Extracted from:

Practical Vim

Edit Text at the Speed of Thought

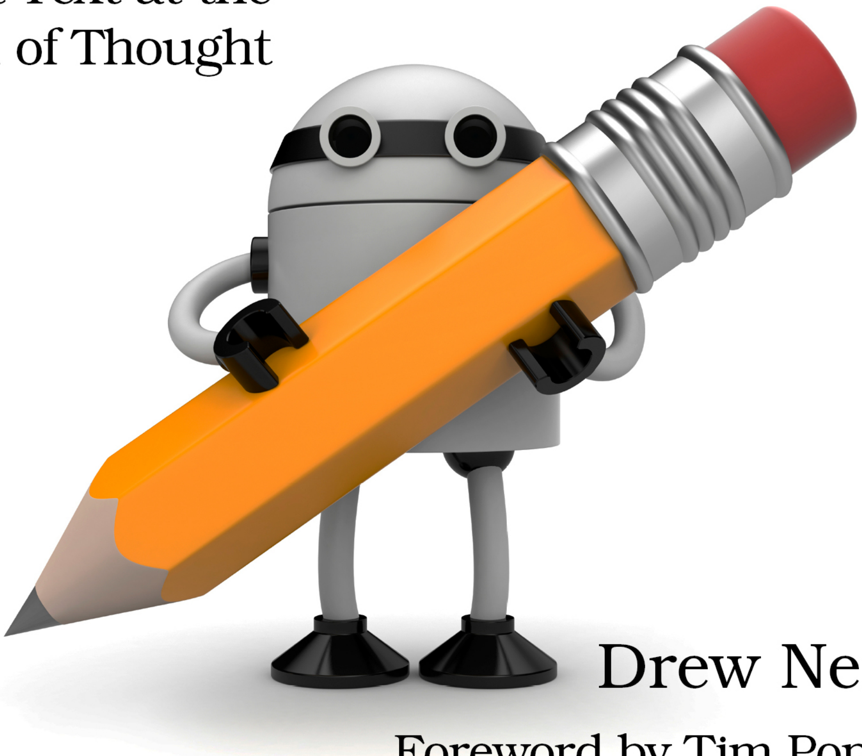
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Practical Vim

Edit Text at the Speed of Thought



Drew Neil

Foreword by Tim Pope

Edited by Kay Keppler

The team that produced this book includes:

Kay Keppler (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-98-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2012

Tip 1

Meet the Dot Command

The dot command lets us repeat the last change. It is the most powerful and versatile command in Vim.

Vim's documentation simply states that the dot command "repeats the last change" (see `:h .` ⓘ). It doesn't sound like much, but in that simple definition we'll find the kernel of what makes Vim's modal editing model so effective. First we have to ask, "What is a change?"

To understand the power of the dot command, we have to realize that the "last change" could be one of many things. A change could act at the level of individual characters, entire lines, or even the whole file.

To demonstrate, we'll use this snippet of text:

the_vim_way/0_mechanics.txt

```
Line one
Line two
Line three
Line four
```

The `x` command deletes the character under the cursor. When we use the dot command in this context, "repeat last change" tells Vim to delete the character under the cursor:

Keystrokes	Buffer Contents

{start}	`Line one`
	`Line two`
	`Line three`
	`Line four`

`x`	`ine one`
	`Line two`
	`Line three`
	`Line four`

`.`	`ne one`
	`Line two`
	`Line three`
	`Line four`

`..`	` one`
	`Line two`
	`Line three`

Keystrokes	Buffer Contents
	`Line four`

We can restore the file to its original state by pressing the `u` key a few times to undo the changes.

The `dd` command also performs a deletion, but this one acts on the current line as a whole. If we use the dot command after `dd`, then “repeat last change” instructs Vim to delete the current line:

Keystrokes	Buffer Contents
{start}	`Line one`
	`Line two`
	`Line three`
	`Line four`
`dd`	`Line two`
	`Line three`
	`Line four`
`.`	`Line three`
	`Line four`

Finally, the `>G` command increases the indentation from the current line until the end of the file. If we follow this command with the dot command, then "repeat last change" tells Vim to increase the indentation level from the current position to the end of the file. In this example, we'll start with the cursor on the second line to highlight the difference

Keystrokes	Buffer Contents
{start}	`Line one`
	`Line two`
	`Line three`
	`Line four`
`>G`	`Line one`
	`  Line two`
	`  Line three`
	`  Line four`
`j`	`Line one`
	`  Line two`
	`  Line three`
	`  Line four`
`.`	`Line one`
	`  Line two`
	`    Line three`
	`    Line four`
`j.`	`Line one`
	`  Line two`
	`    Line three`
	`      Line four`

The `x`, `dd`, and `>` commands are all executed from Normal mode, but we also create a change each time we dip into Insert mode. From the moment we enter Insert mode (by pressing `i`, for example) until we return to Normal mode (by pressing `<Esc>`), Vim records every keystroke. After making a change such as this, the dot command will replay our keystrokes (see *Moving Around in Insert Mode Resets the Change, on page ?*, for a caveat).

The Dot Command Is a Micro Macro

Later, in Chapter 11, *Macros*, on page ?, we'll see that Vim can record any arbitrary number of keystrokes to be played back later. This allows us to capture our most repetitive workflows and replay them at a keystroke. We can think of the dot command as being a miniature macro, or a "micro" if you prefer.

We'll see a few applications of the dot command throughout this chapter. We'll also learn a couple of best practices for working with the dot command in Tip 9, on page ?, and Tip 23, on page ?.

Tip 2

Don't Repeat Yourself

For such a common use case as appending a semicolon at the end of a series of lines, Vim provides a dedicated command that combines two steps into one.

