ecmascript

This package lets you use new JavaScript language features that are part of the ECMAScript 2015 specification but are not yet supported by all engines or browsers. Unsupported syntax is automatically translated into standard JavaScript that behaves the same way.

This video from the July 2015 Meteor Devshop gives an overview of how the package works, and what it provides.

Usage

The `ecmascript` package registers a compiler plugin that transpiles ECMAScript 2015+ to ECMAScript 5 (standard JS) in all `.js` files. By default, this package is pre-installed for all new apps and packages.

To add this package to an existing app, run the following command from your app directory:

```
meteor add ecmascript
```

To add the `ecmascript` package to an existing package, include the statement `api.use('ecmascript');` in the `Package.onUse` callback in your `package.js` file:

```
Package.onUse((api) => {
  api.use('ecmascript');
});
```

Supported ES2015 Features

Syntax

The `ecmascript` package uses Babel to compile ES2015 syntax to ES5 syntax. Many but not all ES2015 features can be simulated by Babel, and `ecmascript` enables most of the features supported by Babel.

Here is a list of the Babel transformers that are currently enabled:

- `es3.propertyLiterals` Makes it safe to use reserved keywords like `catch` as unquoted keys in object literals. For example, `{ catch: 123 }` is translated to `{ "catch": 123 }`.

- `es3.memberExpressionLiterals` Makes it safe to use reserved keywords as property names. For example, `object.catch` is translated to `object["catch"]`.
- `es6.arrowFunctions` Provides a shorthand for function expressions. For example, `[1, 2, 3].map(x => x + 1)` evaluates to `[2, 3, 4]`. If `this` is used in the body of the arrow function, it will be automatically bound to the value of `this` in the enclosing scope.
- `es6.literals` Adds support for binary and octal numeric literals. For example, `0b111110111 === 503` and `0o767 === 503`.
- `es6.templateLiterals` Enables multi-line strings delimited by backticks instead of quotation marks, with variable interpolation:

  ```
  var name = 'Ben';
  var message = `My name is:
  ${name}`;
  ```

- `es6.classes` Enables `class` syntax:

  ```
  class Base {
    constructor(a, b) {
      this.value = a * b;
    }
  }

  class Derived extends Base {
    constructor(a, b) {
      super(a + 1, b + 1);
    }
  }

  var d = new Derived(2, 3);
  d.value; // 12
  ```

- `es6.constants` Allows defining block-scoped variables that are not allowed to be redefined:

  ```
  const GOLDEN_RATIO = (1 + Math.sqrt(5)) / 2;

  // This reassignment will be forbidden by the compiler:
  GOLDEN_RATIO = 'new value';
  ```

- `es6.blockScoping` Enables the `let` and `const` keywords as alternatives to `var`. The key difference is that variables defined using `let` or `const` are visible only within the block where they are declared, rather than being visible anywhere in the enclosing function. For example:

  ```
  function example(condition) {
    let x = 0;
  ```

```
  if (condition) {
    let x = 1;
    console.log(x);
  } else {
    console.log(x);
    x = 2;
  }
  return x;
}

example(true); // logs 1, returns 0
example(false); // logs 0, returns 2
```

- `es6.properties.shorthand` Allows omitting the value of an object literal property when the desired value is held by a variable that has the same name as the property key. For example, instead of writing `{ x: x, y: y, z: "asdf" }` you can just write `{ x, y, z: "asdf" }`. Methods can also be written without the `: function` property syntax:

  ```
  var obj = {
    oldWay: function (a, b) { ... },
    newWay(a, b) { ... }
  };
  ```

- `es6.properties.computed` Allows object literal properties with dynamically computed keys:

  ```
  var counter = 0;
  function getKeyName() {
    return 'key' + counter++;
  }

  var obj = {
    [getKeyName()]: 'zero',
    [getKeyName()]: 'one',
  };

  obj.key0; // 'zero'
  obj.key1; // 'one'
  ```

- `es6.parameters` Default expressions for function parameters, evaluated whenever the parameter is `undefined`, `...rest` parameters for capturing remaining arguments without using the `arguments` object:

  ```
  function add(a = 0, ...rest) {
    rest.forEach(n => a += n);
    return a;
  }
  ```

```
add(); // 0
add(1, 2, 3); // 6
```

