Check out following website for icons & images

<https://www.flaticon.com/search?word=arrow>

Explainn code :

window.addEventListener("keydown", (e) => {

overlay.classList.add("hide");

displayKey.innerHTML = e.key;

displayKeyCode.innerText = e.keyCode;

if (e.keyCode === 32) {

displayKey.innerHTML = `'space'`;

}

});

window.addEventListener("keydown", (e)

The e is the event object, and it holds all the information about the specific event that was triggered—in this case, a keydown event.

overlay.classList.add("hide");

When any key is pressed, it adds a class called "hide" to that element.

Likely, in CSS, .hide sets display: none or visibility: hidden, effectively hiding the overlay.

e.key always returns a string then Why don’t we see quotes around strings in the output?

Because innerText or innerHTML just prints the value as it is, without quotes, unless you add them yourself.

If e.key is "a" — it shows a, not "a" — because it's just printing the string's content, not its data type.

Quotes don’t appear in the output unless you include them

Output is for humans, not for JS type checking — browser won't show "a" unless you code it to

This concept applies everywhere

Strings in JavaScript are wrapped in quotes

But when displayed on a webpage (using .innerText or .innerHTML), the browser doesn't show the quotes

It only shows the text content, not the data type

why did we applied event on window object ?

You're telling the browser:

"Hey, no matter where the user is focused on the page — if they press a key, I want to catch it!"

This is super useful when:

You're building keyboard shortcuts

You want to respond to keys regardless of where the cursor is

You're creating games, modals, or overlays

So in your case, applying it to window ensures it fires no matter where the user is focused on the page — perfect for keyboard-based interactions.

Key presses can be caught even if nothing is focused.

displayKey.innerHTML = e.key;

displayKeyCode.innerText = e.keyCode;

why did we used innerhtml for displaykey but innertext for key code

You wanted the word 'space' to appear with single quotes around it. Using innerHTML ensures those quotes are rendered exactly, not escaped or interpreted differently.

Because e.keyCode is just a number (like 65, 13, 32) — plain text. You're not expecting or inserting any HTML here.

innerText is safer and better for raw values that don’t need formatting or tags.

Use:

innerText when you're inserting just plain text.

innerHTML when you're inserting HTML content or want more control over appearance (like bold, italic, line breaks, etc).

See the difference

let name = "<b>Alice</b>";

someDiv.innerHTML = name; // Will display: Alice (in bold)

someDiv.innerText = name; // Will display: <b>Alice</b> (as text)

why did we used this if (e.keyCode === 32) {

displayKey.innerHTML = `'space'`;

}

When someone presses the spacebar, e.key usually returns a space character:

**e.key === " "**

This would look empty or invisible if we just did:

**displayKey.innerHTML = e.key; // Shows nothing on the screen**

32 is the key code for the spacebar.

Since the space character itself doesn’t visibly show up, we handle it as a special case.

This way, instead of showing a blank space, we clearly display the word 'space' in the output — which is much more understandable for the user.

why did we used === rather than ==

e.keyCode is a number (32)

So we're comparing it to another number (32)

Using === ensures no type conversion happens — it’s a safe, accurate check

=== (Strict Equality)

Compares both value and data type

If the values are not of the same type, it returns false — no automatic conversion.

⚠️ == (Loose Equality)

Converts the values to the same type before comparing

Can lead to unexpected results due to type coercion

explain each line and terms of the following code snippet

keyCodeDiv.addEventListener("click", (e) => {

const textArea = document.createElement("textarea");

textArea.setAttribute("readonly", "");

textArea.style.position = "absolute";

textArea.value = keyCodeDiv.querySelector("h2").innerText;

document.body.appendChild(textArea);

textArea.select();

document.execCommand("copy");

document.body.removeChild(textArea);

keyCodeDiv.querySelector("p").innerText = "Copied";

setTimeout(() => {

keyCodeDiv.querySelector("p").innerText = "Click to Copy";

}, 2000);

});

const textArea = document.createElement("textarea");

Creating the <textarea> is kind of like using a temporary variable (x) in math just to help you calculate something, even though it's not part of the final result. It acts as an intermediary — a temporary helper to get the job done (copying the text) — and then it's removed because it's no longer needed.

Browsers generally only allow copying to clipboard from user-selected text. So when you want to programmatically copy some text (like the content of an <h2>), you need to:

* Put that text inside an input or textarea (because they allow selection).
* Select the text inside that field.
* Run the copy command.

So the <textarea> is created only to:

* Hold the text temporarily.
* Be selected.
* Be used as a source for copying.

elements like <p> and <h2> don’t support the kind of text selection that .execCommand("copy") requires. That method expects a selection to be inside a form element (<input> or <textarea>), not just any text.

