Lesson 1: Introduction to Object Oriented Programming

What is OOP?

Object-Oriented Programming is more than merely using objects in your programs; it's a style of programming that uses objects to organize a program.

In JS101, we thought of a program in terms of variable declarations, conditionals, loops, and function calls. This approach is called procedural programming. Our programs also used some functional programming techniques, such as passing functions to array methods like **map** and **filter**.

OOP is a **programming paradigm** in which we think about a problem in terms of objects. The way we think about a problem changes from a series of steps to a collection of objects that interact with each other.

We think about problems in OOP as a set of objects with **state** and **behaviour**.

Large, complex problems can be a challenge to maintain due to dependecies throughout the program. When done right, OOP makes our code flexible, easy to understand, and easy to change.

However, OOP doesn't necessarily let programmers write smaller programs than the equivalent non-OO program. Instead, OO programs are often much larger than the equivalent non-OO programs.

OOP also doesn't always lead to more efficient code. It can, in some cases, but in can also lead to less eficient code; OOP programs can require more memory, more disk space, and more computing power. However, the advantages of OOP usually outweigh these concerns.

Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming. At its core, encapsulation describes the idea of bundling or combining the data and the operations that work on that data into a single entity (i.e., an object).

Encapsulation is about bundling state (data) and behaviour (operations) to form an object.

In most OOP languages, encapsulation has a broader purpose. It also refers to restricting access to the state and certain behaviours; an object only exposes the data and behaviours that other parts of the application need to work.

In other words, objects expose a **public interface** for interacting with other objects and keep their implementation details hidden. Thus, other objects can't change the data of an object without going through the proper interface.

Unfortunately, JS doesn't support access restrictions.

Practice Problems: OOP and Encapsulation

Question 1 — What is OOP?

- OOP is a programming paradigm in which we think about a problem in terms of objects. We think about problems in terms of a collection of objects with **states** and **behaviours** that interact with each other.

Question 2 — Advantages and Disadvantages of OOP?

- Advantages
    - It lets programmers think about a problem at a higher-level of abstraction, which helps them break down and solve the problem.
    - OOP helps programmers write programs that reduce the dependencies in a program, which makes maintenance easier.
    - Done right, OOP makes code flexible, easy to understand, and easy to change.

- Disadvanatges
    - OOP programs are often much larger than the equivalent procedural program.
    - OOP may lead to less efficient code; OO programs may require more memory, disk space, and computing power.

Question 3 — What does encapsulation refer to in JS?

- Encapsulation is the idea of bundling data and operations associated with that data in a single entity; that is, it's the grouping of related properties and methods in a single object.

Question 4 — How does encapsulation differ from encapsulation in most other OO languages?

- In other languages, encapsulation concerns hiding details of an object from code that uses the object. An object should only expose the methods and properties that other objects need to use the encapsulated object. However, JS does not directly provide the means to limit exposure of methods and properties. There are ways to achieve a degree of access restriction, but they're not perfect.

Creating Objects

OOP is about combining data and behaviour into an object.

When object properties have function values, we call them **methods**.

The OO style strongly discourages changing property values directly. Instead, it encourages using methods to interface with the object.

In a racing game, the object **raceCar** might look like:

```
let raceCar = {

        make: 'BMW',
        fuelLevel: 0.5,
        engineOn: false,

        startEngine: function() {
                raceCar.engineOn = true;
        },

        drive: function() {
                raceCar.fuelLevel -= 0.1;
        },

        stopEngine: function() {
                raceCar.engineOn = false;
        },

        refuel: function(percent) {
                if ((raceCar.fuelLevel + (percent / 100)) <= 1) {
                        raceCar.fuelLevel += (percent / 100);
                } else {
                        reaceCar.fuelLevel = 1;
                }
        },
};
```

Compact Method Syntax

Using functions as object values (i.e., methods) is so common that there's a shorthand syntax called the compact syntax for it:

- You can omit the **:** and the **function** keyword and use parenthesis to denote a method. There is a subtle difference between these two syntaxes, however.

- Instead of **refuel: function(percent)**, we would only write **refuel(percent)**.

The this Keyword

Thus far, we refer to the object from inside the methods by directly using the variable name of the object. Suppose we change the variable name or pass the object to a function that refers to its arguments by a different name. In that case, calling a method with the original variable name will throw a reference error.

We need some way to refer to the object that contains a method from other methods in that object. The keyword **this** provides the desired functionality.

For now, you can assume that when you use **this** inside a method, it refers to the object that contains the method.

Summary

In JS, we achieve encapsulation by making use of an object. The properties of the object hold the state (data), and methods represent behaviour. Inside the methods, the **this** keyword lets us refer to the properties and other methods of the object.

Collaborator Objects

An object's state is stored in properties that refer to other values or objects. The state is often a collection of strings, numbers, and booleans.

An object's properties can hold any value or object; strings, numbers, arrays, and even other objects.

Objects that help provide state within another object are called **collaborator objects**, or more simply, **collaborators**. Collaboration is all about objects working together in some manner. A collaborator works in conjunction -- in collaboration -- with another object.

Collaborator objects play an important role in object-oriented design; they represent the connections between the different classes in your program. When working on an object-oriented program, be sure to consider what collaborators your objects need and whether those associations make sense, both from a technical standpoint and in terms of modelling the problem your program aims to solve.

We often talk of collaborators in the context of custom objects, but collaborators don't have to be custom objects. They can be built-in objects like arrays and dates, as well.

Collaborator objects let you chop up and modularize the problem domain into cohesive pieces. They play an important role in modelling complicated problem domains in OO programming.

Functions as Object Factories

In this assignment, we'll learn how to automate the process of creating objects.

One way to automate object creation is to use **object factories**: functions that create and return objects of a particular type.

In OOP, we often need to create multiple objects of the same type. Object factory functions provide a straightforward abstraction for object creation.

Benefits of using object factories:

- Object factories let you avoid most of the tedium and errors that result from copying and pasting to create multiple objects of the same type.
- Object factories help reduce code duplication.

Practice Problems: Objects and Factories

You can use a shorthand notation when a property and an argument of the function have the same name. For example:

```
function createBook(title, author) {
    return {
        title,
        ...
    };
}
```

this is the same as writing title : title

The template literal lets us interpolate any expression into the string using **${ }**.

Assignment: OO Rock Paper Scissors

The classical approach to planning an object-oriented application includes several steps:

- Write a textual description of the problem or exercise;
- Extract the significant nouns and verbs from the description; and
- Organize and associate the verbs with the nouns.

Note that the nouns are the objects or **types** of objects and the verbs are the behaviors or methods. In OO design, you shouldn't think about the game flow logic during this early design phase.

OOP is all about organizing and modularizing the code into a cohesive structure - objects. Only after you know what objects you need can you look at orchestrating the program's flow.

Once we have the nouns and verbs, we must organize them by associating each verb with the noun that performs the action represented by the verb.

Once we've organized our nouns and verbs into objects, we need an **engine** to orchestrate the objects. The engine is where the procedural program flow should be.

One of the hardest things to understand about OOP is that there is no absolute **right** solution. OOP always comes down to making tradeoffs.

Walk-through: OO RPS Design Choice

In OOP, sub-types often share multiple properties and methods. JS provides some constructs that help extract such duplications to one place.

