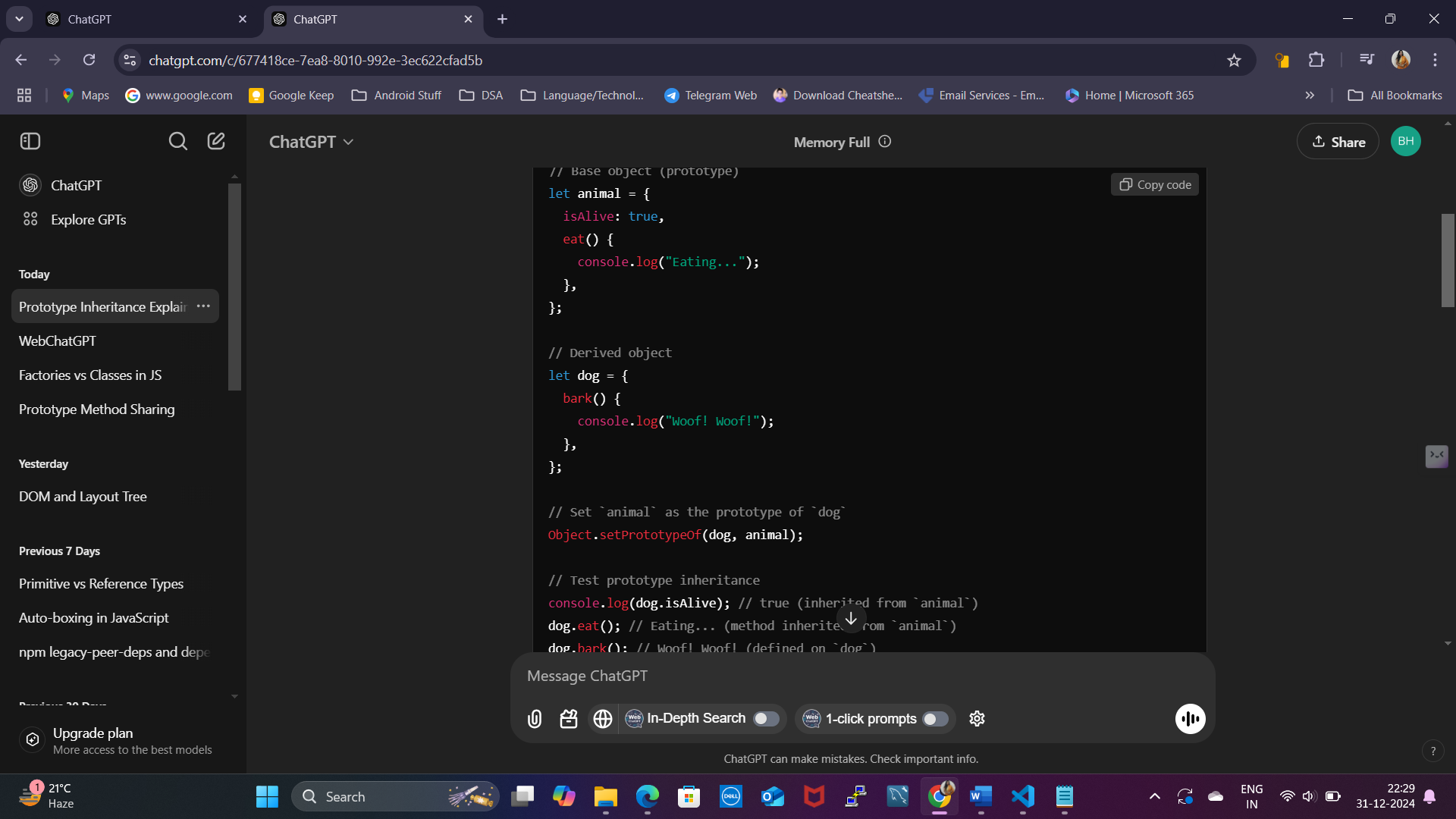
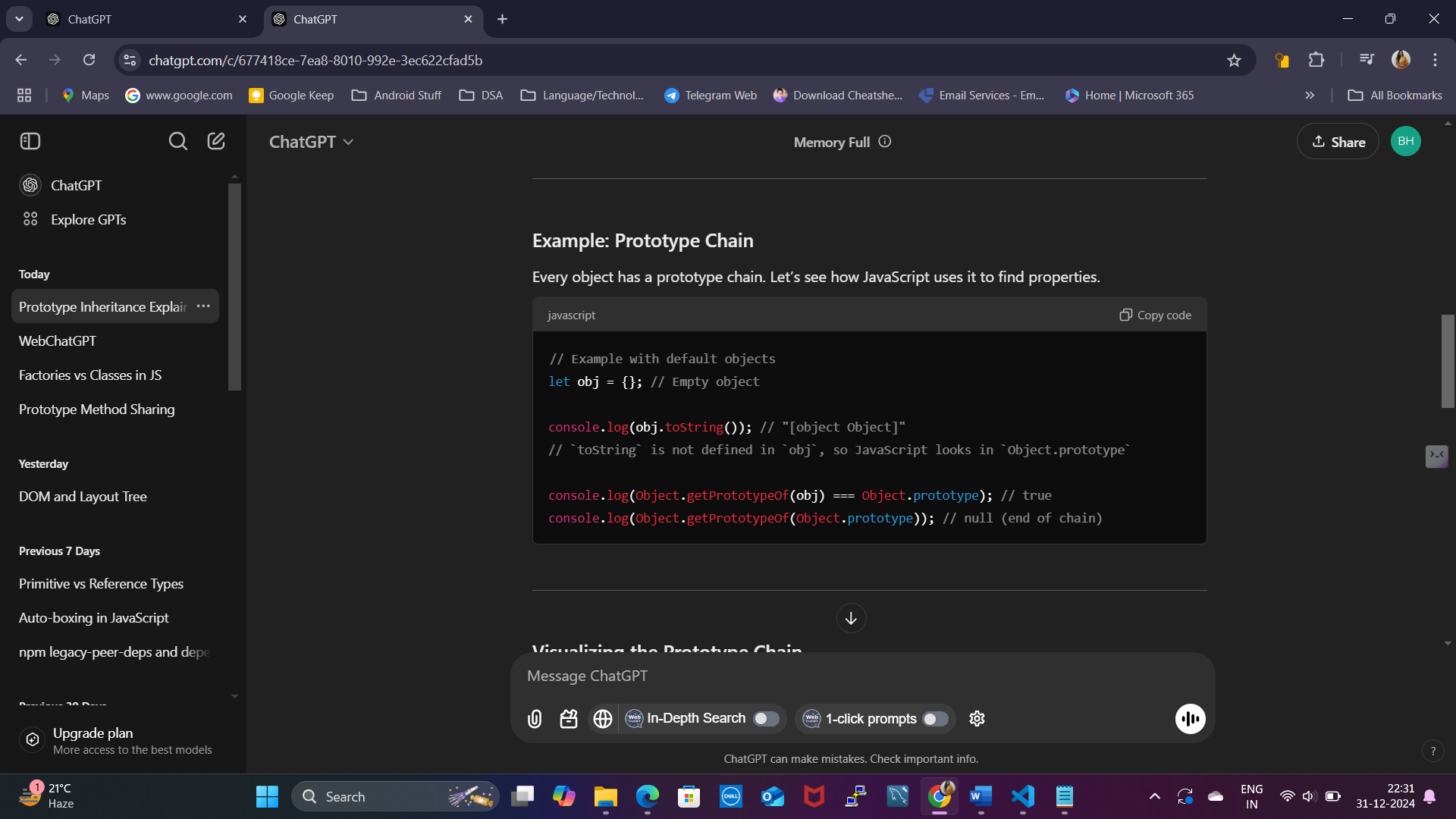
Prototype Inheritance and Prototype Chain

