JavaScript ES6

Lesson 18:
ES6 Meta Programming using Reflection & Proxy API

Lesson Objectives

At the end of this module you will be able to:

- Perform operations related to inspecting and manipulating methods and properties of objects.
- Create proxies using the Proxy API
- Understand what proxies are and how to use them
- Intercept various operations on the objects using traps

ES6 – Reflect API

ES6 introduces Reflect API to inspect and manipulating the properties of objects.

ES6 Reflect API is well organized and makes it easier to read and write code, as it doesn't throw exceptions on failure. Instead, it returns the Boolean value, representing if the operation was true or false.

All the methods of the ES6 Reflect API are wrapped in the Reflect object to make it look well organized.

Since developers are adapting to the Reflect API for the object reflection, it's important to learn this API in depth.

Reflect.construct()

Reflect.construct() method is used to invoke a function as a constructor.

Reflect.construct() method returns the new instance created by the target constructor.

Reflect.construct(**constructor, args, prototype**) method takes three arguments:

- The first argument is the target constructor.
- The second argument is an array, specifying the arguments of the target constructor
- The third argument is another constructor whose prototype will be used as the prototype of the target constructor.

Demo

Reflect-API-construct

Reflect.apply()

Reflect.apply() method is used to invoke a function with a given this value.

Function invoked by Reflect.apply() is called as the target function.

Reflect.apply(function, this, args) method takes three arguments

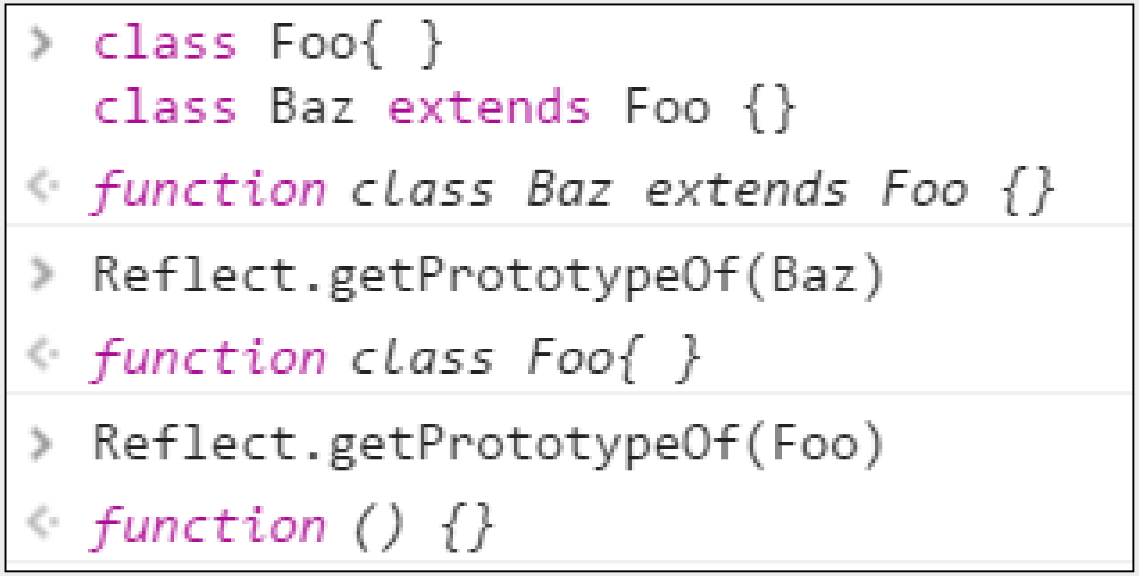
- The first argument represents the target function.
- The second argument represents the value of *this* inside the target function.
- The third argument is an array object, specifying the arguments of the target function.

Demo

Reflect-API-apply

Reflect.getPrototypeOf(object)

Reflect.getPrototypeOf() method is used to retrieve prototype of an object i.e. the value of the internal [[prototype]] property of an object.

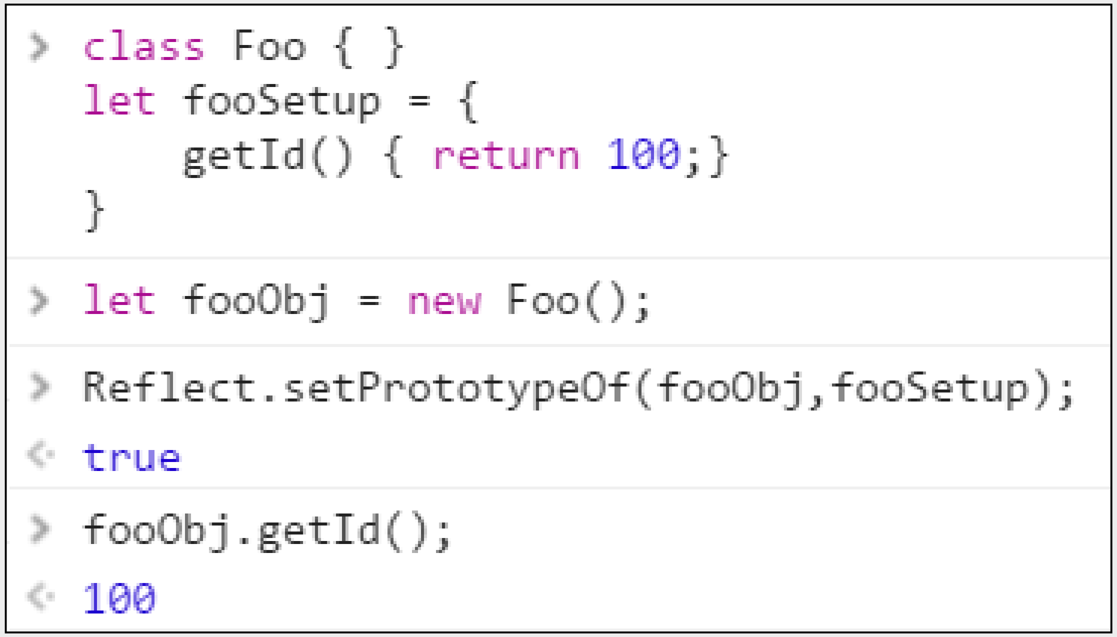


```
> class Foo{ }
  class Baz extends Foo {}
< function class Baz extends Foo {}
> Reflect.getPrototypeOf(Baz)
< function class Foo{ }
> Reflect.getPrototypeOf(Foo)
< function () {}
```

Reflect.setPrototypeOf(object, prototype)

Reflect.setPrototypeOf() is used to set the internal [[prototype]] property's value of an object.

The Reflect.setPrototypeOf() method will return true if the internal [[prototype]] property's value was set successfully. Otherwise, it will return false.