The general principle of extracting duplicated code to a single place is always worth considering. It makes changes to the code less error-prone and tedious. In the long run, it often leads to less work.

Summary

Important points:

- Encapsulation is the idea of bundling data and operations related to that data in a cohesive unit called an object. In OOP, encapsulation also refers to the idea of restricting access to state and some behaviours, but JS objects don't support that type of encapsulation.

- The simplest way to create a JS object is to use the object literal syntax. Adding methods to an object is as simple as adding a function as the value of a property.

- You can access the properties and methods of an object from within a method using the **this** keyword.

- Objects collaborate with other objects by using them as part of their state. We say that two objects have a collaborator relationship if one of them is part of the state of the other.

- One way to automate the creation of objects is to use the factory function pattern. A factory function returns an object with a particular set of methods and properties. The methods remain the same across the objects, while the property values can be customized by providing them as arguments.

- One object factory can reuse another object factory by mixing the object returned by another factory function into itself by using **Object.assign**.

Quiz

#7:

Collaborators are objects that store state within another object.

Lesson 2: Functions, Objects, and Prototypes

Review - Objects

Objects are one of the eight fundamental types in JS:

- String
- Number
- Boolean
- Null
- Undefined
- Object
- BigInt
- Symbol

They are basically a collection of properties where each property has a key and value. While values can be any of the JS types, **property keys are always strings**.

When dealing with objects, we are basically doing either one of two things: setting a property or accessing a property.

Dot notation is also called **member access notation**, while bracket notation is called **computed member access notation**.

What happens if we access a non-existent property on an object? We get **undefined**. However, we also get the same value when we try to access a property that is explicitly set to **undefined**. That's a dilemma. How do we distinguish one from each other? There are two ways to do that:

- **in** operator; and ——————→ "key" in myObject
- **Object.prototype.hasOwnProperty()**.

Both methods check if a property exists in an object. If it does, **true** is returned, and **false** otherwise.

Another indirect way of checking for property existence is to enumerate the properties of an object via **Object.keys** or **Object.getOwnPropertyNames**. Both return an array of the object's properties. The difference is that **Object.keys** returns an array of enumerable properties while **Object.getOwnPropertyNames** returns all properties regardless if they're enumerable or not.

Object Prototypes

An object factory serves two purposes:

- It returns objects that represent data of a specific type.
- It reuses code.

Factory functions give us the ability to create objects of the same type by merely calling a function.

As useful as factory functions are, there are other ways to extract code into one place so that multiple objects can use it. In JS, we rely heavily on prototypes to accomplish this.

Prototypes

In JS, objects can inhereit properties and behaviour from other objects. If another object, for instance, has a **language** property and a **speak** behaviour, a new object can access and use **language** and **speak** without explicitly defining them in the new object.

More specifically, JS objects use something called **prototypal inheritance**. The object that you inherit properties and methods from is called the **prototype**.

The function **Object.create** creates a new object that inherits properties from an existing object. It takes an object that is called the **prototype object** as an argument, and returns a new object that inherits properties from the prototype object.

The newly created object has access to all the properties and methods that the prototype object provides.

An unusual aspect of this relationship is that the inheriting object doesn't receive any properties or methods of its own. Instead, it **delegates** property and method access to its prototype.

JS objects use an internal **[[Prototype]]** property to keep track of their prototype. When you create an object with **Object.create**, the new object's **[[Prototype]]** property gets assigned to the prototype object.

Note that **[[Prototype]]** is an internal property: you cannot access it directly in your code. However, you can access and replace its value with **Object** functions.

For instance, **Object.getPrototypeOf** takes an object as an argument and returns its prototype object.

You can use **Object.setPrototypeOf** to set the prototype object of an object (i.e., the first argument is the 'new' object — it should already have been declared — and the second argument is the prototype object).

An important consideration when dealing with prototypes is that objects hold a reference to their prototype objects through their internal **[[Prototype]]** property. If the object's prototype changes in some way, the changes are observable in the inheriting object as well.

The Default Prototype

The default prototype is the prototype object of the **Object** constructor, **Object.prototype**. For now, all you need to know is that **Object.prototype** provides the default prototype.

Iterating Over Objects with Prototypes

A **for/in** loop iterates over an object's properties. The iteration includes properties from the objects in its prototype chain. Use **hasOwnProperty** to skip the prototype properties.

Object.keys returns an object's "own" property keys -- you do not need to use **hasOwnProperty**.

Note that both **for/in** and **Object.keys** deal with **enumerable properties**, which is merely a way of talking about properties you can iterate over. Not all properties are enumerable.

In particular, most properties and methods of the built-in types are not. Usually, any properties or methods you define for an object are enumerable.

You can check whether a property is enumerable with the **Object.prototype.propertyIsEnumerable** method.

The Prototype Chain

Since the prototype of an object is itself an object, the prototype can also have a prototype from which it inherits.

If **a** is the prototype of **b** and **b** is the prototype of **c**, we say that objects **b** and **a** are part of the **prototype chain** of object **c**. The complete prototype chain also includes the default prototype, which is the prototype of object **a** in this case. Since the prototype of **Object.prototype** is **null**, the complete prototype chain looks like this:

c —> **b** —> **a** —> **Object.prototype** —> **null**

The __proto__ Property

Many older JS programs use a property named **__proto__**, which is pronounced **dunder proto**, instead of **Object.setPrototypeOf** and **Object.getPrototypeOf**.

“dunder” is a shortened version of “double underscore”.

The **__proto__** property is a deprecated, non-hidden version of the **[[Prototype]]** property.

As a rule, you should only use **__proto__** if you need to support very old browsers or old versions of Node, or as a convenient shortcut with temporary code or debugging operations.

Property Look-Up in the Prototype Chain

When you access a property on an object, JS first looks for an “own” property with that name on the object. If the object does not define the specified property, JS looks for it in the object’s prototype. If it can’t find the property there, it next looks in the prototype’s prototype.This process continues until it finds the property or it reaches **Object.prototype**. If **Object.prototype** also doesn’t define the property, the property access evaluates to **undefined**.

The implication here is that when two objects in the same prototype chain have a property with the same name, the object that’s closer to the calling object takes precedence.

When assigning a property on a JS object, it always treats the property as an “own” property.

The discussion of inheriting properties from other objects applies to methods as well. Methods in JS are merely properties that refer to functions. Thus, when we talk about object properties, we also mean methods.

Methods on Object.prototype

The **Object.prototype** object is at the top of all JS prototype chains. Thus, its methods are available from any JS object provided you don’t explicitly use something like **null** as the prototype object. Here are 3 useful methods:

- **Object.prototype.toString()** returns a string representation of the object.
- **Object.prototype.isPrototypeOf(obj)** determines whether the object is part of another object’s prototype chain.
- **Object.prototype.hasOwnProperty(prop)** determines whether the object contains the property.

Objects Without Prototypes

It turns out that you can create an object that doesn’t have a prototype by setting the prototype to **null**. We would do this like the following:

```
let a = Object.create(null);
```

This technique is a bit unusual and not seen very often. However, it lets you create a “clean” or “bare” object for use as a general key/value data structure.

The bare object doesn’t carry around a bunch of excess baggage in the form of unneeded properties and prototypes.

