13.1. What are Modules?

Just like functions should be kept small and accomplish only one thing, we want to apply the same idea for the different parts of our program. **Modules** allow us to keep the features of our program in separate, smaller pieces. We code these smaller chunks and then connect the modules together to create the big project.

Modules are like Legos. Each piece has its own distinct shape and function, and the same set of pieces can be combined in lots of different ways to create unique results.

13.1.1. One Possible Scenario

Imagine we want to create a program that quizzes students on their JavaScript skills.

What would go into this app? Features could include:

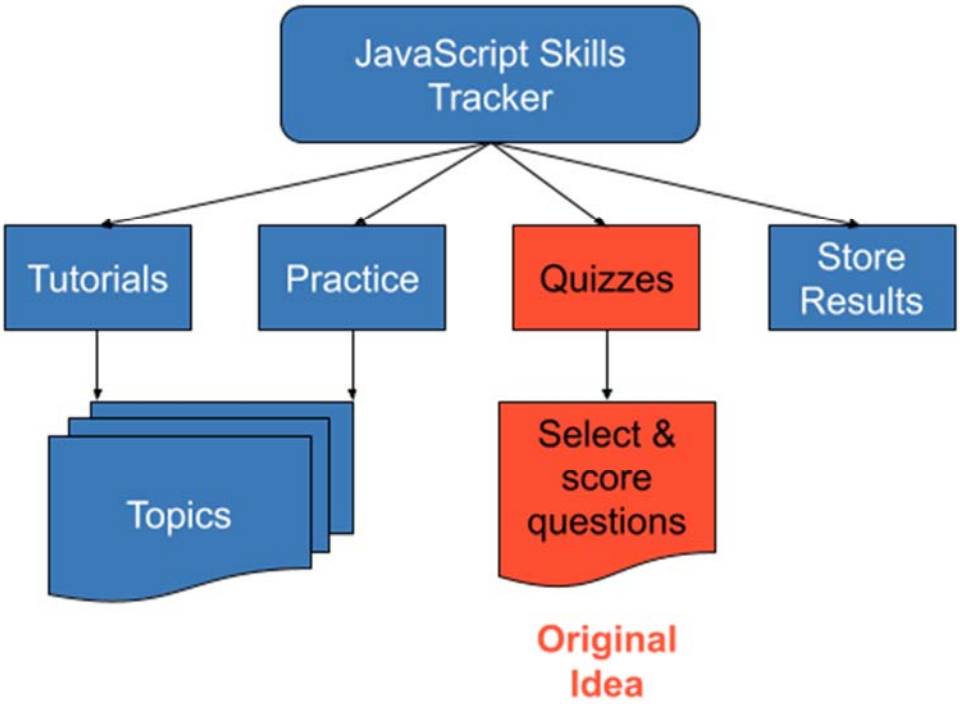
1. Selecting questions from a stored array or object.
2. Presenting the questions to the students and collecting their answers.
3. Scoring the quizzes.
4. Storing the results.

This would be a useful app, but we could make it better by adding some other features. Instead of just quizzing students, maybe we could add some tutorial pages. Our app now provides some teaching and assessment content.

Next, how about adding some non-graded practice to make sure the students are ready for their final quiz? Once we accomplish that, we could continue adding to our app to make it better and better.

Let's pause a moment to consider what happened to the size of our project. As the program evolved from the straightforward quiz app to one that included tutorials and practice tasks, the number of lines of code increased. Now imagine we replicate all of these features for two or three other programming languages.

We can picture our app as follows:



Imagine if the project included two or three additional programming languages!

The result is a mammoth program that contains thousands of lines of code. How would this impact debugging? How about keeping the code DRY? Do any of the features overlap? How easy is it to add new features?

13.1.2. Why Use Modules

Modules help us keep our project organized. If we find a bug in the quiz part of our program, then we can focus our attention on the quiz module rather than the entire codebase.

Modules also save us effort in other projects - another example of the DRY concept. We have already practiced condensing repetitive tasks into loops or functions. Similarly, if we design our quiz module in a generic way, then we can use that same module in other programs.

Even better, we can SHARE our modules with other programmers and use someone else's work (with permission) to enhance our own. Writing the imaginary quiz/tutorial/practice app from scratch would take us many, many weeks. However, someone in the coding community might already have modules that we can immediately incorporate into our own project—saving us time and effort.

Modules keep us from reinventing the wheel.