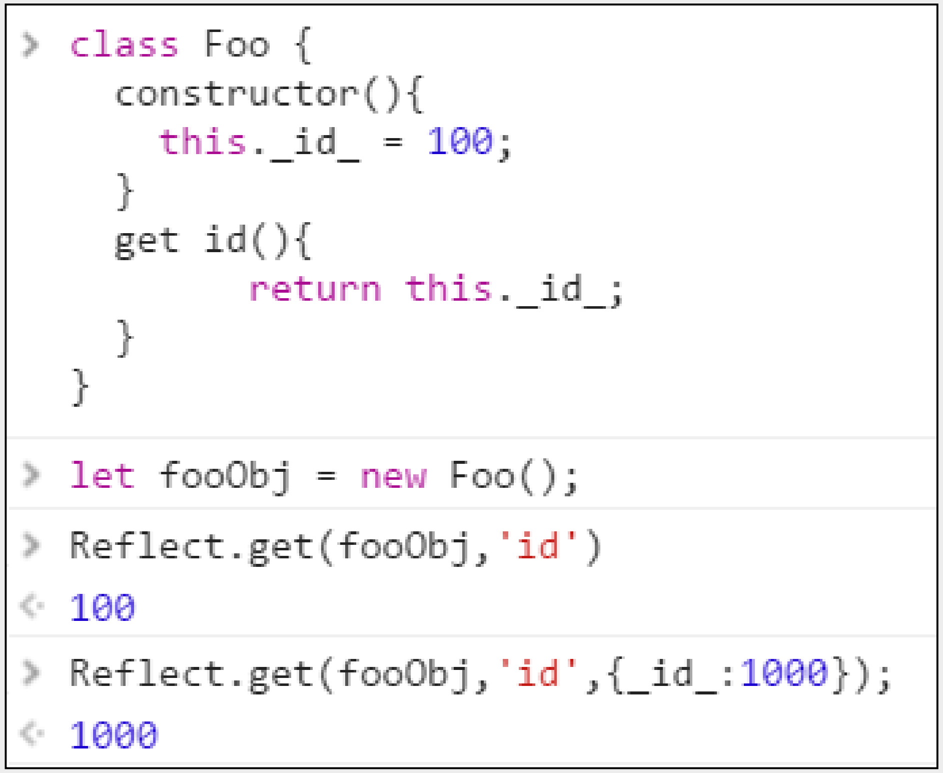


```
> class Foo { }
  let fooSetup = {
      getId() { return 100;}
  }
> let fooObj = new Foo();
> Reflect.setPrototypeOf(fooObj,fooSetup);
<· true
> fooObj.getId();
<· 100
```

Reflect.get(object, property, this)

Reflect.get() method is used to retrieve the value of an object's property.

Reflect.get(object, property, this) has three arguments : First argument is the target object, second argument is the property name and third will be the value of *this* inside get().

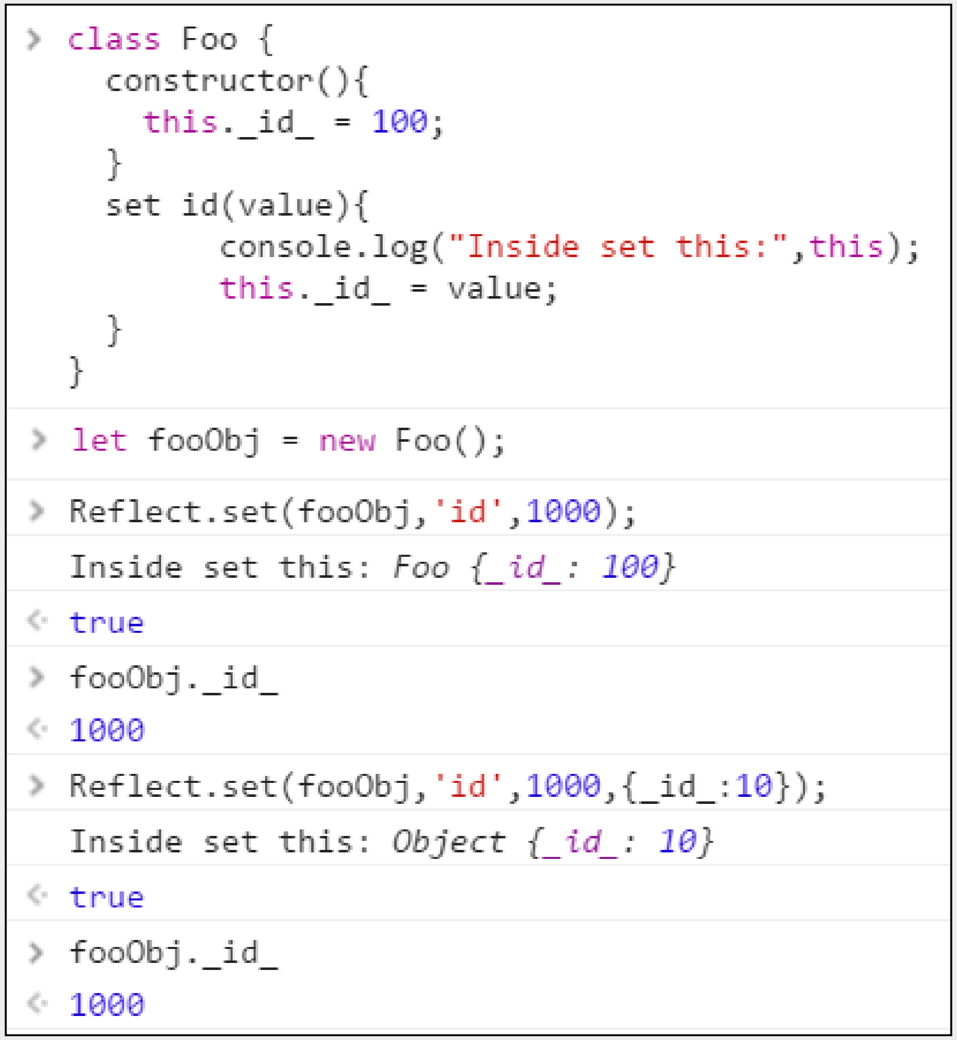


```
> class Foo {
    constructor(){
      this._id_ = 100;
    }
    get id(){
          return this._id_;
    }
  }
> let fooObj = new Foo();
> Reflect.get(fooObj,'id')
< 100
> Reflect.get(fooObj,'id',{_id_:1000});
< 1000
```

Reflect.set(object, property, value, this)

Reflect.set() method is used to set the value of an object's property.

Reflect.set(object, property, value, this) has 4 arguments : first argument is the object, the second argument is the property name, third argument is the property value and the fourth will be the value of *this* inside set().

