Objects VS Maps

In JavaScript, both Objects and Maps are used to store key-value pairs, but they have distinct differences in their capabilities and typical use cases.

Key Differences:

Key Types:

- **Objects:** Keys are limited to strings or Symbols. If a non-string or non-Symbol value is used as a key, it will be implicitly converted to a string.
- **Maps:** Keys can be of any data type, including objects, functions, numbers, booleans, and more. This provides greater flexibility.

Iteration Order:

- **Objects:** The order of keys is not guaranteed in older JavaScript versions, though modern engines generally preserve insertion order for non-integer keys.
- **Maps:** Maintain the insertion order of their elements, meaning iteration will always follow the order in which elements were added.

Iteration Methods:

- **Objects:** Require methods like `Object.keys()`, `Object.values()`, or `Object.entries()` to iterate over their contents.
- **Maps:** Provide built-in iteration methods like `map.keys()`, `map.values()`, and `map.entries()`, which return iterators.

Default Keys/Prototype Chain:

- **Objects:** Inherit from `Object.prototype`, which can introduce accidental keys or collisions if not handled carefully (e.g., by using `Object.create(null)`).
- **Maps:** Do not have a prototype chain and only contain the keys explicitly added to them, making them cleaner for key-value storage without unintended inheritance.

Performance (at scale):

- **Maps:** Are generally optimized for frequent additions and deletions, and can offer better performance than Objects when dealing with a large number of entries and high churn.

When to Use Which:

Use Objects for:

- Simple data structures where keys are primarily strings and order is not critical.
- When working with JSON, as Objects directly map to JSON structures.
- When you need to define methods or properties directly on the data structure.

Use Maps for:

- Storing key-value pairs where keys can be of any data type.
- When the order of elements needs to be preserved.
- When frequent additions, deletions, or iterations are expected, especially with a large number of entries.
- To avoid potential key collisions from the prototype chain.