Some modules also provide us with useful shortcuts. readline-sync allowed us to collect input from a user, and this module contains lots of other methods besides the `.question` we used in our examples. Rather than making every developer write their own code for interacting with the user through the console, `readline-sync` makes the process easier for all by providing a set of ready-to-use functions. We do not need to worry about HOW the module works. We just need to be able to pull it into our projects and use its functions.

13.2. Require Modules

In order to take advantage of modules, we must *import* them with the **require** command. You have seen this before with `readline-sync`.

```
1 const input = require('readline-sync');
2
3 let name = input.question("What is your name?");
4 console.log(`Hello, ${name}`);
```

Line 1 imports the `readline-sync` module and assigns its functions to the `input` variable.

Modules are either *single functions* or *objects that contain multiple functions*. If importing a module returns a single function, we use the variable name to call that function. If the module returns an object, we use dot notation to call the functions stored in the object. In line 3, we see an example of this. `input.question` calls the `question` function stored in the `readline-sync` module.

Later, we will see examples of importing and using single function modules.

Example

Let's check the type of `input` after we import the `readline-sync` module.

```
const input = require('readline-sync');

console.log(typeof input);
```

Console Output

```
object
```

The `readline-sync` module contains several key/value pairs, each of which matches a key (e.g. `question`) with a specific function.

13.2.1. Where Do We Find Modules?

Modules come from three places:

1. A local file on your computer.
2. Node itself, known as Core modules.
3. An external registry such as NPM.

13.2.2. How Does Node Know Where to Look?

The string value passed into `require` tells Node where to look for a module.

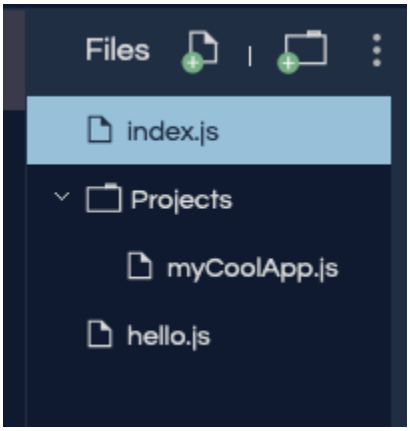
13.2.2.1. User Created Modules

If a module is stored on your computer, the string passed into `require` must provide a *path* and a *filename*. This path tells Node where to find the module, and it describes how to move up and down within the folders on your computer. Paths can be extremely detailed, but best practice recommends that you keep local modules either in the same folder as your project or only one level from your project. Simple paths are better!

A **relative path** starts with `./` or `../`.

1. `./` tells Node, *Search for the module in the current project folder*.
2. `../` tells Node, *Search for the module in the folder one level UP from the project.*

As an example, let's assume we have a folder structure like:



Following best practice gives us three scenarios for importing one file into another:

1. **The module is in the same folder:** If we want to import `hello.js` into `index.js`, then we add `const hello = require('./hello.js');` on line 1 of `index.js`.
2. **The module is one level up:** If we want to import `hello.js` into `myCoolApp.js`, then we add `const hello = require('../hello.js');` on line 1 of `myCoolApp.js`.
3. **The module is one level down:** If we want to import `myCoolApp.js` into `index.js`, then we add `const coolApp = require('./Projects/myCoolApp.js');` on line 1 of `index.js`. This tells Node to search for `myCoolApp.js` in the `Projects` sub-folder, which is in the same folder as `index.js`.

13.2.2.2. Other Modules

If the filename passed to `require` does NOT start with `./` or `../`, then Node checks two resources for the module requested.

1. Node looks for a Core module with a matching name.
2. Node looks for a module installed from an external resource like NPM.

Core modules are installed in Node itself, and as such do not require a path description. These modules are *local*, but Node knows where to find them. Core modules take precedence over ANY other modules with the same name.

Note

W3 schools provides a convenient list of the Core Node modules.
If Node does find the requested module after checking Core, it looks to the NPM registry, which contains hundreds of thousands of free code packages for developers.

In the next section, we will learn more about NPM and how to use it.

13.2.3. Package.json File

Node keeps track of all the modules you import into your project. This list of modules is stored inside a `package.json` file.

