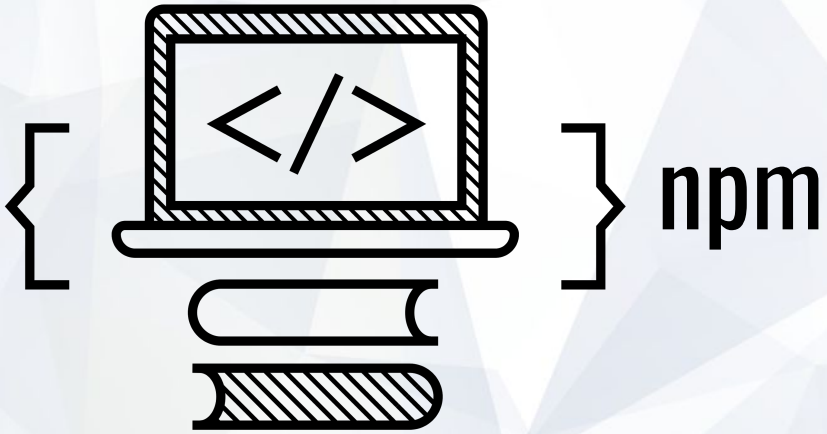


Course: Java

S1

WELCOME

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `npm` to initialize a JavaScript package.

02 Add third-party package dependencies by using `npm install`.

03 Add development dependencies.

04 Execute `npm` workflows, including `start` and `test`.

Node Package Manager (npm)

Node Package Manager (npm)

The term “npm” refers to three things:

01

A JavaScript package repository that’s hosted at https://registry.npmjs.org/ and that has a user interface at the npm, Inc. site.

02

A command-line tool that we can use to create JavaScript projects, resolve dependencies, and automate development workflows.

03

The **npm, Inc.** company, which maintains the repository and the command-line tool. GitHub acquired this company in 2020, which means that Microsoft ultimately owns it.

The npm Command-Line Tool

We spend most of our time working with the command-line tool. Even when we fetch third-party packages, we might not need to think about them.

```
# display all commands
> npm -l

# involved overview (opens docs)
> npm help npm

# command details (opens docs)
> npm help [command]
```

The Goal: Create a Tic-Tac-Toe Game

Our goal today is to create a tic-tac-toe game from the command-line by using `npm`.

The game doesn't need to be fancy. It just needs to demonstrate the following:

- Creating a package.
- Adding a production dependency.
- Adding a development dependency.
- Running `start` and `test`.

Step 1: Run npm init

Create a `tic-tac-toe` directory, navigate into it, run `npm init`, and accept the defaults:

```
> mkdir tic-tac-toe

> cd ./tic-tac-toe

> npm init
This utility will walk you through
creating a package.json file.
It only covers the most common items
and tries to guess sensible defaults.

See `npm help init` for definitive
documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterward to
install a package and save it as a
dependency in the package.json file.
```

```
Press ^C at any time to quit.
package name: (tic-tac-toe)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to
.\tic-tac-toe\package.json:
```

```
{
  "name": "tic-tac-toe",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

  Is this OK? (yes)
```

Step 2: Edit package.json

We're encouraged to edit the `package.json` project/package file. VS Code has the auto-prompt and auto-complete features built in for `package.json`. Be aware that some project files are more risky to edit.

```
{
  "name": "tic-tac-toe",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

`name`: Package name

- Rules: No upper-case characters; < 214 characters; must start with a letter unless it's a scoped package; must be URL-friendly. If we publish to the `npm` repository, it must be unique across all names in the repository.

`scripts`: Development workflows.

`version`: Version number: [major].[minor].[patch]

`description`: A brief overview.

`main`: File containing exports that can be imported by using the package name.

`author`: A single author. (To add multiple, use the `contributors` property.)

`license`: The package's license. (Important for publication, so that people understand valid usage.)

Step 3: Add the First Line of JavaScript

Start small. We're confirming that we can get something to run.

Add an `index.js` file, which `npm init` does not create.

```
"use strict";

console.log("Welcome to Tic-Tac-Toe!");
```

In `package.json`, add a `start` entry to `scripts`. The script tells Node.js to run `index.js`.

