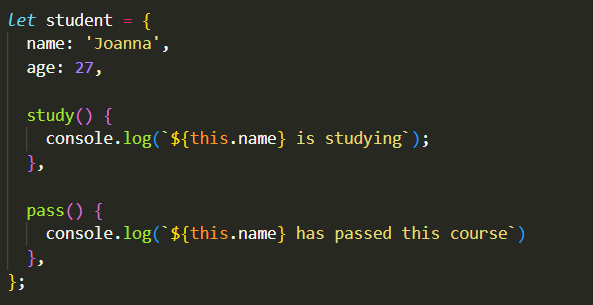
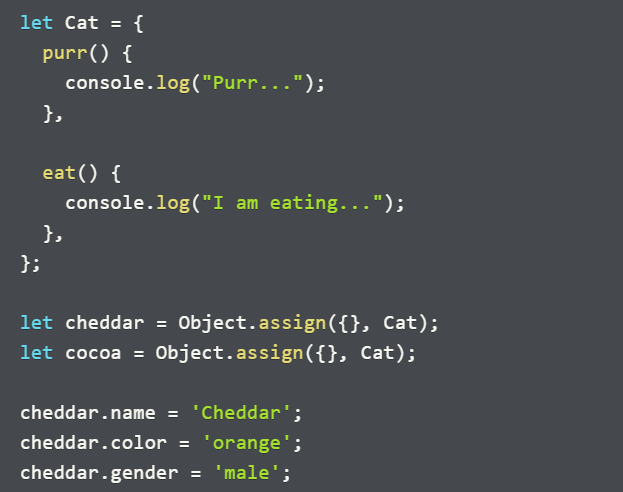
* JS implements OOP using *prototypes* (mainly) which is different from class based OOP in most other languages. This prototypal inheritance is more flexible and less restrictive than the class based OOP. Since ES 2015 class syntax has been introduced in JS making OOP more homologous with other languages.
* OOP paradigm lets programmers create complex software as a collection of objects that encapsulate data and behavior. These objects communicate and interact with one another. This simplifies the design, development and maintenance of complex systems.

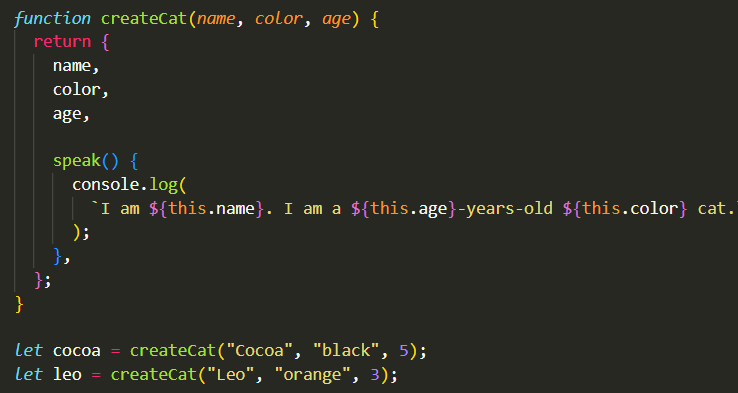
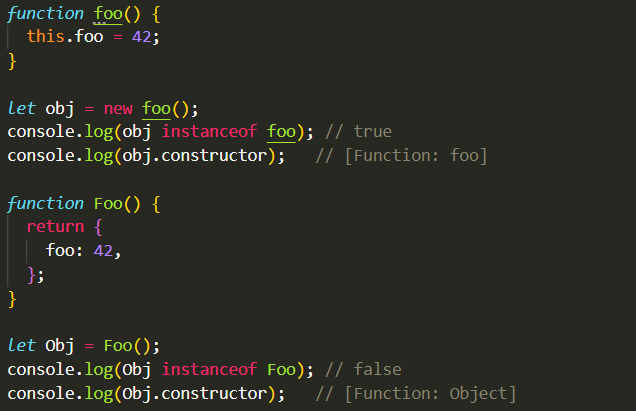
# Types and Objects

1. Constructor function: Objects, also called instances/instance objects are created by constructor function. Constructor functions:  
   
   1. Store both properties and methods
   2. Are called by new keyword (without it, it returns as a normal function)
   3. Can create as many objects/instances as you want
2. Using objects to store data and properties:
   1. Defining one object for each instance:  
      
   2. Defining a common object for multiple instances:  
        
      Even though cheddar and cocoa are two different objects, with different properties, they are of the same type and hence have the same behavior.