Suppose that we have a snippet of JavaScript code like this:

the_vim_way/2_foo_bar.js
```
var foo = 1
var bar = 'a'
var foobar = foo + bar
```

We need to append a semicolon at the end of each line. Doing so involves moving our cursor to the end of the line and then switching to Insert mode to make the change. The `$` command will handle the motion for us, and then we can run `a`;<Esc> to make the change.

To finish the job, we could run the exact same sequence of keystrokes on the next two lines, but that would be missing a trick. The dot command will repeat that last change, so instead of duplicating our efforts, we could just run `j$.` twice. One keystroke (`.`) buys us three (`a`;<Esc>). It's a small saving, but these efficiencies accumulate when repeated.

But let's take a closer look at this pattern: `j$.`. The `j` command moves the cursor down one line, and then the `$` command moves it to the end of the line. We've used two keystrokes just to maneuver our cursor into position so that we can use the dot command. Do you sense that there's room for improvement here?

Reduce Extraneous Movement

While the `a` command appends after the current cursor position, the `A` command appends at the end of the current line. It doesn’t matter where our cursor is at the time, pressing `A` will switch to Insert mode and move the cursor to the end of the line. In other words, it squashes `$a` into a single keystroke. In *Two for the Price of One*, on page 6, we see that Vim has a handful of compound commands.

Here is a refinement of our previous example:

Keystrokes	Buffer Contents
{start}	`var foo = 1`
	`var bar = 'a'`
	`var foobar = foo + bar`
`A`;<Esc>	`var foo = 1;`
	`var bar = 'a'`
	`var foobar = foo + bar`
`j`	`var foo = 1;`
	`var bar = 'a'`
	`var foobar = foo + bar`
`.`	`var foo = 1;`
	`var bar = 'a';`
	`var foobar = foo + bar`
`j.`	`var foo = 1;`
	`var bar = 'a';`
	`var foobar = foo + bar;`

By using `A` instead of `$a`, we give the dot command a boost. Instead of having to position the cursor at the *end* of the line we want to change, we just have to make sure it is *somewhere* (anywhere!) on that line. Now we can repeat the change on consecutive lines just by typing `j.` as many times as it takes.

One keystroke to move, another to execute. That’s about as good as it gets! Watch for this pattern of usage, because we’ll see it popping up in a couple more examples.

Although this formula looks terrific for our short example, it’s not a universal solution. Imagine if we had to append a semicolon to fifty consecutive lines. Pressing `j.` for each change starts to look like a lot of work. For an alternative approach, skip ahead to Tip 30, on page ?.

Two for the Price of One

We could say that the `A` command compounds two actions (`$a`) into a single keystroke. It's not alone in doing this. Many of Vim's single-key commands can be seen as a condensed version of two or more other commands. The table below shows an approximation of some examples. Can you identify anything else that they all have in common?

Compound Command	Equivalent in Longhand
`C`	`c$`
`s`	`cl`
`S`	`^C`
`I`	`^i`
`A`	`$a`
`o`	`A<CR>`
`O`	`ko`

If you catch yourself running `ko` (or worse, `k$a<CR>`), stop! Think about what you're doing. Then recognize that you could have used the `O` command instead.

Did you identify the other property that these commands share? They all switch from Normal to Insert mode. Think about that and how it might affect the dot command.

Tip 3

Take One Step Back, Then Three Forward

We can pad a single character with two spaces (one in front, the other behind) by using an idiomatic Vim solution. At first it might look slightly odd, but the solution has the benefit of being repeatable, which allows us to complete the task effortlessly.

Suppose that we have a line of code that looks like this:

the_vim_way/3_concat.js

```
var foo = "method("+argument1+","+argument2+")";
```

Concatenating strings in JavaScript never looks pretty, but we could make this a little easier on the eye by padding each + sign with spaces to make it look like this:

```
var foo = "method(" + argument1 + "," + argument2 + ")";
```

Make the Change Repeatable

This idiomatic approach solves the problem:

Keystrokes	Buffer Contents
{start}	`var foo = "method("+argument1+","+argument2+")";`
`f+`	`var foo = "method("+argument1+","+argument2+")";`
`s␣+␣<Esc>`	`var foo = "method(" + argument1+","+argument2+")";`
`;`	`var foo = "method(" + argument1+","+argument2+")";`
`.`	`var foo = "method(" + argument1 + ","+argument2+")";`
`;.`	`var foo = "method(" + argument1 + "," + argument2+")";`
`;.`	`var foo = "method(" + argument1 + "," + argument2 + ")";`

The `s` command compounds two steps into one: it deletes the character under the cursor and then enters Insert mode. Having deleted the + sign, we then type `␣+␣` and leave Insert mode.

One step back and then three steps forward. It's a strange little dance that might seem unintuitive, but we get a big win by doing it this way: we can repeat the change with the dot command; all we need to do is position our cursor on the next + symbol, and the dot command will repeat that little dance.

Make the Motion Repeatable

There's another trick in this example. The `f{char}` command tells Vim to look ahead for the next occurrence of the specified character and then move the cursor directly to it if a match is found (see :h f ⓘ). So when we type `f+`, our cursor goes straight to the next + symbol. We'll learn more about the `f{char}` command in Tip 49, on page ?.

Having made our first change, we could jump to the next occurrence by repeating the `f+` command, but there's a better way. The `;` command will repeat the last search that the `f` command performed. So instead of typing `f+` four times, we can use that command once and then follow up by using the `;` command three times.

All Together Now

The `;` command takes us to our next target, and the `.` command repeats the last change, so we can complete the changes by typing `;.` three times. Does that look familiar?

Instead of fighting Vim's modal input model, we're working with it, and look how much easier it makes this particular task.