 Each object has an internal link to another object called its *prototype*.

What Is Prototype Inheritance?

In JavaScript, **prototype inheritance** allows objects to "inherit" properties and methods from other objects. Every JavaScript object has a "hidden" property called [[Prototype]], which points to another object (or null). This forms a **prototype chain**, where if an object doesn't have a property or method, JavaScript looks for it in its prototype, then in the prototype's prototype, and so on, until it finds the property or reaches the end of the chain (null).





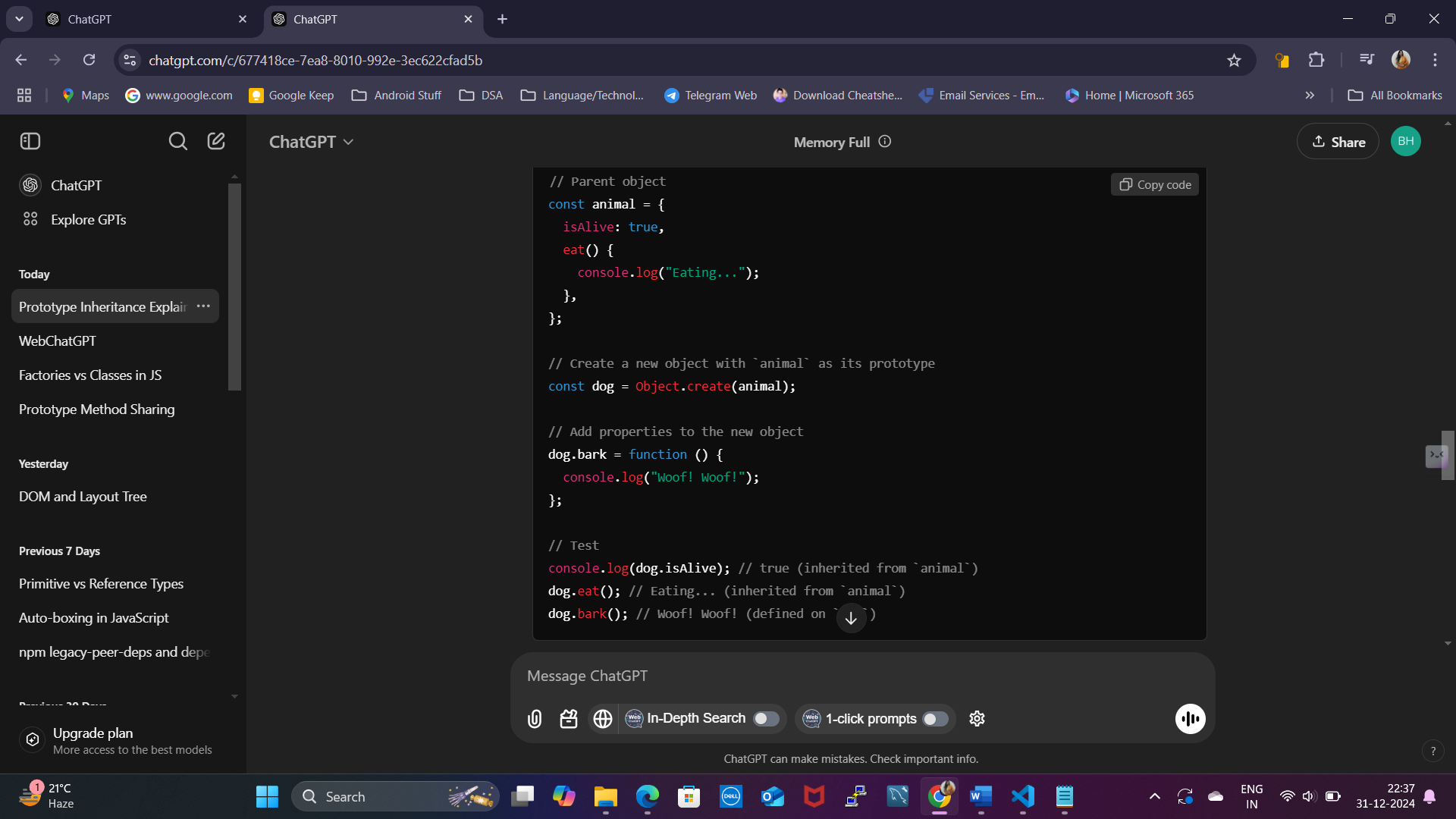
**Key Points**

1. **Prototype Inheritance**: Objects inherit properties and methods from their prototype.
2. **Prototype Chain**: JavaScript searches for properties and methods along the chain of prototypes until it finds them or reaches null.
3. **Object.setPrototypeOf**: Sets the prototype of an object.
4. **Object.getPrototypeOf**: Gets the prototype of an object.

