can rewrite this example as:

```
1 say $str.subst(/',' <?alpha>/, ', ',  :g);
```

A leading exclamation mark negates the meaning, such that the lookahead must *not* find the regex fragment. Another variant is:

```
say $str.subst(/',' <!space>/, ', ',  :g);
```

You can also look behind to assert that the string only matches *after* another regex fragment. This assertion is `<?after>`. You can write the equivalent of many built-in anchors with look-ahead and look-behind assertions, though they won't be as efficient.

Table 9.3: Emulation of anchors with look-around assertions

Anchor	Meaning	Equivalent Assertion
^	start of string	`<!after .>`
^^	start of line	`<?after ^ | \n >`
$	end of string	`<!before .>`
>>	right word boundary	`<?after \w> <!before \w>`

9.8 Match objects

```
sub line-and-column(Match $m) {
    my $line   = ($m.orig.substr(0, $m.from).split("\n")).elems;
    # RAKUDO workaround for RT #70003, $m.orig.rindex(...) directly fails
    my $column = $m.from - ('' ~ $m.orig).rindex("\n", $m.from);
    $line, $column;
}

my $s = "the quick\nbrown fox jumped\nover the the lazy dog";

my token word { \w+ [ \' \w+]? }
my regex dup { <word> \W+ $<word> }

if $s ~~ m/ <dup> / {
    my ($line, $column) = line-and-column($/);
    say "Found '{$<dup><word>}' twice in a row";
    say "at line $line, column $column";
}

# output:
# Found 'the' twice in a row
# at line 3, column 6
```

Every regex match returns an object of type `Match`. In boolean context, a match object returns `True` for successful matches and `False` for failed ones. Most properties are only interesting after successful matches.

The `orig` method returns the string that was matched against. The `from` and `to` methods return the positions of the start and end points of the match.

In the previous example, the `line-and-column` function determines the line number in which the match occurred by extracting the string up to the match position (`$m.orig.substr(0, $m.from)`), splitting it by newlines, and counting the elements. It calculates the column by searching backwards from the match position and calculating the difference to the match position.

> The `index` method searches a string for another substring and returns the position of the search string. The `rindex` method does the same, but searches backwards from the end of the string, so it finds the position of the final occurrence of the substring.

Using a match object as an array yields access to the positional captures. Using it as a hash reveals the named captures. In the previous example, `$<dup>` is a shortcut for `$/<dup>` or `$/{ 'dup' }`. These captures are again `Match` objects, so match objects are really trees of matches.

The `caps` method returns all captures, named and positional, in the order in which their matched text appears in the source string. The return value is a list of `Pair` objects, the keys of which are the names or numbers of the capture and the values the corresponding `Match` objects.

```
1  if 'abc' ~~ m/(.) <alpha> (.) / {
2      for $/.caps {
3          say .key, ' => ', .value;
4
5      }
6  }
7
8  # Output:
9  # 0 => a
10 # alpha => b
11 # 1 => c
```

In this case the captures occur in the same order as they are in the regex, but quantifiers can change that. Even so, `$/.caps` follows the ordering of the string, not of the regex. Any parts of the string which match but not as part of captures will not appear in the values that `caps` returns.

To access the non-captured parts too, use `$/.chunks` instead. It returns both the captured and the non-captured part of the matched string, in the same format as `caps`, but with a tilde `~` as key. If there are no overlapping captures (as occurs from look-around assertions), the concatenation of all the pair values that `chunks` returns is the same as the matched part of the string.

