**Name:Hassan Hamdy**

**ID:20193368**

**The deffrence between rem and em**

em Unit: The em unit allows setting the font size of an element relative to the font size of its parent. When the size of the parent element changes, the size of the child changes automatically.

Note: When em units are used on font-size property, the size is relative to the font-size of the parent. When used on other properties, it’s relative to the font-size of that element itself. Here, only the first declaration takes the reference of the parent.

* The font-size of the .child element will be 40px (2\*20px).
* The margin of .child will be 60px. That’s 1.5 times the font-size of our element (1.5\*40px).

 rem Unit: The rem is based upon the font-size value of the root element, which is the <html> element. And if the <html> element doesn’t have a specified font-size, the browser default value of 16px is used. So here only the value of the root is considered, and there is no relation with a parent element.

Unlike em, here size is relative for all declarations, not only first. Let’s understand this with our previous example.

* The font-size of the .child element will be 60px (2\*30px).
* The margin of .child will be 45px. That’s 1.5 times the font-size of the html element (1.5\*30px)

**Css position**

Relative to its original position the element above will now be nudged down from the top by 20px. If we were to animate these properties we can see just how much control this gives us (although this isn’t a good idea for performance reasons):

**Values**

* static: every element has a static position by default, so the element will stick to the normal page flow. So if there is a [left](https://css-tricks.com/almanac/properties/l/left/)/[right](https://css-tricks.com/almanac/properties/r/right/)/[top](https://css-tricks.com/almanac/properties/t/top/)/[bottom](https://css-tricks.com/almanac/properties/b/bottom/)/[z-index](https://css-tricks.com/almanac/properties/z/z-index/) set then there will be no effect on that element.
* relative: an element’s original position remains in the flow of the document, just like the static value. But now [left](https://css-tricks.com/almanac/properties/l/left/)/[right](https://css-tricks.com/almanac/properties/r/right/)/[top](https://css-tricks.com/almanac/properties/t/top/)/[bottom](https://css-tricks.com/almanac/properties/b/bottom/)/[z-index](https://css-tricks.com/almanac/properties/z/z-index/) will work. The positional properties “nudge” the element from the original position in that direction.
* absolute: the element is removed from the flow of the document and other elements will behave as if it’s not even there whilst all the other positional properties will work on it.
* fixed: the element is removed from the flow of the document like absolutely positioned elements. In fact they behave almost the same, only fixed positioned elements are always relative to the document, not any particular parent, and are unaffected by scrolling.
* sticky: the element is treated like a relative value until the scroll location of the viewport reaches a specified threshold, at which point the element takes a fixed position where it is told to stick.
* inherit: the position value doesn’t cascade, so this can be used to specifically force it to, and inherit the positioning value from its parent.

### Absolute

If a child element has an absolute value then the parent element will behave as if the child isn’t there at all

### Fixed

The fixed value is similar to absolute as it can help you position an element anywhere relative to the document, however this value is unaffected by scrolling. See the child element in the demo below and how, once you scroll, it continues to stick to the bottom of the page

### Sticky

The sticky value is like a compromise between the relative and fixed values. As of this writing, it is currently an experimental value, meaning it is not part of the official spec and only partially adopted by select browsers. In other words, it’s probably not the best idea to use this on a live production website.

What does it do? Well, it allows you to position an element relative to anything on the document and then, once a user has scrolled past a certain point in the viewport, fix the position of the element to that location so it remains persistently displayed like an element with a fixed value.

**The deference between while vs for loop**

|  |  |
| --- | --- |
| For loop | While loop |
| Initialization may be either in loop statement or outside the loop | Initialization is always outside the loop. |
| Once the statement(s) is executed then after increment is done. | Increment can be done before or after the execution of the statement(s). |
| It is normally used when the number of iterations is known. | It is normally used when the number of iterations is unknown. |
| Condition is a relational expression | Condition may be expression or non-zero value. |
| It is used when initialization and increment is simple. | It is used for complex initialization. |
| For is entry controlled loop. | While is also entry controlled loop. |
| for ( init ; condition ; iteration ) { statement(s); } | while ( condition ) { statement(s); } |

**for loop:**

for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping. **Syntax:**

for (initialization condition; testing condition;

increment/decrement)

{

statement(s)

}

**while loop:**

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement. **Syntax :**

while (boolean condition)

{

loop statements...

}

**Object Method:**

The **Object** type represents one of [JavaScript's data types](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures). It is used to store various keyed collections and more complex entities. Objects can be created using the [Object()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Object) constructor or the [object initializer / literal syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer).