If you have bare objects in your program, you must remember that the usual Object properties and methods don’t exist on those objects. That’s why you sometimes see code like this:

```
if (Object.getPrototypeOf(obj) && obj.isPrototypeOf(car)) {

    // obj has a non-null prototype; and
    // obj is in the prototype chain of car

}
```

Summary

JS objects can inherit properties from other objects. The object that another object inherits properties from is its prototype.

In most cases, we use **Object.create** to create objects whose prototype we need to set explicitly. We can also use **Object.setPrototypeOf** to set the prototype of an object that already exists.

By default, all object literals inherit from **Object.prototype**.

When you access a property on an object, JS looks for the property first in the object, then its prototype chain, all the way up to **Object.prototype**.

Function Expressions

We can call a function that we defined through a function declaration before the declaration in the program. This is due to the fact that the JS engine runs our code in two passes.

During the first pass, it does some preparatory work, while the second executes the code. One action that occurs during the first pass is called **hoisting**; the engine effectively moves function declarations to the top of the program file in which they're defined, or the top of the function in which they are nested.

Hoisting is an internal step performed by the engine; it doesn't move any code around.

Function definitions that are the first thing on a line are known as **function declarations**. On the other hand, **function expressions** are function definitions that are part of an expression.

You can't call a function expression until after the expression is evaluated.

Typically, we assign a function expression to a variable or object property, pass it to another function, or return it to a calling function.

We can define function expressions without giving them a name. Such unnamed functions are called **anonymous functions**.

Function expressions don't have to be anonymous. You can name a function expression:

```
let squaredNums = [1,2,3].map(function squareNum(num) {

    return num * num;

});
```

The main advanatage of naming a function expression occurs when the function throws an error (i.e., raises an exception). If the function has a name, the stack trace uses that name to help you determine where the error occurred. Without the name, JS merely reports the location as "anonymous".

The function name given to a function expression is **not visible** in the scope that includes the function expression.

The function name on a function expression is visible inside the function, which is useful when working with recursive functions.

There's no declaration syntax for arrow functions; arrow functions are always function expressions. Arrow functions are always anonymous: there's no way to define a named arrow function. Arrow functions are either immediately invoked, assigned to variables or properties, or passed around as arguments and return values.

First-Class Functions

Functions of all kinds, including declared functions, can be treated as values. We can do the following:

```
function logNum(num) {
    console.log('Number: ' + num);
}
```

[1,2,3].forEach(logNum);

Notice that we don't use the invocation syntax, **( )**, when passing **logNum** as an argument to **forEach**. If we did, it would throw a **TypeError** exception since **forEach** expects a function; instead of passing a function, though, we would be passing **undefined**, the return value of **logNum()**.

The takeaway is that you should not invoke functions when you want to use them as values. Use invocation only when you need to run the code in the function.

You can treat any function as you would any other JS value: remove the invocation syntax, and you've got an expression whose value is a function. You can do whatever you want with that value (at least within the limtis of what JS can do).

Type of a Function Value

Functions are of the type **function**. Functions are objects: they are a compound type that has its own properties and methods.

Higher Order Functions

The fact that JS treats functions as values means that we can have a special kind of function in our programs: a **higher-order function**. A higher-order function is a function that has at least one of the following properties:

- It takes a function as an argument.
- It returns a function.

You can think of a function that returns another function as a function factory: it creates and returns a new function.

Typically, the function factory uses the arguments you pass to it to determine the specific job performed by the function it returns.

Higher-order functions let the programmer use powerful and flexible abstractions.

The Global Object

JS creates a global object when it starts running. It serves as the **implicit execution context** for function invocations.

The global object is available everywhere in your program and houses important global properties. We talked about global values such as **Infinity** and **NaN**, and global functions, such as **isNaN** and **parseInt**. All these entities are properties of the global object.

As with other JS objects, you can add properties to the global object at any time.

The global object has an interesting property: whenever you assign a value to a variable without using the **let**, **const**, or **var** keywords, the variable gets added to the global object as a property. You can even access such variables without using the global object as the caller.

Whenever you try to access a variable for which there are no local or global variables with the variable's name, JS looks at the global object and looks for a property with that name.

Implicit and Explicit Execution Context

Execution Context

The execution context -- or **context** -- is a concept that refers to the **environment** in which a function executes. In JS, it most commonly refers to the current value of the **this** keyword.

When we talk about the execution context of a function or method call, we're talking about the value of **this** when that code executes. The context depends on how the function or method was invoked, not on where the function was defined.

The only factor that determines the context is how you call the function or method.

There are two basic ways to set the context when calling a function or method:

- **Explicit**: The execution context that you set explicitly.
- **Implicit**: The execution context that JS sets implicitly when your code doesn't provide an explicit context.

Setting the execution context is also called **binding this** or **setting the binding**. A binding is something that ties two things together. In this case, it refers to the fact that a call binds **this** to a specific object when the function or method is called.

Function Execution Context (Implicit)

Every JS function call has an execution context. In other words, the **this** keyword is available to every function in your JS program. Every time you call that function, JS binds some object to **this**.

Within a regular function call, JS sets the binding for **this** to the global object. Remember that in Node, the global object is called **global**; in a browser, it is **window**.

Since all function calls have an execution context, and since a regular function call does not provide an explicit context, JS supplies an implicit context: the global object. We say that this execution context is implicit since the function invocation doesn't supply an explicit alternative.

Strict Mode and Implicit Context

When strict mode is enabled, the implicit **this** is assigned to **undefined** instead of the global object.

Method Execution Context (Implicit)

When you call a method that belongs to an object, the execution context inside that method call is the object used to call the method. We call that **method execution context**.

Method execution syntax is usually said to provide an implicit context; we're using an explicit object to call the method, but JS is interpreting that object as the implicit context. For this reason, we usually say that method calls provide an implicit execution context.

Be careful, however. The first-class nature of JS functions has ramifications for the execution context. Remember that the context is determined solely by how you call the function or method.

Explicit Function and Method Execution Context

Using parenthesis after a function or method name is not the only way to invoke it.

You can provide an explicit context to any function or method, and it doesn't have to be the global object or the object that contains the method. Instead, you can use any object -- or even **null** -- as the execution context for any function or method.

There are two main ways to do that in JS: **call** and **apply**.

Explicit Execution Context with call

JS functions are objects: they have properties and methods just like any other object. One method that all JS functions have is the **call** method. The **call** method calls a function with an explicit execution context.

For example:

```
function logNum() {
    console.log(this.num);
}

let obj = {
    num: 42
};

logNum.call(obj); // logs 42
```

The first argument to **call** provides the explicit context for the function invocation. We don't mutate the object when we use **call**.

You can also use **call** to explicitly set execution context on methods, not just functions:

```
let obj1 = {
        logNum() {
                console.log(this.num);
        }
};

let obj2 = {
        num: 42
};

obj1.logNum.call(obj2); // logs 42
```

The **call** method lets us pass arguments as a comma-separated list to our function.

The general syntax for **call** is as follows:

```
someObject.someMethod.call(context, arg1, arg2, arg3, ...)
```

Explicit Execution Context with **apply**

The **apply** method works in much the same way as **call**. The only difference is that **apply** uses an array to pass any arguments to the function. Here's the general syntax:

```
someObject.someMethod.apply(context, [arg1, arg2, arg3, ...])
```

apply is handy when you have the list of arguments in an array. However, you can also use **call** in conjunction with the spread operator to accomplish the same thing:

```
let args = [arg1, arg2, arg3];
someObject.someMethod.call(context, ...args);
```

Summary

It's important to remember that the rules for **this** are entirely different from the rules for variable scope. While a variable's scope is determined by where you write code, **this** depends on how you invoke it.