## Inheritance

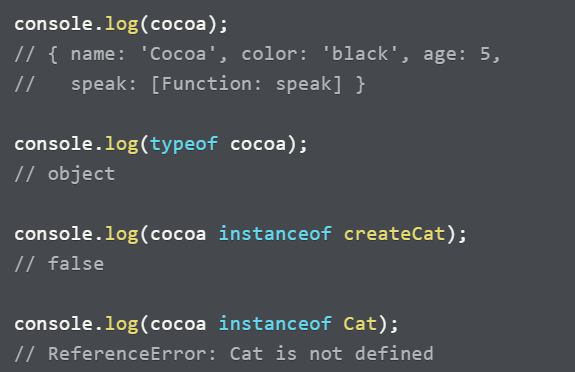
1. Since JS doesn’t support traditional classes, it also doesn’t support traditional inheritance. Instead, it supports something called prototypal inheritance.
2. In prototypal inheritance, each object in JS links to a prototype object, from which it inherits behavior. This prototype object in turn inherits additional behavior from other prototype objects.

# Object Factory

1. Object factories are functions that let you create multiple instances of similar objects without defining a specific type.
2. It’s a function that returns a new object, everytime it is called (optionally with specified properties).  
   
3. Object factories offer a functional alternative to more complicated object creation approaches such as constructor functions and classes.
4. Advantages of object factories:
   1. A simple functional approach to object creation.
   2. Creates highly customized objects with different properties (based on arguments passed) and methods. This can be useful for applications that want to generate objects dynamically.
   3. When inheritance is needed, you have a much straight forward way to create objects without dealing with the complexities of prototype chains.
   4. Bypasses the complexities of Execution Context with this keyword. The function is returning an object that has everything built into it and there is no dependencies on the E.C.
5. Dis-Advantages of object factories (O.F):
   1. Each object created, begets *copies* of the methods defined by the returned object. Thus factory functions can end up using too much memory.
   2. Implementing inheritance is more cumbersome and less performance efficient. Especially when, you need to manually copy the methods from one object type to another for the purpose of inheritance.
   3. Objects created by O.F do not have a *type*, hence you cannot use instanceof operator or constructor property to determine the type of an object (unlike in constructor functions).   
      

# Classes

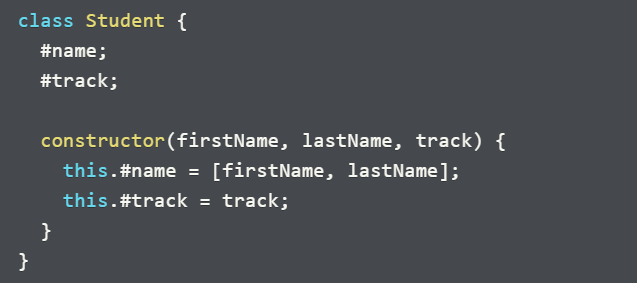
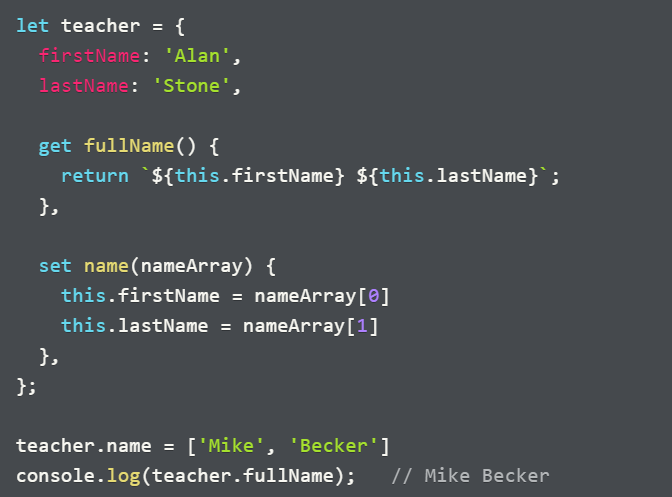
## Defining Classes

