

# 2758. Next Day

Easy

[Leetcode Link](#)

## Problem Description

In this coding task, the goal is to extend the functionality of the JavaScript `Date` object by introducing a new method called `nextDay()`. When called on a `Date` object, this method should calculate the date for the following day and return it as a string formatted as "YYYY-MM-DD". For example, if the method is invoked on a `Date` object representing June 20, 2014, it should return the string "2014-06-21" which represents June 21, 2014.

## Intuition

To achieve the result of getting the next day's date, we employ a straightforward approach. By using the `Date` object's built-in methods, we can carry out this task with a few steps:

- Get the current date from the `this` reference inside our new `nextDay` method. The `this` keyword refers to the `Date` object that `nextDay` is called upon.
- Create a copy of the current date to avoid mutating the original `Date` object.
- Increment the date of this copied object by one, using the `getDate` and `setDate` methods. This action effectively moves the date forward by one day.
- Convert the updated `Date` object into a string formatted as ISO 8601 using `toISOString`. This will yield a string in the format "YYYY-MM-DDTHH:mm:ss.sssZ".
- Extract and return only the date portion from this ISO string (i.e., the first 10 characters, "YYYY-MM-DD").

By following these steps, we conform to JavaScript's date handling and ensure correctness while providing a neatly formatted result.

## Solution Approach

The solution leverages the native JavaScript `Date` object's methods to achieve the desired functionality. Here's a step-by-step breakdown of the algorithm, patterns, and data structure used in the implementation:

- Monkey Patching:** The solution uses a pattern known as "monkey patching," where a method is added to an existing built-in object at runtime. By extending the `Date` object's prototype, we introduce a new method `nextDay` that becomes available to all instances of `Date`.
- Creating a New Date Instance:** A new `Date` object is instantiated using `this.valueOf()`. The `valueOf` method returns the primitive value of the specified object. In the case of a `Date` object, it returns the number of milliseconds since January 1, 1970, 00:00:00 UTC. This number is used to create a new `Date` object which is a copy of the original, ensuring that the original `Date` object remains unchanged.
- Incrementing the Date:** By calling `getDate` on the newly created `Date` object, we get the day of the month for the specified date according to local time. We then use `setDate` to advance this value by one, which moves the date to the next day.
- Formatting the Result:** The `toISOString` method is used, which converts a `Date` object into a string in ISO format (YYYY-MM-DDTHH:mm:ss.sssZ). Since the problem only requires the date component in "YYYY-MM-DD" format, `slice(0, 10)` is applied to this string to extract the first ten characters, discarding the time portion.
- Returning the Result:** The string representation of the next day in "YYYY-MM-DD" format is then returned.

The solution doesn't directly employ complex data structures or algorithms as it mainly revolves around date manipulation using built-in `Date` object methods. The pattern used here is simple and efficient since JavaScript's `Date` object internally handles all the complexities related to leap years, time zones, daylight saving time changes, and other date-related quirks.

Here's the implementation based on the approach described:

```
1 declare global {
2   interface Date {
3     nextDay(): string;
4   }
5 }
6
7 Date.prototype.nextDay = function () {
8   const date = new Date(this.valueOf());
9   date.setDate(date.getDate() + 1);
10  return date.toISOString().slice(0, 10);
11 };
12
13 /**
14  * Example usage:
15  * const date = new Date("2014-06-20");
16  * console.log(date.nextDay()); // "2014-06-21"
17  */
```

By following these steps, the `nextDay` method reliably calculates the next day, formats it appropriately, and integrates seamlessly into the `Date` object for convenient usage.

## Example Walkthrough

To provide a better understanding of the `nextDay` method implementation, let's walk through a small example where we apply the solution approach to a `Date` object representing April 30, 2021. This date is of particular interest because the following day is May 1, 2021, which is the start of a new month. Demonstrating the function works across month boundaries ensures it handles more complex cases of date manipulation.

- Creating the Date Object:** We first construct a `Date` object representing April 30, 2021.

```
1 const date = new Date("2021-04-30");
```
- Adding the nextDay Method:** We could imagine the `nextDay` method has already been added to the `Date` prototype as described in the solution approach.
- Calling nextDay:** When we call the `nextDay` method on our date object:

```
1 const nextDayStr = date.nextDay();
```
- Inside nextDay:** Here's what happens within the `nextDay` method:
  - It creates a new `Date` object that represents the same moment in time as `date` by using `this.valueOf()`.
  - It then calls `getDate()` on the new `Date` object to retrieve the day of the month, which is 30. It then uses `setDate()` to increment this value by one. The JavaScript `Date` object automatically handles the transition to the next month, so the internal date representation becomes May 1, 2021.
  - Finally, it uses `toISOString()` to convert the `Date` object to a string in ISO format, followed by `slice(0, 10)` to extract the year, month, and day parts.
- Result:** The `nextDay` method returns the string "2021-05-01" which is the formatted string representing the next day, conforming to the "YYYY-MM-DD" format. This result is stored in `nextDayStr`.

In summary, on invoking `date.nextDay()`, where `date` is April 30, 2021, our output will be:

```
1 console.log(nextDayStr); // "2021-05-01"
```

This confirmed our function correctly moves from the last day of April to the first day of May, thereby validating the method's ability to handle month changes seamlessly.