10

Grammars

Grammars organize regexes, just like classes organize methods. The following example demonstrates how to parse JSON, a data exchange format already introduced (see *sec:multis* on page 67).

```
# file lib/JSON/Tiny/Grammar.pm

grammar JSON::Tiny::Grammar {
    rule TOP        { ^[ <object> | <array> ]$ }
    rule object     { '{' ~ '}' <pairlist>     }
    rule pairlist   { <pair>* % [ \, ]         }
    rule pair       { <string> ':' <value>     }
    rule array      { '[' ~ ']' [ <value>* % [ \, ] ]  }

    proto token value { <...> };

    token value:sym<number> {
        '-'?
        [ 0 | <[1..9]> <[0..9]>* ]
        [ \. <[0..9]>+ ]?
        [ <[eE]> [\+|\-]? <[0..9]>+ ]?
    }
```

```
18
19      token value:sym<true>    { <sym>    };
20      token value:sym<false>   { <sym>    };
21      token value:sym<null>    { <sym>    };
22      token value:sym<object>  { <object> };
23      token value:sym<array>   { <array>  };
24      token value:sym<string>  { <string> }
25
26      token string {
27          \" ~ \" [ <str> | \\ <str_escape> ]*
28      }
29
30      token str {
31          [
32              <!before \t>
33              <!before \n>
34              <!before \\>
35              <!before \">
36              .
37          ]+
38      #    <-["\\\t\n]>+
39      }
40
41      token str_escape {
42          <["\\/bfnrt]> | u <xdigit>**4
43      }
44
45  }
46
47
48  # test it:
49  my $tester = '{
50      "country":  "Austria",
51      "cities": [ "Wien", "Salzburg", "Innsbruck" ],
52      "population": 8353243
53  }';
54
55  if JSON::Tiny::Grammar.parse($tester) {
56      say "It's valid JSON";
57  } else {
58      # TODO: error reporting
59      say "Not quite...";
60  }
```

A grammar contains various named regex. Regex names may be constructed the same as subroutine names or method names. While regex names are completely up to the grammar writer, a regex named `TOP` will, by default, be invoked when the `.parse()` method is executed on a grammar[1]. So, the call to `JSON::Tiny::Grammar.parse($tester)` attempts to match the regex named `TOP` to the string `$tester`.

However, the example code does not seem to have a regex named `TOP`; it has a *rule* named `TOP`. Looking at the grammar, you'll also note that there are also `token` declarations as well. What's the difference? Each of the `token` and `rule` declarations are just `regex` declarations with special default behavior. A `token` declares a regex that does not backtrack by default, so that when a partial pattern match fails, the regex engine will not go back up and try another alternative (this is equivalent to using the `:ratchet` modifier). A `rule` will also not backtrack by default, additionally, sequences of whitespace are deemed significant and will match actual whitespace within the string using the built-in `<ws>` rule (this is equivalent to using the `:sigspace` modifier).

See `sec:regexes` for more information on modifiers and how they may be used or look at `S05`.

Usually, when talking about regex that are rules or tokens, we tend to call them rules or tokens rather than the more general term "regex", to distinguish their special behaviors.

In this example, the `TOP` rule anchors the match to the start (with `^`) and end of the string (with `$`), so that the whole string has to be in valid JSON format for the match to succeed. After matching the anchor at the start of the string, the regex attempts to match either an `<array>` or an `<object>`. Enclosing a regex name in angle brackets causes the regex engine to attempt to match a regex by that name within the same grammar. Subsequent matches are straightforward and reflect the structure in which JSON components can appear.

Regexes can be recursive. An `array` contains `value`. In turn a `value` can be an `array`. This will not cause an infinite loop as long as at least one regex per recursive call consumes at least one character. If a set of regexes were to call each other recursively without progressing in the string, the recursion could go on infinitely and never proceed to other parts of the grammar.

[1] The name of the regex that is automatically invoked may also be specified as a parameter to `.parse()`. For instance, `JSON::Tiny::Grammar.parse($tester, :rule<object>)` will start parsing at the regex named `object`

The example JSON grammar introduces the *goal matching syntax* which can be presented abstractly as: `A ~B C`. In `JSON::Tiny::Grammar`, `A` is `'{'`, `B` is `'}'` and `C` is `<pairlist>`. The atom on the left of the tilde (`A`) is matched normally, but the atom to the right of the tilde (`B`) is set as the goal, and then the final atom (`C`) is matched. Once the final atom matches, the regex engine attempts to match the goal (`B`). This has the effect of switching the match order of the final two atoms (`B` and `C`), but since Perl knows that the regex engine should be looking for the goal, a better error message can be given when the goal does not match. This is very helpful for bracketing constructs as it puts the brackets near one another.

