
# Memo Decorator

This decorator applies memoization to a method of a class.

## Usage

Apply the decorator to a method of a class. The cache is local for the method but shared among all instances of the class. Strongly recommend you to **use this decorator only on pure methods.**

Installation:

```
npm i memo-decorator --save
```

### Configuration

```
export interface Config {
  resolver?: Resolver;
  cache?: MapLike;
}
```

- `Resolver` is a function, which returns the key to be used for given set of arguments. By default, the resolver will use the first argument of the method as the key.
- `MapLike` is a cache instance. By default, the library would use `Map`.

Example:

```
import memo from 'memo-decorator';

class Qux {
  @memo({
    resolver: (...args: any[]) => args[1],
    cache: new WeakMap(),
  })
  foo(a: number, b: number) {
    return a * b;
  }
}
```

### Demo

```
import memo from 'memo-decorator';

class Qux {
  @memo()
```

```
  foo(a: number) {
    console.log('foo: called');
    return 42;
  }

  @memo({
    resolver: (_) => 1,
  })
  bar(a: number) {
    console.log('bar: called');
    return 42;
  }
}

const a = new Qux();
// Create a new cache entry and associate `1` with the result `42`.
a.foo(1);
// Do not invoke the original method `foo` because there's already a cache
// entry for the key `1` associated with the result of the method.
a.foo(1);
// Invoke the original `foo` because the cache doesn't contain an entry
// for the key `2`.
a.foo(2);

// Invoke `bar` and return the result `42` gotten from the original `bar`
implementation.
a.bar(1);
// Does not invoke the original `bar` implementation because of the specified
`resolver`
// which is passed to `memo`. For any arguments of the function, the resolver will
return
// result `1` which will be used as the key.
a.bar(2);

const b = new Qux();
// Does not invoke the method `foo` because there's already an entry
// in the cache which associates the key `1` to the result `42` from the
// invocation of the method `foo` by the instance `a`.
b.foo(1);

// Outputs:
// foo: called
// foo: called
// bar: called
```

## License

MIT