*Day 3

- **Method** – Methods facilitate communication between objects.
  - **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

- **Object** – An object is a real-time representation of any entity
- **Encapsulation** – The process of wrapping property and function within a single unit is known as encapsulation.
- **Inheritance** – It is a concept in which some property and methods of an Object is being used by another Object
- **Class** – A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
  - **Constructors** – Responsible for allocating memory for the objects of the class.

Object-Oriented Programming Concepts

* Classes

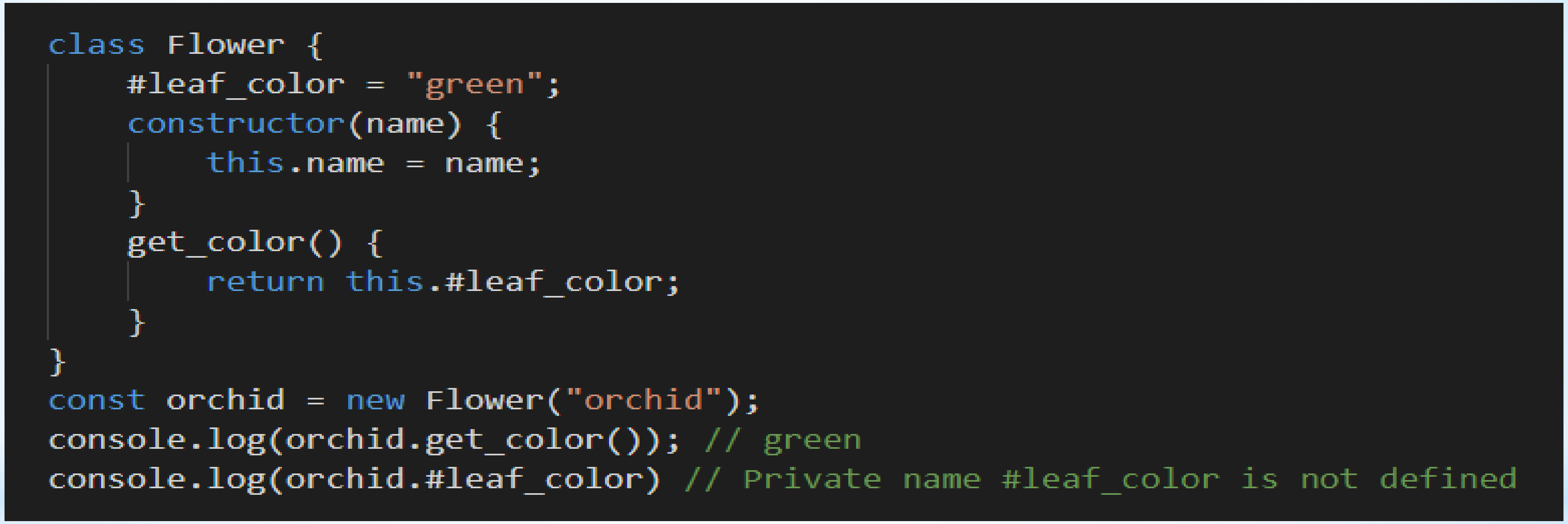
* Object Orientation, considers a program as a collection of objects that communicates with each other via mechanism called **methods**.
* A class definition can include the following –
  * **Constructors** – Responsible for allocating memory for the objects of the class.
  * **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

* Creating Objects
  * To create an instance of the class, use the new keyword followed by the class name.
* Accessing Functions
  * A class's attributes and functions can be accessed through the object. Use the '.' **dot notation** (called as the period) to access the data members of a class.