Another novelty is the declaration of a *proto token*:

```
1  proto token value { <...> };
2
3  token value:sym<number> {
4      '-'?
5      [ 0 | <[1..9]> <[0..9]>* ]
6      [ \. <[0..9]>+ ]?
7      [ <[eE]> [\+|\-]? <[0..9]>+ ]?
8  }
9
10 token value:sym<true>    { <sym> };
11 token value:sym<false>   { <sym> };
```

The `proto token` syntax indicates that `value` will be a set of alternatives instead of a single regex. Each alternative has a name of the form `token value:sym<thing>`, which can be read as *alternative of `value` with parameter `sym` set to `thing`*. The body of such an alternative is a normal regex, where the call `<sym>` matches the value of the parameter, in this example `thing`.

When calling the rule `<value>`, the grammar engine attempts to match the alternatives in parallel and the longest match wins. This is exactly like normal alternation, but as we'll see in the next section, has the advantage of being extensible.

10.1 Grammar Inheritance

The similarity of grammars to classes goes deeper than storing regexes in a namespace as a class might store methods. You can inherit from and extend grammars, mix roles into them, and take advantage of polymorphism. In fact, a grammar is a class which by default inherits from `Grammar` instead of `Any`. The `Grammar` base grammar contains broadly useful rules predefined. For instance, there is a rule to match alphabetic characters (`<alpha>`), and another to match digits (`<digit>`), and another to match whitespace (`<ws>`), etc.

Suppose you want to enhance the JSON grammar to allow single-line C++ or JavaScript comments, which begin with `//` and continue until the end of the line. The simplest enhancement is to allow such a comment in any place where whitespace is valid.

However, `JSON::Tiny::Grammar` only implicitly matches whitespace through the use of *rules*. Implicit whitespace is matched with the inherited regex `<ws>`, so the simplest approach to enable single- line comments is to override that named regex:

```
grammar JSON::Tiny::Grammar::WithComments
    is JSON::Tiny::Grammar {

    token ws {
        \s* [ '//' \N* \n ]?
    }
}

my $tester = '{
    "country":  "Austria",
    "cities": [ "Wien", "Salzburg", "Innsbruck" ],
    "population": 8353243 // data from 2009-01
}';

if JSON::Tiny::Grammar::WithComments.parse($tester) {
    say "It's valid (modified) JSON";
}
```

The first two lines introduce a grammar that inherits from `JSON::Tiny::Grammar`. Just as subclasses inherit methods from superclasses, so grammars inherit rules from its base grammar. Any rule used within the grammar will be looked for first in the grammar in which it was used, then within its parent(s).

In this minimal JSON grammar, whitespace is never mandatory, so `ws` can match nothing at all. After optional spaces, two slashes `'//'` introduce a comment, after which must follow an arbitrary number of non- newline characters, and then a newline. In prose, the comment starts with `'//'` and extends to the rest of the line.

Inherited grammars may also add variants to proto tokens:

```
grammar JSON::ExtendedNumeric is JSON::Tiny::Grammar  {
    token value:sym<nan> { <sym> }
    token value:sym<inf> { <[+-]>? <sym> }
}
```

In this grammar, a call to `<value>` matches either one of the newly added alternatives, or any of the old alternatives from the parent grammar `JSON::Tiny::Grammar`. Such extensibility is difficult to achieve with ordinary, `|` delimited alternatives.

10.2 Extracting data

The `.parse` method of a grammar returns a `Match` object through which you can access all the relevant information of the match. Named regex that match within the grammar may be accessed via the `Match` object similar to a hash where the keys are the regex names and the values are the `Match` object that represents that part of the overall regex match. Similarly, portions of the match that are captured with parentheses are available as positional elements of the `Match` object (as if it were an array).

