2618. Check if Object Instance of Class

Medium

Leetcode Link

Problem Description

The problem requires writing a function that checks if a certain value (referred to as `obj`) is an instance of a specified class or one of its superclasses. An object is deemed an instance of a class if it has access to the methods defined by that class. This means we need to determine if `obj` inherits from the prototype of the given class (let's call this `classFunction`). It's necessary to account for different types of inputs, including cases where the `obj` or `classFunction` might be `undefined`.

Intuition

The intuition behind the solution involves understanding JavaScript's prototypal inheritance. An object in JavaScript inherits properties and methods from its prototype. The prototype chain is a series of linked prototypes; an object has a prototype, that prototype has its own prototype, and so on, until an object's prototype is `null`.

Knowing this, we must check whether the prototype of the given object is the same as, or linked through a series of prototypes to, the prototype property of the class function (constructor).

The approach can be broken down into several steps:

1. Test if the `classFunction` is `null` or `undefined`. If it is, then definitely `obj` isn't an instance of it, so we return `false`.
2. Loop through the prototype chain of `obj` using `Object.getPrototypeOf()`. At each step, we:

   a. Compare the prototype of `obj` with the prototype property of `classFunction`.

   b. If a match is found, `obj` is an instance of `classFunction` or one of the classes in its prototype chain, and we return `true`.

   c. If a match isn't found, update `obj` to its own prototype and keep checking up the chain.
3. If we reach the end of the prototype chain (`obj`'s prototype is `null`), `obj` is not an instance of `classFunction` or a superclass, and we return `false`.

This approach does full justice to the problem by checking the entire inheritance chain, ensuring that the result is accurate even if inheritance spans multiple levels.

Solution Approach

The implementation of the solution involves a fundamental understanding of JavaScript prototypes and iteration. Here is a more detailed walkthrough of the approach taken in the reference solution:

1. Check if `classFunction` is `null` or `undefined`. If so, return `false` immediately. This is because in JavaScript, `null` and `undefined` do not have a prototype chain, and thus, they are not a valid constructor that can create instances.
2. Initiate a loop to traverse the prototype chain of the `obj`. The loop continues until `obj` itself is `null` or `undefined`. The condition that breaks the loop indicates that we have reached the end of the prototype chain (since the prototype of the last object in a chain is `null`).
3. Use `Object.getPrototypeOf(obj)` to access the prototype of `obj`. This function returns the prototype (`[[Prototype]]`) or `null` of the given object.
4. Compare the current prototype of `obj` to the prototype property of `classFunction` using `proto === classFunction.prototype`. If they are equal, this means that the `classFunction` is in the prototype chain of `obj`, and therefore `obj` is an instance of `classFunction` or one of its ancestors. Return `true` in this case.
5. If the current prototype does not match, the loop continues. To do so, set `obj` to its prototype for the next iteration: `obj = proto;`. This step effectively moves up the prototype chain.
6. If the loop exits without returning `true`, this means that `classFunction.prototype` was not found in the prototype chain of `obj`. Therefore, we return `false`.

This solution uses a while loop to check the prototype chain, a fundamental pattern used in prototype-based languages like JavaScript for inheritance checks. It takes advantage of the `Object.getPrototypeOf()` function to access the prototype chain of objects.

The solution doesn't use any additional data structures. The time complexity is O(n), where n is the length of the prototype chain of the `obj`. This is because in the worst case, the loop will traverse the entire chain. The space complexity is O(1) because no additional space is used besides the variables for iteration and comparison.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have a class hierarchy where we have a class `Animal`, a subclass `Mammal` that extends `Animal`, and another subclass `Dog` that extends `Mammal`. We're interested in checking if an instance of `Dog` is also an instance of `Animal` using the solution approach described.

```
function Animal() {}
function Mammal() {}
Mammal.prototype = Object.create(Animal.prototype);
function Dog() {}
Dog.prototype = Object.create(Mammal.prototype);

const myDog = new Dog();
```

In this setup, `Animal` is the superclass, `Mammal` is a subclass that inherits from `Animal`, and `Dog` is a subclass that inherits from `Mammal`. We create an instance of `Dog` called `myDog`. We expect that `myDog` is an instance of `Dog`, `Mammal`, and `Animal`.

Now, let's walk through the solution approach to check if `myDog` is an instance of `Animal`:

1. We call our hypothetical function to check: `isInstanceOf(myDog, Animal)`.
2. Inside the function, we first check if `Animal` is `null` or `undefined`. In our case, it's not, so we continue.
3. We start a loop where `obj` is initially `myDog`. We will loop through its prototype chain to check if any prototype along the way is the prototype of `Animal`.
4. We use `Object.getPrototypeOf(obj)` to get the prototype of `myDog`, which will be `Dog.prototype`.
5. We compare `Dog.prototype` with `Animal.prototype` using `===`. They are not the same, so we move up the prototype chain.
6. We set `obj` to its own prototype with `obj = Object.getPrototypeOf(obj)`, now `obj` refers to `Mammal.prototype`.
7. We compare `Mammal.prototype` with `Animal.prototype`. Again, they are not the same, so we continue up the prototype chain.
8. We update `obj` again with `obj = Object.getPrototypeOf(obj)`, and `obj` now refers to `Animal.prototype`.
9. This time, when we compare `Animal.prototype` with `Animal.prototype`, we find them to be the same.
10. Since we found a match, our function returns `true`, correctly identifying that `myDog` is an instance of `Animal` based on the prototype chain.

