2.5 The most important JavaScript concepts you need first □□

Before GSAP becomes intuitive, you need a small set of JavaScript ideas that appear in *almost every animation setup*. The good news: you do **not** need to become a full JavaScript developer first. You mainly need to understand:

1. what JavaScript does in the browser,
2. how to store values,
3. how to group actions into functions,
4. how to find elements on the page,
5. how to react to user actions or browser actions,
6. and how to make sure your code runs at the right time.

This section is designed specifically for your use case: **WordPress + BricksBuilder + future GSAP/Glaze work**.

2.5.1 What JavaScript does in the browser

JavaScript is the language that gives a web page *behavior*.

If HTML is the **structure** of the page and CSS is the **appearance**, then JavaScript is the **logic and interaction layer**.

A simple way to think about it

Imagine a website as a stage:

- **HTML** creates the stage, actors, and props

- **CSS** decides how everything looks
- **JavaScript** tells things *when to move, change, react, appear, disappear, animate, or respond*

GSAP itself is a JavaScript animation library. That means GSAP does not replace JavaScript—it works *through* JavaScript.

What JavaScript can do in the browser

JavaScript can:

1. **Find elements on the page**
   - Example: find a heading, button, image, section, or card
2. **Change content**
   - Example: update text, numbers, labels
3. **Change styles**
   - Example: set opacity, width, color, transforms
4. **Add or remove classes**
   - Example: activate a menu, highlight a selected tab
5. **React to events**
   - Example: click, hover, scroll, page load, resize
6. **Run animations**
   - Example: fade in a section, move a box, stagger cards
7. **Read information from the page**
   - Example: current scroll position, element size, form values
8. **Coordinate timing**
   - Example: wait 1 second, then animate another element

Why this matters for GSAP

When you write GSAP code, you usually do something like this:

1. select an element,
2. wait until the page is ready,
3. then animate that element.

That means GSAP code is often sitting inside regular JavaScript structure.

For example:

```
<script>
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
</script>
```

Even if the animation itself is simple, there are already several JavaScript concepts involved:

- `gsap.to(...)` is a **function call**
- `".box"` is a **selector string**
- `{ x: 200, duration: 1 }` is an **object** containing settings

You do not need to master every advanced JavaScript topic right away. But you do need to become comfortable reading this kind of code.

2.5.2 Variables in simple terms

A **variable** is a named container for a value.

Think of it like a labeled box.

- You give the box a name
- You put a value into it
- Later, you can use that value again

Why variables matter

In animation work, variables help you store:

- selected elements
- distances
- durations
- colors
- settings
- values you want to reuse

Instead of writing the same thing again and again, you store it once.

Simple example

```
let name = "Peter";
```

This means:

- create a variable called `name`
- store the text `"Peter"` inside it

Then you can use it:

```
let name = "Peter";
console.log(name);
```

Common kinds of values

1. Text

```
let title = "Welcome";
```

Text values use quotes.

2. Numbers

```
let duration = 1.5;
let distance = 200;
```

Numbers do **not** use quotes.

3. True or false

```
let menuOpen = true;
let animationFinished = false;
```

These are called **booleans**.

4. Elements

You can also store HTML elements in variables:

```
let button = document.querySelector(".my-button");
```

Now `button` refers to the element found on the page.

This is extremely common in GSAP setups.

`let`, `const`, and `var` in simple language

You will mostly see three ways of creating variables:

- `let`
- `const`
- `var`

For modern web work, focus mainly on:

1. **`let`**
   - use this when the value may change
2. **`const`**
   - use this when the value should stay the same
3. **`var`**
   - older style; you may still see it, but usually prefer `let` or `const`

Example with `let`

```
let count = 1;
count = 2;
```

This is allowed because `let` variables can be reassigned.

Example with `const`

```
const title = "Hello";
```

This is good when the variable should not point to a new value later.

For example, if you select an element and do not plan to replace it, `const` is often a good choice:

```
const box = document.querySelector(".box");
```

A practical animation-style example

```
const box = document.querySelector(".box");
const moveDistance = 200;
```

```
const animationTime = 1;

gsap.to(box, {
  x: moveDistance,
  duration: animationTime
});
```

Here:

- `box` stores the selected element
- `moveDistance` stores how far it should move
- `animationTime` stores the duration

This makes your code easier to read.

Why variables are useful in BricksBuilder projects

When working in Bricks, you often have repeated sections, reusable classes, and structured page parts. Variables help you avoid messy code.

