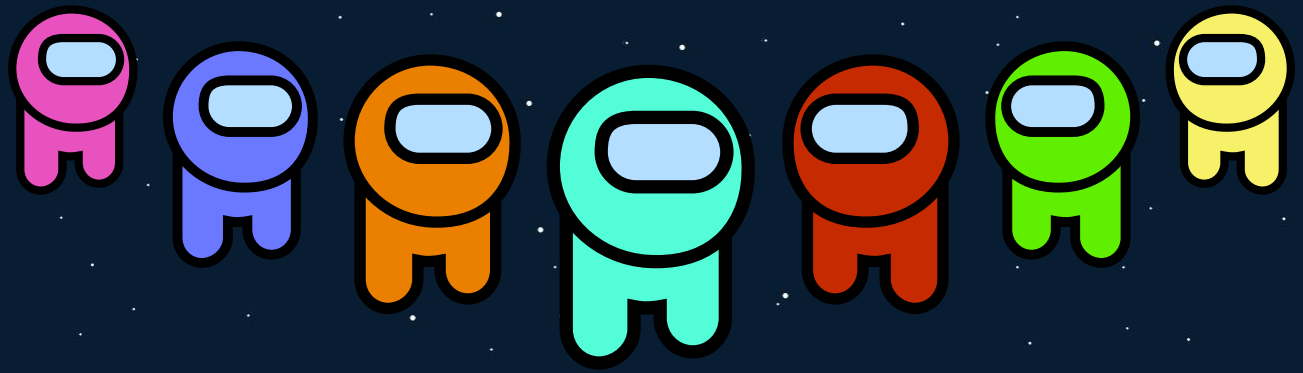
# Java script's primitives and reference types, and objects

أحمد إبراهيم بيت المال 172157



# Contents





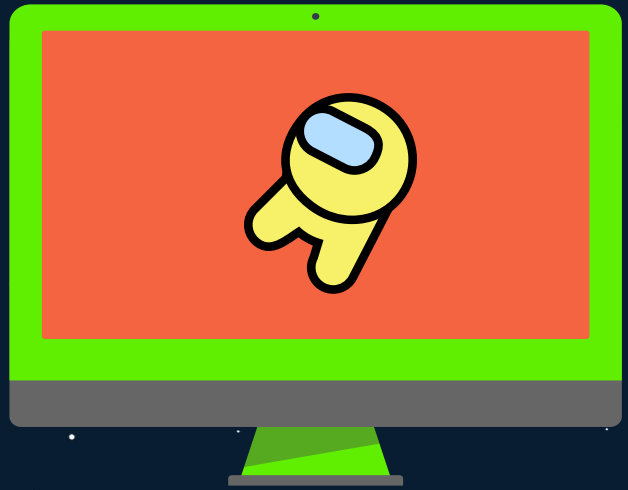


01

# Primitive and reference types

# Data types

*Data types* are the classifications we give to the different kinds of data that we use in programming. In JavaScript, there are seven fundamental data types:

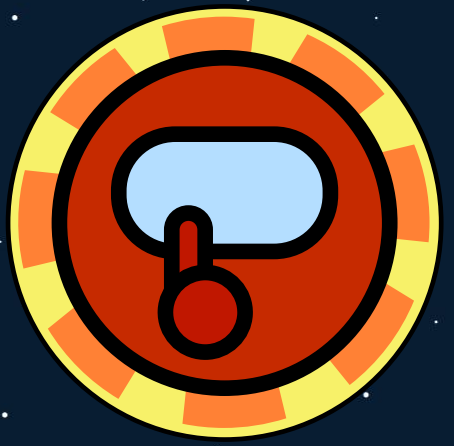


# DATA TYPES

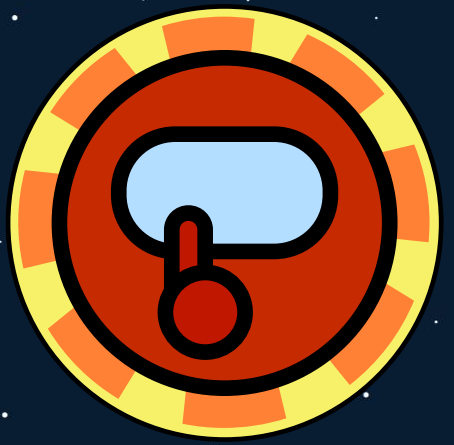
| | |
|---|---|
| Number | Any number, including numbers with decimals: 4, 8, 1516, 23.42. |
| String | Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " ... ". |
| Boolean | This data type only has two possible values– either true or false (without quotes). It's helpful to think of booleans as on and off switches or as the answers to a "yes" or "no" question. |
| Null | This data type represents the intentional absence of a value, and is represented by the keyword null (without quotes). |
| Undefined | This data type is denoted by the keyword undefined (without quotes). It also represents the absence of a value though it has a different use than null. |
| Sympol | A newer feature to the language, symbols are unique identifiers, useful in more complex coding. |
| Object | Collections of related data. |

# Whoa!

The first 6 of those types are considered primitive data types. They are the most basic data types in the language. Objects are more complex,and are considered reference types.



Javascript has 3 data types that are passed by reference: Array , Function , and Object . These are all technically Objects, so we refer to them collectively as Objects.

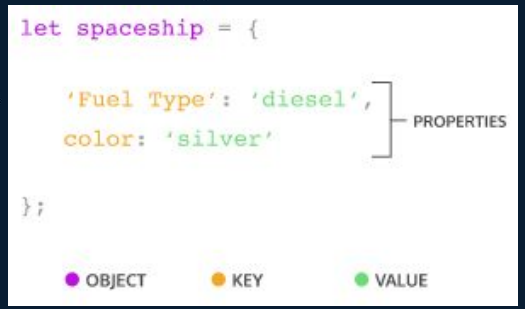


# Reference types

01

## Object

Objects store collections of key-value pairs, Each key-value pair is a property—when a property is a function it is known as a method.




02

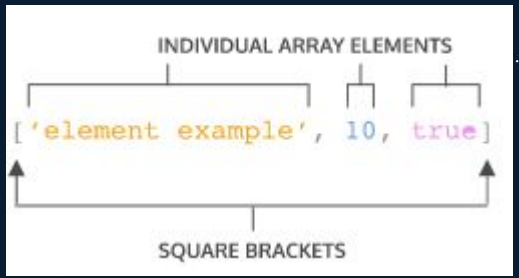
## Function

A function is a reusable block of code that groups together a sequence of statements to perform a specific task.


03

## Array

Arrays are lists that store data in JavaScript.



```
const unknown1 = 'foo';
console.log(typeof unknown1); // Output:
string

const unknown2 = 10;
console.log(typeof unknown2); // Output:
number

const unknown3 = true;
console.log(typeof unknown3); // Output:
boolean
```

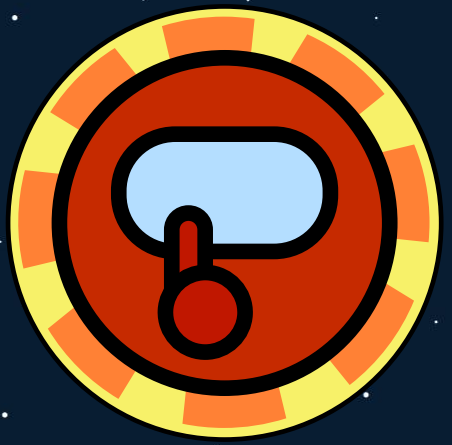
# IDENTIFYING PRIMITIVE TYPES

While writing code, it can be useful to keep track of the data types of the variables in your program. If you need to check the data type of a variable's value, you can use the typeof operator.

The typeof operator checks the value to its right and returns, or passes back, a string of the data type.

