Semantic Versioning Cheatsheet

What Is A Semantic Version?

A strictly non-decreasing `major.minor.patch` version number, e.g. `1.2.3`.

Format

```
major.minor.patch[-prerelease][+build]
```

Version Examples

```
0.1.2
1.0.0-alpha
1.1.0-beta.2
1.2.3-rc.4+20220201AB34EF
```

Ordering Examples

```
  1.0.0-alpha+1234
= 1.0.0-alpha+5678
= 1.0.0-alpha
< 1.0.0-alpha.1
< 1.0.0-alpha.beta
< 1.0.0-beta
< 1.0.0-beta.2
< 1.0.0-beta.11
< 1.0.0-rc.1
< 1.0.0
< 2.0.0
< 2.1.0
< 2.1.1
```

Rules & Actions

SCENARIO	WHAT TO DO	MUST OR MAY
Backwards incompatible changes to public API	Increment `major`, reset `minor` and `patch` to `0`	Must
Backwards compatible changes to public API	Increment `minor`, reset `patch` to `0`	Must
Deprecations in public API	Increment `minor`, reset `patch` to `0`	Must
Substantial changes to private code	Increment `minor`, reset `patch` to `0`	May
Backwards compatible bug fixes	Increment `patch`	Must

Once a versioned software package has been released, the contents of that version ***must not*** be modified.

^ A major-level change may include minor- and patch-level changes. A minor-level change may include patch-level changes.

Dependency Management

> *Say you're writing an application called `firetruck`, which depends on a library `ladder`, which complies with semantic versioning. At the time of development, `ladder` is at version 3.1.0. What should you expect?*

You should expect `firetruck` to work with any `ladder` releases with versions in `[3.1.0, 4.0.0)`.

Recommendations

- Use `0.1.0` as your first version
- Use major version `0` for rapid development, when your API might change every day. Once the software has a stable API / is being used in production, it should be `1.0.0`.

See

https://semver.org