Node Tool System:

NPM:

npm is our Node Package Manager. We install packages in our local project, or in our global folder, to add additional features to our JS development environment.

- Install package for product launch: *npm i <name> –prod:*
- *I*nstall package for development environment: *npm <name> –dev*
- *list all packages installed: npm ls*
- *Install globally : npm i -g <name>:.*
- *Install a specific version of a package.: npm i <name>@X.Y.Z .*
- *Run a node Script: npm run <scriptname>*
- *Uninstall a package: npm uninstall <name>*

Npm saves time on building by using a system cache – this is useful if you have the same packages being locally installed for many different projects. However, the cache files can become old, or dependency issues can sometimes occur. Use the following commands:

- Run through all packages and verify proper install: *npm cache verify*
- *Clean a Dirty Cache :Npm cache clean --force*

NPX:

Npx is our Node Package Runner. If pre-requesites are not installed, it will temporarily install them in memory, to run the package. These will **not be saved** after the package finishes running.

- npx <packageName>: to run a package in command line

NVM:

nvm is used to manage multiple versions of node. As the inter-dependencies and legacy issues for different nodeJS packages is highly complex – it is **critical** for a developer to not limit themselves to a fixed version of node. Note: nvm is acutally a shell script. It is not a package, or library.

- List all node versions available: *nvm ls*
- *Use a particular node version: nvm use <name>*
- nvm use default: just use the latest version (18+).

node is our interpreter and compiler for backend Javascript files, and packages. It allows us to run JS in the backend, and gives us access to system calls and functionality not available outside of a browser's sandbox security model (such as the ability to make FIFO Pipes, or write/delete files).

NODE:

On its own, the node command will compile any CommonJS .js or .cjs file that you give it. It is designed to run back-end JS files.

- Run a JS script or node Module: *node <module>*

Import and Export Systems:

ESModules: [2]
Exports:

- what is *exported* will be available outside the module. Things not exported cannot be directly accessed by outsiders (private).
- The *"default"* keyword can only be used once, and is the primary object that is exported by the module. Often times, we can use this to export a main() or starter() method from the module.
  - Usage: *export default <entity_name>;*
- **Export Notation:**
  - In-line export: *export <function/constant> name = {};*
  - Export of multiple declared entities in file: *export default {fun1, fun2, const1, const2..}*

Imports:

- *Default values* that are imported, **need** to have names assigned to them.
  - Example: import <givenname> from "./pathto/file.js"
  - **Note:** If the default method is a starter method – we don't need to know much about the internals of the module to run it.
- Import all members (including default member), and wrap them in an object/use namespacing:
  - Example: *import * as <given name> from "./pathto/file;*
  - **Note:** You have to know what the members are called (internally in the module) to do this.
- Import named members:
  - Example: *import { a, b, c }... from "./pathto/file";*
  - *Note: If you know the internals of the module, or have many methods that need called to get it to work, use this.*
  - Exported Member re-aliased: *import object as alias from "./pathto/file;*
- *Mixing default and named exports: Is totally possible.*
  - *Example: import <given name to default>, {named1, named2} from "./pathto/file.js"*
- *Side-Effect import (just run the code in the module, do not import anything):*
  - *Example: import "./pathto/file.js";*

CommonJS:

- any JS file is a CommonJS module, if it exports one of its symbols.

Export Notation:

- **Exporting from a file:**
- Example: *module.exports = { fun1, fun2, const1, const2.... }*
- *Note: this should work for almost every case.*

Import Notation:

- Object Wrapper Import: *let <name> = require("./pathto/file.js");*
- Import Multiple symbols from module: *const {a,b,c} = require("./pathto/file.js");*

Differences Between CommonJS and EcmaScript Module Systems:

Require():	**Import ... :**
Uses original CommonJS loader.	Uses (newer) EcmaScript Loader
Common JS is closer to regular Javascript – needs minimal transpilation to work in browser	Import and ES Import system not apart of JavaScript specification – further transpilation needed – with Babel and Parceling libraries.
For .js and .cjs files	For .mjs files (experimental mode). Can do .js files now.
Require() statement can be loaded anywhere in code – more flexible.	Import can only be used at the beginning of a file – less flexible
Supported in all NodeJS versions.	Only supported in nodeJS 8.0+

Package.json file:

Sections and Tags:

- **dependencies:** These are dependencies that are used in production and development (all neccessary packages)..
- **dev-dependencies:** these are packages used in development only (such as support packages like rimraf, or light-server).
- s**cripts:** activated with *"npm <script name>" on command line. Use to run cleanup, test, dev and production launches with ease. Rules can chain and refer to other rules, as well.*
- type: "module"/"commonjs": The package type – decides whether we load .js files with CommonJS or EcmaScript Loader. If omitted, .js is done with requires and CommonJS.
- packages are just folders with a package.json file, in the end.
- module: any file or directory that exports a symbol for use outside of itself..
- we run modules with the node compiler. We run packages with the npx command.

Other FAQs:

- What is the difference between node <scriptname> and npm start?

Node will execute .js and .mjs files directly, and output them to cmd line. Npm <script tag> will run any package.json script line you tell it. The funcitonality is a superset of the node compiler command. In addition to executing scirpts, you can run the main starter methods of other packages, or even use common shell commands (such as | or >).
For reference, we can make npm start behave as node would, with "start": "node <startfile.js>" in our package.json. [1]

- How do we load both CommonJS and ES modules in the same file or project?

The commonJS system cannot import any ES Modules. You can never import an ES Mod using the require() statement.

ES Modules, can for the most part import commonJS modules. The import statement can be used to import all exports, or specific named functions as you would normally do. Some specific rules from [3]:

1.You can only use `import` and `export` in an ES module. Specifically, this means you can only use `import` and `export` in a `.mjs` file, or in a `.js` file if `"type": "module"`.

2.You cannot use `require` in an ES module, you must use `import`.

3.You cannot use `require` to load an ES module.

4.**So, what does work?**

1.An ES module can import exports from a CommonJS module. As far as mixing goes, that's it.

References:

[1]: https://stackoverflow.com/questions/31362021/npm-start-vs-node-app-js

[2]: https://ui.dev/esmodules

[3]: *https://pencilflip.medium.com/using-es-modules-with-commonjs-modules-in-node-js-1015786dab03*