Hard Binding Functions with Contexts

JS has a third way to specify the context: the **bind** method on function objects.

bind works a little differently, however. For example:

```
function sumNum(num1) {
        return this.num + num1;
}

let obj = {
        num: 42
};

let sumNum2 = sumNum.bind(obj);
sumNum2(5); // logs 47
```

bind returns a new function. The new function is **permanently** bound to the object passed as **bind**'s first argument. You can then pass that method around and call it without worrying about losing its context since it's permanently bound to the provided object.

An interesting and important property of permanently bound functions is that you cannot change their execution context, even if you use **call** or **apply** or call **bind** a second time.

bind's context is the original function, and it returns a new function that calls the original function with the context supplied to bind as its first argument.

With **bind**, the original function isn't changed and doesn't have its context changed.

Unlike **call** and **apply**, **bind** doesn't invoke the function used to call it. Instead, it returns a new function that is permanently bound to the context argument.

Dealing with Context Loss I

Functions and methods can "lose context". We've used quotes since functions don't lose their execution context in reality -- they always have one, but it may not be the context that you expect.

We'll study what happens when a method is copied out of an object and used elsewhere.

Method Copied from Object

Methods like **forEach**, **map**, and **filter**, take a callback function as an argument and an optional **thisArg** context object that gets used as the callback's execution context.

However, it's not always possible to pass a context argument to a function or method; you may not even be able to change the function if, say, it belongs to a third-party library. Besides, it's generally not a good idea to pass a lot of arguments to your functions; the more arguments a function can accept, the harder the function is to use.

Another approach you can use is to hard-bind the method's context using **bind**.

bind has one significant advantage: once you bind a context to a function, that binding is permanent and does not need to be repeated if it gets called more than once.

The disadvantage of **bind** is that it is no longer possible to determine the context by looking at the invocation of the final function.

Loss of surrounding context is a common issue when dealing with functions nested within object methods.

Dealing with Context Loss II

We'll see how nested functions suffer from context loss.

Inner Function Not Using the Surrounding Context

```
let obj = {
        a: 'hello',
        b: 'world',
        foo: function () {
                function bar() {
                        console.log(this.a + ' ' + this.b);
                }
                bar();
        },
};

obj.foo();
```

Here, **bar** is invoked as a standalone function. Thus, its execution context is the global object, not the **obj** object.

Solution 1: Preserve Context with a Variable in Outer Scope

One common way to preserve the context in this scenario is to use something like **let self = this** or **let that = this** in the outer function.

If you define the **self** or **that** variable -- these names are idiomatic, and not a required name -- in the outer scope, you can use that variable and whatever value it contains inside your nested inner function(s).

```
let obj = {
        a: 'hello',
        b: 'world',
        foo: function () {
                let self = this;
                function bar() {
                        console.log(self.a + ' ' + self.b);
                }
                bar();
        },
};

obj.foo();
```

We assign **this** to the local variable **self**. Since JS uses lexical scoping rules for variables, **bar** can access **self** within its body; that lets us use it instead of **this** to access the correct context object.

Solution 2: Call Inner Function with Explicit Context

You can also use **call** or **apply** to explicitly provide a context when calling the inner function (i.e., **bar.call(this)**).

Solution 3: Use **bind**

A third approach is to call **bind** on the inner function and get a new function with its execution context permanently set to the object. For example,

```
let bar = function() {
        console.log(this.a + ' ' + this.b);
        }.bind(this);
```

One advantage of **bind** is that you can do it once and then call it as often as needed without an explicit context.

Solution 4: Using an Arrow Function

Arrow functions have a property that comes in very handy when dealing with context loss; they inherit their execution context from the surrounding context. That means that an arrow function defined inside another function always has the same context as the outer function's context.

For example:

```
let bar = () => {
        console.log(this.a + ' ' + this.b);
};
```

Using arrow functions like this is similar to using **bind** in that you don't have to worry about arrow functions losing their surrounding context. An arrow function, once created, always has the same context as the function that surrounded it when it was created.

Using arrow functions is the most common solution these days.

Despite how useful arrow functions are for avoiding context loss, you should not try to use them as methods on an object. They don't work as expected.

Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

Dealing with Context Loss III

We'll see how passing functions as arguments can strip them of their intended context objects.

Function as Argument Losing Surrounding Context

```
let obj = {
        a: 'hello',
        b: 'world',
        foo: function() {
                [1,2,3].forEach(function(number) {
                        console.log(String(number) + ' ' + this.a + ' ' + this.b);
                });
        },
};

obj.foo();
```

The problem is that **forEach** executes the function expression passed to it, so it gets executed with the global object as context.

Solution 1: Preserve the Context with a Variable in Outer Scope

```
let self = this;
```

Solution 2: Use bind

```
[1,2,3].forEach(function(number) {
        console.log(String(number) + ' ' + this.a + ' ' + this.b);
}.bind(this));
```

Solution 3: Use an Arrow Function

```
…forEach(number => …
```

Solution 4: Use the Optional thisArg Argument

Some methods that take function arguments allow an optional argument that specifies the context to use when invoking the function.

Array.prototype.forEach, for instance, has an optional **thisArg** argument for the context.

```
[1,2,3].forEach(function(number) {
        console.log(String(number) + ' ' + this.a + ' ' + this.b);
}, this);
```

The array methods **map**, **every**, and **some** and others also take an optional **thisArg** argument.

Summary

Passing a function as an argument to another function strips it of its execution context, which means the function argument gets invoked with the context set to the global object.

This problem is identical to the problem with copying a method from an object and using it as a bare function.

Summary

- Every object has an internal **[[Prototype]]** property that points to a special object, the object's prototype. It is used to look up properties that don't exist on the object itself.
    - **Object.create** returns a new object with the passed-in argument as its prototype.
    - You can use **Object.getPrototypeOf** and **obj.isPrototypeOf** to check for prototype relationships between objects.
- Looking up a property in the prototype chain is the basis for prototypal inheritance, or property sharing through the prototype chain. Objects lower down in the chain inherit properties and behaviours from objects in the chain above. That is, downstream objects can delegate properties or behaviours to upstream objects.

    - A downstream object overrides an inherited property if it has a property with the same name. Overriding is similar to shadowing, but it doesn't completely hide the overhidden properties.
    - **Object.getOwnPropertyNames** and **obj.hasOwnProperty** can be used to test whether an object owns a given property.

- Function invocations (i.e., **parseInt(numberString)**) rely upon implicit execution context that resolves to the global object. Method invocations (i.e., **array.forEach(processElement)**) rely upon implicit execution context that resolves to the object that holds the method.

- All JS code executes within a context. The top-level context is the **window** object in browsers and the **global** object in Node. All global methods and objects, such as **parseInt** and **Math**, are properties of **window** or **global**.

- The value of **this** is the current execution context of a function or method.

- The value of **this** changes based on how you invoke a function, not how you define it.

- JS has first-class functions that have the following characteristics:

    - You can add them to objects and execute them in the respective object's context.
    - You can remove them from their objects, pass them around, and execute them in entirely different contexts.
    - They're initially unbound but dynamically bound to a context object at execution time.