Object.create and Object.assign

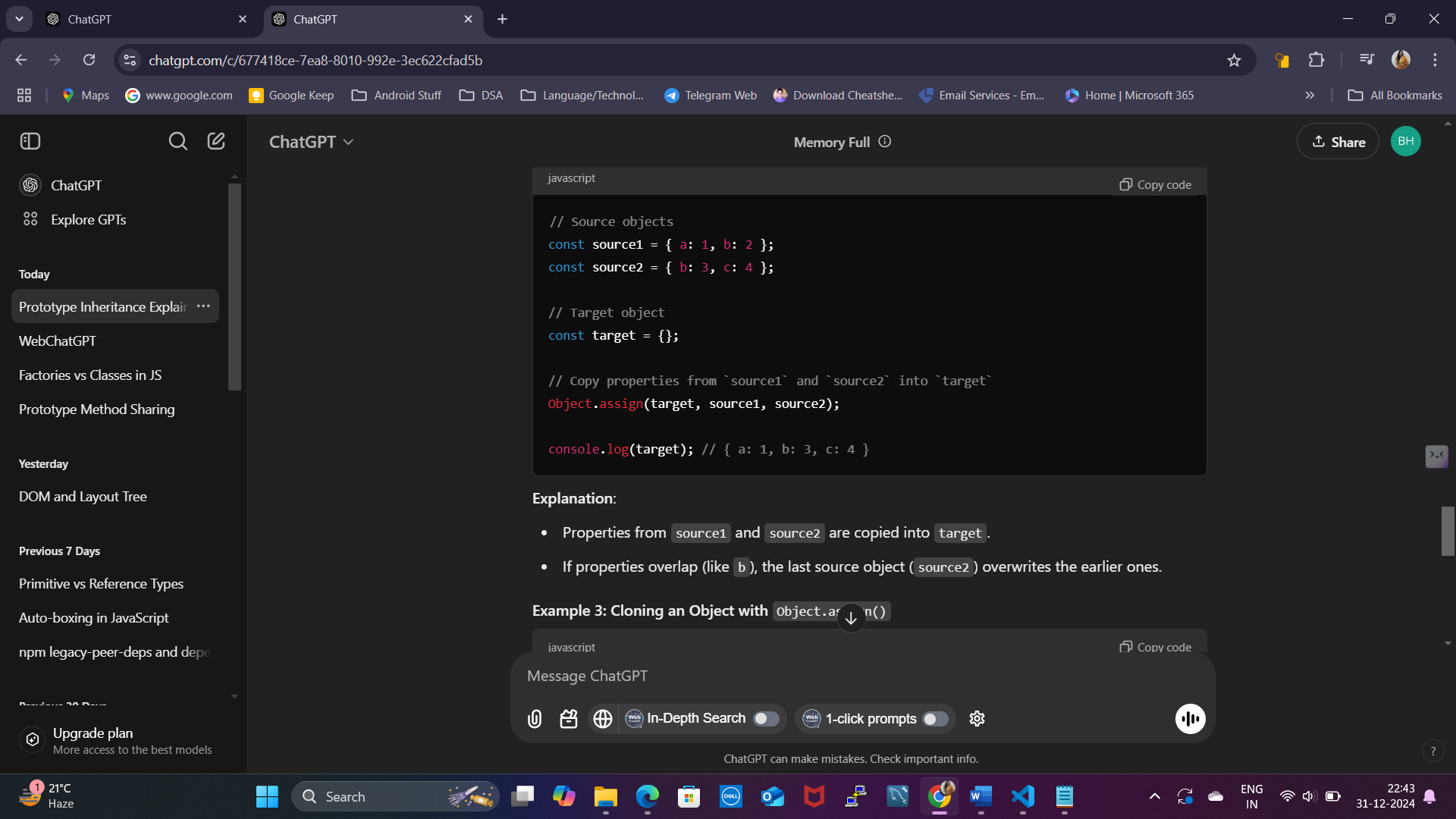
**1. Object.create()**

* **What it does**: Creates a new object and sets the prototype (i.e., the "parent") of the new object to an existing object.
* **Why use it**: To create objects that inherit directly from another object.



**2. Object.assign()**

* **What it does**: Copies properties from one or more objects into a target object.
* **Why use it**: To merge objects or clone an object’s properties into a new object.



map, reduce, filter

**1. map()**

* **What it does**: Transforms each element in an array and returns a new array of the same length.

**2. filter()**

* **What it does**: Filters elements of an array based on a condition and returns a new array with elements that pass the test.

**3. reduce()**

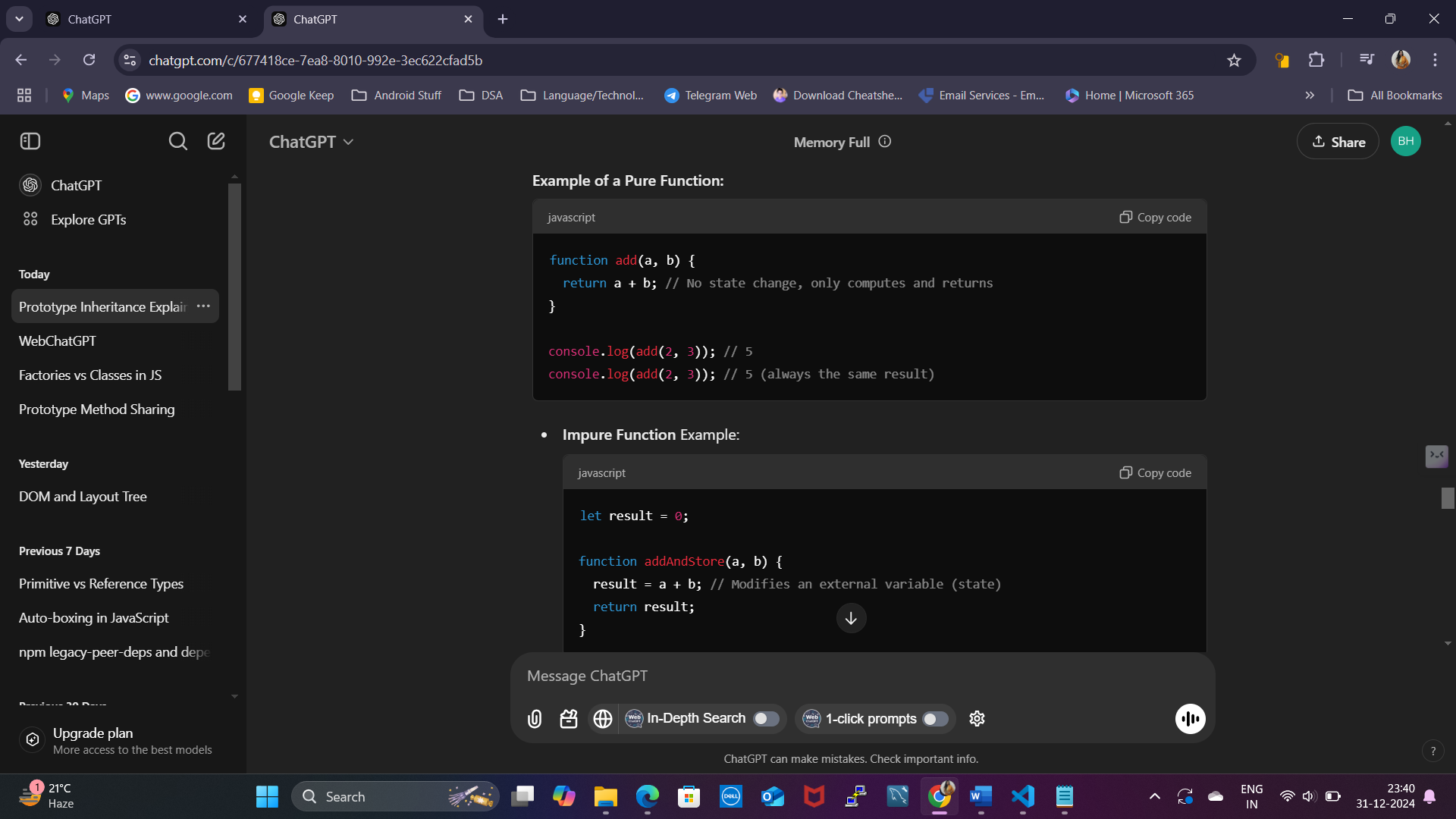
* **What it does**: Processes an array to reduce it to a single value (e.g., sum, product, or object).

