Lesson | Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Semantic Versioning

Text

When we install a JavaScript package, it will include a version number that looks something like this:

```
4.46.0
```

In fact, this is the version of webpack we'll be installing in future lessons.

But what do these numbers even mean? Why are there multiple decimal points?

This convention is used for **semantic versioning**. Semantic versioning just means that we are specifying the **major** version, the **minor** version, and the **patch** number. Semantic versioning is used in all sorts of software, not just npm packages.

It's important to understand these differences because you can run into a lot of trouble with your development environment otherwise.

The first number `4` is the **major** version. Major versions represent "breaking" changes to software. This doesn't mean the software is broken — it just means that the package is no longer backwards compatible. If you are using version `3` and you upgrade to version `4` without actually updating your code to account for the changes, it will probably break your code. That's what "breaking changes" means. This is why software companies often use older **legacy** code. Legacy code just means the code isn't using the latest versions of all packages — and in some cases, the legacy code may be very old indeed. If a software company has a lot of code that's reliant on a specific package, they might not want to update to the latest version, even if it has great new features, because making that update will break the existing code.

In short, major versions mean major changes.

The second number (after the first decimal) is the **minor** version. In the number above, the minor version is `46`. Minor version changes aren't breaking changes — and they are backwards compatible with other minor versions in that major version number. In other words, `4.46` should be backwards compatible with `4.01` because they have the same major version number. However, `4.46` wouldn't be backwards compatible with `3.99` because they have a different major version number.

Minor versions can still have updates or new functionality — it's just that they'll be backwards compatible with other minor versions in that major version. Keep in mind that minor versions can potentially introduce new bugs (as can new major versions), so having the latest version doesn't automatically mean that everything will be "better" — it just means updates have been made.

Finally, the third number (after the second decimal) is the **patch number**. This is the number of patches that's been done on this minor version. A patch is something that "patches up" a bug or

problem in the code.

Let's look at the number above again:

```
4.46.0
```

The major version is 4, the minor version is 46, and there have been 0 patches to this minor version.

As we mentioned before, this version number is for webpack, one of the most popular packages for JavaScript, React, and other JavaScript frameworks. It is well-funded and a large chunk of the JavaScript development community depends on it. Even so, it is *constantly* being updated and changed — and even though it's a great piece of software, it has plenty of bugs, too.

At this point, you might wonder how semantic versioning applies to you. If you aren't building your own software and packages, why does it matter?

Well, software packages don't always play nicely with each other. Let's say that the makers of package A make a major version change. In the process, they think about compatibility with package B and package C because they are both widely used and industry standard. However, they don't think about compatibility with package Z, which is instrumental to your project. If we update package A in our project and it doesn't play nicely with package Z, our project will break. We may get an instructive error message or we might not — after all, the makers of package Z might not even know about this breaking change yet — or they might be open source and not have the resources to address the issue quickly. To make matters worse, there's often very little documentation (or none at all) about conflicts between package versions, especially if it's a very recent issue or it involves a less common combination of packages.

This can even happen with minor versions or patch numbers. A change to the minor version or patch number just ensures that the changes are backwards compatible with itself — not every other package you might be using. That means even a change to a minor version or patch number in one package could cause an issue with another package we're working with, breaking our application. In fact, the more packages we work with, the more likely this becomes. We'll discuss how we can avoid these issues soon.

It's important to remember that developing software is an iterative process — software is constantly changing, which means new bugs are being introduced (and fixed). Sometimes that process is exciting, sometimes it's frustrating, and sometimes it's overwhelming. In fact, if a product is no longer being changed, it's probably not because it's stable and finished. More likely, it's obsolete and no longer being maintained!

Also, before we move on — keep in mind that LearnHowToProgram.com follows the same iterative process. We, too, are constantly updating lessons to keep up with changes. In the process, we introduce (and fix) bugs. We also keep some legacy lessons and code because it doesn't always make sense to update to the latest, hottest version.

As a developer, your job is to solve problems and overcome these challenges. Look at any bugs or issues you might run into as part of the process, whether they come from external projects, your own code, or even from LearnHowToProgram.com. We find that students do best when they keep their growth mindset and look at any issues they run into as positive challenges to learn from. After all, these are the exact challenges you'll run into the industry as well.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/creating-a-package-json-with-npm)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/installing-dependencies-with-npm-webpack-and-webpack-cli)

Lesson 7 of 49
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)