So while we could show and store the text in <p> or <h2>, we can't copy directly from them using .execCommand("copy").

the user never sees that <textarea> on the screen, because it's added temporarily and hidden from view.

position: absolute removes it from the normal page layout and allows us to place it anywhere on the screen — even off-screen.

By default, an absolutely positioned element with no top, left, width, etc.:

Isn’t placed anywhere visible.

Has no content shown

By default elemment will be in top left corner

So the textarea appears visually at the top-left corner of the screen — although very briefly.

Gets removed quickly after copying, so even if it flashed for a millisecond, it wouldn’t stay.

**Want to make it truly invisible (even safer)?**

Some devs also do this to make sure the textarea is completely hidden:

textArea.style.left = "-9999px";

This pushes the textarea far off-screen, making it completely invisible but still focusable and selectable for copying.

So if you want to be extra careful and ensure no flicker

textArea.style.position = "absolute";

textArea.style.left = "-9999px";

why did we used this attribute and also did not set any value for it textArea.setAttribute("readonly", "");

We set the textarea as readonly to prevent the user (or even some browser extensions or UI behaviors) from editing the text while it's selected for copying.

In HTML, boolean attributes (like readonly, disabled, checked, etc.) don’t need a value.

Just their presence is enough to activate them.

In html following will also work

<textarea readonly></textarea>

<textarea readonly="true"></textarea>

"" = an empty string is fine for boolean attributes in HTML. It's just telling the browser: “This attribute exists.”

**Elements where readonly is valid:**

It only works on form elements that accept user input and can display text but are not meant to be edited.

Supported elements:

<input> (but only with certain types like text, email, number, etc.)

<textarea>

**It won’t have any effect on elements like:**

<p>, <h1>, <div> (these aren’t form fields)

<input type="button"> or <input type="checkbox">

<select> or <option>

**If you don’t set the textarea to readonly, here’s what might happen:**

1. Browser auto-corrects or modifies the text

2. If the textarea is temporarily visible for even a split second (slow device or laggy browser), the user might click or tap it and unintentionally change something before it gets copied.

If the textarea is invisible, why even bother appending it to the page?

document.body.appendChild(textArea);

Browsers can only select or copy content from elements that are part of the DOM (i.e., on the page).

Even if the textarea is invisible, it must still be:

✅ Inserted into the page (the DOM)

❌ Just existing in JavaScript memory isn't enough

textArea.select();

This selects (highlights) all the text inside the textarea.

It's like the browser doing: "CTRL + A" inside that box.

You must select the text before you can copy it.

document.execCommand("copy");

This tells the browser: "Copy whatever is currently selected."

It works like pressing "CTRL + C" manually.

Since the text inside textArea is selected, that’s what gets copied.

keyCodeDiv.addEventListener("click", async (e) => {

const textToCopy = keyCodeDiv.querySelector("h2").innerText;

try {

await navigator.clipboard.writeText(textToCopy);

keyCodeDiv.querySelector("p").innerText = "Copied";

} catch (err) {

console.error("Failed to copy!", err);

keyCodeDiv.querySelector("p").innerText = "Failed to Copy";

}

setTimeout(() => {

keyCodeDiv.querySelector("p").innerText = "Click to Copy";

}, 2000);

});

As exec-command is deprecated we will replace the code with this one , which removes the need of extra element ‘textarea’

why did we inserted keyword async before call back function ?

We added async before the callback function because inside it we are using await.

navigator.clipboard.writeText(textToCopy) is a Promise — it’s asynchronous.

To use await, the function must be declared as async.

await tells JavaScript: "Pause here until the Promise resolves, then continue."

why did we used try and catch block ?

We used try and catch because when you use await, if something goes wrong (an error happens), it throws an error immediately.

try lets us attempt to run the code.

catch lets us handle the error if it fails — without crashing the app.

So it portrays : Try to copy. If it fails, catch the error and show a nice message instead of crashing.

what action do this line performs , explain each term in depth navigator.clipboard.writeText(textToCopy)

Navigator is an object.

Inside the navigator object, clipboard is another object ( as a property of navigator )

Inside clipboard object there

clipboard has methods like writeText and readText?

Just like any object can have properties and methods

clipboard.writeText(text)

It copies the given text to the clipboard.

After calling this, if the user does Ctrl+V, they can paste the text you copied.

It returns a Promise (so you can use .then() or await).

clipboard.readText()

It reads (or fetches) the current text from the clipboard.

It also returns a Promise (because accessing the clipboard can take time or need permission).

why we used navigator , can not we used clipboard only ?

Clipboard is not a standalone thing available directly in JavaScript.