1. Class syntax makes it easier to use and understand OO in JS. However it is not a new OO inheritance model, it is only a syntactic sugar on top of the already existing prototype system.
2. Limitation of object factories: If you come across an object instance defined by OF in some other part of the program, it is not possible to determine its type. Ex: A cocoa instance for Cat class, made by createCat O.F:
3. Apart from this, making many instances of the same class require copying the methods (memory issue) and implementation of inheritance is also cumbersome. Classes solves all of these problems.
4. Example of a cat class:  
   
5. Most classes require a constructor method. It initializes all the properties of an object instance. **Note:** You must use this to reference the property names within constructor as well as to access the property in any of the methods that you define within the class.
6. The new keyword tells JS to instantiate a new Cat object by calling its constructor method. The new keyword:
   1. Creates a new empty object
   2. Sets the value of this in constructor to reference the new object.
   3. Calls the constructor with the specified argument.
   4. Finally, JS returns the completed object to the caller when the constructor finishes running.

## Inheritance

1. A way to form classes using existing class.
   1. Existing class is also called: Super Class, Base Class or Parent Class
   2. Inheriting class is also called: Subclass, derived class or child class
2. If we want to make a class for lion. Since lion is a cat, we can make it a subclass of Cat. Whenever there is “is a” relationship, it is appropriate to use inheritance.  
   
3. The extends keyword tells JS that the current class will inherit from the class named after extends. Objects of class lion will be able to access and use all the properties and methods of Cat class.
4. The first thing you need to do in the inheriting class’s constructor method is to call super(); super tells JS that it needs to call the same constructor method from the superclass.
5. We can also add more properties in the subclass’s constructor.
6. You can also use super() to call a method from the superclass within the subclass method. Ex: super.parentMethod()

# Non-ordinary properties and methods

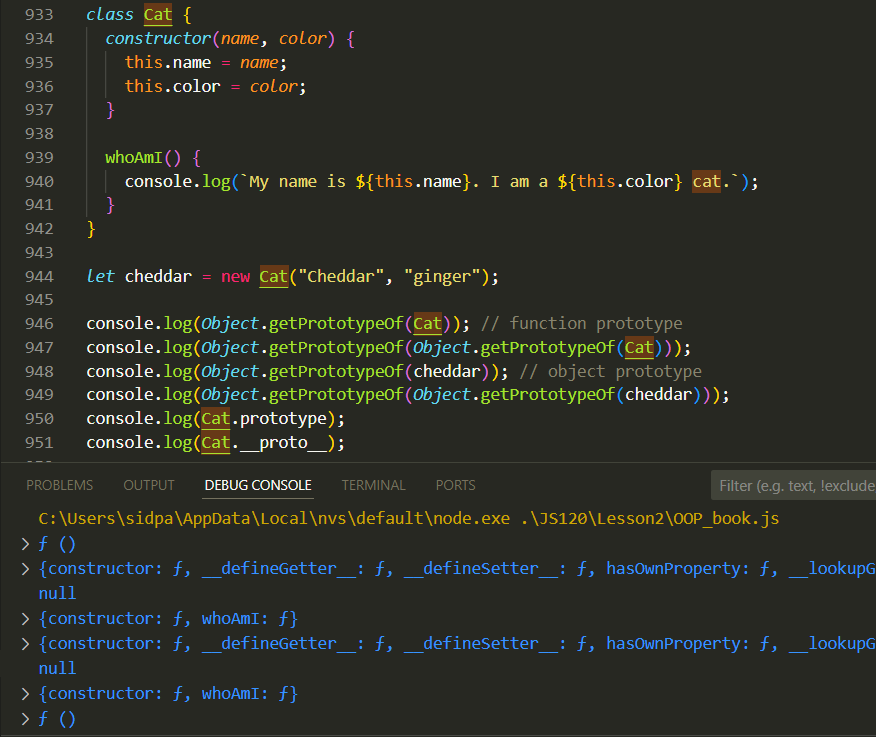
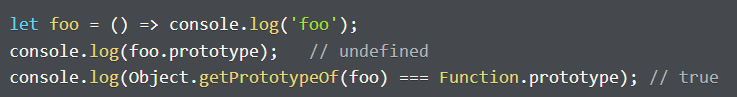
1. Ordinary object properties are also called:
   1. instance properties
   2. public fields
   3. public class fields
2. Ordinary object methods are also called:
   1. instance methods
   2. public methods
   3. public class methods
3. **Private fields (since JS 2022):**
   1.  Also called private instance properties or private class fields are object properties that are private to an object, i.e. no other objects can access it directly.
   2. Aids encapsulation, prevents misuse and allows changing code implementation without breaking other user’s code.
   3. properties are declared (optionally initialized) with a “#” outside the constructor, and then also declared similarly within the constructor.
   4. Cannot be accessed outside of the class, unless a method to retrieve it is made.
4. **Private methods:**
   1. Also called private instance methods or private class methods are instance methods that are private to an object. They can only be used within the class.
   2. Their names begin with “#”.
   3. Used if you are performing certain operations inside the object but don’t want to make those operations available to the users.
5. Getters and setters
   1. Useful; so that
      1. You can safely do future updates in object properties without breaking the code of users that are using your code.
      2. Prevent unrestricted access and editing of the code.
   2. Methods that work like ordinary properties, i.e. you access them without “()” in case of getters OR set the properties as: instance.setterMethod = newProperty; in case of setters.
   3. Getters can be defined on any object (not just classes) by prepending the method definition with the “get” keyword. Also, the method must always be defined in a concise method syntax.
   4. For Setters we prepend the method with the keyword “set”.  
      