For example:

```
const cards = document.querySelectorAll(".feature-card");
const duration = 0.8;
```

Now you can reuse those values later.

This becomes especially helpful when:

- testing animations,
- changing settings,
- adjusting timing,
- targeting multiple sections.

Naming variables well

Good variable names make animation code much easier to understand.

Bad names

```
let x = 200;
let a = document.querySelector(".box");
```

These are vague.

Better names

```
const box = document.querySelector(".box");
const moveDistance = 200;
```

Now the meaning is obvious.

Useful naming tips

1. use names that describe the value
2. use English words consistently
3. avoid very short names unless the meaning is obvious
4. use `camelCase`

Examples of `camelCase`:

- `heroTitle`
- `scrollDistance`
- `animationDuration`

A small mental model

When you see code like this:

```
const heading = document.querySelector(".hero-title");
```

read it in plain language as:

> “Create a constant called `heading` and store the page element with class `.hero-title` inside it.”

That is enough understanding for now.

2.5.3 Functions in simple terms

A **function** is a reusable block of code that performs a task.

Think of a function as a **named instruction set**.

Instead of writing the same steps again and again, you put them into a function and run it when needed.

Why functions matter for animation

Functions are used constantly in animation work:

- start an animation
- reset an animation
- animate all cards
- react to a click
- run something after the page loads
- handle scroll or resize logic

Simple function example

```
function sayHello() {
  console.log("Hello");
}
```

This *defines* the function.

But it does not run yet.

To run it, you call it:

```
sayHello();
```

How to read this

```
function sayHello() {
  console.log("Hello");
}
```

means:

- create a function named `sayHello`
- when this function runs, log `"Hello"`

A more practical example

```
function animateBox() {
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
}
```

This creates a function called `animateBox`.

It will animate `.box` when called:

```
animateBox();
```

This is useful because you can now trigger that same animation:

- on page load,
- on click,
- after a delay,
- or in response to some other event.

Functions can take input

Sometimes a function should work with different values. That is where **parameters** come in.

```
function moveBox(distance) {
  gsap.to(".box", {
    x: distance,
    duration: 1
```

```
  });
}
```

Now you can call it with different values:

```
moveBox(100);
moveBox(300);
```

How to understand this

- `distance` is a placeholder
- when the function runs, the actual value replaces it

This is powerful because it makes your code reusable.

Why functions help you avoid repetition

Without functions:

```
gsap.to(".box", { x: 200, duration: 1 });
gsap.to(".box", { x: 200, duration: 1 });
gsap.to(".box", { x: 200, duration: 1 });
```

With a function:

```
function animateBox() {
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
}

animateBox();
animateBox();
animateBox();
```

You define the logic once and reuse it.

Functions and events often work together

A very common pattern is:

1. define a function,
2. attach it to an event.

Example:

```
function openMenuAnimation() {
  gsap.to(".menu", {
    opacity: 1,
    y: 0,
    duration: 0.5
  });
}

document.querySelector(".menu-button").addEventListener("click", openMenuAnimation);
```

This means:

- when the button is clicked,
- run `openMenuAnimation`

This pattern is everywhere in JavaScript and GSAP work.

Function expressions and arrow functions

You may also see functions written in different styles.

Function declaration

```
function animateBox() {
  console.log("animate");
}
```

Function expression

```
const animateBox = function() {
  console.log("animate");
};
```

Arrow function

```
const animateBox = () => {
  console.log("animate");
};
```

For now, you do not need to deeply worry about the differences. Just know that they all define functions.

In modern examples, especially with event listeners and shorter code, arrow functions are common.

Example:

```
document.querySelector(".button").addEventListener("click", () => {
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
});
```

This means:

- find `.button`
- when it is clicked
- run this code

A beginner-friendly recommendation

At your current stage, try to become comfortable with:

1. regular functions

   ```
   function animateBox() {
     gsap.to(".box", {
   ```

```
    x: 200,
    duration: 1
  });
}
```

2. simple arrow functions in event listeners

```
button.addEventListener("click", () => {
  animateBox();
});
```

That is enough to build a strong foundation.

2.5.4 Selecting elements from the page

Before you can animate something, JavaScript must be able to **find it**.

This is one of the most important concepts in all browser-based animation work.

If JavaScript cannot correctly select the element, GSAP cannot animate it.

