**Introduction**

You’re probably prompted to update your web browser every few months. Do you know why? A few reasons include addressing security vulnerabilities, adding features, and supporting new HTML, CSS, and JavaScript syntax.

The reasons above imply there is a period before a software update is released when there are security vulnerabilities and unsupported language syntax.

This lesson focuses on the latter. Specifically, how developers address the gap between the new JavaScript syntax that they use and the JavaScript syntax that web browsers recognize.

This has become a widespread concern for web developers since Ecma International, the organization responsible for standardizing JavaScript, released a new version of it in 2015, called ECMAScript2015, commonly referred to as ES6. Note, the 6 refers to the version of JavaScript and is not related to the year it was released (the previous version was ES5).

Upon release, web developers quickly adopted the new ES6 syntax, as it improved readability and efficiency. However, ES6 was not supported by most web browsers, so developers ran into browser compatibility issues.

In this lesson, you will learn about two important tools for addressing browser compatibility issues.

* caniuse.com — A website that provides data on web browser compatibility for HTML, CSS, and JavaScript features. You will learn how to use it to look up ES6 feature support.
* Babel — A Javascript library that you can use to convert new, unsupported JavaScript (ES6), into an older version (ES5) that is recognized by most modern browsers.

Let’s get started by running ES6 JavaScript on a fake old web browser version.

**Instructions**

**1.**

Copy the following code into **main.js**. The code includes three features of ES6 syntax: const, let, and ES6 string interpolation.

var pasta = "Spaghetti"; // ES5 syntax

const meat = "Pancetta"; // ES6 syntax

let sauce = "Eggs and cheese"; // ES6 syntax

// Template literals, like the one below, were introduced in ES6

const carbonara = `You can make carbonara with ${pasta}, ${meat}, and a sauce made with ${sauce}.`;

Run the code when you are ready. Look at the errors displayed in our LetScape

// Set the variable below to a number

let esFivePercentageSupport = 97;

// Set the variable below to a number

let esSixTemplateLiterals = 78;

**Why ES6?**

Before we learn how to set up a JavaScript project that converts ES6 to an older version, it’s worth understanding a few of the reasons Ecma made such substantial updates.

The version of JavaScript that preceded ES6 is called JavaScript ES5. Three reasons for the ES5 to ES6 updates are listed below:

* Readability and economy of code — The new syntax is often easier to understand (more readable) and requires fewer characters to create the same functionality (economy of code).
* Addresses sources of ES5 bugs — Some ES5 syntax led to common bugs. With ES6, Ecma introduced syntax that mitigates some of the most common pitfalls.
* A similarity to other programming languages — JavaScript ES6 is syntactically more similar to other object-oriented programming languages. This leads to less friction when experienced, non-JavaScript developers want to learn JavaScript.

Because ES6 addressed the above issues, Ecma knew that adoption by web developers would occur quickly, while web browser support lagged behind.

To limit the impact of ES6 browser compatibility issues, Ecma made the new syntax backwards compatible, which means you can map JavaScript ES6 code to ES5.

**Instructions**

**1.**

In these tasks, you will manually change the code in **main.js** to ES5 syntax.

The let and const keywords were introduced in ES6. Before that, we declared all variables with the var keyword.

Change the let and const keywords in **main.js** to var.

Hint

**main.js** should contain the following code:

var pasta = "Spaghetti"; // ES5 syntax

var meat = "Pancetta"; // ES6 syntax

var sauce = "Eggs and cheese"; // ES6 syntax

// Template literals, like the one below, were introduced in ES6

var carbonara = ...

The elipses in the example above is used in place of the interpolated string.

**2.**

The following line of code is written with ES6 string interpolation:

`You can make carbonara with ${pasta}, ${meat}, and a sauce made with ${sauce}.`

In ES5, we used plus signs (+) to interpolate variables into strings.

In **main.js**, use ES5 string interpolation to set carbonara to the same string as above.

Hint

Use the syntax below to interpolate a string using ES5 syntax:

let varOne = 5;

let esFiveInterpolation = "This number " + varOne + "is interpolated using ES5 syntax."

In the example above, the plus signs (+) are used to interpolate varOne into the string.

var pasta = "Spaghetti"; // ES5 syntax

var meat = "Pancetta"; // ES6 syntax

var sauce = "Eggs and cheese"; // ES6 syntax

// Template literals, like the one below, were introduced in ES6

var carbonara = "You can make carbonara with " + pasta + ", " + meat + ", " + " and a sauce made with " + sauce + ".";

# Transpilation With Babel

In the last exercise, you manually converted ES6 code to ES5. Although manual conversion only took you a few minutes, it is unsustainable as the size of the JavaScript file increases.

Because ES6 is predictably backwards compatible, a collection of JavaScript programmers developed a JavaScript library called Babel that transpiles ES6 JavaScript to ES5.

Transpilation is the process of converting one programming language to another.

