18.1. What Are Classes?

Recall that objects are data structures that hold many values, which consist of *properties* and *methods*.

We often need to create many objects of the same *type*. To do this in an efficient way, we define a **class**, which allows us to set up the general structure for an object. We can then reuse that structure to build multiple objects. These objects all have the same set of *keys*, but the *values* assigned to each key will vary.

Let's revisit the animal astronauts from earlier exercises to see how this works.

18.1.1. An Astronaut Object

When we create an object to hold an astronaut's data, it might look something like:

```
let fox = {
  name: 'Fox',
  age: 7,
  mass: 12,
  catchPhrase: function(repeats) {
     let phrase = 'LaunchCode';
     for (let i = 0; i < repeats; i++) {
        phrase += ' Rocks';
     }
     return phrase;
  }
}

console.log(`${fox.name} is ${fox.age} years old and has a mass of ${fox.mass} kg.`);
console.log(`${fox.name} says, "${fox.catchPhrase(3)}."`);
```

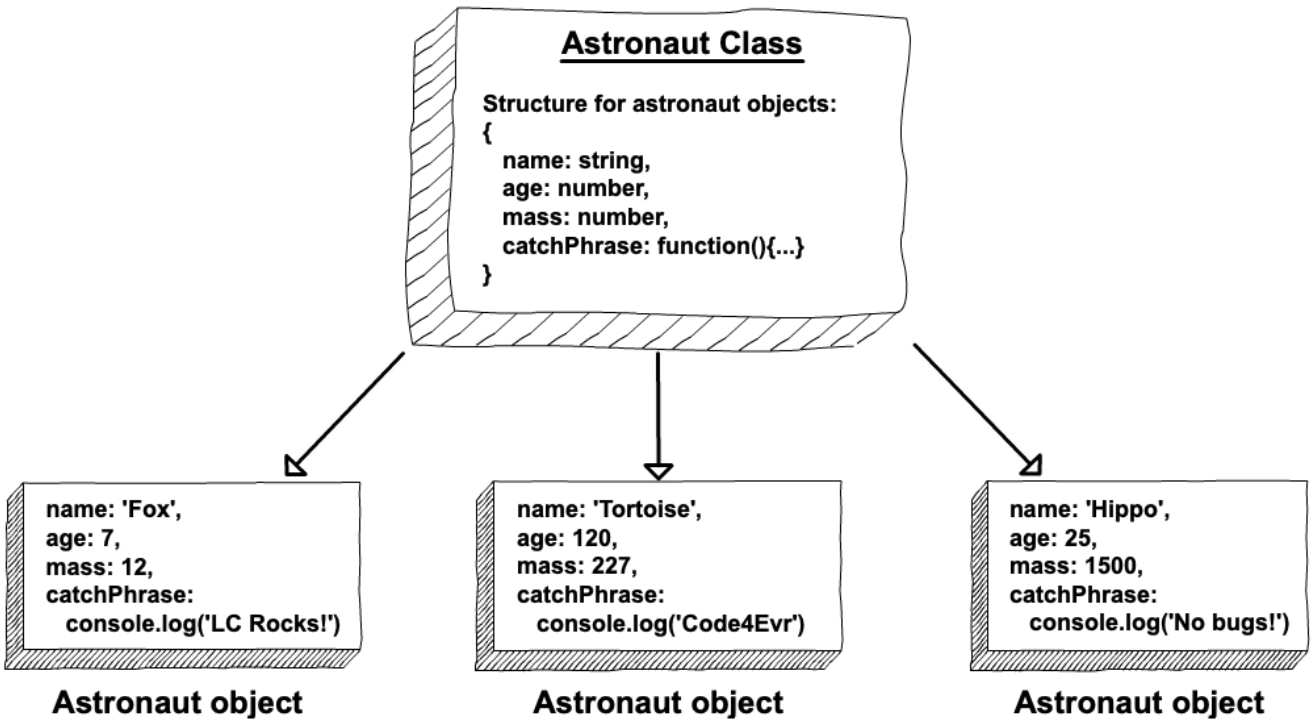
Console Output

```
Fox is 7 years old and has a mass of 12 kg.
Fox says, "LaunchCode Rocks Rocks Rocks."
```

The `fox` object contains all the data and functions for the astronaut named `'Fox'`.