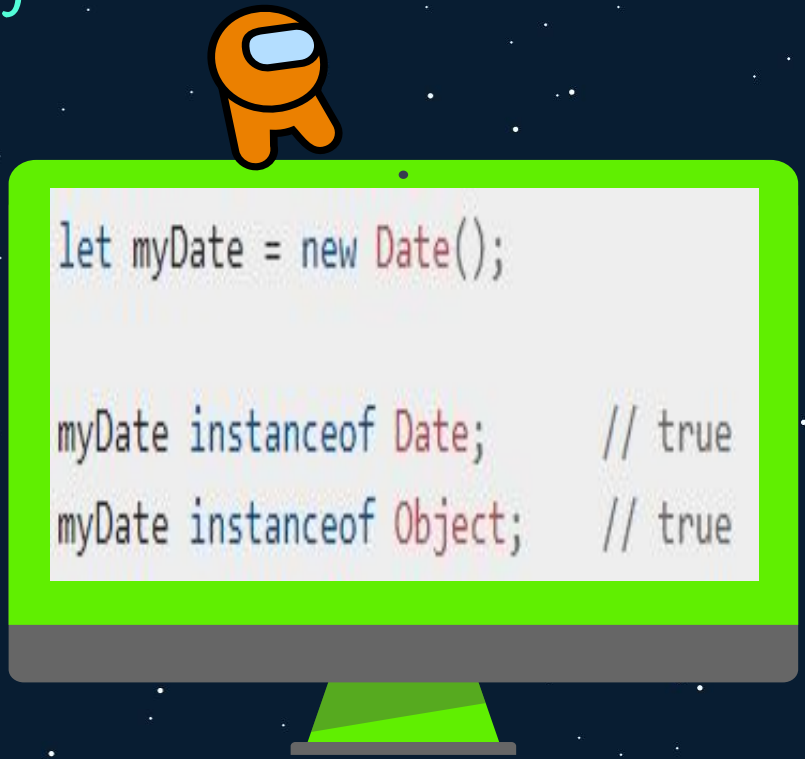
# Whoa!

When it comes to identifying reference types, only functions can be identified using the operator (typeof), other reference types would return (object) as their type, and that’s not very helpful, that’s when (instanceof) comes in handy.



# Identifying reference types

*instanceof is an operator used to test if an object (a reference type) is of a given type. The result of the operation is either true or false.*



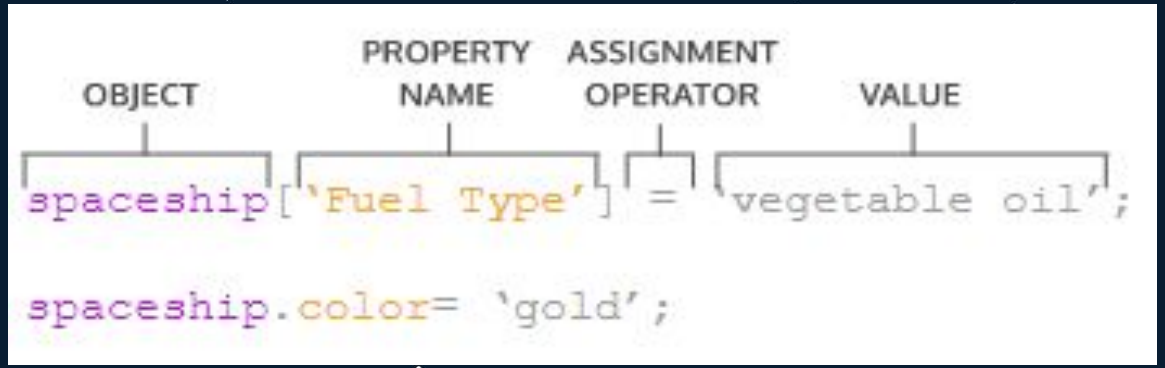
# Creating objects

Objects can be assigned to variables just like any JavaScript type. We use curly braces, {}, to designate an object literal:

```
let spaceship = {}; // spaceship is an empty object
```

# Property Assignment

We can use either dot notation, ., or bracket notation, [], and the assignment operator, = to add new key-value pairs to an object or change an existing property.



# One of two things can happen with property assignment:

If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.

If there was no property with that name, a new property will be added to the object.