It’s a part (a property) of the Navigator object.

Clipboard API is inside the Navigator for security reasons and organization.

navigator object

It's a property inside the window object.

It's specifically responsible for giving information about:

The browser itself (name, version, platform)

Whether the device is online/offline

Managing clipboard access (via Clipboard API)

Accessing Bluetooth devices

Handling Push Notifications

Many other browser/device features

✅ Think of navigator like "browser information and capability manager."

window object

In a browser, the window object is the top-level global object.

It represents the browser window or tab itself.

Almost everything you use (like document, console, navigator, alert, setTimeout) — lives inside the window object.

When navigator.clipboard.writeText(text) is called:

And promise is resolved , we sucessfully copied value

It’s a void Promise — meaning it doesn’t return any specific value.

No value is returned by writeText, so .then() just runs after the operation completes.

When promise rejects

It is rejected with an error object if something goes wrong ( like the user hasn't granted permission to access the clipboard, or you're trying to run this on an insecure page (HTTP instead of HTTPS) — the Promise is rejected.), and the catch block handles the error, giving you useful details about why it failed.

what will be the output look like , what will error replaced with in case page is insecure console.log("Failed to copy text:", error);

Failed to copy text: SecurityError: Clipboard API is only available in secure contexts (HTTPS)

error.name → "SecurityError"

name and message are properties of the error object.

You can access them separately, together, or only the one you need.

error.message → "Clipboard API is only available in secure contexts (HTTPS)"

So the error is actually an Error object (specifically a SecurityError).

When you print it, the browser shows it like:

SecurityError: Clipboard API is only available in secure contexts (HTTPS)

Why do we prefer console.error instead of traditional console.log when logging errors?

console.log → normal text, looks boring.

console.error → big red scary error with stack trace (easy to notice).console.error tells the browser and tools "this is an ERROR", not just random text.

console.error vs console.log

console is an object.

Console has several methods inside it:

console.log() → normal logging (gray text).

console.error() → error logging (red, scary).

console.warn() → warning logging (yellow/orange).

console.info() → informational logging (sometimes blue).

Console is an object.

It has different methods like log, error, warn, etc.

These methods print values differently based on their purpose.

console.error("Failed to copy!", err);

→ Only shows the error in the browser’s console (Inspect → Console tab).

→ It does not show anything to users on the actual web page (UI).

background-color: transparent !important;

box-shadow: none !important;

padding: 0 100px !important;

cursor: default !important;

Why did we used important keyword ?

background-color, box-shadow, and padding were already set for .container div because .arrow is inside the same .container.

.arrow is a special div — it’s not supposed to look like the other clickable boxes. ✅ Normally, the .container div rules would affect .arrow too (because .arrow is a div inside .container), but you don't want that.

So you add !important to make sure .arrow ignores the default .container div styles.

In short:

Without !important, .arrow would get a grey background, box-shadow, and clickable pointer style like other boxes.

With !important, .arrow stays transparent, no shadow, no clickable hand cursor.

Normal CSS cannot beat an earlier !important. Only another !important can override a previous !important.

!important is a CSS keyword used to give higher priority to a style rule, overriding normal specificity and source order.

filter: drop-shadow() adds a shadow effect to only the visible parts of an element (like images, icons, text).

It’s different from box-shadow, which adds a shadow around the whole box of the element (including transparent areas).

filter: drop-shadow(offset-x offset-y blur-radius color);

Why is filter useful?

✅ You don't need to edit images in Photoshop — you can add effects with just CSS.

✅ It makes UI animations, hovers, and transitions look professional and dynamic.

✅ It works on images, videos, backgrounds, and even text sometimes!

The filter property adds visual effects (like blur, brightness, shadows) to elements directly using CSS.

Multiple filters can be applied in one filter property

img {

filter: blur(5px) brightness(150%) grayscale(50%);

}

Filter types :

Filter Function | What it does

blur(px) | Makes the element blurry

brightness(%) | Makes the element brighter or darker

contrast(%) | Increases or decreases contrast

grayscale(%) | Converts to black & white

sepia(%) | Gives a brownish old-photo effect

drop-shadow(x, y, blur, color) | Adds a shadow around the visible parts

The filter property in CSS applies graphical effects like blurring, brightness, contrast, shadows, etc., to an element (usually images, backgrounds, or any visual content).

It visually changes how an element looks without modifying the actual image or content.

We applied position: fixed to the .overlay so that it stays on top of everything and covers the entire screen, no matter how much the user scrolls.

**Why use it?**

To create a fullscreen effect, like:

A loading screen

A welcome message

A modal or popup background

A lock screen to hide content until a user clicks something

It removes the element from normal layout flow, so it won't push or affect nearby elements.