*Accessing Functions

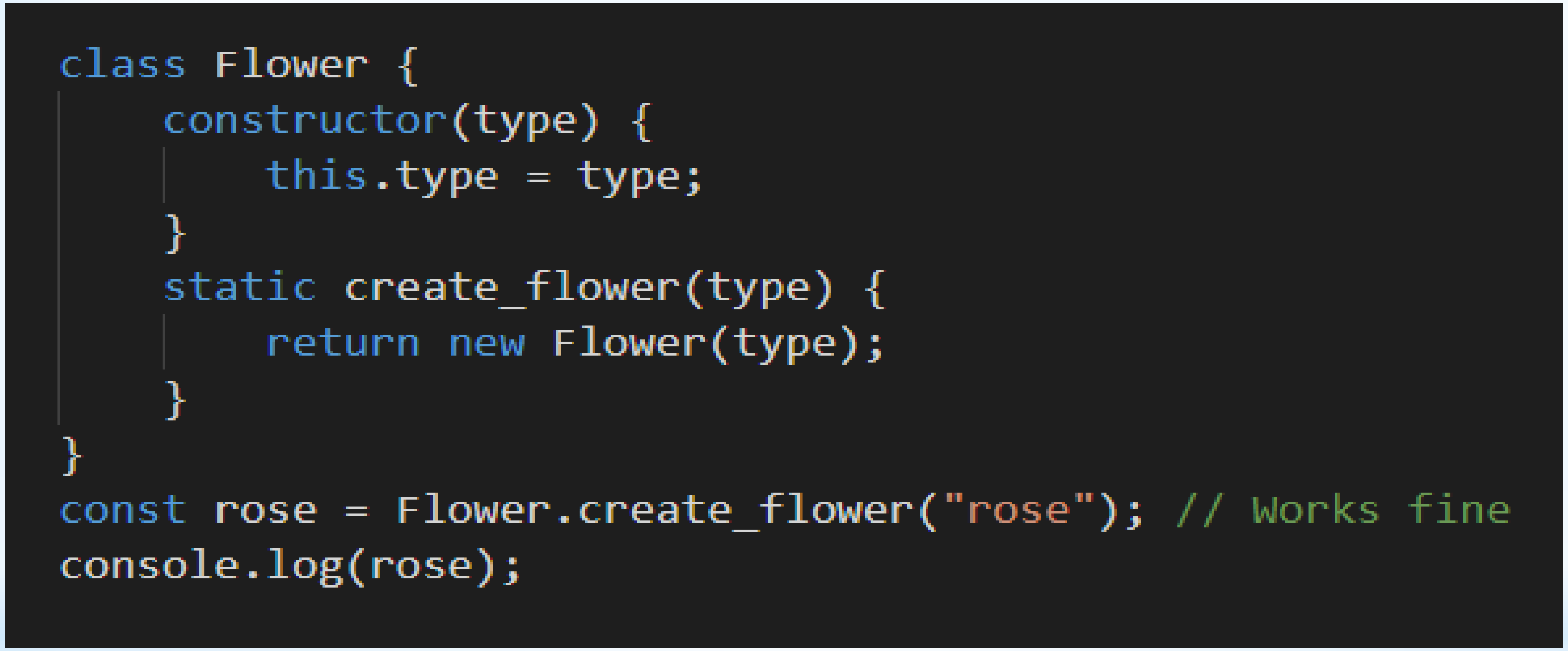
*Class Private Fields

* JavaScript supports class syntax since ES6, but only now were private fields introduced. To define a private property, it has to be prefixed with the hash symbol: #.



```javascript
class Flower {
    #leaf_color = "green";
    constructor(name) {
        this.name = name;
    }
    get_color() {
        return this.#leaf_color;
    }
}
const orchid = new Flower("orchid");
console.log(orchid.get_color()); // green
console.log(orchid.#leaf_color) // Private name #leaf_color is not defined
```

* a class method can now be declared with the static keyword and called from outside of a class.



```
class Flower {
    constructor(type) {
        this.type = type;
    }
    static create_flower(type) {
        return new Flower(type);
    }
}
const rose = Flower.create_flower("rose"); // Works fine
console.log(rose);
```