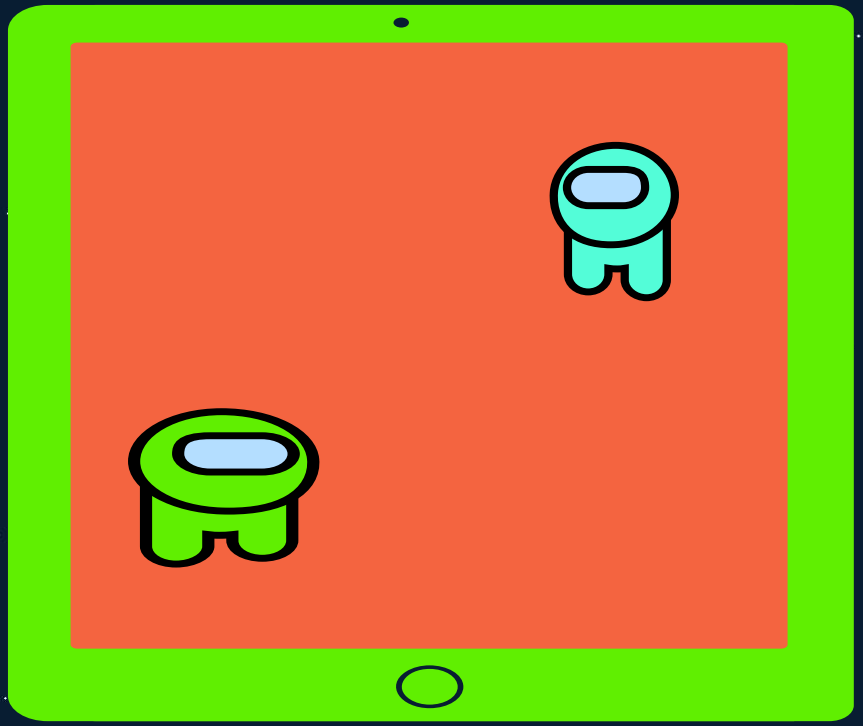




02

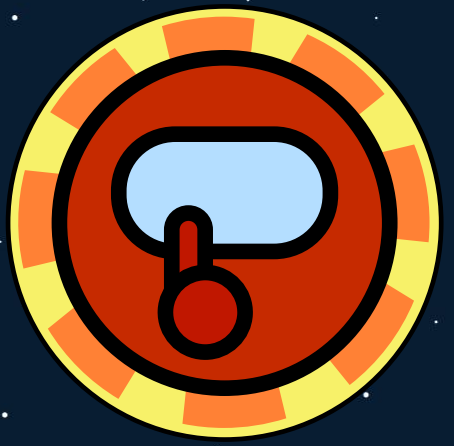
# OBJECTS

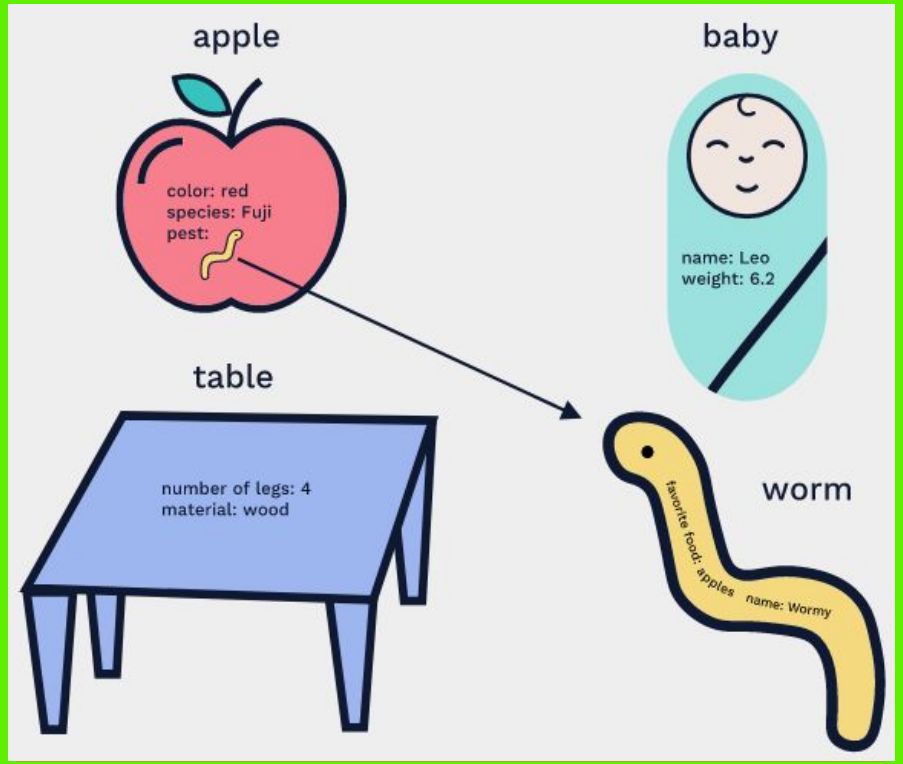
# OBJECTS



JavaScript loves objects! Many components of the language are actually objects under the hood, and even the parts that aren't— like strings or numbers— can still act like objects in some instances.

objects are containers storing related data and functionality.