Pure Functions, Side Effects, State Mutation, and Event Propagation

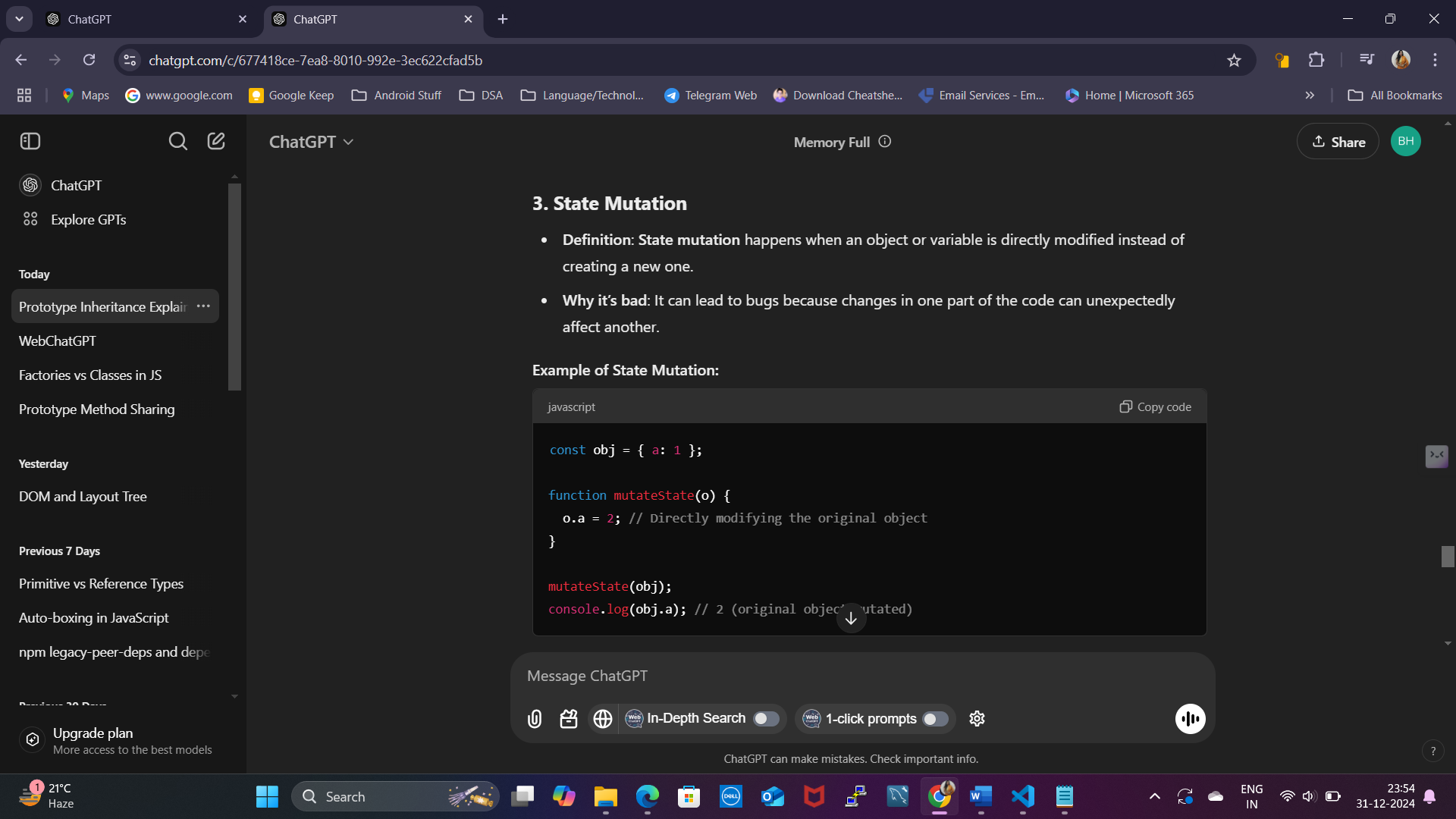
1. Pure Functions

**Definition**: A function is **pure** if:

1. It always gives the same output for the same input.
2. It does not change anything outside its scope (no side effects).