## Python Solution

```
1 from datetime import datetime, timedelta
2
3 # Extend the datetime class by creating a subclass that includes the next_day method.
4 class ExtendedDate(datetime):
5     def next_day(self):
6         """
7         Calculate and return the ISO format string of the next day.
8
9         Returns:
10            str: The date string in the format YYYY-MM-DD for the next day.
11
12            """
13         # Add one day to the current date
14         next_date = self + timedelta(days=1)
15         # Format the date as an ISO date string and return
16         return next_date.strftime('%Y-%m-%d')
17
18 # Example usage:
19 # Create an instance of ExtendedDate with a specific date
20 today = ExtendedDate(2014, 6, 20)
21 # Print the string representation of the next day
22 print(today.next_day()) # Outputs: "2014-06-21"
```

## Java Solution

```
1 import java.util.Calendar;
2 import java.util.Date;
3 import java.text.SimpleDateFormat;
4
5 public class ExtendedDate {
6     private Date date;
7
8     // Constructor that initializes the ExtendedDate object with a specific date
9     public ExtendedDate(Date date) {
10         this.date = date;
11     }
12
13     // This method returns the ISO string representation of the next day
14     public String nextDay() {
15         // Getting a Calendar instance based on the current date
16         Calendar calendar = Calendar.getInstance();
17         calendar.setTime(date);
18
19         // Adding one day to the current date in the Calendar
20         calendar.add(Calendar.DAY_OF_MONTH, 1);
21
22         // Converting the next day's date to the ISO format (YYYY-MM-DD)
23         SimpleDateFormat isoFormat = new SimpleDateFormat("yyyy-MM-dd");
24         return isoFormat.format(calendar.getTime());
25     }
26
27     // Example usage of the ExtendedDate class
28     public static void main(String[] args) {
29         // Initialize a Calendar object with a specific date
30         Calendar calendar = Calendar.getInstance();
31         calendar.set(2014, Calendar.JUNE, 20);
32
33         // Create an ExtendedDate object with the specific date
34         ExtendedDate today = new ExtendedDate(calendar.getTime());
35
36         // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD)
37         System.out.println(today.nextDay());
38     }
39 }
40
```

## C++ Solution

```
1 #include <iostream>
2 #include <ctime>
3 #include <sstream>
4 #include <iomanip>
5
6 // Extending the std::tm structure to include the next_day method, by creating a wrapper class around std::tm.
7 class Date {
8 public:
9     std::tm timeStruct;
10
11     // Constructor that initializes Date object from a string in the format "YYYY-MM-DD".
12     Date(const std::string &isoDate) {
13         std::istringstream ss(isoDate);
14         ss >> std::get_time(&timeStruct, "%Y-%m-%d");
15         // Manual check if get_time failed, if so, populate with current time.
16         if (ss.fail()) {
17             std::time_t t = std::time(nullptr);
18             localtime_s(&timeStruct, &t); // For safety, using localtime_s as it's thread-safe.
19         }
20     }
21
22     // Method to calculate the next day and return it as an ISO string (YYYY-MM-DD).
23     std::string nextDay() const {
24         // Making a copy of timeStruct to modify
25         std::tm nextDayTm = timeStruct;
26         // Increment the day by one; mktime will normalize the tm struct if the day goes beyond the last day of the month
27         nextDayTm.tm_mday++;
28         std::time_t nextDayTime = std::mktime(&nextDayTm);
29
30         // Create a string stream to hold the ISO string representation of the next day
31         std::stringstream oss;
32         oss << std::put_time(localtime(&nextDayTime), "%Y-%m-%d");
33
34         // Return the next day as an ISO string
35         return oss.str();
36     }
37 };
38
39 /**
40  * Example usage:
41  * Date today("2014-06-20"); // Initialize a Date object with a specific date.
42  * std::cout << today.nextDay() << std::endl; // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD).
43  */
44
45 int main() {
46     Date today("2014-06-20"); // Initialize a Date object with a specific date.
47     std::cout << today.nextDay() << std::endl; // Outputs: "2014-06-21"
48     return 0;
49 }
50
```

## Typescript Solution

```
1 // Extending the global Date interface to include the nextDay method.
2 interface Date {
3     nextDay(): string;
4 }
5
6 // Implementing the nextDay method for the Date prototype to get the ISO string of the next day.
7 Date.prototype.nextDay = function (): string {
8     const currentDate = new Date(this.valueOf()); // Creating a new date object that represents the current date.
9     currentDate.setDate(currentDate.getDate() + 1); // Adding one day to the current date.
10    return currentDate.toISOString().slice(0, 10); // Converting the next day's date to an ISO string and slicing to get the date part
11 };
12
13 /**
14  * Example usage:
15  * const today = new Date("2014-06-20"); // Initialize a Date object with a specific date.
16  * console.log(today.nextDay()); // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD).
17  */
18
```

## Time and Space Complexity

The time complexity of the `nextDay` method is  $O(1)$ , indicating that it takes a constant amount of time to compute the next day regardless of the size of the input. This is because the internal operations, such as `getDate`, `setDate`, and `toISOString`, are all constant-time operations provided by the `Date` object's API.

The space complexity is also  $O(1)$  because the additional space used does not grow with the size of the input. A single new `Date` instance is created, and its use of memory does not depend on the input date.