# OBJECTS

We can use JavaScript objects to model real-world things, like a basketball, or we can use objects to build the data structures that make the web possible.

# Creating properties

We fill an object with unordered data.
This data is organized into key-value pairs. A key is like a variable name that points to a location in memory that holds a value.
A key's value can be of any data type in the language including functions or other objects.
We make a key-value pair by writing the key's name, or identifier, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (,).

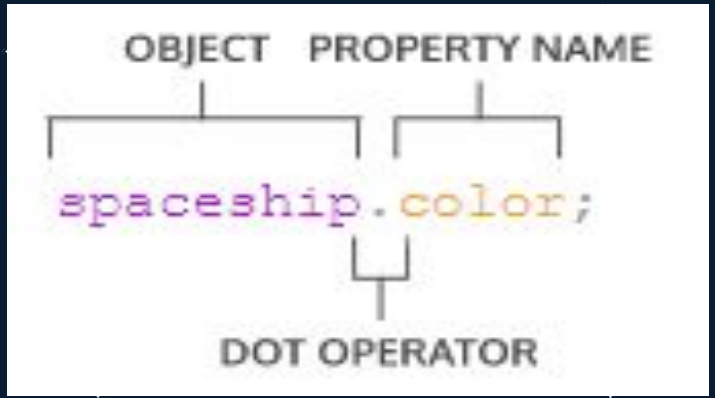
```
// An object literal with two key-value pairs
let spaceship = {
  'Fuel Type': 'diesel',
  color: 'silver'
};
```

Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks

# Accessing properties

we can access an object's property With property dot notation, we write the object's name, followed by the dot operator and then the property name (key):

```
let spaceship = {
  homePlanet: 'Earth',
  color: 'silver'
};
spaceship.homePlanet; // Returns 'Earth',
spaceship.color; // Returns 'silver',
```



# Whoa!

If we try to access a property that does not exist on that object, undefined will be returned.

```
spaceship.favoriteIcecream; // Returns
undefined
```

# Object's methods

When the data stored on an object is a function we call that a method. A property is what an object has, while a method is what an object does.







With the new method syntax introduced in ES6 we can omit the colon and the function keyword.

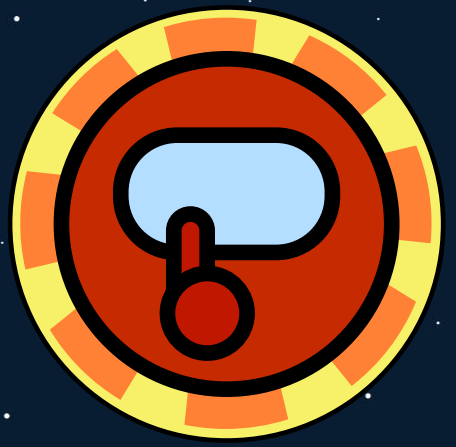
```
const alienShip = {
  invade () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```







Object methods are invoked by appending the object's name with the dot operator followed by the method name and parentheses:



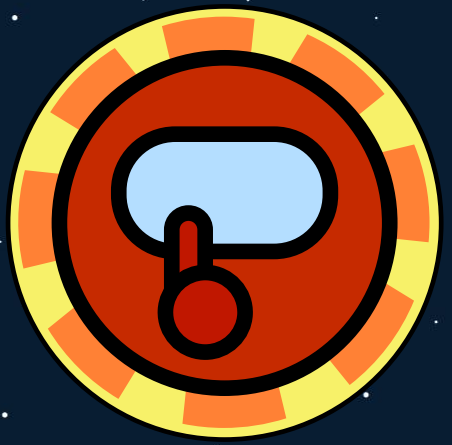
```
alienShip.invade(); // Prints 'Hello! We have
come to dominate your planet. Instead of
Earth, it shall be called New Xaculon.'
```

# Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects aren't ordered! JavaScript has given us alternative solution for iterating through objects with the for...in syntax

# Whoa!

for...in will execute a given block of code for each property in an object.



# FOR...IN

Our for...in will iterate through each element of the spaceship.crew object. In each iteration, the variable crewMember is set to one of spaceship.crew's keys, enabling us to log a list of crew members' role and name.