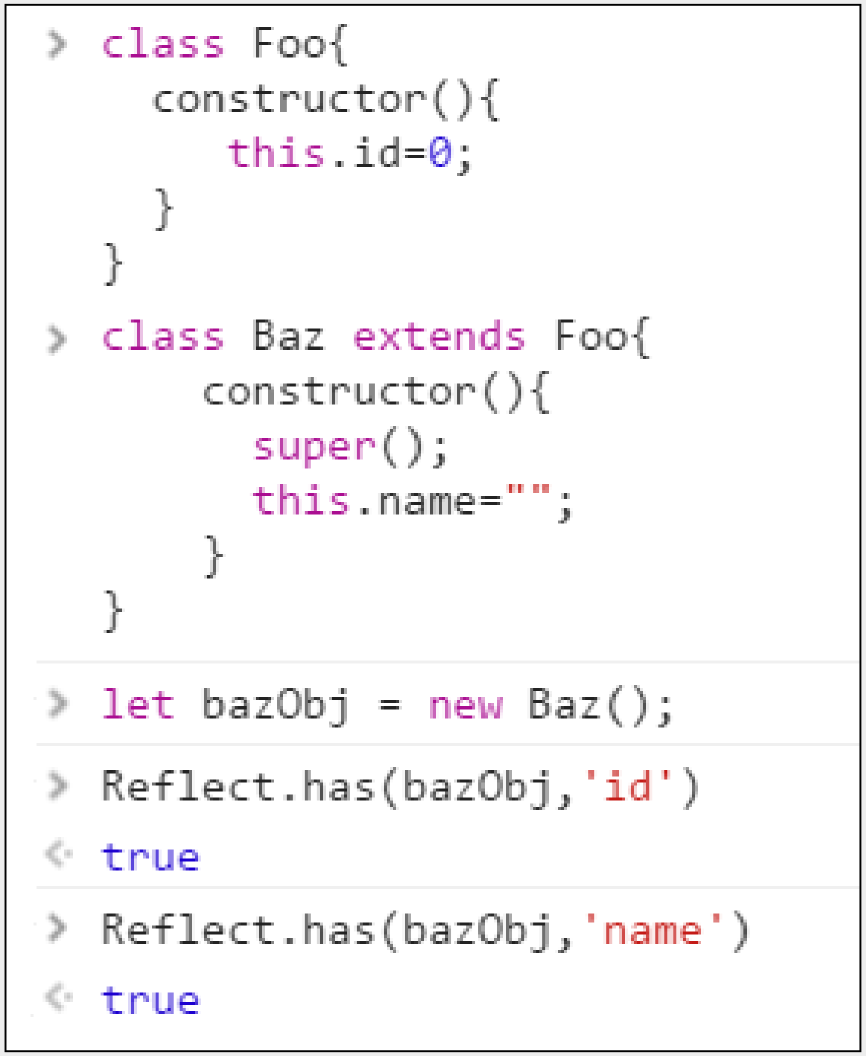


```
> class Foo {
    constructor(){
      this._id_ = 100;
    }
    set id(value){
          console.log("Inside set this:",this);
          this._id_ = value;
    }
  }
> let fooObj = new Foo();
> Reflect.set(fooObj,'id',1000);
  Inside set this: Foo {_id_: 100}
< true
> fooObj._id_
< 1000
> Reflect.set(fooObj,'id',1000,{_id_:10});
  Inside set this: Object {_id_: 10}
< true
> fooObj._id_
< 1000
```

Reflect.has(object, property)

Reflect.has() is used to check whether a property exists in an object.

It also checks for the inherited properties. It returns true if the property exists otherwise false.

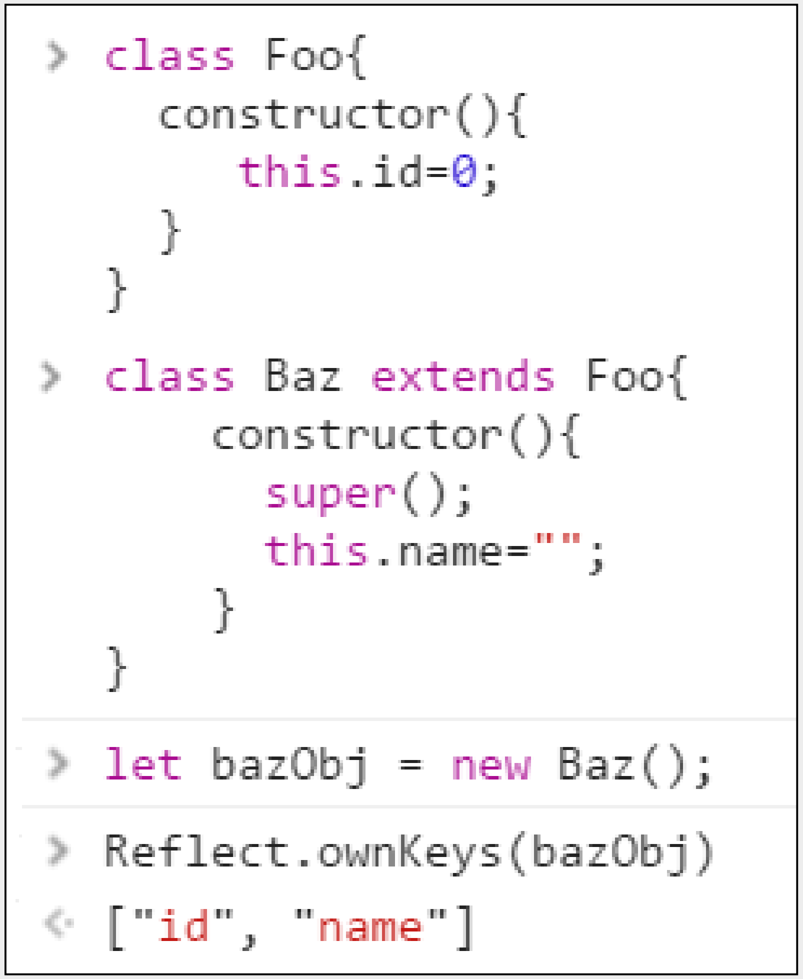


```
> class Foo{
    constructor(){
      this.id=0;
    }
  }
> class Baz extends Foo{
      constructor(){
        super();
        this.name="";
      }
  }
> let bazObj = new Baz();
> Reflect.has(bazObj,'id')
< true
> Reflect.has(bazObj,'name')
< true
```

Reflect.ownKeys(object)

Reflect.ownKeys() method returns an array whose values represent the keys of the properties of an provided object.

