1)

```
let randomValue = { name: "Lydia" }
randomValue = 23

if (!typeof randomValue === "string") {
	console.log("It's not a string!")
} else {
	console.log("Yay it's a string!")
}
```

Output:

```
Yay it's a string!
```

Explanation:

The condition within the if statement checks whether the value of !typeof randomValue is equal to "string". The ! operator converts the value to a boolean value. If the value is true, the returned value will be false, if the value is false, the returned value will be true. In this case, the returned value of typeof randomValue is the true value "string", which means that the value of !typeof randomValue is the boolean value false.

!typeof randomValue === "string" always returns false, since we're actually checking false === "string". Since the condition returned false, the code block of the else statement gets run, and Yay it's a string! gets logged.

2)

```
const user = {
	email: "my@email.com",
	updateEmail: email => {
		this.email = email
	}
}

user.updateEmail("new@email.com")
console.log(user.email)
```

Output:

```
my@email.com
```

Explanation:

The updateEmail function is an arrow function, and is not bound to the user object. This means that the keyword this is not referring to the user object, but in this case refers to the global scope. The value of email within the user object does not get updated. When the value of user.email is logged, the original value of my@email.com gets returned.

3)

```
const user = {
	email: "e@mail.com",
	password: "12345"
}

const updateUser = ({ email, password }) => {
	if (email) {
		Object.assign(user, { email })
	}

	if (password) {
		user.password = password
	}

	return user
}

const updatedUser = updateUser({ email: "new@email.com" })

console.log(updatedUser === user)
```

Output:

```
true
```

Explanation:

The updateUser function updates the values of the email and password properties on user, if the values of the users email and password attributes are passed to the function. The returned value of the updateUser function is the user object, which means that the value of updatedUser is a reference to the same user object pointed to by user. updatedUser === user equals true.

4)

```
const add = x => x + x;

function myFunc(num = 2, value = add(num)) {
    console.log(num, value);
}

myFunc();
myFunc(3);
```

Output:

```
2 4
3 6
```

Explanation:

First, we invoked myFunc() without passing any arguments. Since we didn't pass arguments, num and value got their default values: num is 2, and value as a function add return value. To the add function, we pass a value of 2 as the value for the num as an argument.   Function add returns 4, which is the value of value.

Then, we invoked myFunc(3) and passed the value 3 as the value for the argument num. We didn't pass an argument for value. Since we didn't pass a value for the value argument, it got the default value: the returned value of the add function. To add, we pass num, which has the value of 3. Function add returns 6, which is the value of value.

5)

```
const handler = {
    set: () => console.log('Added a new property!'),
    get: () => console.log('Accessed a property!'),
};

const person = new Proxy({}, handler);
person.name = 'Lydia';
person.name;
```

Output:

```
Added a new property!
Accessed a new property!
```

Explanation:

Using   Proxy object, we can add custom behavior to an object. In this case we pass to it as the second argument as an handler object which contained properties: set and get. set gets invoked whenever we set property values, get gets invoked whenever we get (access) property values.

The first argument is an empty object {}, which is the value of person. For this object, the custom behavior specified in the handler object gets added. If we add a property to the person object, set will get invoked. If we access a property on the person object, get gets invoked.

First, we added a new property name to the proxy object (person.name = "Lydia"). set gets invoked, and logs "Added a new property!".

Then, we access a property value on the proxy object, the get property on the handler object got invoked. "Accessed a property!" gets logged.

6)

```
const myPromise = Promise.resolve('Woah some cool data');

(async () => {
    try {
        console.log(await myPromise);
    } catch {
        throw new Error(`Oops didn't work`);
    } finally {
        console.log('Oh finally!');
    }
})();
```

Output:

```
Woah some cool data
Oh finally!
```

Explanation:

In the try block, we're logging the awaited value of the myPromise variable: "Woah some cool data". Since no errors were thrown in the try block, the code in the catch block doesn't run. The code in the finally block always runs, "Oh finally!" is also printed.

7)

```
const groceries = ['banana', 'apple', 'peanuts'];

if (groceries.indexOf('banana')) {
    console.log('We have to buy bananas!');
} else {
    console.log(`We don't have to buy bananas!`);
}
```

Output:

```
We don't have to buy bananas!
```

Explanation:

We passed a state groceries.indexOf("banana") to the if conditional statement.
groceries.indexOf("banana") returns 0, a value of falsy. Because of the falsy statement, the code in the else block runs, and We don't have to buy bananas! is the output.

8)

```
function* generatorOne() {
    yield ['a', 'b', 'c'];
}

function* generatorTwo() {
    yield* ['a', 'b', 'c'];
}

const one = generatorOne();
const two = generatorTwo();

console.log(one.next().value);
console.log(two.next().value);
```

Output:

```
[ 'a', 'b', 'c' ]
a
```

Explanation:

With the yield keyword, we yield values in a generator function. With the yield* keyword, we can yield values from another generator function, or can transversw object (for example an array).

In generatorOne, we yield the entire array ['a', 'b', 'c'] using the yield keyword. The value of the property of the object returned by the function one through the next value method(one.next().value) is equivalent to an array ['a', 'b', 'c'].
In generatorTwo, we use yield* keyword. Equivalent function two of a yield value equivalent to the first iterator yield value. Array ['a', 'b', 'c'] is this iterator. The first yield value is a, so our first call (two.next().value) return a.