For example, if we only import `readline-sync`, the file looks something like:

```
{
  "main": "index.js",
  "dependencies": {
     "readline-sync": "1.4.9"
  }
}
```

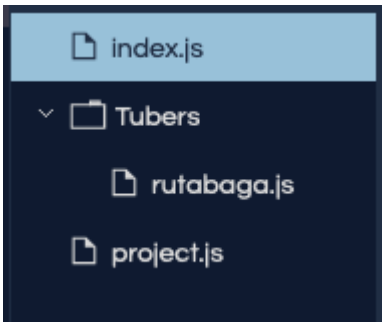
Note

You may not have seen `package.json` yet, because repl.it hides this file by default. We will talk more about this later.

13.2.4. Check Your Understanding

Question

Assume you have the following file structure:



Which statement allows you to import the `rutabaga` module into `project.js`?

a. `const rutabaga = require('/rutabaga.js');`
b. `const rutabaga = require('./rutabaga.js');`
c. `const rutabaga = require('../rutabaga.js');`
d. `const rutabaga = require('./Tubers/rutabaga.js');`

13.3. NPM

NPM, Node Package Manager, is a tool for finding and installing Node modules. NPM has two major parts:

1. A registry of modules.
2. Command line tools to install modules.

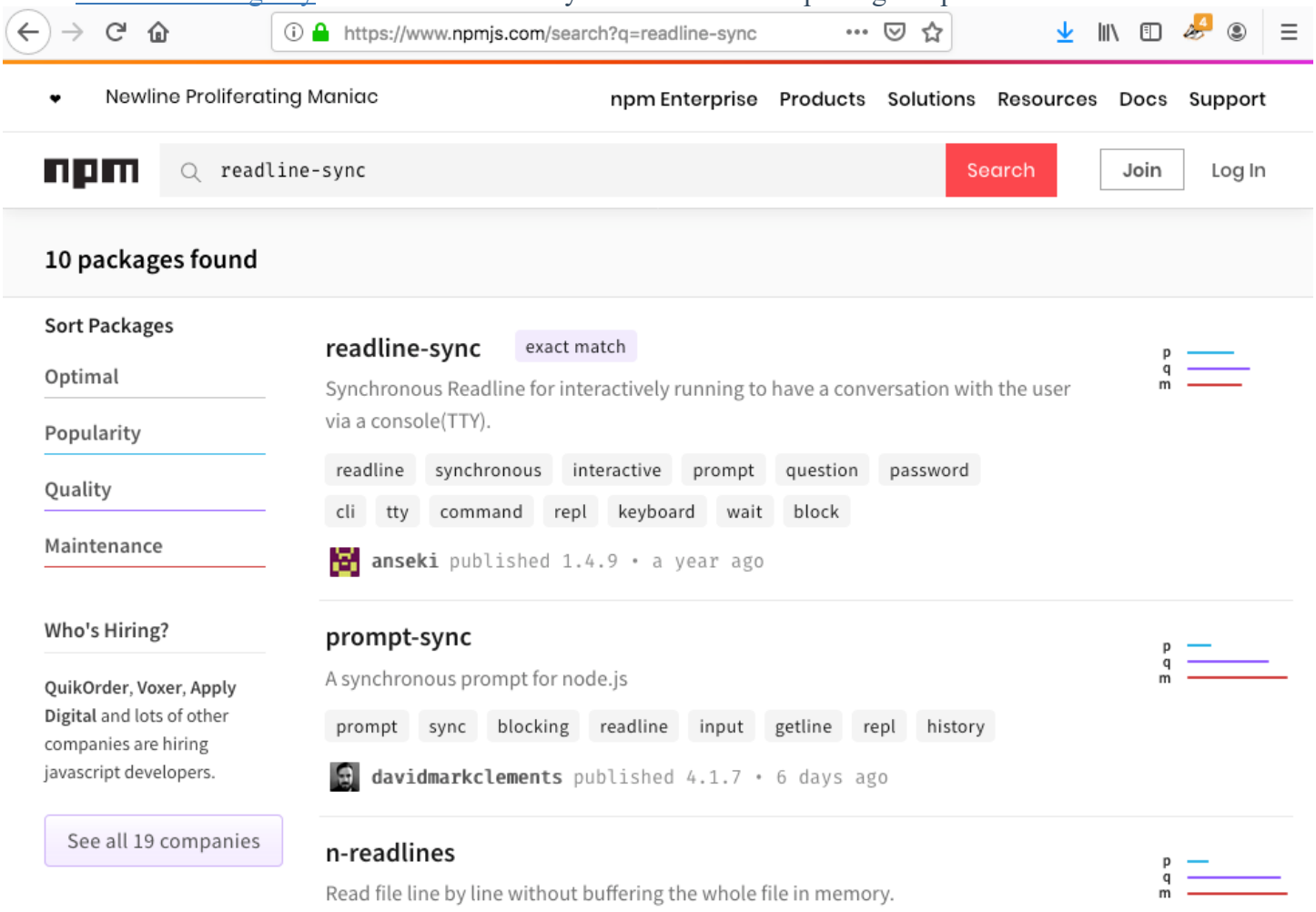
13.3.1. NPM Registry

The NPM registry is a listing of thousands of modules that are stored on a remote server. These can be `required` and downloaded to your project. The modules have been contributed by other developers just like you.