```
{
  "name": "tic-tac-toe",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js", // HERE
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

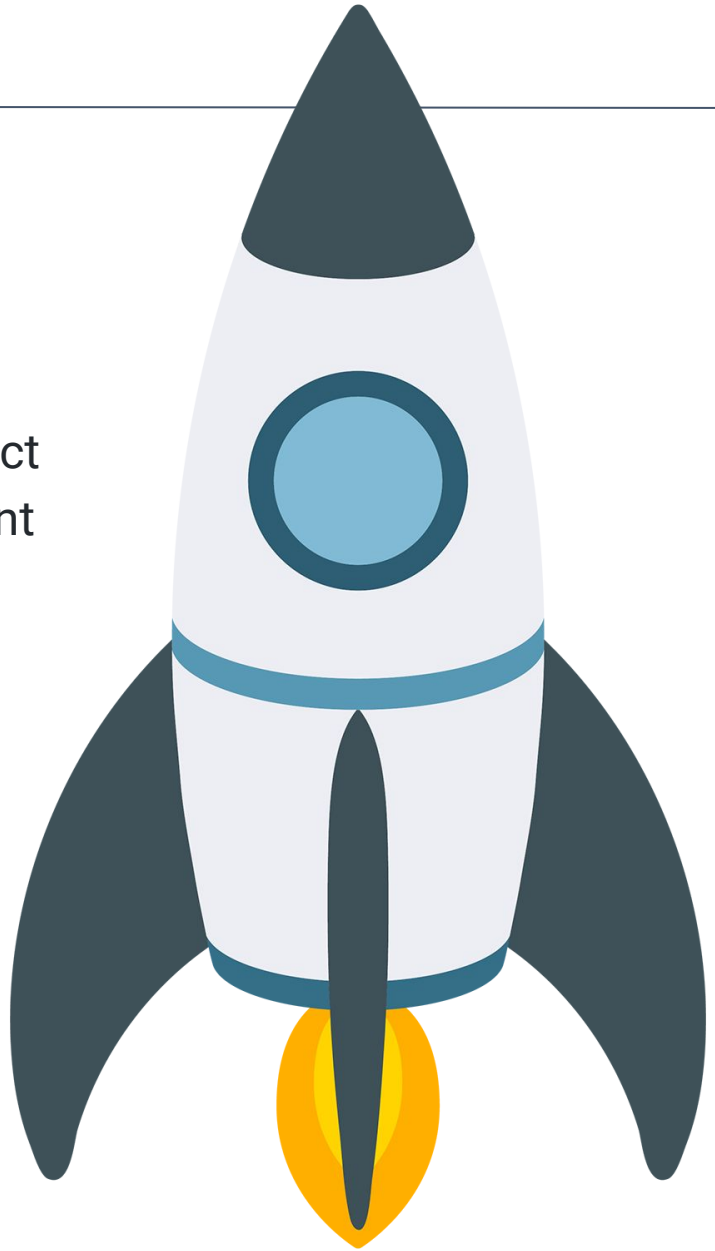
Step 4: Run npm start

- Note that `npm start` runs the start script.
- In this case, the outcome is the `node index.js` command.
- This might seem like an unnecessary step. Why do it?
- Because `start` scripts can become more complex. For example, React applications can compile resources, bundle them, start a development HTTP server, serve the bundle, and then launch the default browser.

```
> npm start

> tic-tac-toe@1.0.0 start .\tic-tac-toe
> node index.js

Welcome to Tic-Tac-Toe!
```




Break

Third-Party Packages

Run npm install

Unlike Maven, `npm` prefers to store dependencies locally in the project. It's possible to install them globally. But, most JS developers don't, because the packages change more quickly than Java's.

- The `npm install` command can locate and install packages in various ways (see `npm help install`).
- Given a package name, it fetches and installs from the npm-hosted repository.
- We'll install prompt-sync 4.2.0, which is a package for synchronously collecting user input.

```
> npm install prompt-sync
npm WARN tic-tac-toe@1.0.0 No description
npm WARN tic-tac-toe@1.0.0 No repository field.

+ prompt-sync@4.2.0
added 3 packages from 3 contributors and audited
3 packages in 0.51s
found 0 vulnerabilities
```

package.json: Dependencies and node_modules

Three things change after the installation:

01

The `package.json` file contains a `dependencies` object that tracks prompt-sync and its version.

02

A new `package-lock.json` file exists.

03

A new `node_modules` directory exists that contains the dependency source code.

`package.json`

```
{
  "name": "tic-tac-toe",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" &&
exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "prompt-sync": "^4.2.0"
  }
  }
```

Using Dependencies

- We use the `require` function to import code from a package.
- A package's code can include functions, objects, and values.
- Use the prompt-sync documentation to determine what to expect.

```
"use strict";
// prompt-sync exports a create function
const createPrompt = require("prompt-sync");
// create a new prompt function with the create
function.
const prompt = createPrompt();

console.log("Welcome to Tic-Tac-Toe!");

const playerOne = prompt("Player #1, What's your
name?:");
console.log(`Player #1: ${playerOne}`);
```

Time to Code

Tic-Tac-Toe

Suggested Time:

15 Minutes


Break

Development Dependencies

Development Dependencies

A development dependency is one that our code doesn't directly use. Instead, the development process uses it. That is, `npm` workflows use it.