- The **call** and **apply** methods invoke a function with an explicit execution context.

- The **bind** method returns a new function that permanently binds a function to a context.

- Arrow functions are permanently bound to the execution context of the enclosing function invocation. When defined at the top level, the context of an arrow function is the global object.

Quiz 1:

#1:

Object.create doesn't create a copy of the object it gets passed as an argument. Instead, it creates a new object whose **[[Prototype]]** property references the argument.

#3:

The **[[Prototype]]** property is a hidden internal property that you cannot access directly, no matter how you try to use it as a property name.

#7:

If you use braces around the body of an arrow function, you must use an explicit **return** statement to provide a return value that isn't **undefined**.

#11:

Function declarations always require a name for the function, so they can never be anonymous.

Quiz 2:

#1:

The global object is available everywhere in a JS program, including both the top level and inside other functions and methods. If you don't provide an explicit execution context, JS uses the global object as the value for **this**, or the calling object for method call syntax. However, you can access the global object anywhere merely by using its name (**global** or **window**).

The global object is used as the implicit execution context when invoking a function with function call syntax. With method call syntax, JS uses the calling object as the implicit execution context.

Lesson 3: Object Creation Patterns

Review - OOP Principles: Encapsulation

Objects provide a means to group related data and functions into one unit. In the context of the object, the data and functions are commonly called state and methods respectively.

This grouping together of related data and functions is what's called encapsulation.

OOP organizes code into logical units.

Review - Factory Functions

Object factories, or factory functions (also called the Factory Object Creating Pattern), provide a simple way to create related objects based on a predefined template.

The factory function lets us create multiple objects of the same "type" with a pre-defined "template". However, it has some disadvantages:

- Every object created with a factory function has a full copy of all the methods. That's redundant, and it can place a heavy load on system memory.

- There is no way to inspect an object and learn whether we created it with a factory function. That effectively makes it impossible to identify the specific "type" of the object; at best, you can only determine that an object has some specific characteristics.

Constructors

Object constructors, or **constructors** for short, are another way to create objects in JS. You can think of a constructor as a little factory that can create an endless number of objects of the same type.

Superficially, a constructor looks and acts a lot like a factory function: you define a constructor once then invoke it each time you want to create an object of that type.

Defining a constructor is nearly identical to defining an ordinary function, but there are differences. Capitalizing the name isn't a language requirement, but it is a convention employed by most JS developers.

Calling a Constructor Function

The most striking features that distinguish constructors from ordinary functions are that:

- we call it with the **new** keyword -- **let corolla = new Car('Toyota', 'Corolla', 2016)**;
- we use **this** to set the object's properties and methods; and
- we don't supply an explicit return value (we can, but usually we don't).

this always refers to an object. Its value depends on how we call the function. Calling constructors is where you see the most significant difference between them and other functions.

The combination of using **new** with a function call treats the function as a constructor.

JS takes the following steps when you invoke a function with **new**:

1. It creates an entirely new object.

2. It sets the prototype of the new object to the object that is referenced by the constructor's **prototype** property.

3. It sets the value of **this** for use inside the function to point to the new object.

4. It invokes the function. Since **this** refers to the new object, we use it within the function to set the object's properties and methods.

5. Finally, once the function finishes running, **new** returns the new object even though we don't explicitly return anything.

If you don't use the **new** keyword, the constructor function won't work as intended. Instead, it acts like an ordinary function. In particular, no new objects are created, so **this** won't point to a new object.

Furthemore, since functions that don't return an explicit value return **undefined**, calling a constructor without **new** also returns **undefined**. When you use **new**, however, the function doesn't have to return anything explicitly: it returns the newly created object automatically.

Who Can be a Constructor

You can use **new** to call almost any JS function that you create. However, you cannot call arrow functions with **new** since they use their surrounding context as the value of **this**.

You can also use **new** on methods that you define in objects. However, calling a method defined with concise syntax, also called concise method, won't work.

In addition, many -- but not all -- built-in objects and methods are incompatible with **new**.

new is also incompatible with special functions known as **generators** (not covered at Launch School).

Constructors With Explicit Return Values

A constructor that explicitly tries to return an object returns that object instead of a new object of the desired type.

In all other situations, it returns the newly created object, provided no errors occur.

In particular, the constructor ignores primitive return values and returns the new object instead.

Supplying Constructor Arguments with Plain Objects

Constructor functions sometimes have to grow with the needs of a program. That often means adding more arguments to the constructor.

The more parameters a function needs, the harder it is to read the code and the more likely that problems will arise.

One common technique that we can use to manage our parameters better involves passing them to our constructor with an object argument.

Determining an Object's Type

JS treats objects and their types in a looser, more dynamic way. You can't determine the specific type of arbitrary JS objects; they are dynamic sturctures with a type of **object**, no matter what properties and methods they have. However, we can get some useful information about an object if we know which constructor created it.

Remember that the **new** operator creates a new object. Suppose that you call the **Car** constructor with **new**. Informally, we can say that the resulting object is a **car**. More formally, we can say that the object is an **instance** of a **Car**.

The **instanceof** operator lets us determine whether a given constructor created an object (i.e., **civic instanceof Car**).

new and Implicit Execution Context

Now that we know about constructors, we can add a constructor call with **new** as a third way to provide an implicit execution context. When you call a function with **new**, its implicit context is the new object.

Problems

#1:

Constructor function names are capitalized.

Constructors With Prototypes

Method Delegation to Prototypes

We learned that we can use prototypes to share code between objects of the same type.

Prototypes are especially useful for sharing methods as all objects of a particular type share the same prototype object.

Furthermore, delegation means that we can share methods by putting them in the prototype object; if an object doesn't contain a requested method, JS searches the prototype chain to find the method.

Thus, we can define a method once in the prototype object, and let the inheriting objects delegate the method calls to the prototype.

We can use prototypes in conjunction with constructors to achieve the same result:

```
let DogPrototype = {
        bark() {
                console.log(this.weight > 20 ? 'Woof!' : 'Yip!');
        }
};

function Dog(name, breed, weight) {
        Object.setPrototypeOf(this, DogPrototype);
        this.name = name;
        this.breed = breed;
        this.weight = weight;
}
```

We can also assign the prototype object to a property of the constructor as follows:

```
function Dog(name, breed, weight) {
        Object.setPrototypeOf(this, Dog.myPrototype);
        this.name = name;
        this.breed = breed;
        this.weight = weight;
}

Dog.myPrototype = {
        bark() {
                console.log(this.weight > 20 ? 'Woof!' : 'Yip!');
        }
};
```

Since JS functions are objects, we can add properties to them. Here, we assign the prototype object to a **myPrototype** property on the **Dog** function object. Thus, we clearly show our intent that all dogs inherit from the **Dog.myPrototype** object.

The Constructor **prototype** Property

What makes constructors special is a characteristic of all function objects in JS: they all have a **prototype** property that we call the **function prototype** to distinguish them from the prototypes used when creating ordinary objects.

When you call a function **Foo** with the **new** keyword, JS sets the new object's prototype to the current value of **Foo**'s **prototype** property. That is, the constructor creates an object that inherits from the constructor function's prototype (**Foo.prototype**). Since inheritance in JS uses prototypes, we can also say that the constructor creates an object whose prototype references **Foo.prototype**.