There is an online version of the registry where you can search for a module by name or desired functionality.

Example

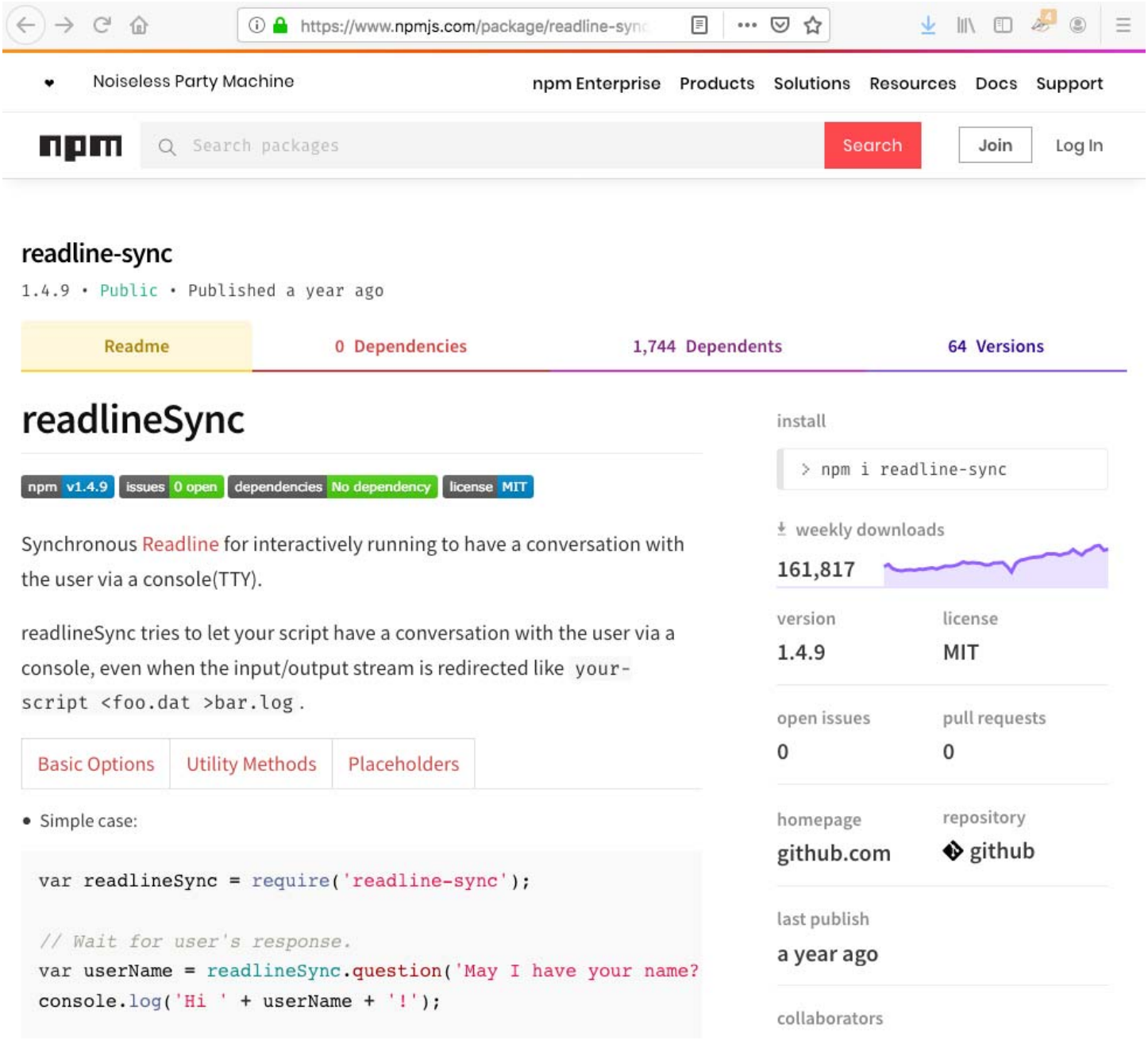
Go to online NPM registry and enter "readline-sync" into the search packages input box.