In the remaining exercises of this lesson, you will learn how to use Babel to transpile the new, easy-to-write version of JavaScript (ES6) to the old, browser-compatible version of JavaScript (ES5).

In the instructions below, you will pass JavaScript ES6 code to Babel, which will transpile it to ES5 and write it to a file in the **lib** directory.

**Instructions**

**1.**

In the terminal window type:

npm install babel-cli

This installs one of the two required Babel packages.

**2.**

In the terminal window type:

npm install babel-preset-env

This installs the second of two required Babel packages.

**3.**

In the terminal, type npm run build and press enter.

You can view the ES5 code in **./lib/main.js**.

You may need to refresh to see the newly created **lib** directory.

npm install babel-cli

npm install babel-preset-env

npm run build

**npm init**

In the last exercise, you wrote one command in your terminal to transpile ES6 code to ES5. In the next five exercises you will learn how to setup a JavaScript project that transpiles code when you run npm run build from the root directory of a JavaScript project.

The first step is to place your ES6 JavaScript file in a directory called **src**. From your root directory, the path to the ES6 file is **./src/main.js**

The initial JavaScript project file structure is:

project

|\_ src

|\_\_\_ main.js

Before we install Babel, we need to setup our project to use the [node package manager (npm)](https://docs.npmjs.com/getting-started/what-is-npm). Developers use *npm* to access and manage Node packages. Node packages are directories that contain JavaScript code written by other developers. You can use these packages to reduce duplication of work and avoid bugs.

Before we can add Babel to our project directory, we need to run npm init. The npm init command creates a **package.json** file in the root directory.

A **package.json** file contains information about the current JavaScript project. Some of this information includes:

* Metadata — This includes a project title, description, authors, and more.
* A list of node packages required for the project — If another developer wants to run your project, npm looks inside **package.json** and downloads the packages in this list.
* Key-value pairs for command line scripts — You can use npm to run these shorthand scripts to perform some process. In a later exercise, we will add a script that runs Babel and transpiles ES6 to ES5.

If you have Node installed on your computer, you can create a **package.json** file by typing npm init into the terminal.

The terminal prompts you to fill in fields for the project’s metadata (name, description, etc.)

You are not required to answer the prompts, though we recommend at minimum, you add your own title and description. If you don’t want to fill in a field, you can press enter. npm will leave fill these fields with default values or leave them empty when it creates the **package.json** file.

After you run npm init your directory structure will contain the following files and folders:

project

|\_ src

|\_\_\_ main.js

|\_ package.json

npm adds the **package.json** file to the same level as the **src** directory.

**Instructions**

**1.**

Run ls in the terminal to see the file structure of your current directory.

**2.**

Use npm to create a new **package.json** file. We recommend you set the name and description properties to the following values:

* name — learning-babel
* description — Use Babel to transpile JavaScript ES6 to ES5

Enter ls into the terminal to see the file structure of your current directory. Notice the addition of **package.json**.

npc

# Install Node Packages

We use the npm install command to install new Node packages locally. The install command creates a folder called **node\_modules** and copies the package files to it. The install command also installs all of the dependencies for the given package.

To install Babel, we need to npm install two packages, babel-cli and babel-preset-env. However, npm installs over one hundred other packages that are dependencies for Babel to run properly.

We install Babel with the following two commands:

$ npm install babel-cli -D

$ npm install babel-preset-env -D

The babel-cli package includes command line Babel tools, and the babel-preset-env package has the code that maps any JavaScript feature, ES6 and above (ES6+), to ES5.

The -D flag instructs npm to add each package to a property called devDependencies in **package.json**. Once the project’s dependencies are listed in devDependencies, other developers can run your project without installing each package separately. Instead, they can simply run npm install — it instructs npm to look inside **package.json** and download all of the packages listed in devDependencies.

Once you npm install packages, you can find the Babel packages and all their dependencies in the **node\_modules** folder. The new directory structure contains the following:

project

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ package.json

The ... in the file structure above is a placeholder for 100+ packages that npm installed.

**Instructions**

**1.**

Enter ls into the terminal to view the current directory’s file structure.

**2.**

Use npm with the -D flag to install the Babel command line package and add it to the devDependencies property in **package.json**.

**3.**

Use npm with the -D flag to install the Babel preset environment package and add it to the devDependencies property in **package.json**.

Enter ls into the terminal to see the file structure of your current directory. Notice the addition of **node\_modules**.

You can also see the new folder by clicking on the file navigation button above the text editor.

# .babelrc

Now that you’ve downloaded the Babel packages, you need to specify the version of the source JavaScript code.

You can specify the initial JavaScript version inside of a file named **.babelrc**. In your root directory, you can run touch .babelrc to create this file.

Your project directory contains the following folders and files:

project

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ .babelrc

|\_ package.json

Inside **.babelrc** you need to define the preset for your source JavaScript file. The preset specifies the version of your initial JavaScript file.