Of course, we have multiple astronauts on our team. To store data for each one, we would need to copy the structure for `fox` multiple times and then change the values to suit each crew member. This is inefficient and repetitive.

By letting us define our own **classes**, JavaScript provides a better way to create multiple, similar objects.


Astronaut Class
Structure for astronaut objects:
{
name: string,
age: number,
mass: number,
catchPhrase: function(){...}
}
name: 'Fox',
age: 7,
mass: 12,
catchPhrase:
console.log('LC Rocks!')
Astronaut object
name: 'Tortoise',
age: 120,
mass: 227,
catchPhrase:
console.log('Code4Evr')
Astronaut object
name: 'Hippo',
age: 25,
mass: 1500,
catchPhrase:
console.log('No bugs!')
Astronaut object

18.2. Declaring and Calling a Class

18.2.1. Creating a Class

Just like the `function` keyword defines a new function, the keyword for defining a new class is `class`. By convention, class names start with capital letters to distinguish them from JavaScript function and variable names (e.g. `MyClass` vs. `myFunction`).

Remember that classes are blueprints for building multiple objects of the same type. The general format for declaring a class is:

```
class ClassName {
  constructor(parameters) {
     //assign properties
  }
  //define methods
}

```

Note the keyword `constructor`. This is a special method for creating objects of the same type, and it assigns the key/value pairs. Parameters are passed into `constructor` rather than the `class` declaration.

18.2.1.1. Assigning Properties

Let's set up an `Astronaut` class to help us store data about our animal crew. Each animal has a `name`, `age`, and `mass`, and we assign these properties in `constructor` as follows:

```
class Astronaut {
  constructor(name, age, mass) {
     this.name = name,
     this.age = age,
     this.mass = mass
  }
}

```

The `this` keyword defines a key/value pair, where the text attached to `this` becomes the key, and the value follows the equal sign (`this.key = value`).

`constructor` uses the three `this` statements (`this.name = name`, etc.) to achieve the same result as the object declaration `let objectName = {name: someString, age: someNumber, mass: someMass}`. Each time the `Astronaut` class is called, `constructor` builds an object with the SAME set of keys, but it assigns different values to the keys based on the arguments.

Note

Each class requires *one* `constructor`. Including more than one `constructor` results in a syntax error.
If `constructor` is left out of a class declaration, JavaScript adds an empty `constructor () {}` automatically.

18.2.2. Creating a New Class Object

To create an object from a class, we use the keyword `new`. The syntax is:

```
let objectName = new ClassName(arguments);
```

`new` creates an **instance** of the class, which means that the object generated shares the same set of keys as every other object made from the class. However, the values assigned to each key differ.

For this reason, objects created with the same class are NOT equal.

Example

Let's create objects for two of our crew members: Fox and Hippo.

```
 1  class Astronaut {
 2    constructor(name, age, mass){
 3       this.name = name,
 4       this.age = age,
 5       this.mass = mass
 6    }
 7  }
 8
 9  let fox = new Astronaut('Fox', 7, 12);
10  let hippo = new Astronaut('Hippo', 25, 1500);
11
12  console.log(typeof hippo, typeof fox);
13
14  console.log(hippo, fox);
```

Console Output

```
object object

Astronaut { name: 'Hippo', age: 25, mass: 1500 }
Astronaut { name: 'Fox', age: 7, mass: 12 }
```

In lines 9 and 10, we call the `Astronaut` class twice and pass in different sets of arguments, creating the `fox` and `hippo` objects.

The output of line 14 shows that `fox` and `hippo` are both the same *type* of object (`Astronaut`). The two share the same *keys*, but they have different values assigned to those keys.

After creating an `Astronaut` object, we can access, modify, or add new key/value pairs as described in the Objects and Math chapter.