An exact match appears as the first result. That is the `readline-sync` module we required. Clicking on the first result leads to the NPM page that describes the `readline-sync` module.

On the details page you will see:

1. Usage statistics (how often the module is used)
2. Instructions on how to use the module (example code)
3. Version information
4. The author(s)
5. Sourcecode repository



13.3.2. NPM Command Line Interface (CLI)

The NPM command line tool, **CLI**, is installed with Node. The NPM CLI is used in a computer's terminal to install modules into a Node project. Before we can talk more about the NPM CLI, however, we need to discuss repl.it and NPM.

We have coded many Node projects inside of repl.it. Repl.it is great. It allows us to simulate a development environment WITHOUT having to install any software on our computers. As such, it automatically handles much of the work for installing external modules.

Let's examine how using the CLI is different when using repl.it.

13.3.3. CLI With Local Development Environment

The following describes the steps for working with NPM outside of repl.it. Do not worry about following along. We are simply reviewing these to familiarize you with the general process.

1. Install Node on your computer, which also installs the NPM CLI tool.
2. Use the CLI tool in a terminal to install modules into your local project. The syntax is `npm install <package_name>`, and running it downloads the module to your computer and adds an entry into a `package.json` file. This entry indicates that your project depends on the module called `package_name`.
3. More detailed instructions can be found in the NPM documentation.

13.3.4. NPM CLI With repl.it

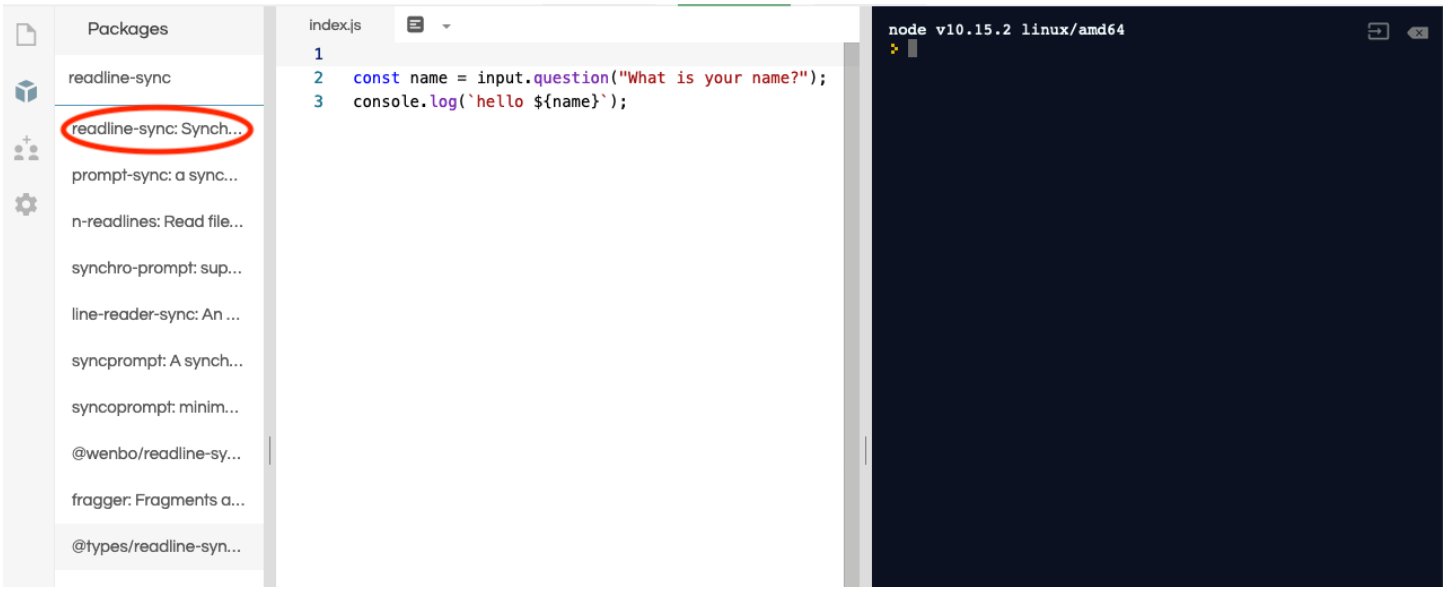
NOW you can follow along, because you use repl.it frequently.

