```
class Class {
    constructor(x=5,y=5) {
        this.x = x;
        this.y = y;
    }

    toString(){
        return `X: ${this.x}, Y: ${this.y}`
    }
}

function fun(a, b) {
    return a**b
}
```

get(target, property, receiver): This trap is called whenever a property of the target object is accessed, including property lookup, property enumeration, and property access through the in operator.

The **target** argument is the target object, the **property** argument is the name of the property being accessed, and the **receiver** argument is the object through which the property access is being performed. The return value of the get trap is used as the result of the property access.

```
get(target, property, receiver){
    return target[property]
}
```

set(target, property, value, receiver): This trap is called whenever a property of the target object is assigned a value.

The **target** argument is the target object, the **property** argument is the name of the property being assigned, the **value** argument is the value being assigned, and the **receiver** argument is the object through which the assignment is being performed. The set trap should return a Boolean value indicating whether the assignment was successful.

```
set(target, property, value, receiver){
    target[property] = value
    return true
}
```

apply(target, thisArg, argumentsList): This trap is called whenever a function-like object is called.

The **target** argument is the target object, the **thisArg** argument is the value of `this` inside the target function, and the **argumentsList** argument is an array-like object

containing the arguments passed to the target function. The return value of the apply trap is used as the result of the function call.

```
apply(target, thisArg, argumentsList){
    return target(...argumentsList) * 10
}
```

```
let o = new Proxy(fun, handler)
console.log(o(2,5)) //320
```

construct(target, argumentsList, newTarget): This trap is called whenever the new operator is used with the proxy as a constructor. The target argument is the target object, the argumentsList argument is an array-like object containing the arguments passed to the constructor, and the newTarget argument is the constructor being used to create the new object. The return value of the construct trap is used as the newly created object.

```
construct(target, argArray, newTarget) {
    return new target(...argArray)
}
```

```
let c = new Class();
let o1 = new Proxy(c.constructor, handler)
let o2 = new o1(10,10) // new object of type Class with x = 10, y = 10
```

has(target, property): called whenever the in operator is used to check if a property exists in an object. It receives the **target** object and the **property** being checked. The `has` trap should return a Boolean indicating whether the property exists.

```
has(target, property){
    return target[property] !== undefined;
}
```

```
console.log('x' in o2) //true
console.log('z' in o2) //false
```

deleteProperty(target, property): called whenever the delete operator is used to delete a property from an object. It receives the **target** object and the **property** being deleted. The `deleteProperty` trap should return a Boolean indicating whether the deletion was successful.

```
deleteProperty(target, property){
    if(target[property] !== undefined)
    {
        delete target[property]
        console.log(`Property ${property} was deleted`)
        return true
    }

    return false
}
```

```
let o3 = new Proxy(new Class(10,10), handler)
delete o3.x //Property x was deleted
console.log('x' in o3) //false
```

defineProperty(target, property, descriptor): called whenever the `Object.defineProperty()` method is used to define a new property on an object or change the properties of an existing one. It receives the **target** object, the **property** being defined or changed, and a property **descriptor** object. The `defineProperty` trap should return a Boolean indicating whether the property definition or change was successful.

```
defineProperty(target, property, descriptor){
    if(target[property] === undefined){
        target[property] = descriptor.value
        return true
    }
}
```

```
Object.defineProperty(o3,'x', { value: 10 })

console.log('x' in o3) //true
```

getOwnPropertyDescriptor(target, property): called whenever the `Object.getOwnPropertyDescriptor()` method is used to retrieve the property descriptor of a property. It receives the **target** object and the **property** being retrieved. The `getOwnPropertyDescriptor` trap should return the property descriptor object.

```
getOwnPropertyDescriptor(target, property){
    if(target[property] !== undefined){

        return {
            value: target[property],
            writable: false,
            enumerable: false,
            configurable: true
        }
    }
}
```

```
console.log(Object.getOwnPropertyDescriptor(o3,'x'))
//{ value: 10, writable: false, enumerable: false, configurable: true }
```

ownKeys(target): called whenever the `Object.getOwnPropertyNames()` or `Object.getOwnPropertySymbols()` methods are used to retrieve the names of all own properties of an object. It receives the **target** object. The ownKeys trap should return an array of the names of all own properties of the target object.

```
ownKeys(target) {
    let arr = []
    for (const targetKey in target) {
        arr.push(targetKey)
    }

    return arr
}
```

```
console.log(Object.getOwnPropertyNames(o3)) // [ 'y', 'x' ]
```

preventExtensions(target): called whenever the `Object.preventExtensions()` method is used to make an object non-extensible. It receives the **target** object. The `preventExtensions` trap should return a Boolean indicating whether the object is now non-extensible.

```
preventExtensions(target){
    target.canEvolve = false;
    Object.preventExtensions(target);
    return true;
},
```

```
Object.preventExtensions(o3)
console.log(o3.canEvolve) // false
```

isExtensible(target): called whenever the `Object.isExtensible()` method is used to check if an object is extensible. It receives the **target** object. The `isExtensible` trap should return a boolean indicating whether the object is extensible.

```
isExtensible(target){
    return target.canEvolve || target.canEvolve === undefined
},
```

```
console.log(Object.isExtensible(o3)) // false
```

getPrototypeOf(target): is called whenever the `Object.getPrototypeOf()` method is used to retrieve the prototype of an object. It receives the **target** object as its argument and should return the prototype of the target object.

setPrototypeOf(target): is called whenever the Object.setPrototypeOf() method is used to set the prototype of an object. It receives the **target** object and the new prototype as its arguments and should return a Boolean indicating whether the prototype was successfully set.

These traps allow for a high degree of control and customization over how objects and functions are manipulated in JavaScript. However, it's important to use them responsibly and with a clear understanding of their behavior, as their use can have unexpected consequences if not done correctly.