*Static Fields

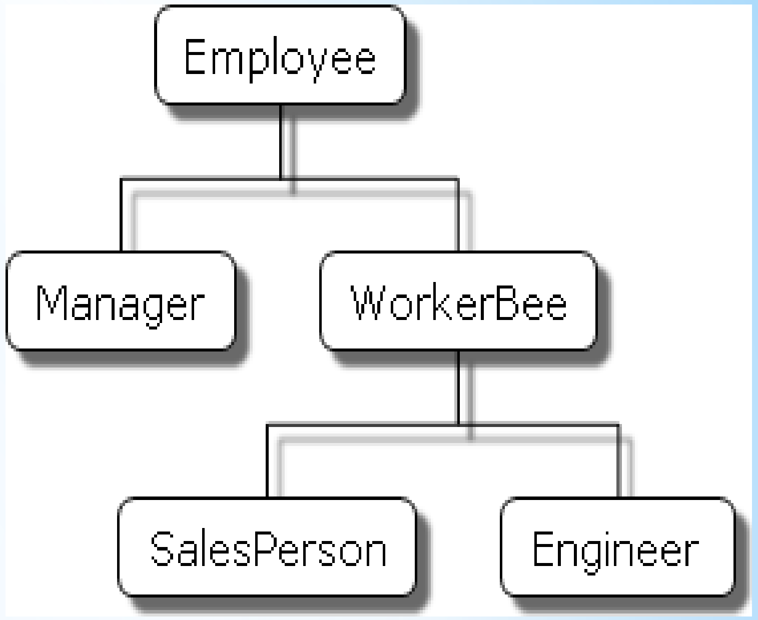
*Class Inheritance

* Inheritance is the ability of a program to create new entities from an existing entity - here a class. The class that is extended to create newer classes is called the **parent class/super class**. The newly created classes are called the **child/sub classes**.
* A class inherits from another class using the ‘extends’ keyword. Child classes inherit all properties and methods except constructors from the parent class.

* **Method Overriding** is a mechanism by which the child class redefines the superclass method.
* a child class to invoke its parent class data member. This is achieved by using the **super** keyword. The super keyword is used to refer to the immediate parent of a class.

*Class Inheritance and Method Overriding

- **Employee** has the properties name (whose value defaults to the empty string) and dept (whose value defaults to "general").
- **Manager** is based on Employee. It adds the reports property (whose value defaults to an empty array, intended to have an array of Employee objects as its value).
- **WorkerBee** is also based on Employee. It adds the projects property (whose value defaults to an empty array, intended to have an array of strings as its value).
- **SalesPerson** is based on WorkerBee. It adds the quota property (whose value defaults to 100). It also overrides the dept property with the value "sales", indicating that all salespersons are in the same department.
- **Engineer** is based on WorkerBee. It adds the machine property (whose value defaults to the empty string) and also overrides the dept property with the value "engineering".