Try It

Play around with modifying and adding properties inside and outside of the `class` declaration.

```
 1  class Astronaut {
 2    constructor(name, age, mass){
 3       this.name = name,
 4       this.age = age,
 5       this.mass = mass
 6    }
 7  }
 8
 9  let fox = new Astronaut('Fox', 7, 12);
10
11  console.log(fox);
12  console.log(fox.age, fox.color);
13
14  fox.age = 9;
15  fox.color = 'red';
```

```
16
17console.log(fox);
18console.log(fox.age, fox.color);
```

repl.it

Console Output

```
Astronaut { name: 'Fox', age: 7, mass: 12 }
7 undefined
Astronaut { name: 'Fox', age: 9, mass: 12, color: 'red' }
9 'red'
```

Attempting to print `fox.color` in line 12 returns `undefined`, since that property is not included in the `Astronaut` class. Line 15 adds the `color` property to the `fox` object, but this change will not affect any other objects created with `Astronaut`.

18.2.2.1. Setting Default Values

What happens if we create a new `Astronaut` without passing in all of the required arguments?

Try It!

```
 class Astronaut {
1  constructor(name, age, mass){
2     this.name = name,
3     this.age = age,
4     this.mass = mass
5  }
6}
7
8let tortoise = new Astronaut('Speedy', 120);
9
10console.log(tortoise.name, tortoise.age, tortoise.mass);
11
```

repl.it

To avoid issues with missing arguments, we can set a *default* value for a parameter as follows:

```
 class Astronaut {
1  constructor(name, age, mass = 54){
2     this.name = name,
3     this.age = age,
4     this.mass = mass
5  }
6}
7
```

Now if we call `Astronaut` but do not specify a mass value, the constructor automatically assigns a value of `54`. If an argument is included for `mass`, then the default value is ignored.

TRY IT! Return to the repl.it in the example above and set default values for one or more of the parameters.

18.2.3. Check Your Understanding

The questions below refer to a class called `Car`.

```
1class Car {
```

```
  constructor(make, model, year, color, mpg){
     this.make = make,
     this.model = model,
     this.year = year,
     this.color = color,
     this.mpg = mpg
  }
}
```

Question

If we call the class with `let myCar = new Car('Chevy', 'Astro', 1985, 'gray', 20)`, what is output by `console.log(typeof myCar.year)`?

a. object
b. string
c. function
d. number
e. property

Question

If we call the class with `let myCar = new Car('Tesla', 'Model S', 2019)`, what is output by `console.log(myCar)`?

a. Car {make: 'Tesla', model: 'Model S', year: 2019, color: undefined, mpg: undefined }
b. Car {make: 'Tesla', model: 'Model S', year: 2019, color: '', mpg: '' }
c. Car {make: 'Tesla', model: 'Model S', year: 2019 }

18.3. Assigning Class Methods

Just as with objects, we may want to add methods to our classes in addition to properties. So far, we have learned how to set the values of the class's properties inside the `constructor`.

When assigning methods in classes, we can either create them *outside* or *inside* the `constructor`.

18.3.1. Assigning Methods Outside `constructor`

When assigning methods outside of the `constructor`, we simply declare our methods the same way we would normally do for objects.

```
class ClassName {
  constructor(parameters) {
    //assign properties with this.key = value
  }

  methodName(parameters) {
    //function code
  }
}
```

Example

```
class Astronaut {
  constructor(name, age, mass){
    this.name = name,
    this.age = age,
    this.mass = mass
  }

  reportStats() {
    let stats = `${this.name} is ${this.age} years old and has a mass of ${this.mass} kg.`;
    return stats;
  }
}

let fox = new Astronaut('Fox', 7, 12);
console.log(fox.reportStats());
```

Console Output

```
Fox is 7 years old and has a mass of 12 kg.
```

We declared our method, `reportStats()` outside of the constructor. When we declare a new instance of the `Astronaut` class, we can use the `reportStats()` method to return a concise string containing all of the info we would need about an astronaut.

18.3.2. Assigning Methods Inside `constructor`

When declaring methods inside the `constructor`, we need to make use of the `this` keyword, just as we would with our properties.

```
class ClassName {
   constructor(parameters) {
      this.methodName = function(parameters) {
         //function code
      }
   }
}
```

Example

Let's consider the `Astronaut` class that we have been working with. When assigning the method, `reportStats()`, inside the `constructor`, we would do so like this:

```
class Astronaut {
   constructor(name, age, mass){
      this.name = name,
      this.age = age,
      this.mass = mass,
      this.reportStats = function() {
         let stats = `${this.name} is ${this.age} years old and has a mass of ${this.mass} kg.`;
         return stats;
      }
   }
}

let fox = new Astronaut('Fox', 7, 12);

console.log(fox.reportStats());
```

Console Output

```
Fox is 7 years old and has a mass of 12 kg.
```

Initially, this may seem to produce the same result as assigning `reportStats()` outside of the constructor. We will weigh the pros and cons of both methods below.