It ignores the inherited properties

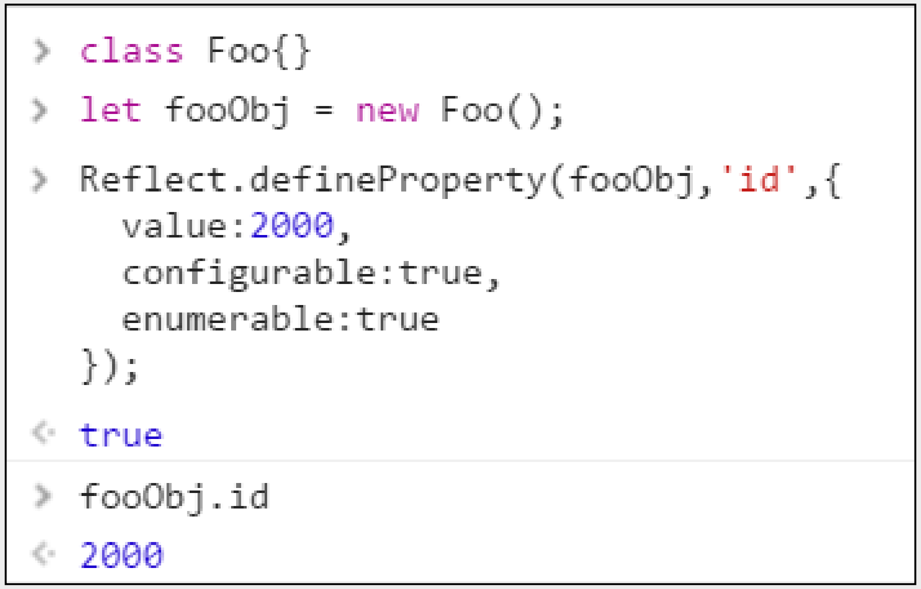


```
> class Foo{
    constructor(){
      this.id=0;
    }
  }
> class Baz extends Foo{
      constructor(){
        super();
        this.name="";
      }
  }
> let bazObj = new Baz();
> Reflect.ownKeys(bazObj)
< ["id", "name"]
```

Reflect.defineProperty(object, property, descriptor)

Reflect.defineProperty() defines a new property directly on an object, or modifies an existing property on an object.

It takes three arguments: object that is used to define or modify a property, symbol or name of the property that is to be defined or modified and descriptor for the property that being defined or modified



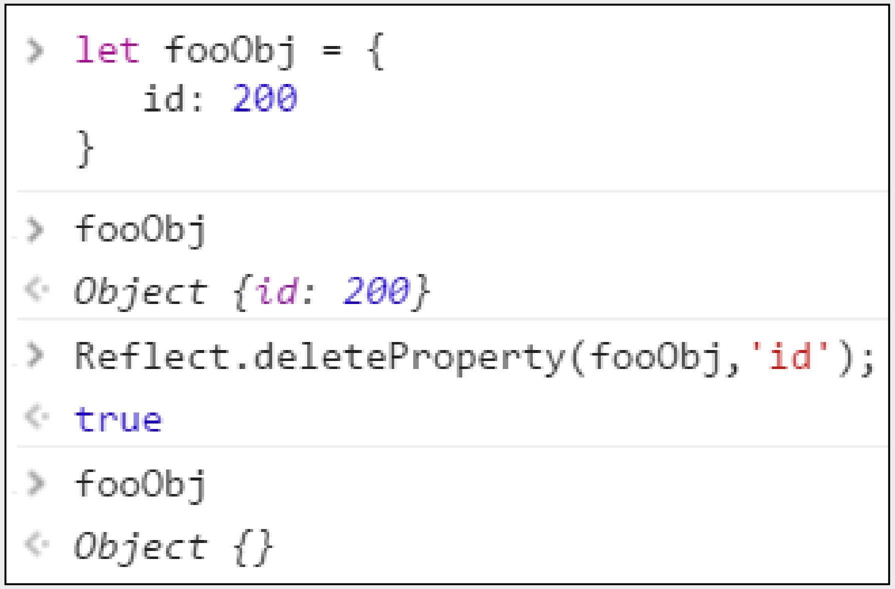
```
> class Foo{}
> let fooObj = new Foo();
> Reflect.defineProperty(fooObj,'id',{
    value:2000,
    configurable:true,
    enumerable:true
  });
< true
> fooObj.id
< 2000
```

Reflect.deleteProperty(object, property)

Reflect.deleteProperty() method is used to delete a property of an object.

This method takes two arguments : first argument is the reference to the object and the second argument is the name of the property to delete.

Returns true if the property is deleted successfully, otherwise, it returns false

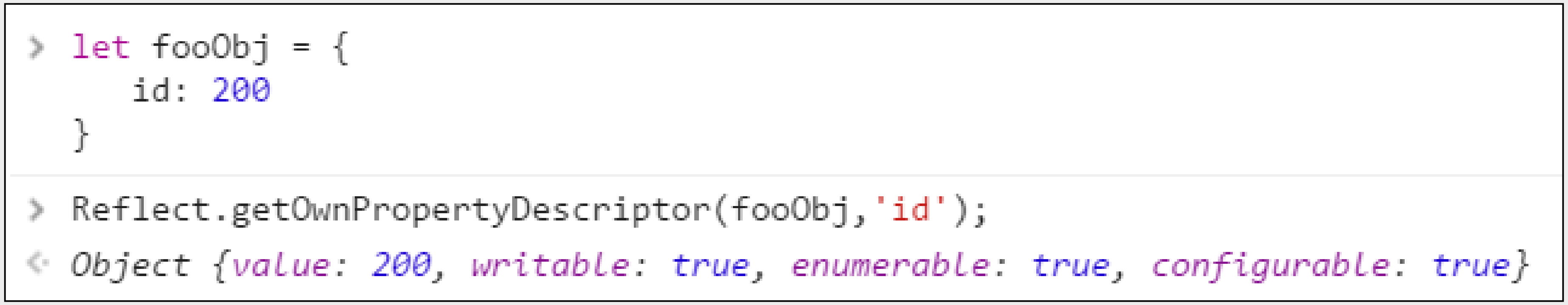


```
> let fooObj = {
      id: 200
  }
> fooObj
< Object {id: 200}
> Reflect.deleteProperty(fooObj,'id');
< true
> fooObj
< Object {}
```

Reflect.getOwnPropertyDescriptor(object, property)

Reflect.getOwnPropertyDescriptor() method is used to retrieve the descriptor of an object's property.