*The employee example

* The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

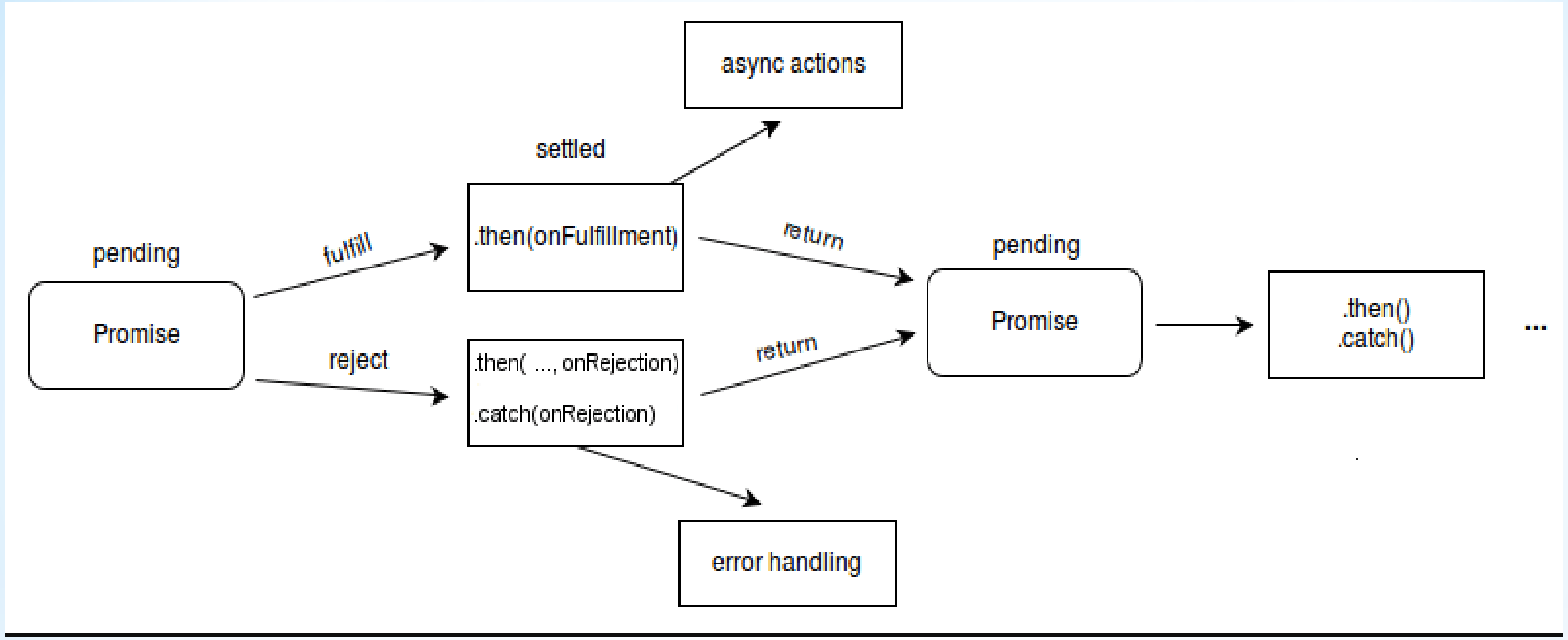
*AJAX

- **onreadystatechange** Defines a function to be called when the readyState property changes
- **readyState**
  - 0: request not initialized
  - 1: server connection established
  - 2: request received
  - 3: processing request
  - 4: request finished and response is ready
- **responseText** Returns the response data as a string
- **responseXML** Returns the response data as XML data
- **status**
  - 200: "OK"
  - 403: "Forbidden"
  - 404: "Not Found"
- **statusText** Returns the status-text (e.g. "OK" or "Not Found")