**3. State Mutation**

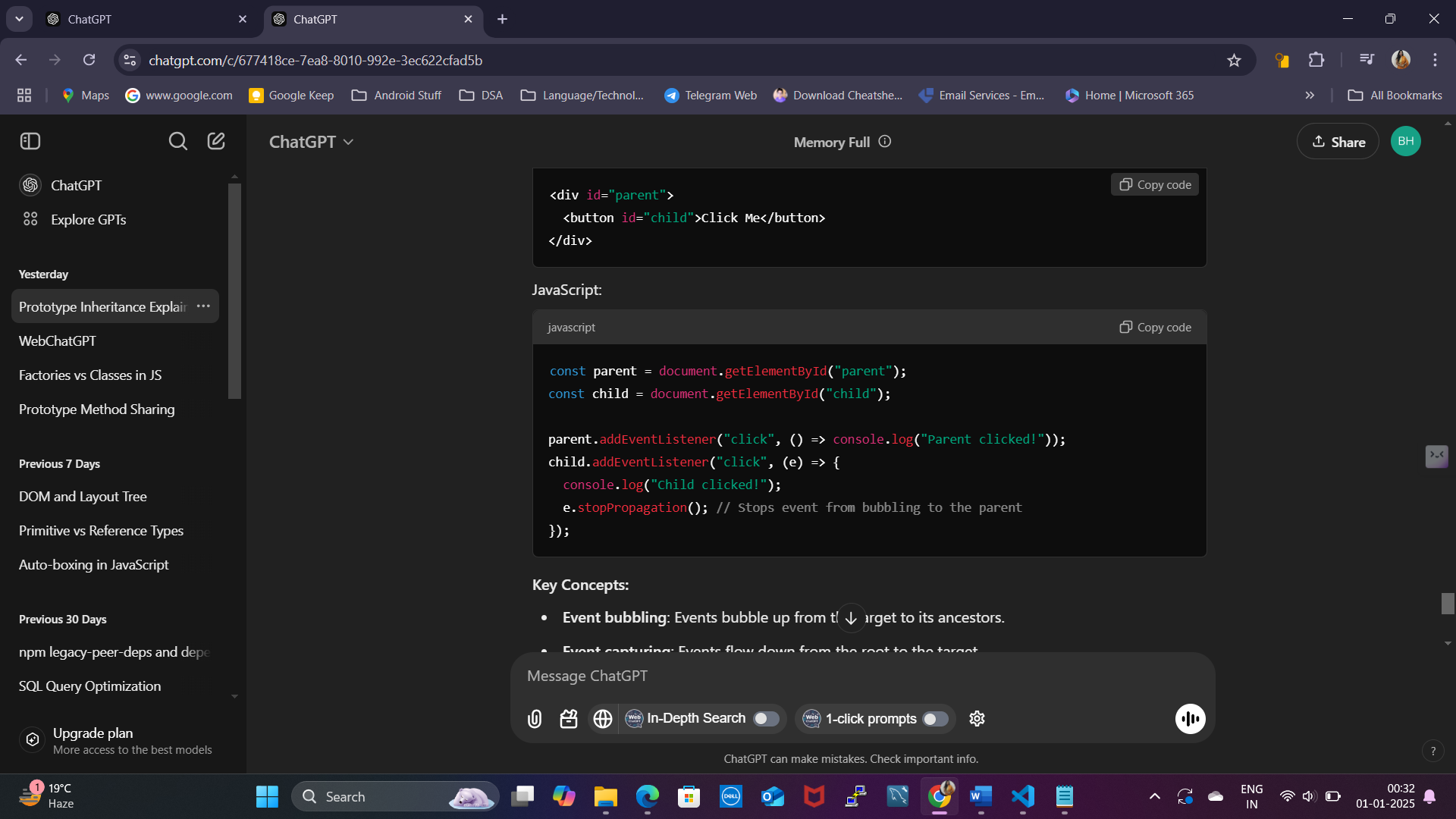


A screenshot of a computer

Description automatically generated

**4. Event Propagation**

* **Definition**: The process by which events move through the DOM. There are two main phases:
  1. **Capturing Phase**: Event goes from the root to the target element.
  2. **Bubbling Phase**: Event goes from the target element back up to the root.



**Key Concepts:**

* **Event bubbling**: Events bubble up from the target to its ancestors.
* **Event capturing**: Events flow down from the root to the target.
* **stopPropagation**: Stops the event from propagating further.

Closures

A **closure** in JavaScript is a feature where an inner function has access to the variables of its outer function, even after the outer function has executed. Closures allow the inner function to "remember" the environment in which it was created.

**Key Characteristics of Closures:**

1. **Access to Outer Variables**: The inner function can access variables of its outer function.
2. **Environment Persistence**: The variables of the outer function remain in memory even after the outer function has returned, thanks to the reference maintained by the inner function.
3. **Encapsulation**: Closures can be used to create private variables and methods.

**Scope Chain in Closures**

A closure has access to:

1. Its own variables.
2. Variables of the outer function in which it was defined.
3. Global variables.

A screenshot of a computer

Description automatically generated

**How it Works**

1. outerFunction is called with 'outside' as the outerVariable, and it returns innerFunction.
2. innerFunction has access to outerVariable even after outerFunction has returned.

High Order Functions

A **Higher-Order Function (HOF)** is a function that does at least one of the following:

1. **Takes another function as an argument** (also called a callback).
2. **Returns a function** as its result.

Higher-order functions are a key feature of JavaScript and functional programming paradigms. They allow for more abstract, flexible, and modular code

**Characteristics of Higher-Order Functions**

1. **Abstraction**: They abstract away common patterns into reusable pieces of code.
2. **Composability**: They allow you to build complex functionality by combining simple functions.
3. **Immutability**: Often used in functional programming to avoid mutating data.

Collections and Generators

Collections in JavaScript refer to data structures used to store and manage groups of data. The most common collection types include arrays, sets, and maps