It takes two arguments: The first argument is the object and the second argument is the property name.



```
> let fooObj = {
      id: 200
  }
> Reflect.getOwnPropertyDescriptor(fooObj,'id');
< Object {value: 200, writable: true, enumerable: true, configurable: true}
```

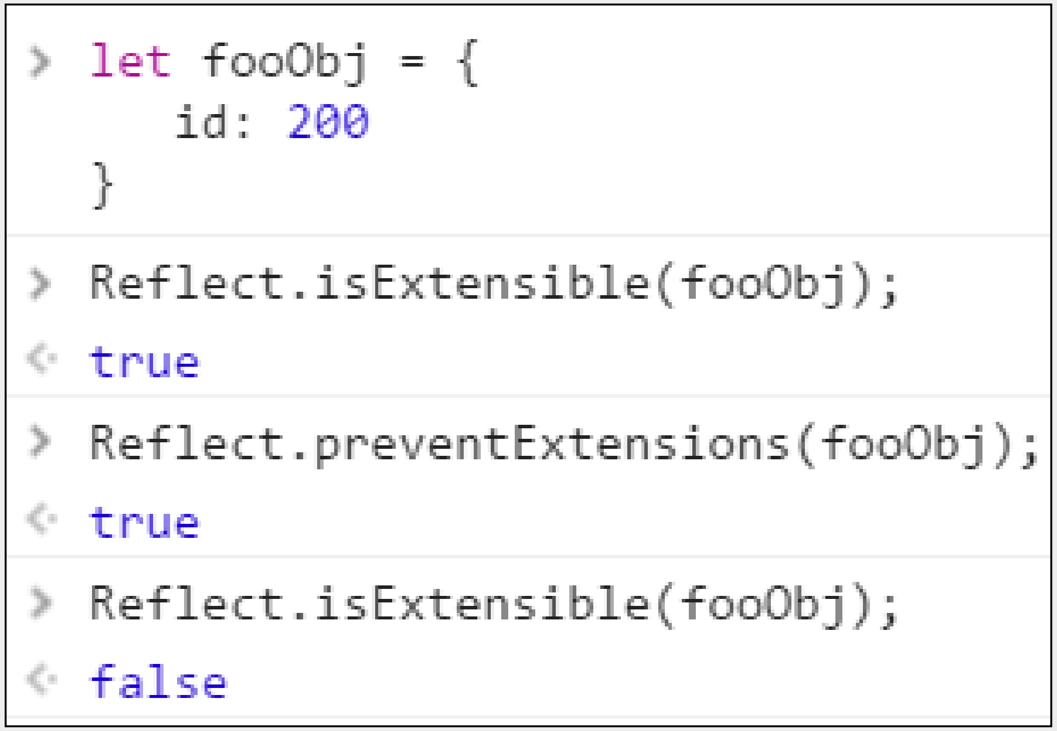
Reflect.preventExtensions(object)

Reflect.preventExtensions() is used to mark an object as non-extensible.

It returns a Boolean, indicating whether the operation was successful or not.

Reflect.isExtensible(object)

Reflect.isExtensible() is used to check whether an object is extensible or not i.e. whether new properties can be added to an object or not.



```
> let fooObj = {
     id: 200
  }
> Reflect.isExtensible(fooObj);
< true
> Reflect.preventExtensions(fooObj);
< true
> Reflect.isExtensible(fooObj);
< false
```

Introduction to Proxies

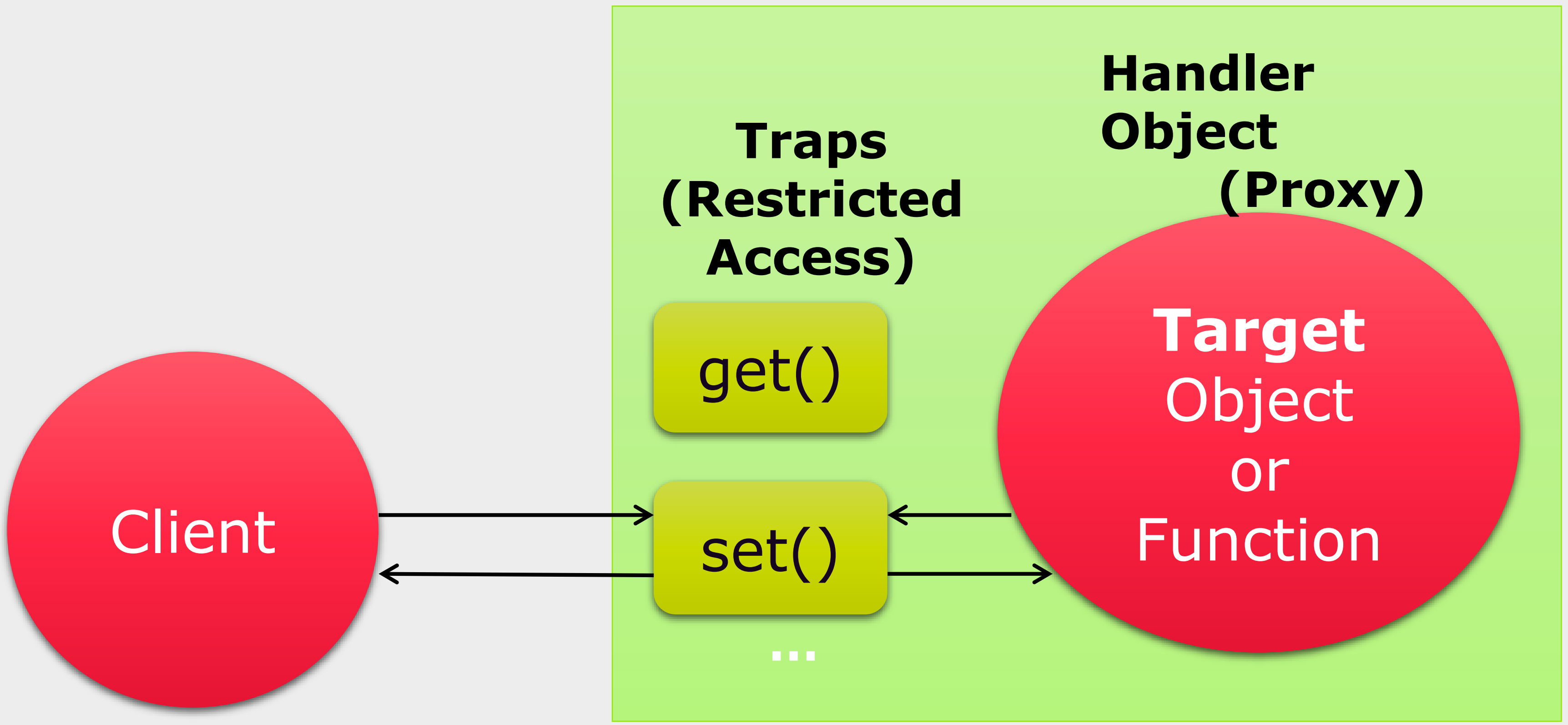
ES6 introduced the Proxy API, using that proxies can be created for objects and functions.

A proxy acts like a wrapper for an object, and defines the custom behavior for the fundamental operations on the object.