18.3.3. Which Way is Preferred?

Try It!

Try comparing the outputs of `fox` and `hippo` to see the effect of assigning a method *inside* the constructor versus *outside* the constructor.

```
// Here we assign the method inside the constructor
class AstronautI {
   constructor(name, age, mass){
      this.name = name,
      this.age = age,
      this.mass = mass,
      this.reportStats = function() {
         let stats = `${this.name} is ${this.age} years old and has a mass of ${this.mass} kg.`;
         return stats;
      }
```

```
  }
}

// Here we assign the method outside fo the constructor
class AstronautO {
  constructor(name, age, mass){
     this.name = name,
     this.age = age,
     this.mass = mass
  }

  reportStats() {
     let stats = `${this.name} is ${this.age} years old and has a mass of ${this
.mass} kg.`;
     return stats;
  }
}

let fox = new AstronautI('Fox', 7, 12);
let hippo = new AstronautO('Hippo', 25, 1000);

  console.log(fox);
  console.log(hippo);
```

repl.it

In the case of assigning the method *inside* the constructor, each `Astronaut` object carries around the code for `reportStats()`. With today's computers, this is a relatively minor concern. However, each `Astronaut` has extra code that may not be needed. This consumes memory, which you need to consider since today's businesses want efficient code that does not tax their systems.

Because of this, if a method is the same for ALL objects of a class, define that method *outside* of the constructor. Each object does not need a copy of identical code. Therefore, the declaration of a method outside of the constructor will not consume as much memory.

18.3.4. Check Your Understanding

Question

What is the assignment for the `grow` method missing?

```
class Plant {
  constructor(type, height) {
     this.type = type,
     this.height = height
  }

  grow  {
     this.height = this.height + 1;
  }
}
```

18.4. Inheritance

Object-oriented programming is a type of software design where the codebase is organized around *objects* and *classes*. Objects contain the functions and central logic of a program.

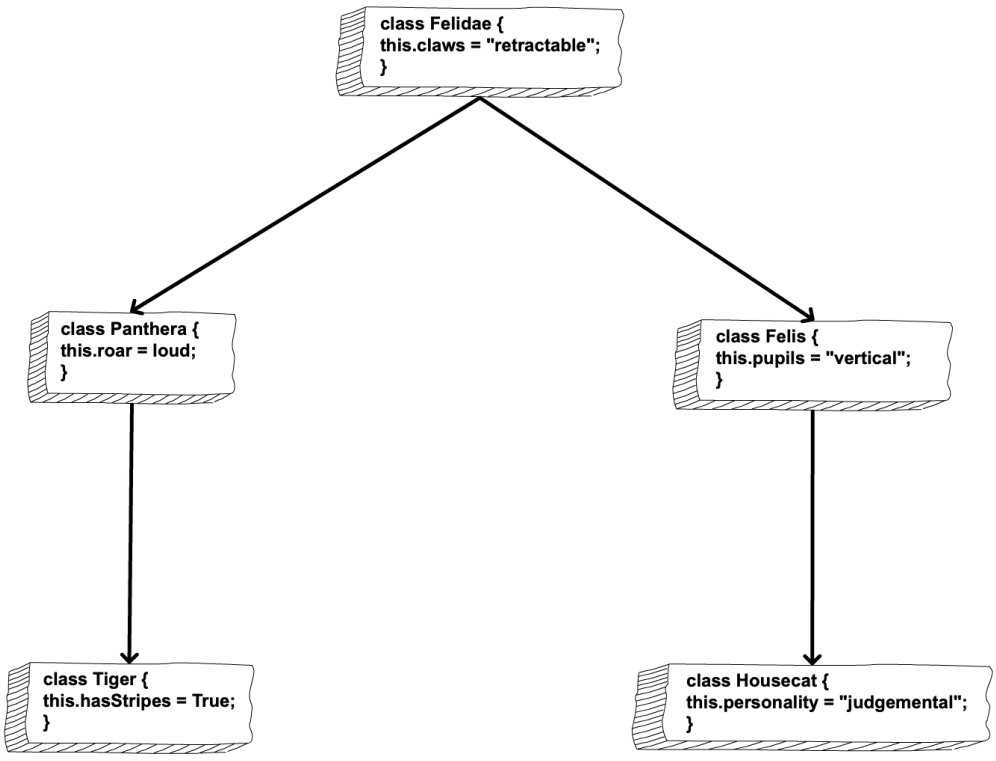
Object-oriented programming stands on top of four principles: abstraction, polymorphism, encapsulation, and inheritance. We will dive into inheritance now and work with the other three principles in Unit Two of this class.