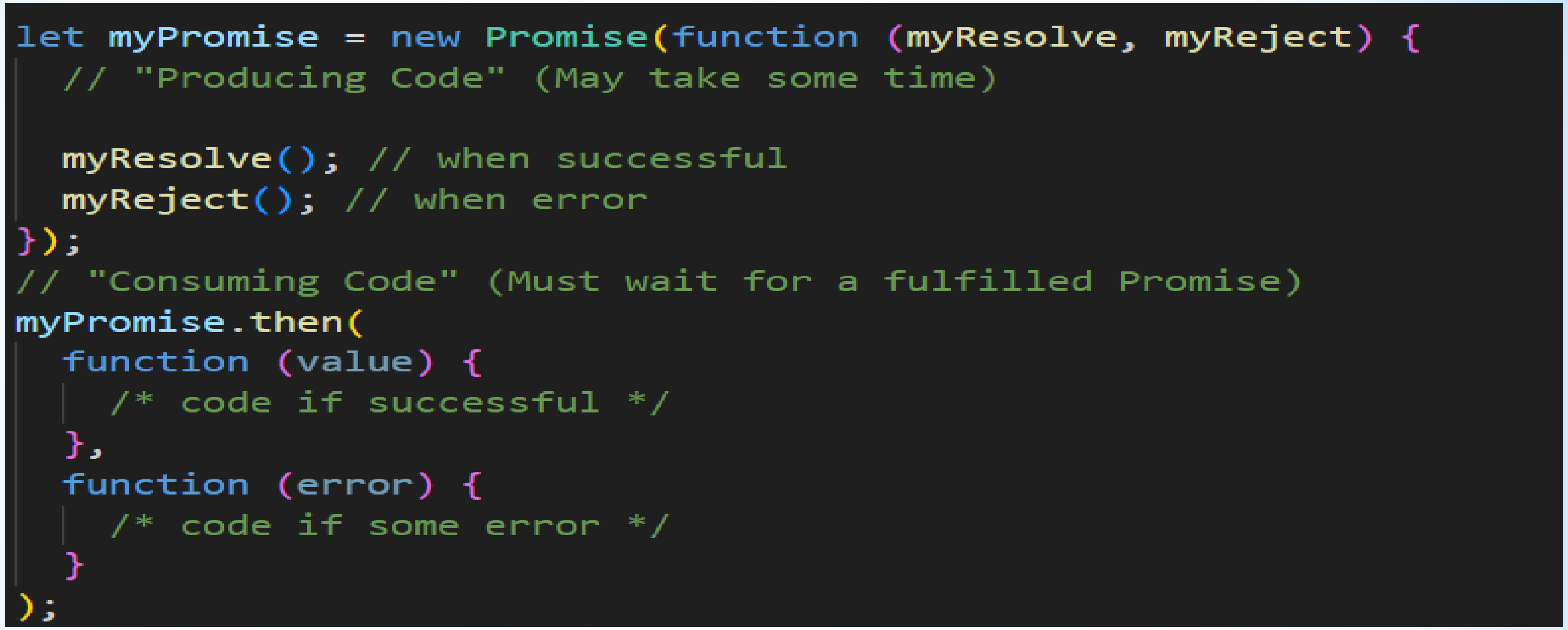
*XMLHttpRequest Object Properties

*Promise

- Promise for a value to be returned in a future
- Can have three mutually exclusive states
  - fulfilled
  - rejected
  - pending
- Can be referred as deferred or future
- Powerful for asynchronous / concurrent applications

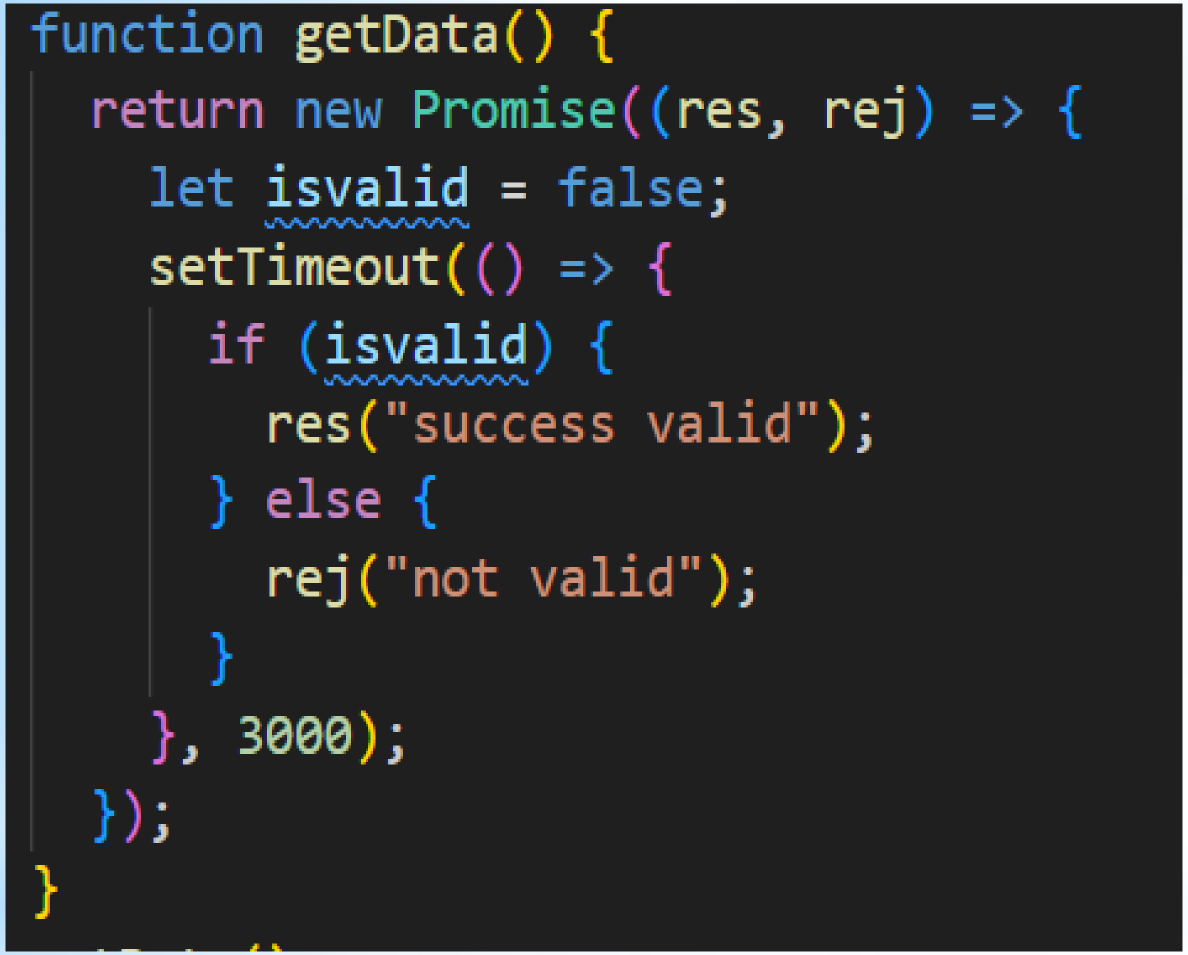

async actions
settled
.then(onFulfillment)
pending
Promise
fulfill
reject
return
return
.then(..., onRejection)
.catch(onRejection)
pending
Promise
.then()
.catch()
...
error handling