6. Static Fields and Methods:
   1. Belong to the class itself and can be called on the class only. Cannot be called on the class instances.
   2. Static fields are used to store constants and utility data that is shared across all class instances. One use case can be to keep track of something related to the class, ex: instance counts.
   3. Static methods or class methods are defined on class rather than instance objects and are not available on object instances. They provide utility functions that perform tasks relevant to the class but do not require access to the instance specific data.  
      One use case can be to report relevant data to the class.  
      

# Prototypal Inheritance

## Prototype Objects

1. They are the key to understanding prototypal inheritance. They are of two types:
   1. object prototypes
   2. function prototypes

They are closely related but different in how they are used.

1. **Object prototype:** 
   1. An object that contains methods that an object inherits.
   2. instances/objects created from a class always have an object prototype (includes methods defined by the class).
   3. If the current class of object is inheriting from another class, then this object prototype will further have an object prototype which contains methods of its superclass.
   4. You can follow this chain of prototype all the way to Object.prototype (whose object prototype is null).
   5. All object literals ({} and new Object()) and object factories also have Object.prototype as their object prototype.
2. **Function prototype:  
   **
   1. Most JS functions and classes have a prototype property that references an object called a function prototype.
   2. When we use new to call the Cat class, following things happen:
      1. JS creates a new empty object that has access to the methods defined within Cat.
      2. It then sets this new object’s object prototype to Cat’s function prototype.
      3. Calls Cat’s constructor method with this set to the new object.
      4. The constructor method initializes the new object and returns the object to the caller.
   3. Thus, function prototype of Cat and object prototype of cheddar refers to the same object. Whether we call it function or object prototype depends on our perspective. The function prototype belongs to the class or constructor function and is used during object construction. On the other hand, object prototype belongs to the object and is used when JS needs to look for methods to execute.
3. Accessing object prototype: **Object.getPrototypeOf(instance)**
4. Reassigning object prototype: **Object.setPrototypeOf(instance, protoObject)**
5. ****The object prototype for instances are called function prototypes for class/functions, but class/fns also have their own object prototypes: f () as shown in the above code example. This is where they look for static methods (apply, call, bind etc).
6. When you set another class/fn as the prototype for a class/fn, it can access its static methods but it cannot access its instance methods.
7. **Note:** Arrow functions do not have function prototypes (object prototypes of instances) but they do have object prototype (Function.prototype OR f ()).  
   

## Constructor/Prototype pattern

1. Here we use constructor functions to create objects.
2. In order to replicate inheritance:
   1. We first create other constructor functions (sub-classes)
   2. In order to establish prototype chain, we replace the prototype property of subclass with that of super-class using Object.create().
   3. To call the constructor function of superclass inside sub-class, we call the super-class function using the context of sub-class: superClass.call(this, arg1, arg2, …)