Example

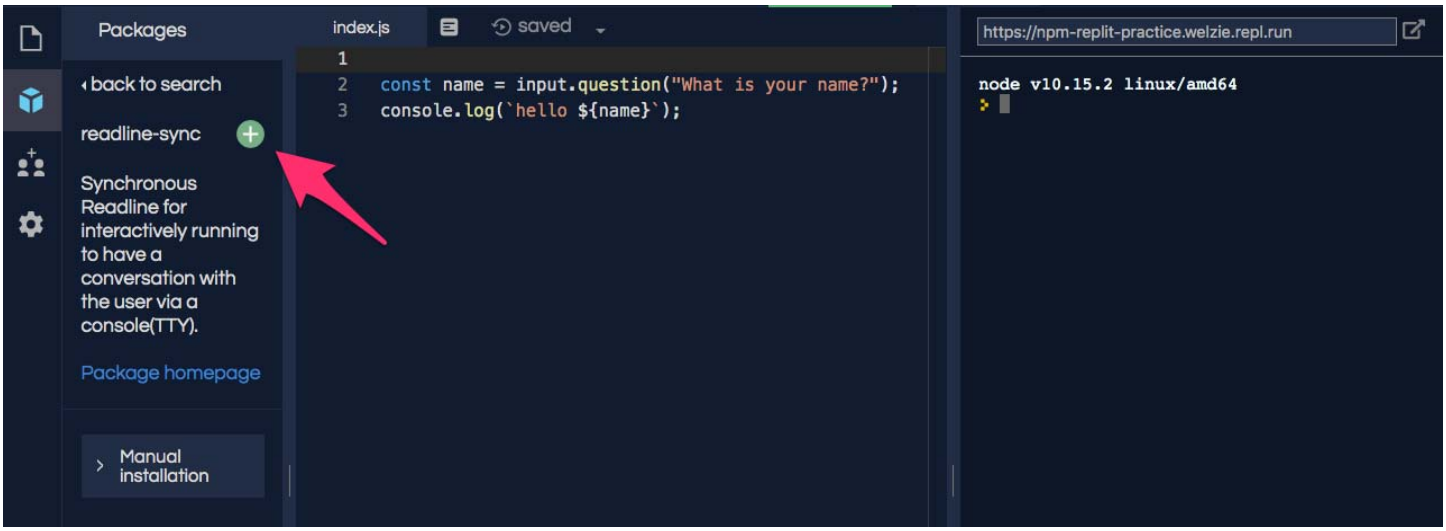
Fork this example repl.it.

Since we are in repl.it, we can skip NPM CLI. Instead, we will use the repl.it interface to add the modules we want.

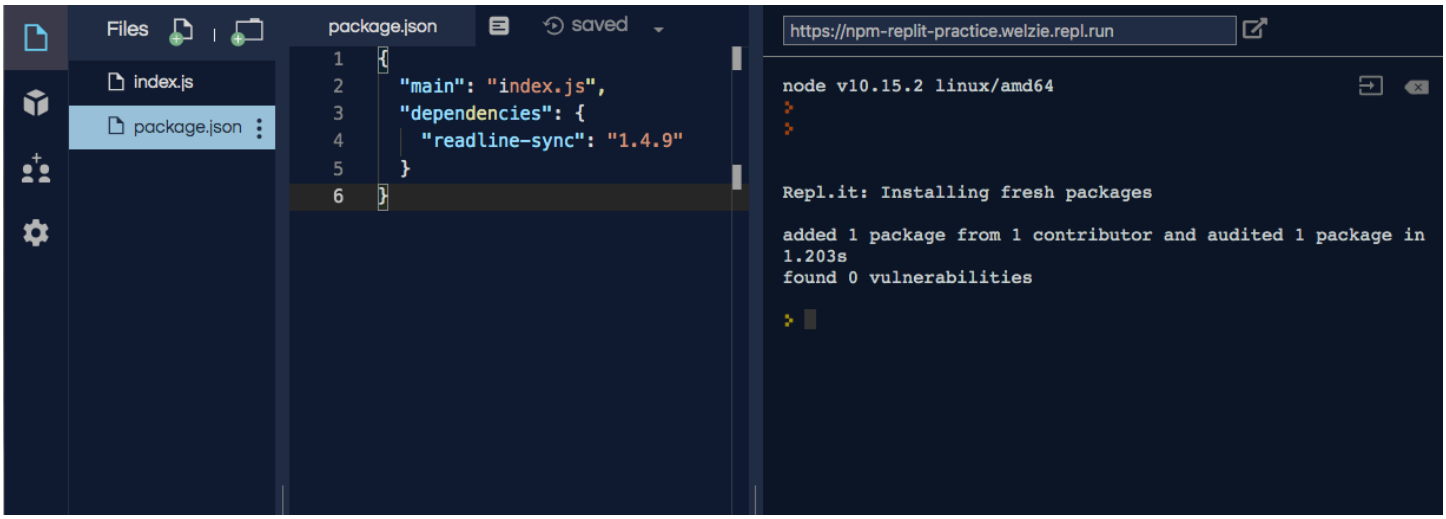
1. Click on the Packages icon in the left menu (it looks like a box).
2. Enter “readline-sync” in the search box.
3. Click on the top matching result.