Once an object is wrapped using a proxy, all the operations that are supposed to be done on the object should now be done on the proxy object, so that the custom behavior can take place.

Some fundamental operations on the objects are property lookup, property assignment, constructor invocation, enumeration, and so on

Introduction to Proxies



Proxies are used to define the custom behavior for the fundamental operations on objects.

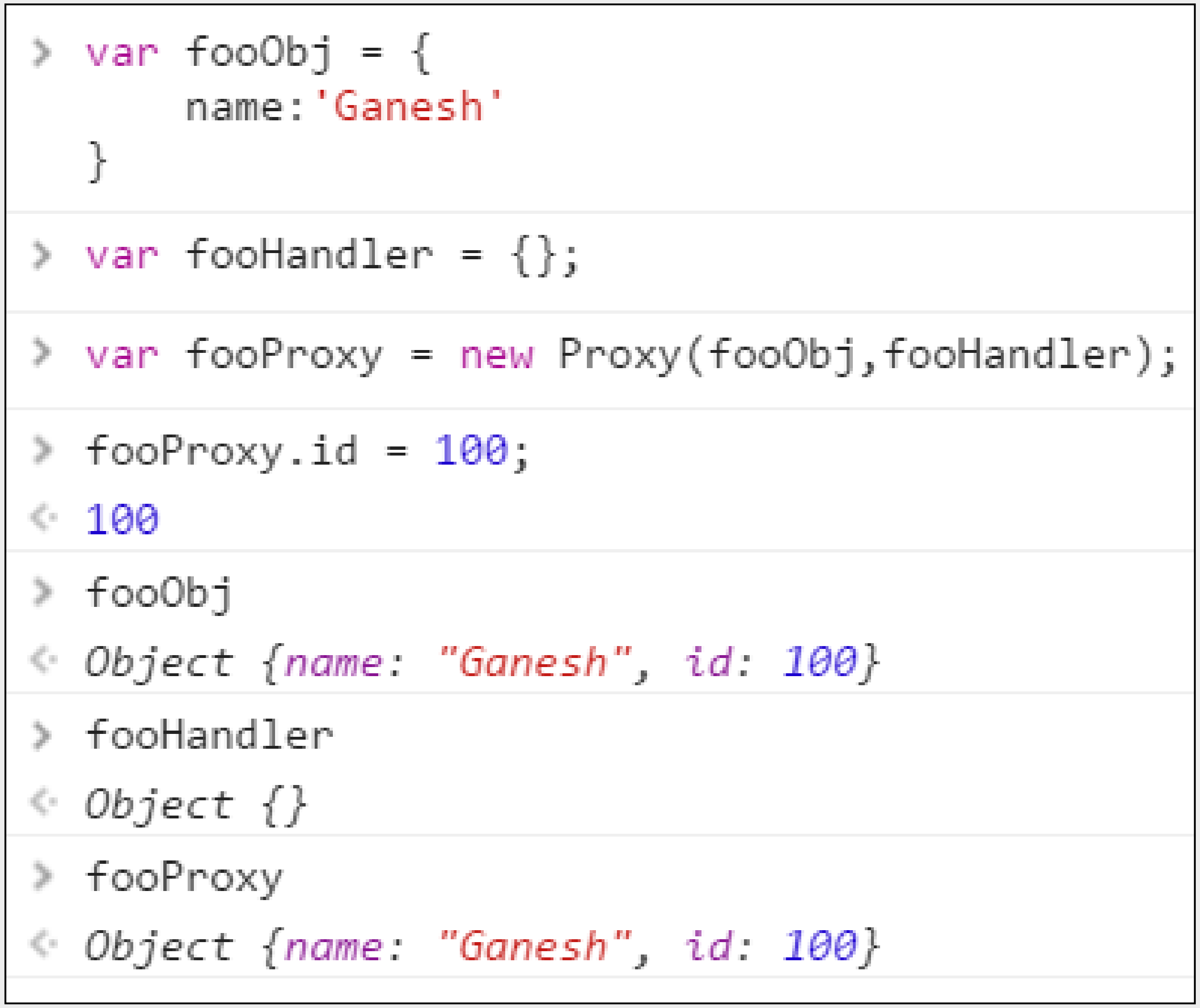
Proxy API

ES6 Proxy API provides the Proxy constructor to create proxies. The Proxy constructor takes two arguments :

- **Target**: This is the object that will be wrapped by the proxy
- **Handler**: This is an object that contains the traps for the target object.

A trap can be defined for every possible operation on the target object. If a trap is not defined, then the default action takes place on the target.

Creating Proxy



```
> var fooObj = {
      name:'Ganesh'
  }
> var fooHandler = {};
> var fooProxy = new Proxy(fooObj,fooHandler);
> fooProxy.id = 100;
<· 100
> fooObj
<· Object {name: "Ganesh", id: 100}
> fooHandler
<· Object {}
> fooProxy
<· Object {name: "Ganesh", id: 100}
```

Traps

Traps are functions that intercept various operations on the target object, and define the custom behavior for those operations.

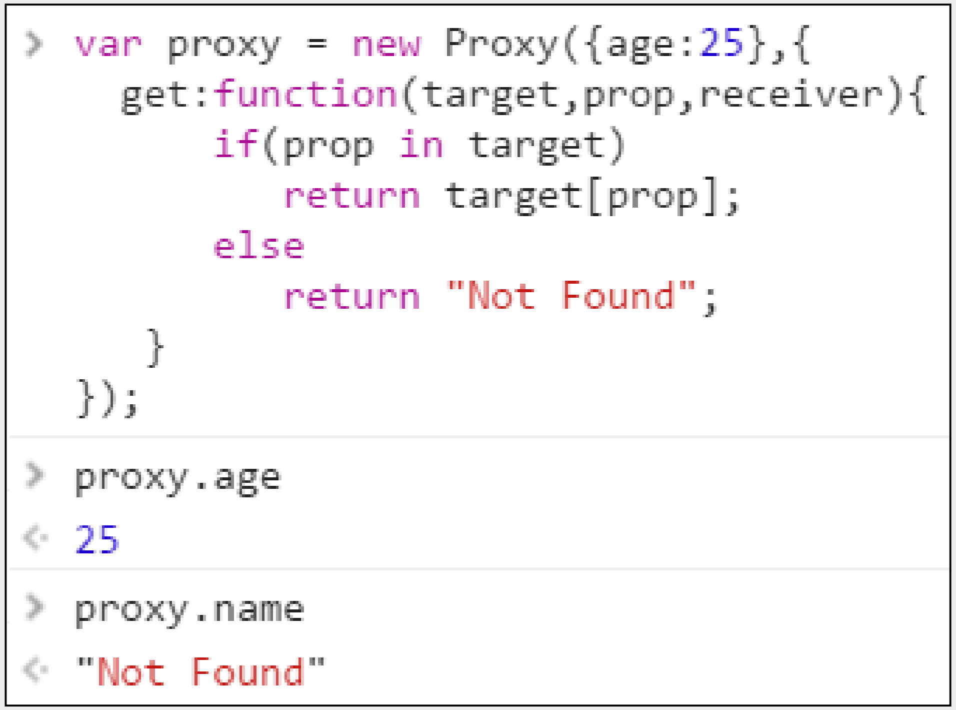
Traps in Proxy API are allegiant to Reflection API.