We use the term "prototype" to refer to 2 distinct but related concepts:

- If **bar** is an object, then the object from which **bar** inherits is the **object prototype**. By default, constructor functions set the object prototype for the objects they create to the constructor's prototype object.

- The **constructor's prototype object**, also called the **function prototype**, is an object that the constructor uses as the object prototype for the objects it creates. In other words, each object that the constructor creates inherits from the constructor's prototype object. The constructor stores its prototype object in its **prototype** property; that is, if the constructor's name is **Foo**, then **Foo.prototype** references the constructor's prototype object.

In most cases, when we talk about a **prototype** without being more explicit, we mean an **object prototype**. We'll talk about the constructor's prototype, the function prototype, or the **prototype** property when talking about a constructor's prototype object.

Note that constructors don't inherit from the constructor's prototype object. Instead, the objects that the constructor creates inherit from it.

Every JS function has a **prototype** property. However, JS only uses it when you call that function as a constructor, that is, by using the **new** keyword.

JS takes the following steps when you invoke a function with **new**:

1. It creates an entirely new object.

2. It sets **Foo.prototype** as the prototype for the new object. That is, the new object inherits from the object referenced by **Foo.prototype**.

3. It sets the execution context (**this**) for the function to point to the new object.

4. It invokes the function.

5. It returns the new object unless the function returns another **object**.

When you call a method on an object, JS binds **this** to the object whose method you used to call it. If it doesn't find the method in that object, but it does find it in the prototype, that doesn't change the value of **this**. **this** always refers to the original object -- that is, the object used to call the method -- even if the method is in the prototype.

A property of interest on a prototype object is the **constructor** property (i.e., **Dog.prototype.constructor;**).

As with the **instanceof** operator, the **constructor** property lets us determine the type of an object (i.e., **maxi.constructor === Dog**).

Be careful, however. It's possible to reassign the **constructor** property to something else.

Overriding the Prototype

Inheriting methods from a prototype doesn't mean that the inheriting object is stuck with those methods. JS objects are incredibly dynamic and flexible.

Two objects created with the same constructor may end up looking completely different from each other because of changes and additions made after constructing the object.

Practice Problems - Constructors and Prototypes

#7:

An object created by a constructor function has a **constructor** property that points back to its constructor.

#8:

Constructor functions built in a way to check whether the function was called with the **new** keyword are called **scope-safe constructors**. Most, but no all, of JS's built-in constructors, such as **Object**, **RegEx**, and **Array**, are scope-safe. **String** is not.

Static and Instance Properties and Methods

In OOP, we often refer to individual objects of a specific data type as **instances** of that type. An instance is just another term for the objects created using any means of defining multiple objects of the same kind.

Instance Properties

It's convenient to describe the properties of an instance as **instance properties**. These properties belong to a specific instance of some type.

They aren't properties of the constructor, they are properties of the instances created by the constructor.

Instance Methods

Since methods are also properties on an object, we can refer to methods stored directly in an object as instance properties too. More commonly, we call them **instance methods** just to distinguish them from ordinary data properties.

However, methods usually aren't stored directly in instances. Instead, they are usually defined in the object's prototype object.

While methods defined in the prototype object aren't stored in the instance object, they still operate on individual instances. Therefore, we usually refer to them as instance methods.

Any method defined in any prototype in the prototype chain of an object is considered to be an instance method of the object.

Static Properties

Static properties are defined and accessed directly on the constructor, not on an instance or a prototype. Typically, static properties belong to the type rather than to the individual instances or the prototype object.

One common use of static properties is to keep track of all of the objects created by a constructor. For instance:

```
function Dog(name, breed, weight) {
        this.name = name;
        this.breed = breed;
        this.weight = weight;
        Dog.allDogs.push(this);
}

Dog.allDogs = [ ];
```

allDogs is information about dogs in general. Therefore, it should be a static property.

Static Methods

Static properties don't have to be ordinary data properties. You can also define static methods.

Object.assign, **Array.isArray**, and **Date.now** are all examples of static methods.

Built-In Constructors

JS comes with a variety of built-in constructors and prototypes that let you instantiate useful objects. These constructors work like constructors for other objects; they're used with the **new** keyword to create objects.

The **Array** Constructor

We can create arrays using the **Array** constructor (i.e., **let emptyArray = new Array(1,2,3,4)**).

When we provide a single number argument, the constructor creates an array with a length equal to the number specified by the argument, but with no actual arguments. Note that the single-number form of the constructor does not accept non-integers or negative numbers.

The single-number constructor, together with the **Array.prototype.fill** method, lets you create arrays with a value that is repeated in every element (i.e., **(new Array(3)).fill('*')**).

The **fill** method takes any value as an argument and replaces all elements of the array with that value.

Array lets you omit the **new** keyword.

As with any JS function, the **Array** constructor has a **prototype** property. All arrays inherit from the object referenced by this property. This relationship implies that every array can use the methods defined in **Array.prototype**.

The instance methods for the Array type are labeled **Array.prototype.aMethodName**.

The array type also has several static methods. Static methods belong directly to the constructor function; they aren't part of the prototype used to create new objects.

The **Array.isArray** method takes one argument and returns **true** if the argument is an array object, and **false** if it is not. Programs often use this method to verify or refute that a given value is an array object.

The **Array.from** method takes an **array-like object** as an argument and returns a new array with the equivalent element values. An array-like object is any object that has a **length** property and provides indexed access to some of its properties with the **[index]** notation. Such objects have properties whose keys are non-negative integers.

The Object Constructor

new Object()

All objects created by the **Object** constructor or with object literal syntax, inherit from **Object.prototype**. Thus, all such objects have access to the instance methods defined in **Object.prototype**.

Since arrays are a subtype of objects, all array objects have access to all the methods on **Object.prototype**.

Almost all JS objects, whether built-in or custom-created, inherit from **Object.prototype**, either directly or further down the prototype chain.

Commonly used static **Object** methods: **Object.assign**, **Object.create**, **Object.entries**, **Object.freeze**, **Object.isFrozen**, **Object.keys**, **Object.values**.

The Date Constructor

The **Date** constructor creates objects, commonly called a **date object**, that represent a specific date and time. Calling **Date** without arguments returns a date object that represents the creation date and time of the object (i.e., **let now = new Date()**).

You can create date objects that represent any given date and time by passing additional arguments to the **Date** constructor.

The **Date.prototype.toString** method returns a string that represents the date.

The **Date.prototype.getFullYear** method returns the year from the date as a number.

The **Date.prototype.getDay** method returns a number that represents the day of the week that corresponds to the date object.

The String Constructor

JS has two kinds of strings: string primitives and **String** objects.

Equality for objects works by identity. Two objects are strictly equal only when they are the same object.

When you try to access a property or invoke a method on a string primitive, JS wraps the string primitive in a **String** object behind the scenes.

Once JS has wrapped your string primitive in a **String** object, it then uses the object to access the property or call the method. When the wrapping object has served its purpose, JS discards it.

As a general rule, you should not create **String** objects explicitly. However, if you're writing code where you may have to operate on **String** objects, you can use **String.prototype.valueOf()** to retrieve the value of the **String** object as a primitive.

In the case of **String**, it simply returns a new string, not an object, when you omit the **new** keyword.