Inheritance refers to the ability of one class to acquire properties and methods from another.

Think of it this way, in the animal kingdom, a *species* is a unique entity that inherits traits from its *genus*. The *genus* also has unique properties, but inherits traits from its *family*. For example, a tiger and a housecat are members of two different species, however, they share similar traits such as retractable claws. The two cats inherited their similar traits from their shared family, *felidae*.

Using inheritance in programming, we can create a structure of classes that inherit properties and methods from other classes.

If we wanted to program classes for our tiger and housecat, we would create a felidae class for the family. We would then create two classes for the panthera genus and the felis genus. We would create classes for the tiger and house cat species as well. The species classes would inherit properties and methods from the genus classes and the genus classes would inherit properties and methods from the family class.



The classes inheriting properties and methods are **child classes**, and the classes passing down properties and methods are **parent classes**.

18.4.1. `extends`

When designating a class as the child class of another in JavaScript, we use the `extends` keyword. We also must use the `super()` constructor to get the properties and methods needed from the parent class.

```
class ChildClass extends ParentClass {
   constructor () {
      super();
      // properties
   }
}
```

In the case of a tiger, tigers have stripes, but they also have loud roars. Their ability to roar loudly is a trait they share with other members of the *panthera* genus. Tigers also got their retractable claws from the *felidae* family.

Example

```
class Felidae {
   constructor() {
      this.claws = "retractable";
   }
}

class Panthera extends Felidae {
   constructor() {
      super();
      this.roar = "loud"
   }
}

class Tiger extends Panthera {
   constructor() {
      super();
      this.hasStripes = "true";
   }
}

let tigger = new Tiger();

console.log(tigger);
```

repl.it

When creating the classes for our tiger, we can use the `extends` keyword to set up `Tiger` as the child class of `Panthera`. The `Tiger` class then inherits the property, `roar`, from the `Panthera` class and has an additional property, `hasStripes`.

Note

The `extends` keyword is not supported in Internet Explorer.

18.4.2. Check Your Understanding

Question

If you had to create classes for a *wolf*, the *canis* genus, and the *carnivora* order, which statement is TRUE about the order of inheritance?

a. `Wolf` and `Canis` are parent classes to `Carnivora`.

b. `Wolf` is a child class of `Canis` and a parent class to `Carnivora`.
c. `Wolf` is child class of `Canis`, and `Canis` is a child class of `Carnivora`.
d. `Wolf` is child class of `Canis`, and `Canis` is a parent class of `Carnivora`.

18.5. Exercises: Classes

Welcome to the space station! It is your first day onboard and as the newest and most junior member of the crew, you have been asked to organize the library of manuals and fun novels for the crew to read. Click on this repl.it link and fork the starter code.

Headquarters have asked that you store the following information about each book.

1. The title
2. The author
3. The copyright date
4. The ISBN
5. The number of pages
6. The number of times the book has been checked out.
7. Whether the book has been discarded.

Headquarters also needs you to track certain actions that you must perform when books get out of date. First, for a manual, the book must be thrown out if it is over 5 years old. Second, for a novel, the book should be thrown out if it has been checked out over 100 times.