Traps in ES6		
handler.construct()	handler.ownKeys()	handler.get()
handler.getPrototypeOf()	handler.setPrototypeOf()	handler.set()
handler.isExtensible()	handler.defineProperty()	handler.has()
handler.preventExtensions()	handler.deleteProperty()	handler.apply()
handler.getOwnPropertyDescriptor()		

get(target, property, receiver)

get trap is executed when we retrieve a property value using the dot or bracket notation, or the Reflect.get() method.

It takes three parameters : the target object, the property name, and the proxy.



```
> var proxy = new Proxy({age:25},{
    get:function(target,prop,receiver){
        if(prop in target)
           return target[prop];
        else
           return "Not Found";
     }
  });
> proxy.age
< 25
> proxy.name
< "Not Found"
```

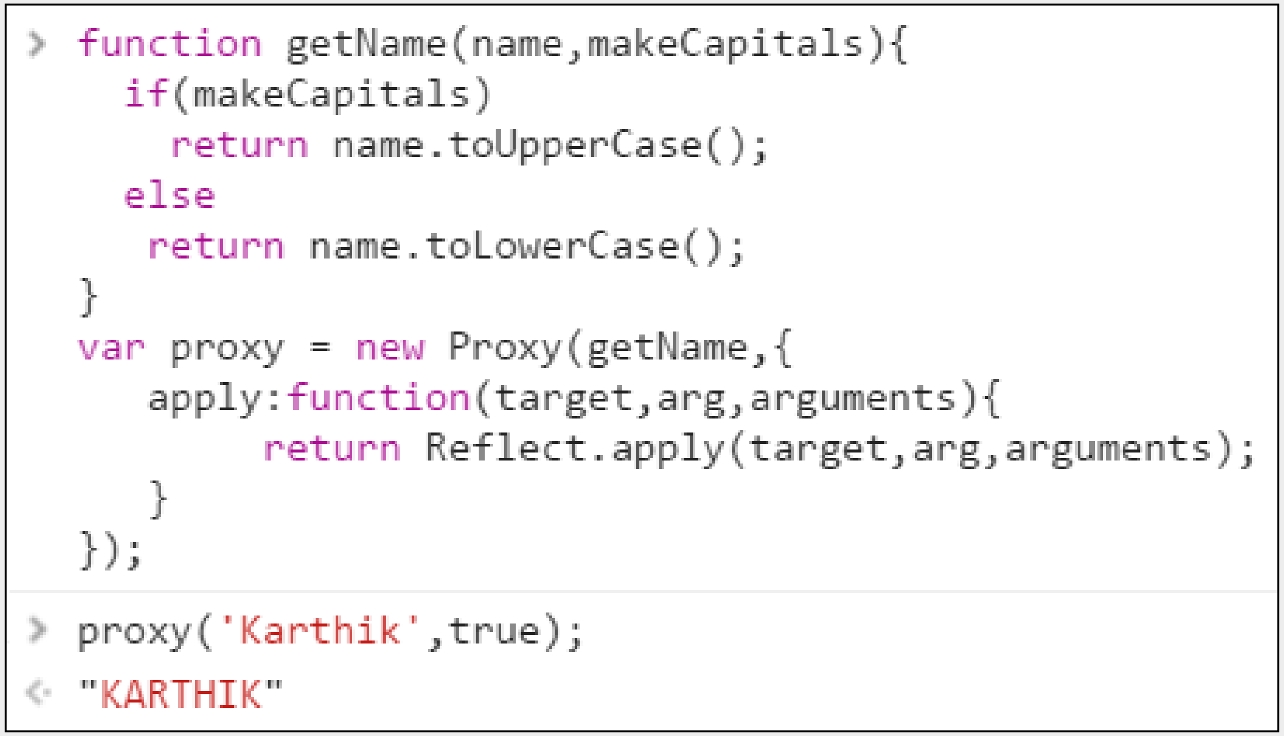
Demo

Proxy-API-get

apply(target, thisvalue, arguments)

apply trap is used to call the functions, it gets executed for function's apply(), call() and the Reflect.apply() method.

It takes three parameters : the target object, arg, and the argument list.

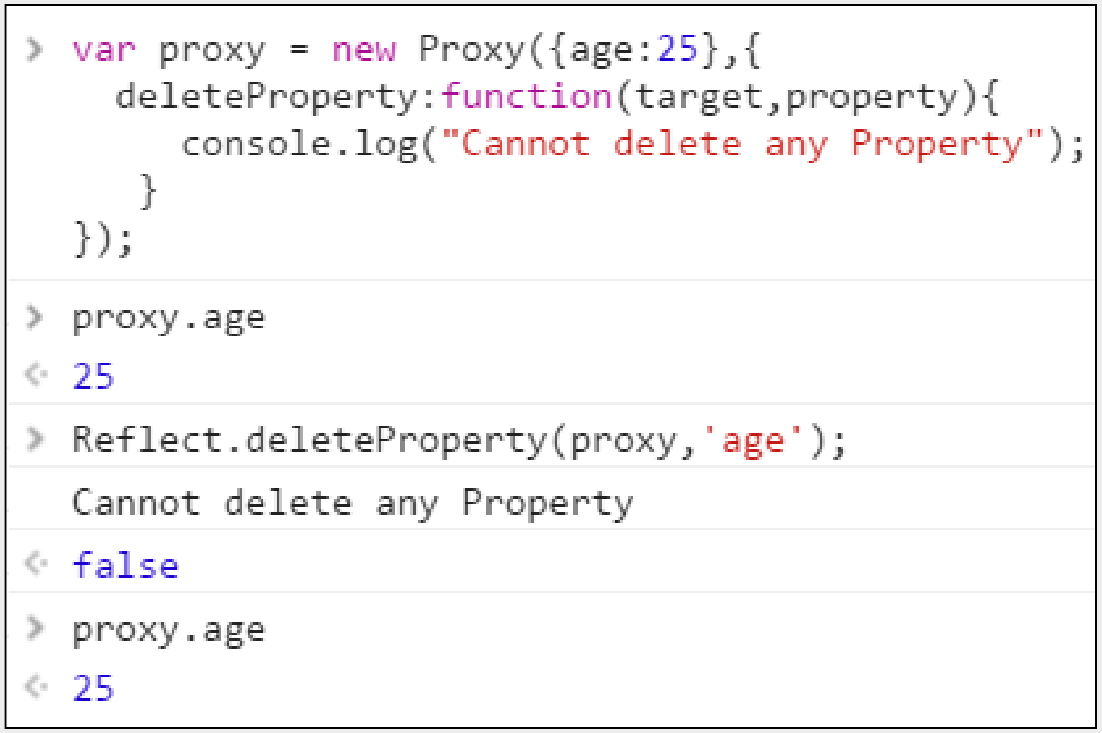


```
> function getName(name,makeCapitals){
    if(makeCapitals)
      return name.toUpperCase();
    else
     return name.toLowerCase();
  }
  var proxy = new Proxy(getName,{
     apply:function(target,arg,arguments){
          return Reflect.apply(target,arg,arguments);
     }
  });
> proxy('Karthik',true);
< "KARTHIK"
```