The DOM in simple terms

When a browser loads your page, it turns the HTML into a structured representation called the **DOM**.

DOM stands for **Document Object Model**.

You do not need the full technical definition. Just think:

> The DOM is JavaScript's way of seeing and working with the page.

So if your HTML contains:

```
<h1 class="hero-title">Welcome</h1>
```

JavaScript can find that element and do things to it.

The most common selection methods

The most important ones for you are:

1. `document.querySelector()`
2. `document.querySelectorAll()`

These two are enough for a lot of real-world work.

`querySelector()`

This finds the **first matching element**.

Example:

```
const title = document.querySelector(".hero-title");
```

This means:

- look through the document
- find the first element with class `hero-title`
- store it in `title`

You can then use it:

```
gsap.to(title, {
  y: 50,
  opacity: 0,
  duration: 1
});
```

`querySelectorAll()`

This finds **all matching elements**.

Example:

```
const cards = document.querySelectorAll(".card");
```

Now `cards` is a collection of all elements with the class `.card`.

This is very useful for repeated elements like:

- cards
- menu items
- testimonials
- gallery items
- sections

GSAP can often work directly with multiple elements:

```
gsap.from(".card", {
  opacity: 0,
  y: 30,
  duration: 0.8,
  stagger: 0.2
});
```

or with the selected collection:

```
gsap.from(cards, {
  opacity: 0,
  y: 30,
  duration: 0.8,
  stagger: 0.2
});
```

How selectors work

Selectors in JavaScript use the same logic you may know from CSS.

Select by class

```
document.querySelector(".box");
```

The dot `.` means class.

Select by ID

```
document.querySelector("#main-button");
```

The hash `#` means ID.

Select by tag

```
document.querySelector("section");
```

This selects the first `<section>`.

More specific selector

```
document.querySelector(".hero .button");
```

This means:
find a `.button` inside `.hero`.

Why classes are often best for animation

In WordPress and BricksBuilder, classes are usually the best animation targets because:

1. they are reusable,
2. they fit well with component-based building,
3. they are cleaner than relying on deeply nested structures,
4. they are easier to maintain.

For example, instead of selecting:

```
document.querySelector("section div div h2");
```

it is better to assign a clear class like:

```
<h2 class="feature-title">Fast performance</h2>
```

and then use:

```
document.querySelector(".feature-title");
```

This is much more reliable.

What happens if nothing is found

Sometimes the selector does not match anything.

Example:

```
const box = document.querySelector(".box");
console.log(box);
```

If there is no `.box`, the result is `null`.

That matters because if you then try to use it incorrectly, errors can happen.

A safer pattern is:

```
const box = document.querySelector(".box");

if (box) {
  gsap.to(box, {
    x: 200,
    duration: 1
  });
}
```

This means:

- if `box` exists,
- run the animation

This is especially useful in WordPress where not every page contains the same elements.

Selecting multiple elements in real projects

Example HTML:

```
<div class="card">Card 1</div>
<div class="card">Card 2</div>
```

```
<div class="card">Card 3</div>
```

JavaScript:

```
const cards = document.querySelectorAll(".card");
console.log(cards);
```

This gives you all three cards.

With GSAP:

```
gsap.from(cards, {
  opacity: 0,
  y: 40,
  duration: 0.8,
  stagger: 0.15
});
```

This is a typical pattern for animating repeated UI elements.

Selecting inside a specific section

Sometimes you do not want all matching elements across the entire page. You only want those inside a specific area.

Example:

```
const section = document.querySelector(".pricing-section");
const buttons = section.querySelectorAll(".button");
```

This means:

- first find `.pricing-section`
- then inside it find all `.button` elements

This is helpful for more precise control.

2.5.5 Events in simple terms

An **event** is something that happens in the browser.

JavaScript can listen for events and react to them.

This is the basis of interactive websites.

Common events

Some very common events are:

1. `click`
2. `mouseenter`
3. `mouseleave`
4. `scroll`
5. `resize`
6. `submit`
7. `DOMContentLoaded`

For animation work, `click`, `hover`, `scroll`, and page-load events are especially important.

Event example: click

Suppose you have a button:

```
<button class="animate-button">Animate</button>
<div class="box"></div>
```

You can listen for a click:

```
const button = document.querySelector(".animate-button");

button.addEventListener("click", () => {
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
});
```

How to read this

- find the button
- listen for a `click`
- when clicked, run the code inside

This is one of the most common JavaScript patterns you will ever use.

