|  |  |
| --- | --- |
| **rem** | **em 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 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.  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. |

**CSS Layout - The position Property**

The position property specifies the type of positioning method used for an element (static, relative, fixed, absolute or sticky).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **position: static;** | **position: relative;** | **position: fixed;** | **position: absolute;** | **position: sticky;** |
| HTML elements are positioned static by default.  Static positioned elements are not affected by the top, bottom, left, and right properties.  An element with position: static; is not positioned in any special way; it is always positioned according to the normal flow of the page: | An element with position: relative; is positioned relative to its normal position.  Setting the top, right, bottom, and left properties of a relatively positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element. | An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.  A fixed element does not leave a gap in the page where it would normally have been located.  Notice the fixed element in the lower-right corner of the page. Here is the CSS that is used: | An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).  However, if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.  Note: Absolute positioned elements are removed from the normal flow, and can overlap elements. | An element with position: sticky; is positioned based on the user's scroll position.  A sticky element toggles between relative and fixed, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position:fixed). |

**for vs while loop**

A for loop is usually used when the number of iterations is known

And while and do...while loops are usually used when the number of iterations are unknown

**OBJECT METHODS**

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

|  |  |  |
| --- | --- | --- |
| **S.No** | **Methods** | **Description** |
| 1 | [Object.assign()](https://www.javatpoint.com/javascript-object-assign-method) | This method is used to copy enumerable and own properties from a source object to a target object |
| 2 | [Object.create()](https://www.javatpoint.com/javascript-object-create-method) | This method is used to create a new object with the specified prototype object and properties. |
| 3 | [Object.defineProperty()](https://www.javatpoint.com/javascript-object-defineproperty-method) | This method is used to describe some behavioral attributes of the property. |
| 4 | [Object.defineProperties()](https://www.javatpoint.com/javascript-object-defineproperties-method) | This method is used to create or configure multiple object properties. |
| 5 | [Object.entries()](https://www.javatpoint.com/javascript-object-entries-method) | This method returns an array with arrays of the key, value pairs. |
| 6 | [Object.freeze()](https://www.javatpoint.com/javascript-object-freeze-method) | This method prevents existing properties from being removed. |
| 7 | [Object.getOwnPropertyDescriptor()](https://www.javatpoint.com/javascript-object-getownpropertydescriptor-method) | This method returns a property descriptor for the specified property of the specified object. |
| 8 | [Object.getOwnPropertyDescriptors()](https://www.javatpoint.com/javascript-object-getownpropertydescriptors-method) | This method returns all own property descriptors of a given object. |
| 9 | [Object.getOwnPropertyNames()](https://www.javatpoint.com/javascript-object-getownpropertynames-method) | This method returns an array of all properties (enumerable or not) found. |
| 10 | [Object.getOwnPropertySymbols()](https://www.javatpoint.com/javascript-object-getownpropertysymbols-method) | This method returns an array of all own symbol key properties. |
| 11 | [Object.getPrototypeOf()](https://www.javatpoint.com/javascript-object-getprototypeof-method) | This method returns the prototype of the specified object. |
| 12 | [Object.is()](https://www.javatpoint.com/javascript-object-is-method) | This method determines whether two values are the same value. |
| 13 | [Object.isExtensible()](https://www.javatpoint.com/javascript-objects) | This method determines if an object is extensible |
| 14 | [Object.isFrozen()](https://www.javatpoint.com/javascript-objects) | This method determines if an object was frozen. |
| 15 | [Object.isSealed()](https://www.javatpoint.com/javascript-objects) | This method determines if an object is sealed. |
| 16 | [Object.keys()](https://www.javatpoint.com/javascript-objects) | This method returns an array of a given object's own property names. |
| 17 | [Object.preventExtensions()](https://www.javatpoint.com/javascript-object-preventextensions-method) | This method is used to prevent any extensions of an object. |
| 18 | [Object.seal()](https://www.javatpoint.com/javascript-object-seal-method) | This method prevents new properties from being added and marks all existing properties as non-configurable. |
| 19 | [Object.setPrototypeOf()](https://www.javatpoint.com/javascript-object-setprototypeof-method) | This method sets the prototype of a specified object to another object. |
| 20 | [Object.values()](https://www.javatpoint.com/javascript-object-values-method) | This method returns an array of values. |

|  |  |  |
| --- | --- | --- |
| Point of comparison | Regular function | Arrow function |
| this value | Inside of a regular JavaScript function, this value (aka the execution context) is dynamic.  The dynamic context means that the value of this depends on how the function is invoked. In JavaScript, there are 4 ways you can invoke a regular function.   1. During a simple invocation the value of this equals to the global object (or undefined if the function runs in strict mode): 2. During a method invocation the value of this is the object owning the method: 3. During an indirect invocation using myFunc.call(thisVal, arg1, ..., argN) or myFunc.apply(thisVal, [arg1, ..., argN]) the value of this equals to the first argument 4. During a constructor invocation using new keyword this equals to the newly created instance | The behavior of this inside of an arrow function differs considerably from the regular function's this behavior. The arrow function doesn't define its own execution context.  No matter how or where being executed, this value inside of an arrow function always equals this value from the outer function. In other words, the arrow function resolves this lexically.  In the following example, myMethod() is an outer function of callback() arrow function  this value inside the arrow function callback() equals to this of the outer function myMethod().  this resolved lexically is one of the great features of arrow functions. When using callbacks inside methods you are sure the arrow function doesn't define its own this: no more const self = this or callback.bind(this) workarounds.  Contrary to a regular function, the indirect invocation of an arrow function using myArrowFunc.call(thisVal) or myArrowFunc.apply(thisVal) doesn't change the value of this: the context value is always resolved lexically. |
| Constructors | As seen in the previous section, the regular function can easily construct objects. | A consequence of this resolved lexically is that an arrow function cannot be used as a constructor. |
| arguments object | nside the body of a regular function, arguments is a special array-like object containing the list of arguments with which the function has been invoked. | no arguments special keyword is defined inside an arrow function.  Again (same as with this value), the arguments object is resolved lexically: the arrow function accesses arguments from the outer function. |
| Implicit return | return expression statement returns the result from a function  If the return statement is missing, or there's no expression after return statement, the regular function implicitely returns undefined | You can return values from the arrow function the same way as from a regular function, but with one useful exception.  If the arrow function contains one expression, and you omit the function's curly braces, then the expression is implicitly returned. These are [the inline arrows function](https://dmitripavlutin.com/javascript-arrow-functions-best-practices/#2-inline-when-possible). |
| Methods | The regular functions are the usual way to define methods on classes.  Usually, the regular functions as methods are the way to go. | Thanks to [Class fields proposal](https://github.com/tc39/proposal-class-fields) (at this moment at stage 3) you can use the arrow function as methods inside classes.  Now, in contrast with regular functions, the method defined using an arrow binds this lexically to the class instance. |

|  |  |
| --- | --- |
| **Object** | **Instance** |
| 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. | 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). |