**Generators in JavaScript**

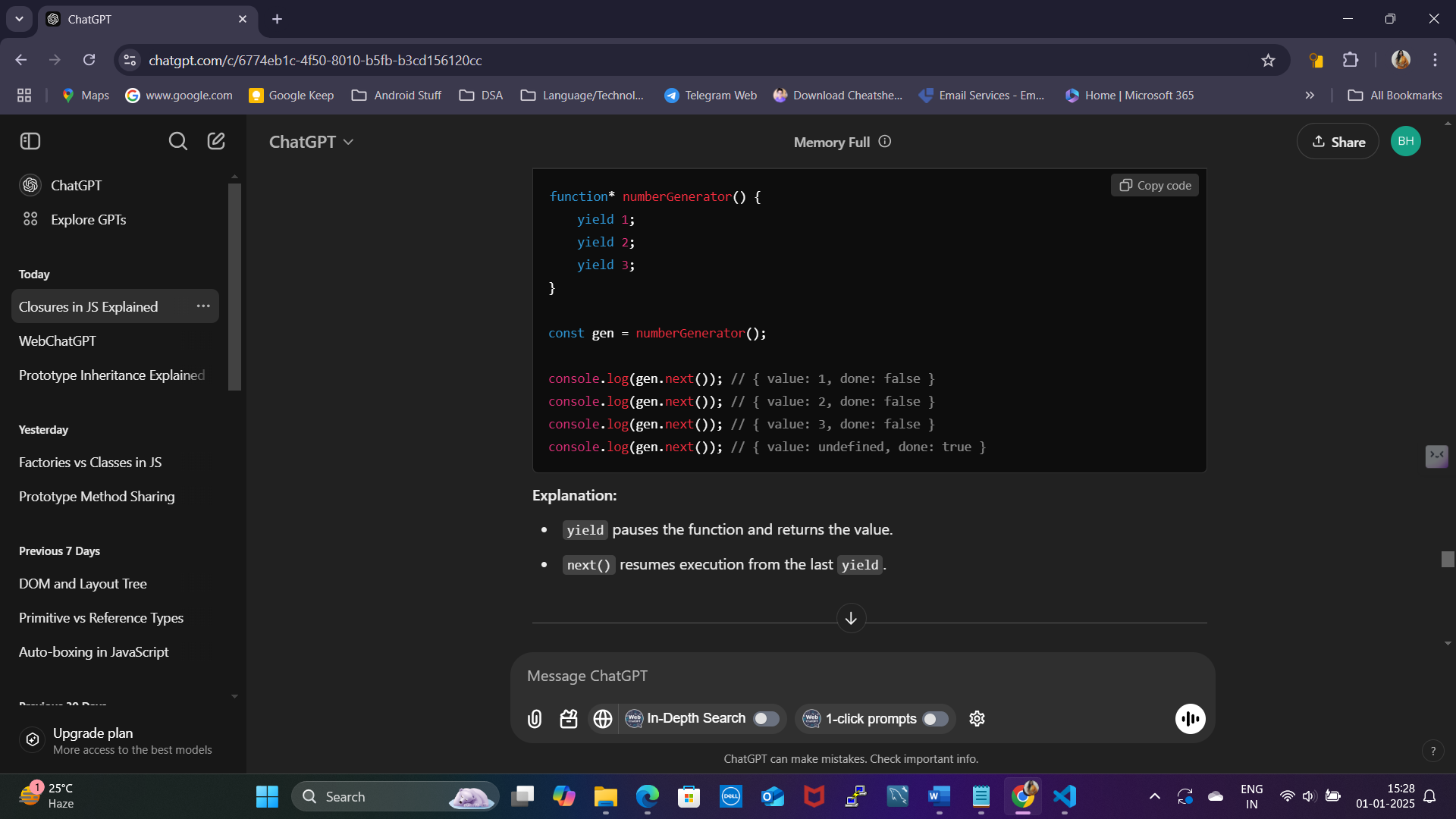
Generators are special functions that can be paused and resumed. They are defined using the function\* syntax and use the yield keyword to pause execution.

**Why Use Generators?**

Generators are useful for:

1. Creating **iterators**.
2. Handling **asynchronous tasks**.
3. Generating infinite sequences or streams of data.

1. Creating a Generator



**Explanation:**

* yield pauses the function and returns the value.
* next() resumes execution from the last yield.

A screenshot of a computer

Description automatically generated

**Design Patterns in JavaScript**

Design patterns are reusable solutions to common software design problems.

Types of Design Patterns

Design patterns are broadly categorized into three types:

1. **Creational Patterns**: Deal with object creation mechanisms, enhancing flexibility and reuse.
2. **Structural Patterns**: Focus on the composition of objects and classes, ensuring better organization and relationships.
3. **Behavioral Patterns**: Focus on communication between objects and how responsibilities are distributed.
4. Creational Patterns
   1. Singleton Pattern

Ensures a class has only one instance and provides a global point of access to it.



**Common Use Cases:**

1. **Logging**: A logging class might be a good candidate for a Singleton. You want to ensure that all parts of your application are logging to the same file or service.
2. **Database Connections**: Managing a single database connection throughout the application ensures that you don't open multiple connections, reducing overhead and the risk of conflicts.
3. **Configuration Management**: A Singleton can be used to load configuration settings that are used across multiple modules of an application.
4. **Cache Management**: A centralized cache, where multiple parts of the application need access to the same data, can benefit from the Singleton pattern.
   1. Factory Pattern

Creates objects without specifying the exact class of object that will be created.

A screenshot of a computer

Description automatically generated

**Usage:**

1. When the type of object to create is determined by some input or configuration.
2. To manage a family of related objects, especially when object creation involves complex logic.
3. When the exact type of object is not known until runtime.
4. Structural Patterns
   1. Decorator Pattern

The **decorator pattern** is a structural design pattern that allows adding new functionality to an existing object without modifying its structure. In JavaScript, it is often implemented using higher-order functions or class decorators.

