The `Function.prototype.apply()` function is a built-in method in JavaScript that allows you to call a function with a specified this value and an array of arguments. Like `Function.prototype.call()`, it allows you to explicitly define the context (`this` value) in which the function will be executed.

Solution

As `Function.prototype.apply()` takes in an array of arguments, we have to define a default empty array parameter for `argArray` if we are to spread it, because we can only spread iterables.

Approach 1: Using `bind`

`bind`, `apply`, and `call` can be viewed as sibling functions. They're highly similar in terms of function signature and usage. Within `Function.prototype` methods, `this` refers to the `Function` object itself. If the `this` context is not used at all, the following will work:

```
Function.prototype.myApply = function (thisArg, argArray = []) {
  return this(...argArray);
};
```

However, `thisArg` is still used widely in modern code, so we need another way to do this. `Function.prototype.bind` creates a new function with a specified `this` value and initial arguments, without executing the original function immediately. It allows us to permanently bind a specific context (`this` value) to the function and partially apply arguments if needed. This is exactly what we need to bridge the gap in the solution above.

JavaScript TypeScript

```
/**
 * Calls the function, substituting the specified object for the this value of the function, and the specified array for
 * @param thisArg The object to be used as the this object.
 * @param argArray A set of arguments to be passed to the function.
 * @return {any}
 */
Function.prototype.myApply = function (thisArg, argArray = []) {
  return this.bind(thisArg)(...argArray);
};
```

Or you can also pass the `argArray` into `bind()` before executing it.

```
/**
 * Calls the function, substituting the specified object for the this value of the function, and the specified array fo
 * @param thisArg The object to be used as the this object.
 * @param argArray A set of arguments to be passed to the function.
 * @return {any}
 */
Function.prototype.myApply = function (thisArg, argArray = []) {
  return this.bind(thisArg, ...argArray)();
};
```

Approach 2: Using `call`

`Function.prototype.call` and `Function.prototype.apply` are very similar. Here's an easy way to remember each function's signature:

- `Function.prototype.call` takes in a **c**omma-separated list of arguments.
- `Function.prototype.apply` takes in an **a**rray of arguments.

```
/**
 * Calls the function, substituting the specified object for the this value of the function, and the specified array fo
 * @param thisArg The object to be used as the this object.
 * @param argArray A set of arguments to be passed to the function.
 * @return {any}
 */
Function.prototype.myApply = function (thisArg, argArray = []) {
  return this.call(thisArg, ...argArray);
};
```

Approach 3: Using `Symbol`

Another approach is to create a `Symbol` and add it as a property to a newly-created `Object` with `thisArg` bound to it. This is very similar to one of the solutions to the `Function.prototype.bind` question.

```
/**
 * Calls the function, substituting the specified object for the this value of the function, and the specified array fo
 * @param thisArg The object to be used as the this object.
 * @param argArray A set of arguments to be passed to the function.
 * @return {any}
 */
Function.prototype.myApply = function (thisArg, argArray = []) {
  const sym = Symbol();
  const wrapperObj = Object(thisArg);
  Object.defineProperty(wrapperObj, sym, {
    enumerable: false,
    value: this,
  });

  return wrapperObj[sym](...argArray);
};
```