Examples include linters, testing libraries, and even HTTP servers. (We might use an HTTP server to launch our app, but the server isn't part of the app itself.)

We'll add ESLint to our tic-tac-toe game to check for syntax and code errors.

- A `linter` is a static code analysis tool that flags programming errors, bugs, stylistic errors, and questionable constructs.

The npm install --save-dev Option

For `npm install`, many options exist. The following are common save options:

- The `--save-prod` option: The default, which app operation requires.
- The `--save-dev` option: The save option that development requires.

- For now, we will only use the `--save-dev` option, which saves ESLint in `devDependencies`.

```
> npm install eslint --save-dev
npm WARN tic-tac-toe@1.0.0 No description
npm WARN tic-tac-toe@1.0.0 No repository field.

+ eslint@7.19.0
added 115 packages from 65 contributors and audited 118 packages in 5.744s

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Use devDependencies

- A few `package.json` properties can store dependencies.
- Nothing restricts dependency use, so it's important to be careful. For example, app code can import a dev dependency.
- ESLint has significantly more dependencies than prompt-sync!

```
{
  "name": "tic-tac-toe",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "prompt-sync": "^4.2.0"
  },
  "devDependencies": {
    "eslint": "^7.19.0"  // HERE
  }
}
```

Initialize the ESLint Configuration

- The **Node Package Executor (npx)** runs a package. If the package is local, it uses it. Otherwise, it fetches the package on demand and deletes it when finished.
- Here we initialize the configuration that ESLint needs to do its job:

```
> npx eslint --init
√ How would you like to use ESLint? · problems
√ What type of modules does your project use? · commonjs
√ Which framework does your project use? · none
√ Does your project use TypeScript? · No / Yes
√ Where does your code run? · node
√ What format do you want your config file to be in? · JSON
Successfully created .eslintrc.json file in .\tic-tac-toe
```

Add a Pretest Script

- As a pretest, we add a script that recursively runs ESLint, looking for files with the .js extension.
- When we run the script, ESLint scans our .js files for problems and writes warnings to the console.

```
"scripts": {
  "start": "node index.js",
  "pretest": "eslint ./", // HERE
  "test": "echo This is where testing would happen."
 }
```

Use npm run

- Knowing how to use `run`, `start`, and `test` are foundational npm skills.
- Your code probably won't trigger a linting error. (See the list of ESLint enforced rules here: https://eslint.org/docs/rules/.) To see a linting failure, double up a semicolon at the end of a statement.
- We can run any script in `scripts`, and we're welcome to create our own.

```
"scripts": {
  "start": "node index.js",
  "pretest": "eslint . --ext .js",
  "my-script": "echo my-script",
  "custom-script": "echo custom-script”
  }
```

`npm help run`

```
> npm run pretest
# (output depends on linting errors)

> npm run my-script

> npm run custom-script

# can run dedicated scripts
> npm run start
```

Using Dedicated Scripts

Some scripts have a shorthand command:

- The `npm start` shorthand command runs the `npm run start` script.
- The `npm test` shorthand command runs the `npm run test` script.
- For more information, run `npm help` scripts.

Additionally, we can add a `pre` or `post` prefix to a named script, and the prefixed script will run before or after the script, respectively.

```
"scripts": {
  "start": "node index.js",
  "pretest": "eslint . --ext .js",
  "precustom": "echo PREcustom",
  "custom": "echo CUSTOM",
  "postcustom": "echo POSTcustom"
  }
```

Slide terminal output is a simplification.
The actual output is below.

```
$ npm run custom

> tic-tac-toe@1.0.0 precustom .\tic-tac-toe
> echo PREcustom

PREcustom

> tic-tac-toe@1.0.0 custom .\tic-tac-toe
> echo CUSTOM

CUSTOM

> tic-tac-toe@1.0.0 postcustom .\tic-tac-toe
> echo POSTcustom

  POSTcustom
```

Use npm test

Note that we'll cover testing later in this module. We'll then use `npm test` to run our tests.

- In any meaningful project, we test our code, and we run those tests by using `npm test`.
- Failed linting in a pretest cancels testing, which saves time.

```
"scripts": {
    "start": "node index.js",
    "pretest": "eslint . --ext .js",
    "test": "echo This is where testing would happen."
  }
```

`npm test`

```
> npm test

> tic-tac-toe@1.0.0 pretest .\tic-tac-toe
> eslint . --ext .js

> tic-tac-toe@1.0.0 test .\tic-tac-toe
> echo This is where testing would happen.

This is where testing would happen.
```


Break

Doggie Day Care

Suggested Time:

60 Minutes


RECAP

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `npm` to initialize a JavaScript package.

02 Add third-party package dependencies by using `npm install`.

03 Add development dependencies.

04 Execute `npm` workflows, including `start` and `test`.

Questions?