- `es6.spread` Allows an array of arguments to be interpolated into a list of arguments to a function call, `new` expression, or array literal, without using `Function.prototype.apply`:

  ```
  add(1, ...[2, 3, 4], 5); // 15
  new Node('name', ...children);
  [1, ...[2, 3, 4], 5]; // [1, 2, 3, 4, 5]
  ```

- `es6.forOf` Provides an easy way to iterate over the elements of a collection:

  ```
  let sum = 0;
  for (var x of [1, 2, 3]) {
    sum += x;
  }
  sum; // 6
  ```

- `es6.destructuring` Destructuring is the technique of using an array or object pattern on the left-hand side of an assignment or declaration, in place of the usual variable or parameter, so that certain sub-properties of the value on the right-hand side will be bound to identifiers that appear within the pattern. Perhaps the simplest example is swapping two variables without using a temporary variable:

  ```
  [a, b] = [b, a];
  ```

  Extracting a specific property from an object:

  ```
  let { username: name } = user;
  // is equivalent to
  let name = user.username;
  ```

  Instead of taking a single opaque `options` parameter, a function can use an object destructuring pattern to name the expected options:

  ```
  function run({ command, args, callback }) { ... }

  run({
    command: 'git',
    args: ['status', '.'],
    callback(error, status) { ... },
    unused: 'whatever'
  });
  ```

- `es7.objectRestSpread` Supports catch-all `...rest` properties in object literal declarations and assignments:

  ```
  let { x, y, ...rest } = { x: 1, y: 2, a: 3, b: 4 };
  x; // 1
  ```

```
y; // 2
rest; // { a: 3, b: 4 }
```

Also enables `...spread` properties in object literal expressions:

```
let n = { x, y, ...rest };
n; // { x: 1, y: 2, a: 3, b: 4 }
```

- `es7.trailingFunctionCommas` Allows the final parameter of a function to be followed by a comma, provided that parameter is not a `...rest` parameter.
- `flow` Permits the use of Flow type annotations. These annotations are simply stripped from the code, so they have no effect on the code's behavior, but you can run the `flow` tool over your code to check the types if desired.

Polyfills

The ECMAScript 2015 standard library has grown to include new APIs and data structures, some of which can be implemented ("polyfilled") using JavaScript that runs in all engines and browsers today. Here are three new constructors that are guaranteed to be available when the `ecmascript` package is installed:

- `Promise` A `Promise` allows its owner to wait for a value that might not be available yet. See this tutorial for more details about the API and motivation. The Meteor `Promise` implementation is especially useful because it runs all callback functions in recycled `Fiber`s, so you can use any Meteor API, including those that yield (e.g. `HTTP.get`, `Meteor.call`, or `MongoCollection`), and you never have to call `Meteor.bindEnvironment`.
- `Map` An associative key-value data structure where the keys can be any JavaScript value (not just strings). Lookup and insertion take constant time.
- `Set` A collection of unique JavaScript values of any type. Lookup and insertion take constant time.
- `Symbol` An implementation of the global `Symbol`s namespace that enables a number of other ES2015 features, such as `for-of` loops and `Symbol.iterator` methods: `[1,2,3][Symbol.iterator]()`.
- Polyfills for the following `Object`-related methods:
  - `Object.assign`
  - `Object.is`
  - `Object.setPrototypeOf`
  - `Object.prototype.toString` (fixes `@@toStringTag` support)

  Complete reference here.
- Polyfills for the following `String`-related methods:

  - `String.fromCodePoint`
  - `String.raw`
  - `String.prototype.includes`
  - `String.prototype.startsWith`
  - `String.prototype.endsWith`
  - `String.prototype.repeat`
  - `String.prototype.codePointAt`
  - `String.prototype.trim`

  Complete reference here.

- Polyfills for the following `Array`-related methods:
  - `Array.from`
  - `Array.of`
  - `Array.prototype.copyWithin`
  - `Array.prototype.fill`
  - `Array.prototype.find`
  - `Array.prototype.findIndex`

  Complete reference here.

- Polyfills for the following `Function`-related properties:
  - `Function.prototype.name` (fixes IE9+)
  - `Function.prototype[Symbol.hasInstance]` (fixes IE9+)

  Complete reference here.