deleteProperty(target, property)

deleteProperty trap is executed, when a property is deleted using either the delete operator or the Reflect.deleteProperty() method..

It takes two parameters: target object and the property name.



```
> var proxy = new Proxy({age:25},{
    deleteProperty:function(target,property){
       console.log("Cannot delete any Property");
     }
  });
> proxy.age
<· 25
> Reflect.deleteProperty(proxy,'age');
  Cannot delete any Property
<· false
> proxy.age
<· 25
```

Proxy.revocable(target, handler)

A revocable proxy is a proxy that can be revoked. i.e. it will be turned off permanently.

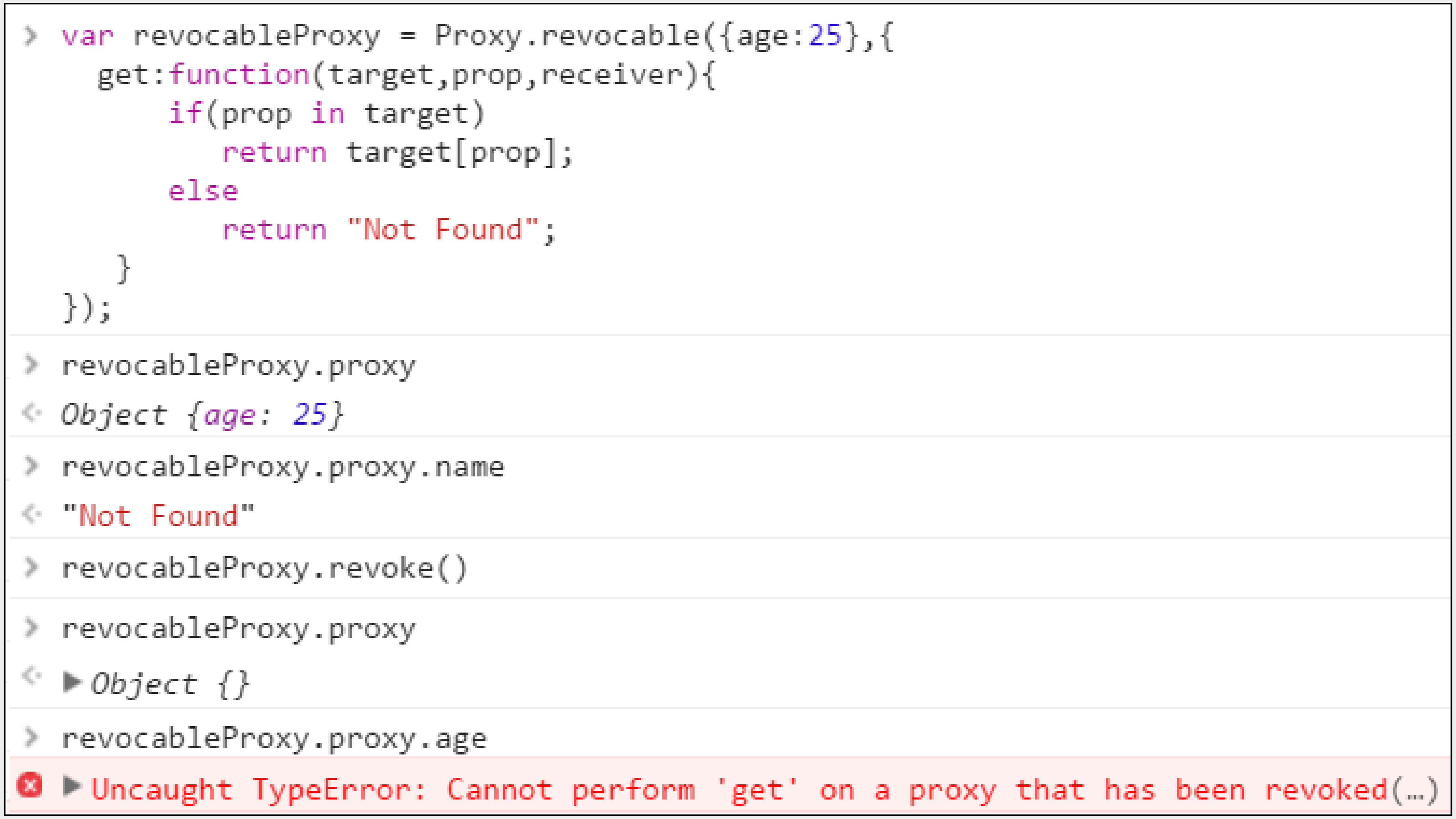
To create the revocable proxies, use Proxy.revocable() method.

This method takes the same arguments as the Proxy constructor, but instead of returning a revocable proxy instance directly, it returns an object with two properties

- **proxy**: This is the revocable proxy object
- **revoke**: When this function is called, it revokes the proxy

Once a revocable proxy is revoked, any attempts to use it will throw a TypeError exception.

Proxy.revocable(target, handler)



```
> var revocableProxy = Proxy.revocable({age:25},{
    get:function(target,prop,receiver){
        if(prop in target)
           return target[prop];
        else
           return "Not Found";
     }
  });
> revocableProxy.proxy
<· Object {age: 25}
> revocableProxy.proxy.name
<· "Not Found"
> revocableProxy.revoke()
> revocableProxy.proxy
<· ▶ Object {}
> revocableProxy.proxy.age
✖ ▶ Uncaught TypeError: Cannot perform 'get' on a proxy that has been revoked(…)
```

Summary

Reflect API is used to inspect and manipulate the properties of objects

Reflect API methods returns the Boolean value, representing the operation was done or not

Proxies are used to define the custom behavior for the fundamental operations on objects.

A proxy acts like a wrapper for an object, and defines the custom behavior for the fundamental operations on the object.

Traps are the functions that intercept various operations on the target object, and define the custom behavior for it.