The **String** function takes non-string values as arguments as well. In that case, it converts the value to a string.

The Number and Boolean Constructors

The **Number** and **Boolean** constructors work in much the same way as the **String** constructor. When called with **new**, they create **Number** and **Boolean** objects. When called without **new**, the **Number** function converts its arguments to a number, and the **Boolean** function converts its argument to a boolean.

As with strings, numbers and booleans both have primitive and object forms, and JS invisibly wraps primitives in objects to access methods and properties. You should also avoid creating **Number** and **Boolean** objects explicitly.

Borrowing Array Methods for Strings

A significant benefit of first-class functions is that methods aren't tied to a particular object type. Using explicit context-binding techniques, we can apply a method to a different object type than the one that defines the method.

For example:

```
let string = 'EEE';
[ ].every.call(string, char => char === 'E'); // returns true
```

Classes

In effect, classes act like **syntactic sugar** -- syntax designed to be easier to read or use -- that makes it easier for programmers to migrate to JS from other OOP languages.

Both functions and classes have two significant definition styles: declarations and expressions.

Class Declarations

The simplest way to create classes in JS is with the **class declaration**, which looks similar to classes in other languages.

```
class Rectangle {
        constructor(length, width) {
                this.length = length;
                this.width = width;
        }

        getArea() {
                return this.length * this.width;
        }
}

let rec = new Rectangle(10,5);
console.log(typeof Rectangle); // function
console.log(rec instanceof Rectangle); //true
console.log(rec.constructor); // [class Rectangle]
console.log(rec.getArea()); // 50
```

Class declarations begin with the **class** keyword, followed by the name of the class. The rest of the syntax looks similar to the simplified (concise) method definition that you can use in object literals. However, there are no commas between the properties of the class.

One significant difference is that the constructor is now a method named **constructor** inside our class instead of being a standalone function. Other methods have no special meaning; you can define as many as you need. In this case, we define **getArea**, and it gets placed in **Rectangle.prototype**.

You **must** use the **new** keyword to call the constructor when using a **class**. JS raises a **TypeError** if you try to call the constructor without the **new** keyword.

Class Expressions

```
let Rectangle = class {
        constructor(length, width) {
                this.length = length;
                this.width = width;
        }

        getArea() {
                return this.length * this.width;
        }
};
```

Aside from the syntax, class expressions are functionally equivalent to class declarations.

Classes as First-Class Citizens

In programming, a **first-class citizen** is a value that can be passed into a function, returned from a function, and assigned to a variable. Like JS functions, JS classes are also first-class values.

Classes are just functions. Since functions are first-class objects, classes must also be first-class objects.

Static Methods and Properties

You can define methods on your custom constructor methods. For example:

```
Rectangle.getDescription = function() {
        return 'A rectangle is a shape with 4 sides';
}
```

You can define static methods with the **class** keyword as well. We need to use the **static** keyword as follows:

```
class Rectangle {
        ...
        static getDescription() {
                return 'A rectangle is a shape with 4 sides';
        }
        ...
}
```

We can also define static properties. Static properties are properties that are defined on the constructor function isntead of the individual objects.

To define a static property with the constructor and prototype pattern, just add it to the constructor function object:

```
Rectangle.description = 'A rectangle is a shape with 4 sides';
```

To do the same thing with a **class**, just use the **static** keyword inside the **class**:

```
class Rectangle {
        static description = 'A rectangle is a shape with 4 sides';
}
```

The **static** modifier, when used with a method declaration, marks the method as static. That is, the method is defined directly on the class, not on the objects the class creates.

Summary

Every function has a **prototype** property that points to an object that contains a **constructor** property. The **constructor** property points back to the function itself. Thus, if **Kumquat** is a constructer function, then **Kumquat.prototype.constructor === Kumquat**.

If the function is used as a constructor, the returned object's **[[Prototype]]** will reference the constructor's **prototype** property. That lets us set properties on the constructor's prototype object so that all objects created by the constructor will share them. We call this the pseudo-classical pattern of object creation.

The pseudo-classical object creation pattern generates objects using a constructor function that defines state and a prototype object that defines shared behaviours.

The pseudo-classical inheritance pattern has types (i.e., classes) that inherit from other types. This way, all objects of a given type can share behaviours from the same source.

Quiz

#1:

The execution context is determined when a function or method is invoked, not by how it is defined. Therefore, factory functions don't set the execution context for the object's methods.

While it's not possible to directly determine the type of an object created by a factory function, that object still represents a specific type of data.

#3:

A constructor that attempts to return an object will return an object of that type.

A constructor that attempts to return a non-object value, such as a string, will instead return a new object of the type associated with the constructor.

A constructor that doesn't return an explicit value will return a new object of the type associated with the constructor.

#7:

The static method **Array.isArray** is useful because the **typeof** operator returns **object** when used with an array, so cannot be used to detect an array. It helps improve readability and show intent. It helps distinguish between arrays and other objects.

#8:

All objects that have a length property are Array-like objects.

#10:

A class declaration always begins with the keyword **class** at the beginning of a statement.

#13:

The **class** statement gets translated behind the scenes to a constructor function and a prototype object, and the class name refers to the constructor function.

Lesson 4: Subclassing and Code Reuse Patterns

Object Creation with Prototypes (OLOO)

Factory functions aren't the only way to create objects in bulk.

Another pattern that we can use is the **OLOO** pattern: **Objects Linking to Other Objects**. It uses prototypes and involves extracting properties common to all objects of the same type to a prototype object. All objects of the same type then inherit from that prototype.

The following is an example:

```
let carPrototype = {
        start: function() {
                this.started = true;
        },

        stop: function() {
                this.started = false;
        },

        init(make, model, year) {
                this.make = make;
                this.model = model;
                this.year = year;
                return this;
        },
};

let car1 = Object.create(carPrototype).init('Toyota', 'Corolla', 2016);
```

Advantage of OLOO over Factory Function

The OLOO pattern has one significant advantage over factory functions: memory efficiency.

Since all objects created with the OLOO pattern inherit methods from a single prototype object, the objects that inherit from that prototype object share the same methods.

Factory functions, on the other hand, create copies of all the methods for each new object. That can have a significant performance impact, especially on smaller devices with limited memory.

However, that doesn't mean that the OLOO is decidedly better than the factory pattern. An advantage of the factory pattern is that it lets us create objects with private state.

Any state stored in the body of the factory function instead of in the returned object is private to the returned object. They can't be accessed or modified unless one of the object methods exposes the state.

With OLOO, there is no way to define private state. All object state can be accessed and modified by outside code.

Subtyping with Constructors and Prototypes

The combination of constructors and prototypes gives us something that resembles a **class**.

We say that a square is a **sub-type** of a rectangle, or that a rectangle is a **super-type** of a square.

Since a function's **prototype** property is writable -- we can change what object it references.

If you reassign the **prototype** property of a constructor, the **prototype.constructor** property will no longer point to the constructor.

We can just reassign the **prototype.constructor** property to point to our constructor of choice.

Prototypal Inheritance vs. Pseudo-Classical Inheritance

The simpler form of inheritance is **prototypal inheritance** or **prototypal delegation**. We sometimes call this form of inheritance **object inheritance** since it works with one object at a time. An object's internal **[[Prototype]]** property points to another object, and the object can delegate method calls to that other object.