Once you have the Match object, what can you *do* with it? You could recursively traverse this object and create data structures based on what you find or execute code. Perl 6 provides another alternative: *action methods*.

```
# JSON::Tiny::Grammar as above
# ...
class JSON::Tiny::Actions {
    method TOP($/)      { make $/.values.[0].ast }
    method object($/)   { make $<pairlist>.ast.hash }
    method pairlist($/) { make $<pair>».ast }
    method pair($/)     { make $<string>.ast => $<value>.ast }
    method array($/)    { make [$<value>».ast] }
    method string($/)   { make join '', $/.caps>>.value>>.ast }

```

```
11      # TODO: make that
12      # make +$/
13      # once prefix:<+> is sufficiently polymorphic
14      method value:sym<number>($/) { make eval $/ }
15      method value:sym<string>($/) { make $<string>.ast }
16      method value:sym<true>  ($/) { make Bool::True  }
17      method value:sym<false> ($/) { make Bool::False }
18      method value:sym<null>  ($/) { make Any }
19      method value:sym<object>($/) { make $<object>.ast }
20      method value:sym<array> ($/) { make $<array>.ast }
21
22      method str($/)               { make ~$/ }
23
24      method str_escape($/) {
25          if $<xdigit> {
26              make chr(:16($<xdigit>.join));
27          } else {
28              my %h = '\\' => "\\",
29                      'n'  => "\n",
30                      't'  => "\t",
31                      'f'  => "\f",
32                      'r'  => "\r";
33              make %h{$/};
34          }
35      }
36  }
37
38  my $actions = JSON::Tiny::Actions.new();
39  JSON::Tiny::Grammar.parse($str, :$actions);
```

This example passes an actions object to the grammar's `parse` method. Whenever the grammar engine finishes parsing a regex, it calls a method on the actions object with the same name as the regex. If no such method exists, the grammar engine continues parsing the rest of the grammar. If a method does exist, the grammar engine passes the current match object as a positional argument.

Each match object has a slot called `ast` (short for *abstract syntax tree*) for a payload object. This slot can hold a custom data structure that you create from the action methods. Calling `make $thing` in an action method sets the `ast` attribute of the current match object to `$thing`.

> An abstract syntax tree, or AST, is a data structure which represents the parsed version of the text. Your grammar describes the structure of the AST: its root element is the `TOP` node, which contains children of the allowed types and so on.

In the case of the JSON parser, the payload is the data structure that the JSON string represents. For each matching rule, the grammar engine calls an action method to populate the `ast` slot of the match object. This process transforms the match tree into a different tree–in this case, the actual JSON tree.

Although the rules and action methods live in different namespaces (and in a real-world project probably even in separate files), here they are adjacent to demonstrate their correspondence:

```
1  rule TOP        { ^ [ <object> | <array> ]$ }
2  method TOP($/)  { make $/.values.[0].ast }
```

The `TOP` rule has an alternation with two branches, `object` and `array`. Both have a named capture. `$/.values` returns a list of all captures, here either the `object` or the `array` capture.

The action method takes the AST attached to the match object of that sub capture, and promotes it as its own AST by calling `make`.

```
1  rule object        { '{' ~ '}' <pairlist>  }
2  method object($/)  { make $<pairlist>.ast.hash }
```

The reduction method for `object` extracts the AST of the `pairlist` submatch and turns it into a hash by calling its `hash` method.

```
1  rule pairlist       { <pair>* % [ \, ]   }
2  method pairlist($/) { make $<pair>».ast; }
```

The `pairlist` rule matches multiple comma-separated pairs. The reduction method calls the `.ast` method on each matched pair and installs the result list in its own AST.

```
1  rule pair       { <string> ':' <value> }
2  method pair($/) { make $<string>.ast => $<value>.ast }
```

A pair consists of a string key and a value, so the action method constructs a Perl 6 pair with the => operator.

The other action methods work the same way. They transform the information they extract from the match object into native Perl 6 data structures, and call `make` to set those native structures as their own ASTs.

The action methods for proto tokens include the full name of each individual rule, including the `sym` part:

```
token value:sym<null>        { <sym>    };
method value:sym<null>($/)   { make Any }

token value:sym<object>      { <object> };
method value:sym<object>($/) { make $<object>.ast }
```

When a `<value>` call matches, the action method with the same symbol as the matching subrule executes.

11

Built-in types, operators and methods

Many operators work on a particular *type* of data. If the type of the operands differs from the type of the operand, Perl will make copies of the operands and convert them to the needed types. For example, `$a + $b` will convert a copy of both `$a` and `$b` to numbers (unless they are numbers already). This implicit conversion is called *coercion*.

Besides operators, other syntactic elements coerce their elements: `if` and `while` coerce to truth values (`Bool`), `for` views things as lists, and so on.