1. Construct three classes that hold the information needed by headquarters as properties. Also, each class needs two methods that update the book's property if the book needs to be discarded. One class should be a `Book` class and two child classes of the `Book` class called `Manual` and `Novel`. *Hint:* This means you need to read through the requirements for the problem and decide what should belong to `Book` and what should belong to the `Novel` and `Manual` classes.

2. Declare an object of the `Novel` class for the following tome from the library:

Novel

Variable	Value
Title	Pride and Prejudice
Author	Jane Austen
Copyright date	1813
ISBN	1111111111111
Number of pages	432
Number of times the book has been checked out	32

Novel

Variable	Value
Whether the book has been discarded	No

3. Declare an object of the `Manual` class for the following tome from the library:

Manual

Variable	Value
Title	Top Secret Shuttle Building Manual
Author	Redacted
Copyright date	2013
ISBN	0000000000000
Number of pages	1147
Number of times the book has been checked out	1
Whether the book has been discarded	No

4. One of the above books needs to be discarded. Call the appropriate method for that book to update the property. That way the crew can throw it into empty space to become debris.

5. The other book has been checked out 5 times since you first created the object. Call the appropriate method to update the number of times the book has been checked out.

18.6. Studio: Classes

18.6.1. Before You Start

If you are enrolled in a LaunchCode program, access this studio by following the repl.it classroom links posted in your class at learn.launchcode.org.

If you are working through this material on your own, use the repl.it links contained on this page.

18.6.2. Getting Started

Let's create a class to handle new animal crew candidates!

Edit the practice file as you complete the studio activity.

18.6.3. Part 1 - Add Class Properties

1. Declare a class called `CrewCandidate` with a `constructor` that takes three parameters—`name`, `mass`, and `scores`. Note that `scores` will be an array of test results.
2. Create objects for the following candidates:
    a. Bubba Bear has a mass of 135 kg and test scores of 88, 85, and 90.
    b. Merry Maltese has a mass of 1.5 kg and test scores of 93, 88, and 97.
    c. Glad Gator has a mass of 225 kg and test scores of 75, 78, and 62.

Use `console.log` for each object to verify that your class correctly assigns the key/value pairs.

18.6.4. Part 2 - Add First Class Method

As our candidates complete more tests, we need to be able to add the new scores to their records.

1. Create an `addScore` method in `CrewCandidate`. The function must take a new score as a parameter. Code this function OUTSIDE of `constructor`. (If you need to review the syntax, revisit Assigning Class Methods).
2. When passed a score, the function adds the value to `this.scores` with the push array method.
3. Test out your new method by adding a score of `83` to Bubba's record, then print out the new score array with `objectName.scores`.

18.6.5. Part 3 - Add More Methods

Now that we can add scores to our candidates' records, we need to be able to evaluate their fitness for our astronaut program. Let's add two more methods to `CrewCandidate`—one to average the test scores and the other to indicate if the candidate should be admitted.

18.6.5.1. Calculating the Test Average

1. Add an `average()` method outside `constructor`. The function does NOT need a parameter.
2. To find the average, add up the entries from `this.scores`, then divide the sum by the number of scores.
3. To make the average easier to look at, round it to 1 decimal place, then return the result from the function.

Verify your code by evaluating and printing Merry's average test score (92.7).

18.6.5.2. Determining Candidate Status

Candidates with averages at or above 90% are automatically accepted to our training program. Reserve candidates average between 80 - 89%, while probationary candidates average between 70 - 79%. Averages below 70% lead to a rejection notice.

1. Add a `status()` method to `CrewCandidate`. The method returns a string (`Accepted`, `Reserve`, `Probationary`, or `Rejected`) depending on a candidate's average.
2. The `status` method requires the average test score, which can be called as a parameter OR from inside the function. That's correct - methods can call other methods inside a class! Just remember to use the `this` keyword.
3. Once `status` has a candidate's average score, evaluate that score, and return the appropriate string.
4. Test the `status` method on each of the three candidates. Use a template literal to print out `'___ earned an average test score of ___% and has a status of ___.'`

18.6.6. Part 4 - Play a Bit

Use the three methods to boost Glad Gator's status to `Reserve` or higher. How many tests will it take to reach `Reserve` status? How many to reach `Accepted`? Remember, scores cannot exceed 100%.

Tip

Rather than adding one score at a time, could you use a loop?