What `addEventListener()` means

This method tells the browser:

> “When this specific thing happens to this specific element, run this code.”

Basic structure:

```
element.addEventListener("eventName", functionToRun);
```

Example:

```
button.addEventListener("click", animateBox);
```

or:

```
button.addEventListener("click", () => {
  animateBox();
});
```

Hover-style events

For hover-related interactions, you might use:

- `mouseenter`
- `mouseleave`

Example:

```
const card = document.querySelector(".card");

card.addEventListener("mouseenter", () => {
```

```
  gsap.to(card, {
    y: -10,
    duration: 0.3
  });
});

card.addEventListener("mouseleave", () => {
  gsap.to(card, {
    y: 0,
    duration: 0.3
  });
});
```

This creates a simple hover motion.

Scroll events

JavaScript can also react to scrolling.

Example:

```
window.addEventListener("scroll", () => {
  console.log("Scrolling");
});
```

However, for many scroll-based animations, **GSAP ScrollTrigger** is usually a better and more powerful solution than manually handling scroll events yourself.

Still, understanding that scroll is just another event is useful.

Form submit events

If a form is submitted, JavaScript can react:

```
const form = document.querySelector(".contact-form");

form.addEventListener("submit", () => {
  console.log("Form submitted");
```

```
});
```

This is less central to GSAP itself, but it helps you understand the broader event model.

The event object

Sometimes the browser provides information about the event.

Example:

```
button.addEventListener("click", (event) => {
  console.log(event);
});
```

The `event` object can contain details about what happened.

For your current animation learning stage, you do not need to go deep into this yet. Just know that it exists and is sometimes useful.

Why events matter so much for GSAP

A lot of animation logic is event-driven:

1. on page load, animate the hero
2. on click, open the menu
3. on hover, lift a card
4. on scroll, reveal sections
5. on resize, recalculate animation values

So even though GSAP is the animation engine, JavaScript events often decide *when* animations begin.

2.5.6 Running code after the page is loaded

This is a very important concept, especially for beginners.

Sometimes JavaScript tries to select or animate elements **before they exist in the DOM**. If that happens, your code may fail or behave unpredictably.

So you often need to wait until the page is ready enough before running your code.

Why timing matters

Suppose your script runs too early:

```
const box = document.querySelector(".box");
console.log(box);
```

If `.box` has not been parsed yet, `box` may be `null`.

Then animation code based on it will not work.

The most common solution: `DOMContentLoaded`

A very common pattern is:

```
document.addEventListener("DOMContentLoaded", () => {
  const box = document.querySelector(".box");

  if (box) {
    gsap.to(box, {
      x: 200,
      duration: 1
    });
  }
});
```

What this means

- wait until the HTML has been loaded and parsed
- then run the code

This is often enough for many animation setups.

Why this is useful in WordPress and BricksBuilder

In WordPress page builders, content is often generated dynamically through the builder and template structure. That means you want to be careful that your JavaScript runs only after the relevant markup is available.

Using `DOMContentLoaded` gives you a safer starting point.

Difference between page loaded and DOM ready

There is an important distinction:

`DOMContentLoaded`

Runs when the HTML is parsed and the DOM is ready.

Full page load

This waits for additional resources too, such as:

- images
- stylesheets
- iframes
- other external assets

A full page load listener looks like this:

```
window.addEventListener("load", () => {
  console.log("Everything is fully loaded");
});
```

When to use which

Use `DOMContentLoaded` when

- you mainly need the HTML structure to exist
- you want to select elements and start many normal animations

Use `load` when

- your animation depends on exact image sizes
- you need all resources fully loaded first
- layout calculations require complete assets

For many beginner GSAP setups, `DOMContentLoaded` is the better default.

Another common approach: putting scripts at the end of the page

Traditionally, developers often placed scripts just before `</body>` so the HTML would already be loaded when the script ran.

Example:

```
<body>
  <div class="box"></div>

  <script>
    const box = document.querySelector(".box");

    gsap.to(box, {
      x: 200,
      duration: 1
    });
  </script>
</body>
```

This can work because the `.box` already exists by the time the script runs.

However, in WordPress and Bricks contexts, scripts may be injected through theme settings, custom code areas, or enqueued files, so relying only on script placement is not always the clearest or safest method for a beginner. Using an explicit load-ready pattern is often easier to reason about.

