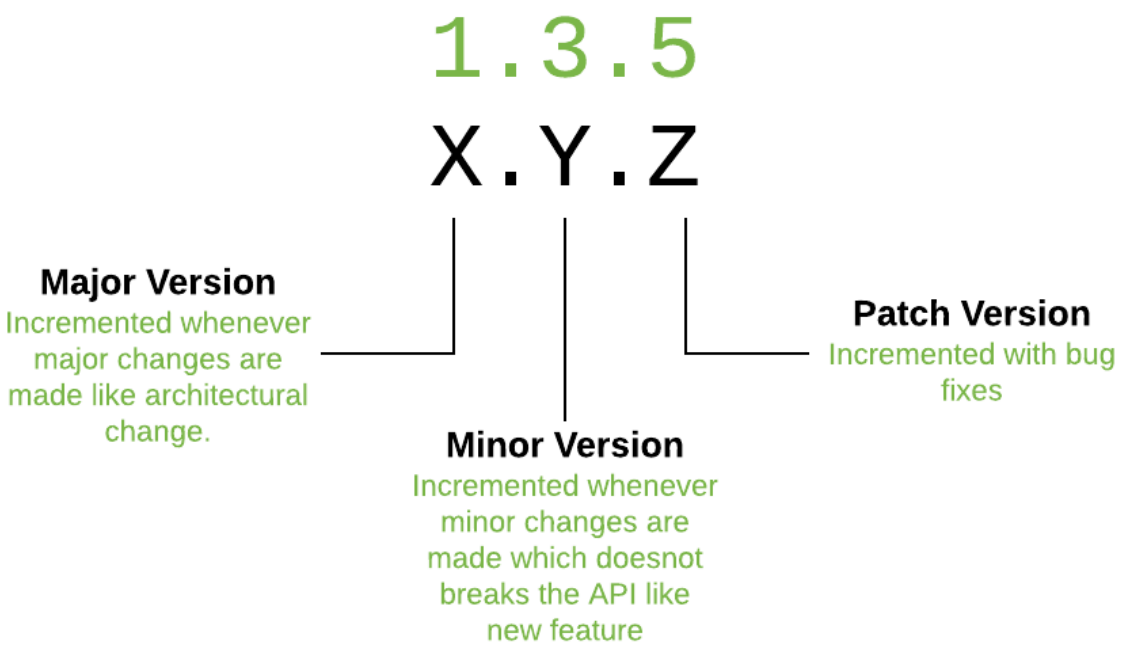
Semantic Versioning

Why?

- Suppose we have a library, say React.
- There are other codebases that are dependent on React
- Semantic versioning will help the user of React to decide if they can safely update to current version, without breaking their existing codebase.
- It is the most widely used versioning system.

What?

- Semantic version is represented using three numbers, separated by period.
- These numbers are Major.Minor.Patch, for example 7.2.0
  - Here, 7 is Major version
  - 2 is the Minor version
  - 0 is the Patch version
- When a bug has been fixed, only the Patch version changes, 7.2.0 becomes 7.2.1, this bug fix doesn’t break the code-bases that are using our library.
- When there is a new feature, the Minor version goes up, 7.2.1 becomes 7.3.0, this addition of feature also doesn’t break the dependent libraries.
- The final one is the Major version, when there is a breaking change in our code, this number increments by 1. This breaking change has the potential to break the code-bases that are dependent on our library. 7.3.0 become 8.0.0
- Note that when a version number increases, all the numbers after it are set to 0.



Why Avoid Breaking Changes?

- Software devs should avoid making breaking changes unless necessary.
- If you have to make breaking changes, better support the old behaviors, and mark them as deprecated. This way users of your package/library will know that certain features won't work in the future.

Special Characters: caret(^) and tilde(~)

- When we run the command

```
npm update
```

- If a version has ^ before its version number, npm updates the package only up to the latest Minor version number.
- 2.3.4 will change to 2.5.0, even if 3.0.0 is the latest version
- ~ works the same way but with patch number

References

- Understanding Semantic Versioning
  - https://www.youtube.com/watch?v=1zBzkT7QCmA
- Updating NPM packages
  - https://www.youtube.com/watch?v=MFhdkLDWjYQ