## [Description](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#description)

Nearly all objects in JavaScript are instances of [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object); a typical object inherits properties (including methods) from Object.prototype, although these properties may be shadowed (a.k.a. overridden). However, an Object may be deliberately created for which this is not true (e.g. by [Object.create(null)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create)), or it may be altered so that this is no longer true (e.g. with [Object.setPrototypeOf](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf)).

Changes to the Object prototype object are seen by **all** objects through prototype chaining, unless the properties and methods subject to those changes are overridden further along the prototype chain. This provides a very powerful although potentially dangerous mechanism to override or extend object behavior.

The Object constructor's behavior depends on the input's type.

* If the value is [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) or [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined), it will create and return an empty object.
* If the value is an object already, it will return the value.
* Otherwise, it will return an object of a Type that corresponds to the given value.

When called in a non-constructor context, Object behaves identically to new Object().

See also the [object initializer / literal syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer).

### [Deleting a property from an object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#deleting_a_property_from_an_object)

There isn't any method in an Object itself to delete its own properties (such as [Map.prototype.delete()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/delete)). To do so, one must use the [delete operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete).

## [Constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#constructor)

[Object()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Object)

Turns the input into an object.

## [Static methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#static_methods)

[Object.assign()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

Copies the values of all enumerable own properties from one or more source objects to a target object.

[Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create)

Creates a new object with the specified prototype object and properties.

[Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty)

Adds the named property described by a given descriptor to an object.

[Object.defineProperties()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties)

Adds the named properties described by the given descriptors to an object.

[Object.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries)

Returns an array containing all of the [key, value] pairs of a given object's **own** enumerable string properties.

[Object.freeze()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze)

Freezes an object. Other code cannot delete or change its properties.

[Object.fromEntries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/fromEntries)

Returns a new object from an iterable of [key, value] pairs. (This is the reverse of [Object.entries](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries)).

[Object.getOwnPropertyDescriptor()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor)

Returns a property descriptor for a named property on an object.

[Object.getOwnPropertyDescriptors()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors)

Returns an object containing all own property descriptors for an object.

[Object.getOwnPropertyNames()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)

Returns an array containing the names of all of the given object's **own** enumerable and non-enumerable properties.

[Object.getOwnPropertySymbols()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols)

Returns an array of all symbol properties found directly upon a given object.

[Object.getPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf)

Returns the prototype (internal [[Prototype]] property) of the specified object.

[Object.is()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)

Compares if two values are the same value. Equates all NaN values (which differs from both IsLooselyEqual used by [==](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Equality) and IsStrictlyEqual used by [===](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality)).

[Object.isExtensible()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible)

Determines if extending of an object is allowed.

[Object.isFrozen()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isFrozen)

Determines if an object was frozen.

[Object.isSealed()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed)

Determines if an object is sealed.

[Object.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)

Returns an array containing the names of all of the given object's **own** enumerable string properties.

[Object.preventExtensions()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions)

Prevents any extensions of an object.

[Object.seal()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/seal)

Prevents other code from deleting properties of an object.

[Object.setPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf)

Sets the object's prototype (its internal [[Prototype]] property).

[Object.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values)

Returns an array containing the values that correspond to all of a given object's **own** enumerable string properties.

## [Instance properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#instance_properties)

[Object.prototype.constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor)

Specifies the function that creates an object's prototype.

[Object.prototype.\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto)

Points to the object which was used as prototype when the object was instantiated.

**regular vs arrow function js**

|  |  |
| --- | --- |
| Regular Function | Arrow Function |
| The function keyword can be used to define a function inside an expression.  Function must have a return statement that specifies the value to return. A function without a return statement will return a default value.  In the case of a constructor called with the new keyword, the default value is the value of its this parameter. For all other functions, the default return value is undefined.  One of the benefits of creating a named function expression is that in case we encountered an error, the stack trace will contain the name of the function, making it easier to find the origin of the error. | Arrow functions don't have their own bindings to this, arguments and shouldn’t be used as methods  . Arrow functions don't have access to the new.target keyword , aren't suitable for call methods  Arrow functions cannot be used as constructors and yield, within its body. The main benefit is that it removes the several pain points associated with the this operator. |

**The object vs instance oop:**

## Object

An Object is created from a Class, like a house is created from a blueprint. You could use a blueprint of a 3 bedroom home to build multiple 3 bedroom homes - 1 blueprint to create multiple 3 bedroom homes. This is what happens with a Class. You can create multiple Objects from your single Class - remember, a Class is just the blueprint for creating Objects of the same type.

## Instance

So we've taken our blueprint (Class), and created multiple 3 bedroom homes (Objects), but how do we refer to one particular home (Object) we've built? We refer to a particular Object as an Instance. If I were to paint the frontdoor of one of the homes, I would be painting the door of a particular Instance of the Objects (home) created from my Class (blueprint).