1. Verify this is the module you want, then click on the plus icon.



Clicking the plus icon adds a `package.json` file that includes a dependency listing for `readline-sync`.



Even though we added `readline-sync` to `package.json`, our code still fails because `input` is not defined. The final step of requiring `readline-sync` is to assign it to a variable.

Add `const input = require("readline-sync");` to line 1.

```
1 const input = require("readline-sync");
2
3 const name = input.question("What is your name?");
4 console.log(`hello ${name}`);
```

Note

So far, we used repl.it without a `package.json` file. That worked because repl.it tries to make the development experience as easy as possible. It hides some details in order to let us pay more attention to our code.

13.5. Wrap-up

In this chapter, we showed how to use `require` to pull a module into your project, and we presented two ways to use `module.exports`. Of course, these are not the only ways to share content.

A quick search online shows that besides functions, we can also share individual variables. There are also alternative syntaxes for `module.exports` - even one that exports as an object, but imports as a function (which means no dot notation).

The skills you practiced in this chapter provide a solid foundation for modules. Learning the alternatives becomes a matter of personal preference and the requirements for your job.

13.6. Exercises: Modules

Practice makes better. You will create a project that accomplishes the following:

a. Steps through a list of Yes/No questions.
b. Calls functions based on the user's responses.

Rather than coding all of the functions from scratch, you are going to use existing modules to help assemble your project.

Open this link and fork the starter code, then complete the following:

13.6.1. Export Finished Modules

Lucky you! Some of your teammates have already coded the necessary functions in the `averages.js` and `display.js` files.

1. In `averages.js`, add code to export all of the functions within an object.
2. In `display.js`, add code to export ONLY `printAll` as a function.

13.6.2. Code & Export New Module

`randomSelect.js` requires your attention.

1. Add code to complete the `randomFromArray` function. It should take an array as an argument and then return a randomly selected element from that array.
2. Do not forget to export the `randomFromArray` function so you can use it in your project.

13.6.3. Import Required Modules

The project code is in `index.js`. Start by importing the required modules:

1. Assign `readline-sync` to the `input` variable.
2. Assign the functions from `averages.js` to the `averages` variable.
3. Assign the `printAll` function from `display.js` to the `printAll` variable.
4. Assign the function from `randomSelect.js` to the `randomSelect` variable.

13.6.4. Finish the Project

Now complete the project code. (Note - The line references assume that you added no blank lines during your work in the previous section. If you did, no worries. The comments in `index.js` will still show you where to add code).

1. Line 21 - Call `printAll` to display all of the tests and student scores. Be sure to pass in the correct arguments.
2. Line 24 - Using dot notation, call `averageForTest` to print the class average for each test. Use `j` and `scores` as arguments.
3. Line 29 - Call `averageForStudent` (with the proper arguments) to print each astronaut's average score.
4. Line 33 - Call `randomSelect` to pick the next spacewalker from the `astronauts` array.

13.6.5. Sanity check!

Properly done, your output should look something like:

```
Would you like to display all scores? Y/N: y
Name        Math      Fitness   Coding    Nav       Communication
Fox         95        86        83        81        76
Turtle      79        71        79        87        72
Cat         94        87        87        83        82
Hippo       99        77        91        79        80
Dog         96        95        99        82        70

Would you like to average the scores for each test? Y/N: y
Math test average = 92.6%.
Fitness test average = 83.2%.
Coding test average = 87.8%.
Nav test average = 82.4%.
Communication test average = 76%.

Would you like to average the scores for each astronaut? Y/N: y
Fox's test average = 84.2%.
Turtle's test average = 77.6%.
Cat's test average = 86.6%.
Hippo's test average = 85.2%.
Dog's test average = 88.4%.

Would you like to select the next spacewalker? Y/N: y
Turtle is the next spacewalker.
```

13.7. Studio: Combating Imposter Syndrome

There is a widely recognized condition called **Impostor Syndrome**. First described in the 1970s, it refers to a situation where someone doubts their accomplishments, and they fear that their success is the result of "faking it". People experiencing imposter syndrome ignore external evidence of their skills, and they attribute their success to luck.