Usually, you want to transpile JavaScript code from versions ES6 and later (ES6+) to ES5. From this point on, we will refer to our source code as ES6+, because Ecma introduces new syntax with each new version of JavaScript.

To specify that we are transpiling code from an ES6+ source, we have to add the following JavaScript object into **.babelrc**:

{

"presets": ["env"]

}

When you run Babel, it looks in **.babelrc** to determine the version of the initial JavaScript file. In this case, ["env"] instructs Babel to transpile any code from versions ES6 and later.

**Instructions**

**1.**

In the root directory, use touch to create a new **.babelrc** file.

**2.**

Open the **.babelrc** file and add an object that specifies the preset to "env".

Run the code once you’ve added the object.

Hint

Open **.babelrc** using the file explorer in the code editor.

Add the following object to the file:

{

"presets": ["env"]

}

**Babel Source Lib**

There’s one last step before we can transpile our code. We need to specify a script in **package.json** that initiates the ES6+ to ES5 transpilation.

Inside of the **package.json** file, there is a property named "scripts" that holds an object for specifying command line shortcuts. It looks like this:

...

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

}, ...

In the code above, the "scripts" property contains an object with one property called "test". Below the "test" property, we will add a script that runs Babel like this:

...

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "babel src -d lib"

}

In the "scripts" object above, we add a property called "build". The property’s value, "babel src -d lib", is a command line method that transpiles ES6+ code to ES5. Let’s consider each argument in the method call:

* babel — The Babel command call responsible for transpiling code.
* src — Instructs Babel to transpile all JavaScript code inside the **src** directory.
* -d — Instructs Babel to write the transpiled code to a directory.
* lib — Babel writes the transpiled code to a directory called lib.

In the next exercise, we’ll run the babel src -d lib method to transpile our ES6+ code.

**Instructions**

**1.**

In **package.json**, add a script called "build".

When run, the "build" script should use Babel to transpile JavaScript code inside of the **src** folder and write it to a folder called **lib**.

Don’t forget to add a comma after the “test” script.

Hint

Open **package.json** and add the following code below the "test" script:

"build": "babel src -d lib"

# Build

We’re ready to transpile our code! In the last exercise, we wrote the following script in **package.json**:

"build": "babel src -d lib"

Now, we need to call "build" from the command line to transpile and write ES5 code to a directory called lib.

From the command line, we type:

npm run build

The command above runs the build script in **package.json**.

Babel writes the ES5 code to a file named **main.js** (it’s always the same name as the original file), inside of a folder called lib. The resulting directory structure is:

project

|\_ lib

|\_\_\_ main.js

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ .babelrc

|\_ package.json

Notice, the directory contains a new folder named **lib**, with one file, called **main.js**.

The npm run build command will transpile all JavaScript files inside of the **src** folder. This is helpful as you build larger JavaScript projects — regardless of the number of JavaScript files, you only need to run one command (npm run build) to transpile all of your code.

**Instructions**

**1.**

Transpile the ES6+ code to lib using the build command.

Once you’ve done this, navigate to the new **lib** folder and open **main.js**.

npm run build

**Review**

In this lesson, you learned about browser compatibility and transpilation. Let’s review the key concepts:

* ES5 — The old JavaScript version that is supported by all modern web browsers.
* ES6 — The new(er) JavaScript version that is *not* supported by all modern web browsers. The syntax is more readable, similar to other programming languages, and addresses the source of common bugs in ES5.
* caniuse.com — a website you can use to look up HTML, CSS, and JavaScript browser compatibility information.
* Babel — A JavaScript package that transpiles JavaScript ES6+ code to ES5.
* npm init — A terminal command that creates a **package.json** file.
* **package.json** — A file that contains information about a JavaScript project.
* npm install — A command that installs Node packages.
* babel-cli — A Node package that contains command line tools for Babel.
* babel-preset-env — A Node package that contains ES6+ to ES5 syntax mapping information.
* **.babelrc** — A file that specifies the version of the JavaScript source code.
* "build" script — A **package.json** script that you use to tranpsile ES6+ code to ES5.
* npm run build — A command that runs the build script and transpiles ES6+ code to ES5.

For future reference, here is a list of the steps needed to set up a project for transpilation:

1. Initialize your project using npm init and create a directory called **src**
2. Install babel dependencies by running
3. npm install babel-cli -D

npm install babel-preset-env -D

1. Create a **.babelrc** file inside your project and add the following code inside it:
2. {
3. "presets": ["env"]

}

1. Add the following script to your scripts object in **package.json**:

"build": "babel src -d lib"

1. Run npm run build whenever you want to transpile your code from your **src** to **lib** directories.

**Instructions**

If you want some more practice, take a moment to add JavaScript files and ES6 syntax to the **src** folder and try transpiling them.

Once you feel comfortable with this process, try setting up Babel on your own computer.