The object creation pattern called **pseudo-classical** object construction, also known as the **constructor/prototype pattern**, uses a constructor function and a prototype object to create objects and provide common methods for those objects. The term "pseudo-classical" refers to the fact that the pattern mimics classes from other OO languages but doesn't actually use classes.

This pattern forms the basis of **pseudo-classical inheritance**, also called **constructor inheritance**. In pseudo-classical inheritance, a constructor's prototype object (the object referenced by its **prototype** property) inherits from another constructor's prototype. That is, a sub-type inherits from a super-type.

Note that using **class** and the **extends** keyword is an alternative form of pseudo-classical inheritance. For example:

```
class Human {
	myName() { return this.name; }
	myAge() { return this.age; }
}

class Person extends Human {
	toString() {
		return `My name is ${this.myName()} and I'm ${this.myAge()} years old.`;
	}
}
```

Both pseudo-classical and prototypal inheritance use prototypal delegation under the hood. If the requested property isn't found, the object delegates the request to the object's prototype object.

Inheritance With Class Declarations

```
class Rectangle {
	constructor(length, width) {
		this.length = length;
		this.width = width;
	}

	getArea() {
		return this.length * this.width;
	}

	toString() {
		return `[Rectangle ${this.width * this.length}]`;
	}
}

class Square extends Rectangle {
	constructor(size) {
		super(size, size);
	}

	toString() {
		return `[Square ${this.width * this.length}]`;
	}
}
```

The **extends** keyword signifies that the class named to the left of **extends** should inherit from the class specified to the right of **extends**

When called inside the **constructor** method, the **super** keyword refers to the constructor method for the parent class (the class that we inherit from). Thus, **super(size, size)** performs the same role performed by this code from our constructor/ prototype example:

```
function Square() {
	Rectangle.call(this, size, size);
}
```

You don't need to use **super** in every subclass, but in most cases you do. In particular, if the superclass's constructor creates any object properties, you must call **super** to ensure that those properties are set properly. For instance, in the **Rectangle** class above, we create two properties in the **Rectangle** constructor, so we must call **super** in **Square**'s constructor.

If you do call **super** in a subclass's constructor, you must call it before you use **this** in that constructor.

Inheritance With Class Expressions

```
let Person = class {
	constructor(name, age) {
		this.name = name;
		this.age = age;
	}

	sayName() {
		console.log(`My name is ${this.name}.`);
	}
};

let Student = class extends Person {
	constructor(name, age, semester) {
		super(name, age);
		this.semester = semester;
	}

	enrollInCourse(courseNumber) {
		console.log(`${this.name} has enrolled in course ${courseNumber}.`};
	}
};

let student = new Student('Kim', 22, 'Fall');
```

Rewriting OO RPS with Constructors and Classes

We typically initialize the state of an object in its constructor and put the instance methods in the constructor's prototype.

Code Reuse with Mixins

One problem with inheritance in JS is that an object can have only one prototype object. We call this **single inheritance**.

Some programming languages allow classes to inherit from multiple classes, a functionality known as **multiple inheritance**. JS doesn't support multiple inheritance, so a class can only inherit from one class.

Enter JS mix-ins -- a pattern that adds methods and properties from one object to another. It's not delegation with prototypes; the mix-in pattern merely copies the properties of one object to another with **Object.assign** or some similar technique.

A **mix-in** is an object that defines one or more methods that can be "mixed in" to a class. This grants that class access to all of the methods in the object. It's the only real workaround for the lack of multiple inheritance short of duplication.

Some JS developers argue that you should use factory functions with mix-ins exclusively. They suggest that inheritance fails at modelling some scenarios, but a combination of factory functions and mix-ins can model any object relationship.

This approach is valid, but is suffers the downsides of all factory functions:

- Every new object receives a new copy of all of its methods, including new copies of both mix-in methods and the methods that belong directly to the object. That can be taxing on memory resources, even more so than the memory requirements of mix-ins.

- You can't determine the type of an object created with a factory function: the **instanceof** operator only recognizes these objects as instances of the **Object** type. This can have a significant impact on debugging.

We suggest a balance of mix-in and classical inheritance pattern instead:

- Inheritance works well when one object type is positively a sub-type of another object. Whenever two object types have an "**is a**" relationship, constructor or class inheritance makes sense.

- When you want to provide your objects with some capability, a mix-in may be the correct choice.

Constructors have a **name** property (i.e., **ClassName.prototype.constructor.name**) that merely contains the name of the class as a string.

Polymorphism

Polymorphism refers to the ability of objects with different types to respond in different ways to the same message or method invocation; that is, data of different types can respond to a common interface. It's a crucial concept in OOP that can lead to more maintanable code.

When two or more object types have a method with the same name, we can invoke that method with any of those objects. **When we don't care what type of object is calling the method, we're using polymorphism**.

There are two chief ways to implement polymorphism.

Polymorphism Through Inheritance

Two object types can respond to the same method call simply by **overriding** a method inherited from a superclass. Overriding is generally treated as an aspect of inheritance, so this is polymorphism through inheritance.

Polymorphism Through Duck Typing

Duck typing occurs when objects of different unrelated types both respond to the same method name. With duck typing, we aren't concerned with the class or type of an object, but we do care whether an object has a particular behaviour.

Specifically, duck typing is a form of polymorphism. As long as the objects involved use the same method name and take the same number of arguments, we can treat the object as belonging to a specific category of objects.

Note that merely having two different objects that have a method with the same name and compatible arguments doesn't mean that you have polymorphism.

In practice, polymorphic methods are intentionally desgined to be polymorphic; if there's no intention, you probably shouldn't use them polymorphically.

Summary

The **Objects Linking to Other Objects (OLOO)** parttern of object creation uses a prototype object, an initializer method, and the **Object.create** method to create objects with shared behaviour. The initializer customizes the state for each object, and is usually named **init**.

The combination of constructors and prototypes provides a way of mimicking classical inheritance with JS. This lets us create **sub-type** objects, which can 'inherit' methods from a **super-type** object. This is one way of facilitating code re-use.

There's a limitation with the inheritance pattern, which is that objects can only directly 'inherit' from one super-type object. In other words, an object can have only one prototype object. Mixins provide a way of addressing this limitation. The mix-in pattern involves creating a mix-in object containing certain methods, and using **Object.assign** to mix that object into another object.

Polymorphism refers to the ability of objects with different types to respond to the same method invocation. It can be implemented through inheritance by method overriding. It can also be implemented through **duck typing**; by ensuring that objects of different types use the same method name to perform different but related function.

Quiz:

#1:

The **instanceof** operator requires the object to the right to have a **prototype** property, such as a function object. In most cases, that means that the object on the right is a constructor function or class. It then tests to see if the **prototype** property of a constructor appears anywhere in the prototype chain of an object. The return value is a boolean value.

#9:

Object.assign with a single argument merely returns a reference to the argument.

Lesson 5: More Object Oriented Programming

An **orchestration engine** is a class that controls the flow of the application or some part of the application. It's common practice to make the orchestration engine the last class in a file, and to give it a name that is likely to be unique.

Writing a **spike** is writing some exploratory code to help you begin sketching out your program's structure and design. Spikes, in general, look similar to pseudocode in their general outline, but they more closely resemble the final code. In fact, it is code, and some of it may not change.