At this point in their learning journey, many new coders feel doubt about their prospects. However, they have PLENTY of company—supreme court justice Sonia Sotomayor, Serena Williams, Tom Hanks, and multiple CEOs have all questioned their success.

The struggle is real, and an open conversation often helps.

13.7.1. You CAN

First, a little perspective. Identify which of the following tasks you have already done or know that you can accomplish:

1. Use code to print "Hello, World" to the screen.
2. Define, initialize, change, and use variables.
3. Convert the string `'1234'` into a number.
4. Construct a `for` loop to repeat a task 100 times.
5. Construct `if/else if/else` statements to decide which of three tasks to perform.
6. Build, modify, and access an array.
7. Design and call a function.
8. Call one function from within another function.
9. Find and fix bugs in a segment of non-working code.

How many of the 9 items listed above did you indicate? There is no 'passing' score for this. Whether you checked all 9 or only 1 or 2, simply saying, *I can do that*, means you have more coding skill than the bulk of the world's population.

Doubt and uncertainty are normal, especially when exploring a new career. However, with the skills you already know, you can legitimately say, *I am a coder*. Combined with the skills you will learn during the rest of the course, there can be no doubt. You ARE NOT pretending.

13.7.2. Discussion

Take a few moments in the studio to consider, share, and discuss the following:

1. Have you ever felt the effects of Imposter Syndrome? When?
2. Have you ever responded to a compliment by diminishing the work that earned you the praise? If so, why did you answer in that way?
3. How do you feel in a test/quiz/studio when someone finishes much earlier than you?
4. What are you most proud of from your time working with LaunchCode?
5. What are your strengths?
6. What gives you confidence?
7. How can you use your effort and strengths to boost your confidence?

13.7.3. Real World Comments

1. “We all have impostor syndrome. Every creative person at the top of their field will admit to it. And, frankly, getting notoriety for what you do only accentuates your feeling of being a fraud… because… you know how stupid you are, how lazy you are.” – *Adam Savage, one of the hosts from Mythbusters*
2. “I thought it was a fluke. I thought everybody would find out, and they’d take it back. They’d come to my house, knocking on the door, ‘Excuse me, we meant to give that to someone else. That was going to Meryl Streep.’” - *Jodie Foster after winning an Oscar for Best Actress*
3. “Very few people, whether you’ve been in that job before or not, get into the seat and believe today that they are now qualified to be the CEO. They’re not going to tell you that, but it’s true.” - *Starbucks CEO Howard Schultz admitting to being insecure*
4. “Feeling a little uncomfortable with your skills is a sign of learning, and continuous learning is what the tech industry thrives on! It’s important to seek out environments where you are supported, but where you have the chance to be uncomfortable and learn new things.” - *Vanessa Hurst, Co-Founder of Girl Develop It*

13.7.4. Helpful Tips

Imposter syndrome is real and common. However, there are things you can do to help boost your confidence:

1. *Acknowledge the thoughts*, especially when you enter a new point in your life. Recognize that your feelings are normal.
2. *Put it into perspective*. You have been in LC101 for 4 - 5 weeks. It is OK if you do not understand everything on Stack Overflow or recognize all the details about the latest technology.
3. *Review your accomplishments*. Think about your life prior to JavaScript when *string*, *object* and *function* all meant something much simpler. Your learning has been real!
4. *Share with a trusted friend, teacher or mentor*. Other people with more experience can provide reassurance, and they probably felt similar doubts when they started.
5. *Accept compliments*. Luck did not earn you your tech job. There were LOTS of candidates, and you shined enough to set you apart. If someone compliments your effort or the quality of your work, graciously accept.
6. *Teach*. This is a great way to reinforce your learning, and it helps you recognize how much you know.
7. *Remember the power of ‘Yet’*. You are not the master of all skills, of course, but you do know how to learn. With more practice, you will fill in any gaps in your knowledge.