11.1 Numbers

Sometimes coercion is transparent. Perl 6 has several numeric types which can intermix freely–such as subtracting a floating point value from an integer, as `123 - 12.1e1`.

The most important types are:

Int

`Int` objects store integer numbers of arbitrary size. If you write a literal that consists only of digits, such as `12`, it is an `Int`.

Num

`Num` is the floating point type. It stores sign, mantissa, and exponent, each with a fixed width. Calculations involving `Num` numbers are usually quite fast, though subject to limited precision.

Numbers in scientific notation such as `6.022e23` are of type `Num`.

Rat

`Rat`, short for *rational*, stores fractional numbers without loss of precision. It does so by tracking its numerator and denominator as integers, so mathematical operations on `Rat`s with large components can become quite slow. For this reason, rationals with large denominators automatically degrade to `Num`.

Writing a fractional value with a dot as the decimal separator, such as `3.14`, produces a `Rat`.

Complex

`Complex` numbers have two parts: a real part and an imaginary part. If either part is `NaN`, then the entire number may possibly be `NaN`.

Numbers in the form `a + bi`, where `bi` is the imaginary component, are of type `Complex`.

The following operators are available for all number types:

Most mathematical functions are available both as methods and functions, so you can write both `(-5).abs` and `abs(-5)`.

The trigonometric functions `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sec`, `cosec`, `cotan`, `asec`, `acosec`, `acotan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sech`, `cosech`, `cotanh`, `asech`, `acosech` and `acotanh` work in units of radians by default. You may specify the unit with an argument of `Degrees`, `Gradians` or `Circles`. For example, `180.sin(Degrees)` is approximately `0`.

Table 11.1: Binary numeric operators

Operator	Description
`**`	exponentiation: `$a**$b` is `$a` to the power of `$b`
`*`	multiplication
`/`	division
`div`	integer division
`%`	modulo (remainder after division)
`+`	addition
`-`	subtraction

Table 11.2: Unary numeric operators

Operator	Description
`+`	conversion to number
`-`	negation

Table 11.3: Mathematical functions and methods

Method	Description
`abs`	absolute value
`sqrt`	square root
`log`	natural logarithm
`log10`	logarithm to base 10
`ceil`	rounding up to an integer
`floor`	rounding down to an integer
`round`	rounding to next integer
`sign`	-1 for negative, 0 for zero, 1 for positive values

11.2 Strings

Strings stored as `Str` are sequences of characters, independent of character encoding. The `Buf` type is available for storing binary data. The `encode` method converts a `Str` to `Buf`. `decode` goes the other direction.

The following operations are available for strings:

Table 11.4: Binary string operators

Operator	Description
`~`	concatenation: `'a' ~'b'` is `'ab'`
`x`	replication: `'a' x 2` is `'aa'`

Table 11.5: Unary string operators

Operator	Description
`~`	conversion to string: `~1` becomes `'1'`

Table 11.6: String methods/functions

Method/function	Description
`.chomp`	remove trailing newline
`.substr($start, $length)`	extract a part of the string. `$length` defaults to the rest of the string
`.chars`	number of characters in the string
`.uc`	upper case
`.lc`	lower case
`.ucfirst`	convert first character to upper case
`.lcfirst`	convert first character to lower case
`.capitalize`	convert the first character of each word to upper case, and all other characters to lower case

11.3 Bool

A Boolean value is either `True` or `False`. Any value can coerce to a boolean in boolean context. The rules for deciding if a value is true or false depend on the type of the value:

Strings

Empty strings and `"0"` evaluate to `False`. All other strings evaluate to `True`.

Numbers

: All numbers except zero evaluate to `True`.

Lists and Hashes

: Container types such as lists and hashes evaluate to `False` if they are empty, and to `True` if they contain at least one value.

Constructs such as `if` automatically evaluate their conditions in boolean context. You can force an explicit boolean context by putting a `?` in front of an expression. The `!` prefix negates the boolean value.

```
my $num = 5;

# implicit boolean context
if $num { say "True" }

# explicit boolean context
my $bool    = ?$num;

# negated boolean context
my $not_num = !$num;
```

Index