A good beginner pattern

This is a solid foundational structure:

```
document.addEventListener("DOMContentLoaded", () => {
  const heroTitle = document.querySelector(".hero-title");
  const heroButton = document.querySelector(".hero-button");

  if (heroTitle) {
    gsap.from(heroTitle, {
      y: 40,
      opacity: 0,
      duration: 1
    });
  }

  if (heroButton) {
    gsap.from(heroButton, {
      y: 20,
      opacity: 0,
      duration: 1,
      delay: 0.3
    });
  }
});
```

This is good because it:

1. waits for the DOM,
2. selects elements,
3. checks if they exist,
4. animates them safely.

This pattern is very relevant for real websites.

Putting it all together

Now let us combine all six concepts into one small example.

Example HTML

```
<button class="animate-button">Animate Box</button>
<div class="box"></div>
```

Example JavaScript

```
document.addEventListener("DOMContentLoaded", () => {
  const button = document.querySelector(".animate-button");
  const box = document.querySelector(".box");
  const moveDistance = 200;

  function animateBox() {
    gsap.to(box, {
      x: moveDistance,
      duration: 1
    });
  }

  if (button && box) {
    button.addEventListener("click", () => {
      animateBox();
    });
  }
});
```

What is happening here

1. **Running after page load**
   - The code waits for `DOMContentLoaded`
2. **Selecting elements**
   - It finds the button and the box
3. **Variables**
   - It stores elements and the distance in variables
4. **Functions**
   - It creates a function called `animateBox`
5. **Events**
   - It listens for a click on the button
6. **Browser behavior**

- When clicked, JavaScript runs the animation through GSAP

This is a very realistic mini-pattern that mirrors how many real interactive animations are built.

A GSAP-oriented mental framework

As you continue learning, try to read many animation setups using this simple formula:

1. **Wait** until the page is ready
2. **Select** the element or elements
3. **Store** what you need in variables
4. **Write** a function if the logic should be reusable
5. **Trigger** it with an event or run it immediately
6. **Animate** with GSAP

That is the basic rhythm of a lot of frontend animation work.

Common beginner mistakes to watch for ⚠

1. Trying to animate an element that does not exist

```
const box = document.querySelector(".box");
gsap.to(box, { x: 200 });
```

If `.box` is missing, this is a problem.

Better:

```
if (box) {
  gsap.to(box, { x: 200 });
}
```

---

2. Forgetting the dot for class selectors

Wrong:

```
document.querySelector("box");
```

Correct:

```
document.querySelector(".box");
```

---

3. Using quotes around numbers accidentally

Not ideal:

```
let duration = "1";
```

Better:

```
let duration = 1;
```

For animation settings, numbers should usually be actual numbers.

---

4. Defining a function but forgetting to call it

```
function animateBox() {
  gsap.to(".box", { x: 200 });
```

```
}
```

This does nothing until you call:

```
animateBox();
```

---

5. Running code too early

If the DOM is not ready, selectors may fail.

Safer:

```
document.addEventListener("DOMContentLoaded", () => {
  // your code here
});
```

What you should be able to do after this section

After understanding this section, you should be able to:

1. explain what JavaScript does on a webpage,
2. read and create simple variables,
3. understand simple functions,
4. select elements with `querySelector` and `querySelectorAll`,
5. attach a click event with `addEventListener`,
6. safely run JavaScript after the DOM is ready.

That is already a *very important* foundation for GSAP.

Mini cheat sheet □□

Select one element

```
const box = document.querySelector(".box");
```

Select many elements

```
const cards = document.querySelectorAll(".card");
```

Create a variable

```
const distance = 200;
```

Create a function

```
function animateBox() {
  gsap.to(".box", {
    x: 200,
    duration: 1
  });
}
```

Run a function on click

```
button.addEventListener("click", animateBox);
```

Run code after DOM is ready

```
document.addEventListener("DOMContentLoaded", () => {
  // code here
});
```

Why this matters specifically for Glaze too

Even though **Glaze** simplifies animation setup, the same browser fundamentals still matter.

You still need to understand:

1. what element you are targeting,
2. when the page is ready,
3. what event may trigger something,
4. how JavaScript connects behavior to markup.

So learning these basics is not “extra theory”—it directly supports both:

- **GSAP as the powerful engine**
- **Glaze as the easier abstraction**

Revision #1
Created 2026-04-25 21:11:02 UTC by art10m
Updated 2026-04-25 21:18:41 UTC by art10m