This walkthrough demonstrates the solution's ability to traverse the entire inheritance chain to identify the relationship between an object and a potential superclass, checking each link in the prototype chain until a match is found or until it reaches the end.

Python Solution

```python
def check_if_instance_of(object_to_check, class_constructor):
    """
    This function checks if a given object is an instance of a specified class/function.

    :param object_to_check: The object to check for being an instance of the class_constructor provided.
    :param class_constructor: The class constructor or function to check against.
    :return: True if object_to_check is an instance of class_constructor; otherwise, False.
    """
    # Return False immediately if class_constructor is None.
    if class_constructor is None:
        return False

    # Traverse the prototype (or class hierarchy in Python) of object_to_check.
    while object_to_check is not None:
        # Check if object_to_check is a direct instance of class_constructor.
        if isinstance(object_to_check, class_constructor):
            return True
        # Move up the class hierarchy (python does not require manual traversal like JavaScript).
        # In python, isinstance already checks the entire class hierarchy.
        break  # Break immediately since further manual traversal is unnecessary.

    # The entire class hierarchy was checked and no instances of class_constructor were found.
    return False

# Usage example:
# check_if_instance_of(datetime.date.today(), datetime.date) # Should return True as today's date is an instance of the date class.

```

Java Solution

```java
/**
 * This method checks if a given object is an instance of a specified class.
 *
 * @param objectToCheck The object to check for being an instance of the classConstructor provided.
 * @param classConstructor The class constructor to check against. It can be null.
 * @return True if the objectToCheck is an instance of the classConstructor; otherwise, false.
 */
public static boolean checkIfInstanceOf(Object objectToCheck, Class<?> classConstructor) {
    // Return false immediately if classConstructor is null.
    if (classConstructor == null) {
        return false;
    }

    // Check if objectToCheck is an instance of the class using the instanceof operator.
    return classConstructor.isInstance(objectToCheck);
}

// Usage example:
// checkIfInstanceOf(new Date(), Date.class); // Returns true, because a Date object is an instance of the Date class.

```

C++ Solution

```cpp
#include <typeinfo>

// This function checks if a given object is an instance of a specified class.
// @tparam ObjectToCheckType - The type of the object to be checked.
// @tparam ClassConstructorType - The class type to check against.
// @param object_to_check - The object to check if it's an instance of the class_constructor provided.
// @param class_constructor - A pointer to an instance of the class type to check against.
// @returns {bool} - True if 'object_to_check' is an instance of 'class_constructor'; otherwise, false.
template<typename ObjectToCheckType, typename ClassConstructorType>
bool CheckIfInstanceOf(const ObjectToCheckType* object_to_check, const ClassConstructorType* class_constructor) {
    // Return false immediately if object_to_check or class_constructor is a null pointer.
    if (object_to_check == nullptr || class_constructor == nullptr) {
        return false;
    }

    // Check if the types match using dynamic_cast to downcast to derived class.
    // dynamic_cast will return nullptr if the cast is not possible (i.e., if the objects are of different types).
    return dynamic_cast<const ClassConstructorType*>(object_to_check) != nullptr;
}

// Usage example:
// CheckIfInstanceOf<Date, Date>(&dateInstance, &dateClassInstance); // Should return true if dateInstance is an instance of Date.

```

Typescript Solution

```typescript
// This function checks if a given object is an instance of a specified class/function.
// @param {any} objectToCheck - The object to check for being an instance of the classFunction provided.
// @param {Function | null | undefined} classConstructor - The class constructor or function to check against.
// @returns {boolean} - True if the objectToCheck is an instance of the classConstructor; otherwise, false.
function checkIfInstanceOf(objectToCheck: any, classConstructor: Function | null | undefined): boolean {
    // Return false immediately if classConstructor is null or undefined.
    if (classConstructor === null || classConstructor === undefined) {
        return false;
    }
    // Traverse the prototype chain of the objectToCheck.
    while (objectToCheck !== null && objectToCheck !== undefined) {
        // Retrieve the prototype of the current object.
        const currentPrototype = Object.getPrototypeOf(objectToCheck);
        // Check if the current prototype equals the prototype of the classConstructor.
        // If yes, objectToCheck is an instance of classConstructor.
        if (currentPrototype === classConstructor.prototype) {
            return true;
        }
        // Move up the prototype chain.
        objectToCheck = currentPrototype;
    }
    // The entire prototype chain was checked and classConstructor.prototype was not found.
    return false;
}

// Usage example:
// checkIfInstanceOf(new Date(), Date); // Should return true since a Date object is an instance of the Date class.

```

Time and Space Complexity

Time Complexity

The time complexity of the `checkIfInstanceOf` function is primarily determined by the while loop that traverses the prototype chain of the `obj` parameter. In the worst-case scenario, this loop will execute once for each link in the prototype chain until it either finds the prototype of `classFunction` or until it reaches the end of the chain (`null`). If `n` represents the number of prototypes in the chain, then the time complexity would be `O(n)`, as each prototype is visited at most once.

Space Complexity

The space complexity of the function is `O(1)`, primarily because it uses a fixed amount of space. Apart from the space used for the parameters and the loop variable `proto`, the function does not allocate any additional space that grows with the input size. The space required for the function call does not depend on the size of the prototype chain, hence it's constant.