A screenshot of a computer

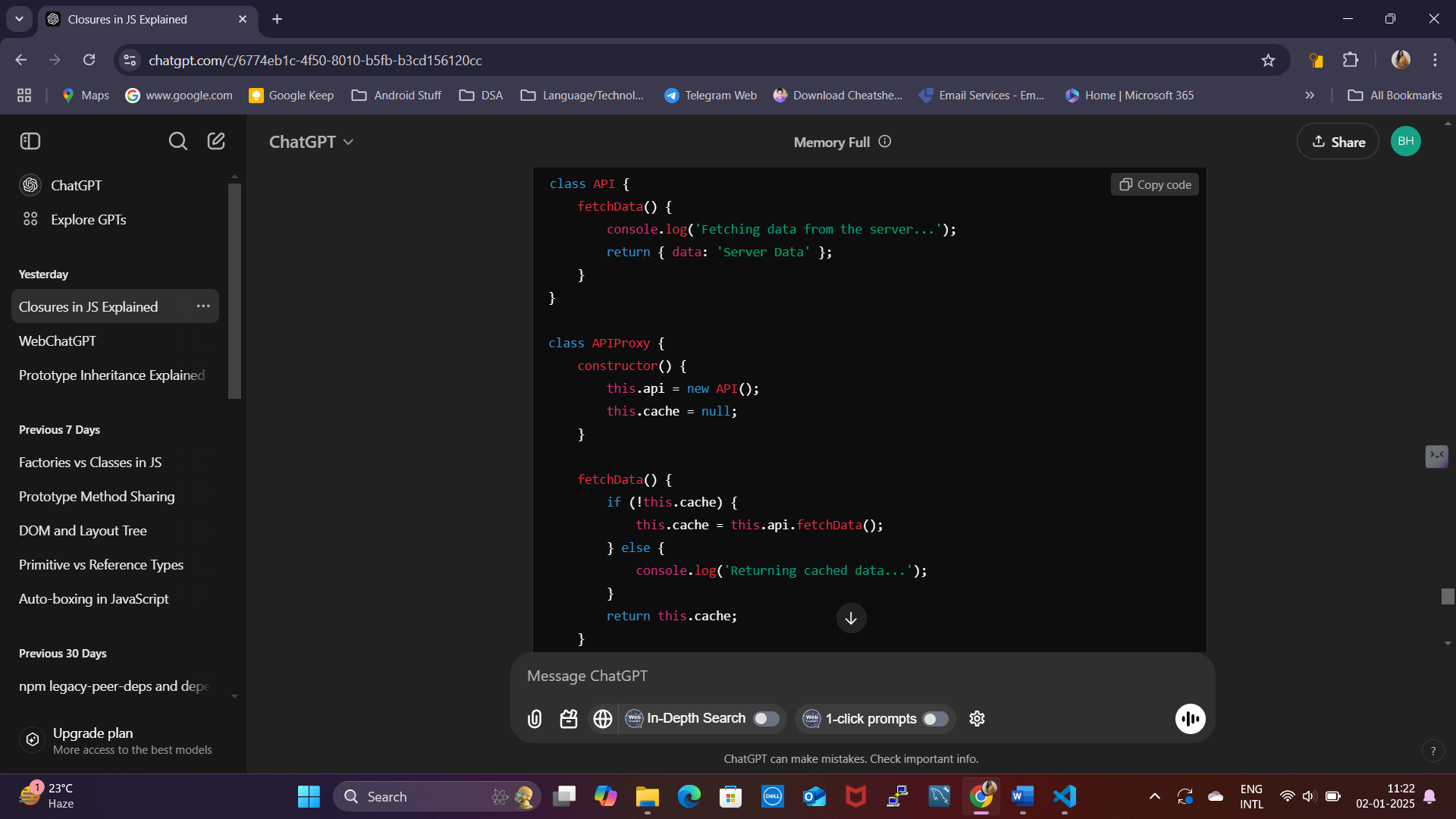
Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. Proxy Pattern

The **Proxy Pattern** is like having a middleman who controls access to another object. Instead of interacting directly with an object, you interact with the proxy. The proxy can add additional behavior, like caching, access control, or logging, before forwarding your request to the real object.



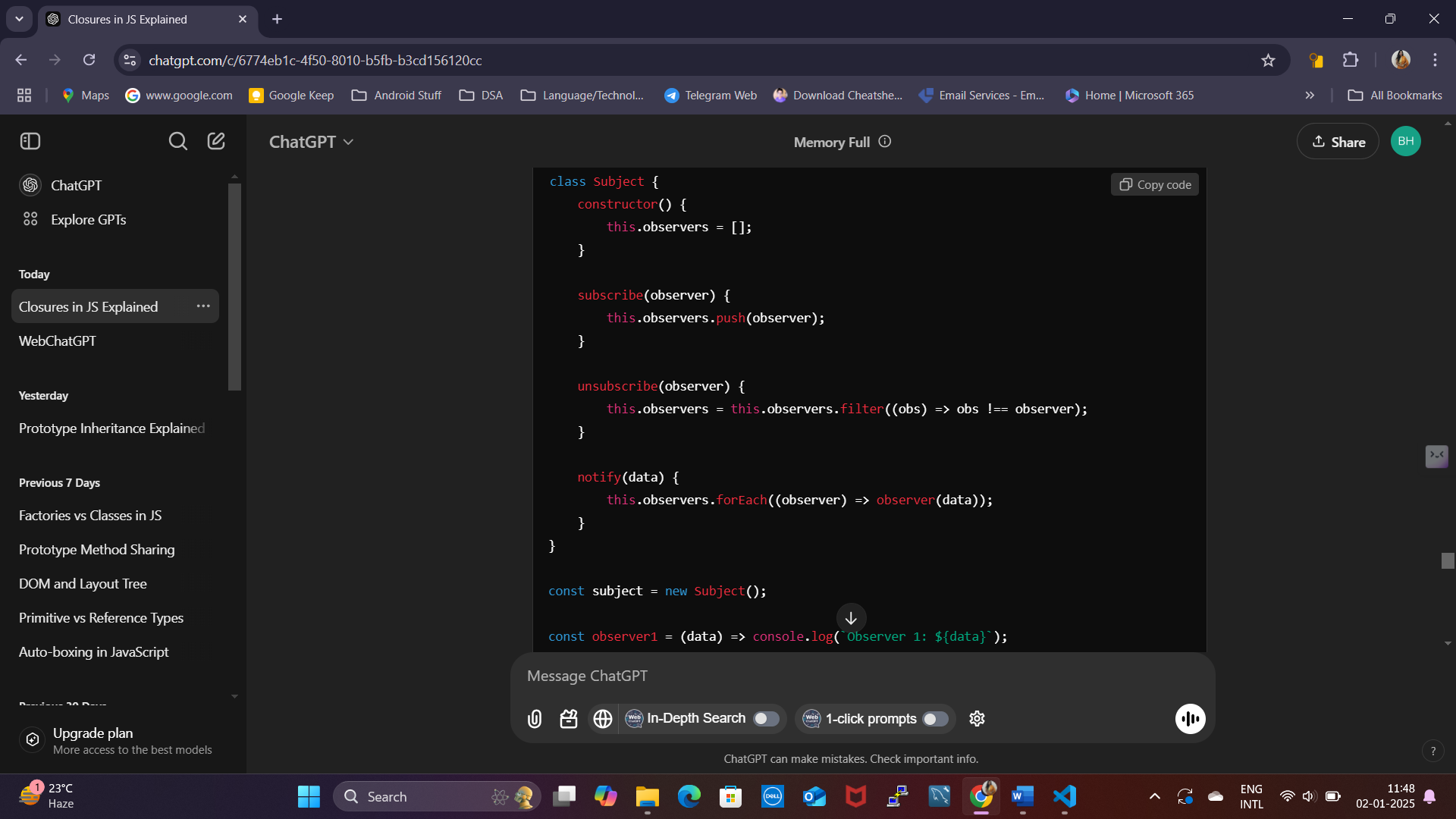
1. Behavioral Patterns

**a. Observer Pattern**

**Allows an object (subject) to notify other objects (observers) of changes in state.**

The **Observer Pattern** is like a subscription service. There’s a "publisher" (subject) that holds some data or state, and "subscribers" (observers) who want to know when that data changes.

