Chapter 62: Arrow Functions

Arrow functions are a concise way of writing anonymous, lexically scoped functions in ECMAScript 2015 (ES6).

Section 62.1: Introduction

In JavaScript, functions may be anonymously defined using the "arrow" (`=>`) syntax, which is sometimes referred to as a *lambda expression* due to Common Lisp similarities.

The simplest form of an arrow function has its arguments on the left side of `=>` and the return value on the right side:

```
item => item + 1 // -> function(item){return item + 1}
```

This function can be immediately invoked by providing an argument to the expression:

```
(item => item + 1)(41) // -> 42
```

If an arrow function takes a single parameter, the parentheses around that parameter are optional. For example, the following expressions assign the same type of function into constant variables:

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

However, if the arrow function takes no parameters, or more than one parameter, a new set of parentheses *must* encase all the arguments:

```
(() => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

If the function body doesn't consist of a single expression, it must be surrounded by brackets and use an explicit **`return`** statement for providing a result:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

If the arrow function's body consists only of an object literal, this object literal has to be enclosed in parentheses:

```
(bar => ({ baz: 1 }))(); // -> Object {baz: 1}
```

The extra parentheses indicate that the opening and closing brackets are part of the object literal, i.e. they are not delimiters of the function body.

Section 62.2: Lexical Scoping & Binding (Value of "this")

Arrow functions are lexically scoped; this means that their **`this`** Binding is bound to the context of the surrounding scope. That is to say, whatever **`this`** refers to can be preserved by using an arrow function.

Take a look at the following example. The class `Cow` has a method that allows for it to print out the sound it makes

after 1 second.

```
class Cow {

  constructor() {
    this.sound = "moo";
  }

  makeSoundLater() {
    setTimeout(() => console.log(this.sound), 1000);
  }
}

const betsy = new Cow();

betsy.makeSoundLater();
```

In the `makeSoundLater()` method, the `this` context refers to the current instance of the `Cow` object, so in the case where I call `betsy.makeSoundLater()`, the `this` context refers to `betsy`.

By using the arrow function, I *preserve* the `this` context so that I can make reference to `this.sound` when it comes time to print it out, which will properly print out "moo".

If you had used a regular function in place of the arrow function, you would lose the context of being within the class, and not be able to directly access the `sound` property.

Section 62.3: Arguments Object

Arrow functions do not expose an arguments object; therefore, `arguments` would simply refer to a variable in the current scope.

```
const arguments = [true];
const foo = x => console.log(arguments[0]);

foo(false); // -> true
```

Due to this, arrow functions are also **not** aware of their caller/callee.

While the lack of an arguments object can be a limitation in some edge cases, rest parameters are generally a suitable alternative.

```
const arguments = [true];
const foo = (...arguments) => console.log(arguments[0]);

foo(false); // -> false
```

Section 62.4: Implicit Return

Arrow functions may implicitly return values by simply omitting the curly braces that traditionally wrap a function's body if their body only contains a single expression.

```
const foo = x => x + 1;
foo(1); // -> 2
```

When using implicit returns, object literals must be wrapped in parenthesis so that the curly braces are not mistaken for the opening of the function's body.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Section 62.5: Arrow functions as a constructor

Arrow functions will throw a `TypeError` when used with the `new` keyword.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Section 62.6: Explicit Return

Arrow functions can behave very similar to classic functions in that you may explicitly return a value from them using the `return` keyword; simply wrap your function's body in curly braces, and return a value:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```