```
let spaceship = {
  crew: {
    captain: {
      name: 'Lily',
      degree: 'Computer Engineering',
      cheerTeam() { console.log('You got this!') }
    },
    'chief officer': {
      name: 'Dan',
      degree: 'Aerospace Engineering',
      agree() { console.log('I agree, captain!') }
    },
    medic: {
      name: 'Clementine',
      degree: 'Physics',
      announce() { console.log(`Jets on!`) } },
    translator: {
      name: 'Shauna',
      degree: 'Conservation Science',
      powerFuel() { console.log('The tank is full!') }
    }
   }
  }
};

// for...in
for (let crewMember in spaceship.crew) {
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`);
}
```

# This keyword

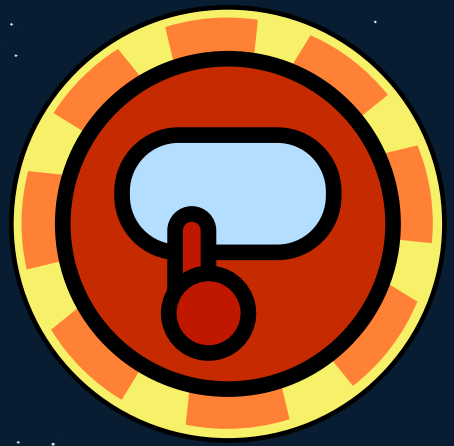
we wanted to add a new method to our goat object called .diet() that prints the goat's dietType

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet();
// Output will be "ReferenceError: dietType is not defined"
```

why is dietType not defined even though it's a property of goat?

# Whoa!

That's because inside the scope of the .diet() method, we don't automatically have access to other properties of the goat object.



```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet();
// Output will be "ReferenceError: dietType is
not defined"
```

# This keyword

Here's where the this keyword comes to the rescue. If we change the .diet() method to use the this, the .diet() works! :

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet() {
    console.log(this.dietType);
  }
};

goat.diet();
// Output: herbivore
```

# This Keyword

The this keyword references the calling object which provides access to the calling object's properties.







# Privacy

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties.

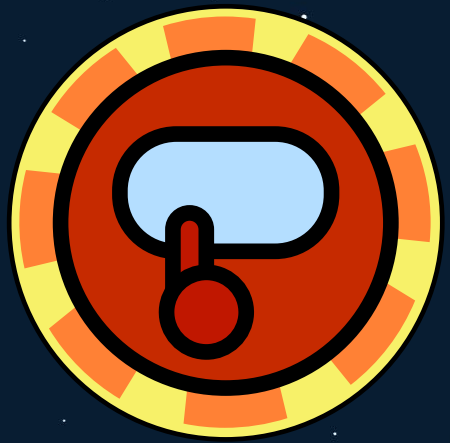
Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore _ before the name of a property to mean that the property should not be altered.

```
const bankAccount = {
  _amount: 1000
}
```

# Whoa!

Even so, it is still possible to reassign _amount:

```
bankAccount._amount = 1000000;
```



# ACCESSOR METHODS

Accessor methods are methods that allow other clients (objects, classes, and users) to access the data of an object.
There are two types of accessor methods:

GETTERS

SETTERS

# GETTERS

Getters are methods that get and return the internal properties of an object.

```
const person = {
  _firstName: 'John',
  _lastName: 'Doe',
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName}
${this._lastName}`;
    } else {
      return 'Missing a first name or a last
name.';
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

# SETTERS

setter methods which reassign values of existing properties within an object.

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a
age');
    }
  }
};
```

# SETTERS

Then to use the setter method:

```
person.age = 40;
console.log(person._age); // Logs: 40
person.age = '40'; // Logs: You must assign
a number to age
```

Setter methods like age do not need to be called with a set of parentheses. Syntactically, it looks like we're reassigning the value of a property.

# FACTORY FUNCTIONS

there are times where we want to create many instances of an object quickly. Here's where factory functions come in...







# Factory functions

A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have . parameters allowing us to customize the object that gets returned.

```
const monsterFactory = (name, age,
energySource, catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

To make an object that represents a specific monster like a ghost, we can call monsterFactory with the necessary arguments and assign the return value to a variable:

```
const ghost = monsterFactory('Ghouly', 251,
'ectoplasm', 'BOO!');
ghost.scare(); // 'BOO!'
```

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don't have to create an object literal every time we need a new monster.

# THANKS