When the data changes, the publisher **notifies** all its subscribers about the update.



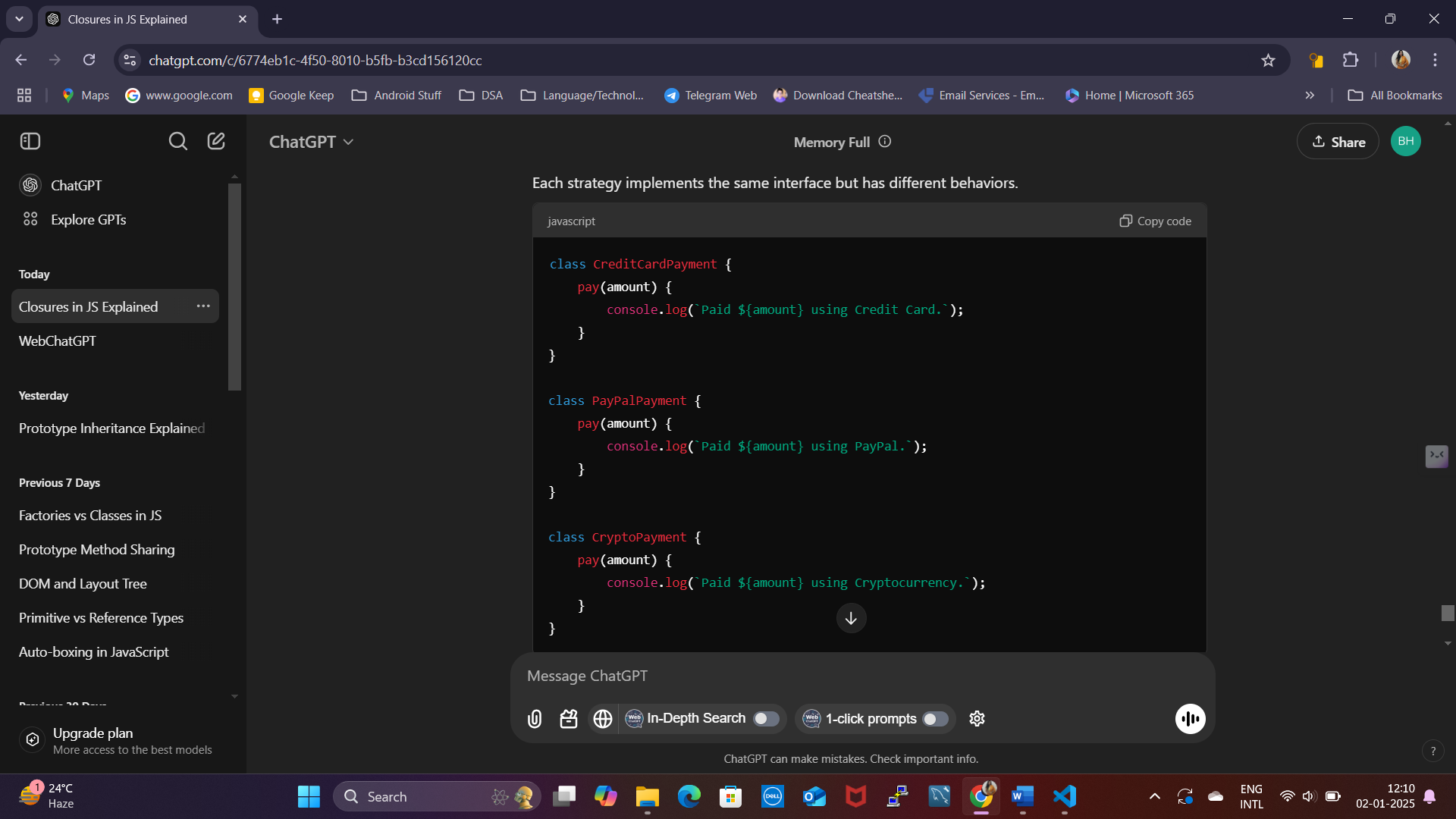
A screenshot of a computer

Description automatically generated

**b. Strategy Pattern**

Allows the selection of an algorithm's behavior at runtime.

The **Strategy Pattern** is like choosing the best tool for the job. It allows you to define a family of interchangeable algorithms (strategies) and select which one to use at runtime without changing the code that uses them.



A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Partial Applications, Currying, Compose, and Pipe

**1. Partial Applications**

**What is it?**

* You take a function with multiple arguments and "lock in" some of those arguments to create a new function.

**Analogy:**

Imagine you go to a pizza shop. The shop has a base price for pizza but charges extra for toppings. You pre-select a base pizza (e.g., Margherita), but you can add different toppings later.

A screenshot of a computer

Description automatically generated

**2. Currying**

**What is it?**

* Currying is a way to break a function with multiple arguments into a series of smaller functions, each taking one argument at a time.

**Analogy:**

You’re ordering a custom sandwich:

1. First, you choose the bread.
2. Next, you choose the fillings.
3. Then, you choose the sauce.

Each step depends on the previous one.

A screenshot of a computer

Description automatically generated

**3. Compose**

**What is it?**

* Compose combines multiple functions into one, processing data from **right to left**.

**Analogy:**

Think of data as going through a series of filters:

1. First, it goes through filter A.
2. Then, the output of A goes to filter B.
3. Finally, the output of B is your result.

A screenshot of a computer

Description automatically generated

**4. Pipe**

**What is it?**

* Pipe is like Compose but processes data from **left to right**

A screenshot of a computer

Description automatically generated

**Keyword Static**

The static keyword in JavaScript is used in classes to define a method or property that belongs to the **class itself** rather than to instances of the class. These are called **static methods** or **static properties**.

**Key Points About static**

1. **Class-Level Access**:
   * Static methods and properties are accessed directly on the class, not on instances of the class.
   * They are commonly used for utility functions or constants that do not depend on instance-specific data.
2. **Cannot Access Instance Members**:
   * Static methods cannot access instance properties or methods directly (this does not refer to the instance in static methods).
3. **Shared Across All Instances**:
   * A static property or method is the same for all instances of the class.