*Promise



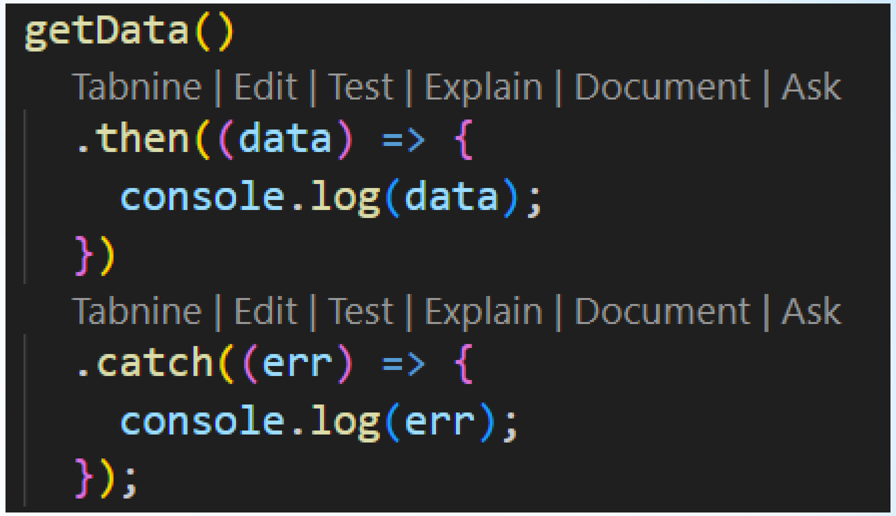
```
let myPromise = new Promise(function (myResolve, myReject) {
  // "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject(); // when error
});
// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function (value) {
    /* code if successful */
  },
  function (error) {
    /* code if some error */
  }
);
```

*Promise



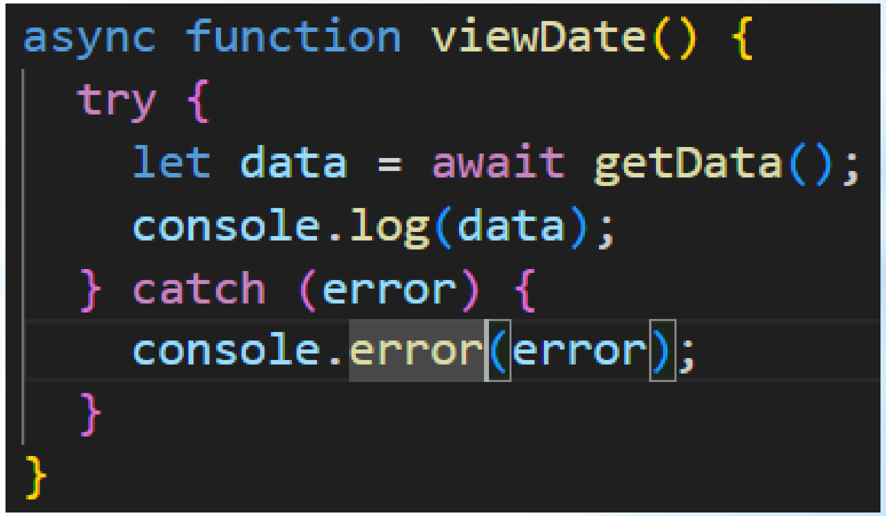
```
function getData() {
  return new Promise((res, rej) => {
    let isvalid = false;
    setTimeout(() => {
      if (isvalid) {
        res("success valid");
      } else {
        rej("not valid");
      }
    }, 3000);
  });
}
```

then



```
getData()
    Tabnine | Edit | Test | Explain | Document | Ask
  .then((data) => {
    console.log(data);
  })
    Tabnine | Edit | Test | Explain | Document | Ask
  .catch((err) => {
    console.log(err);
  });
```

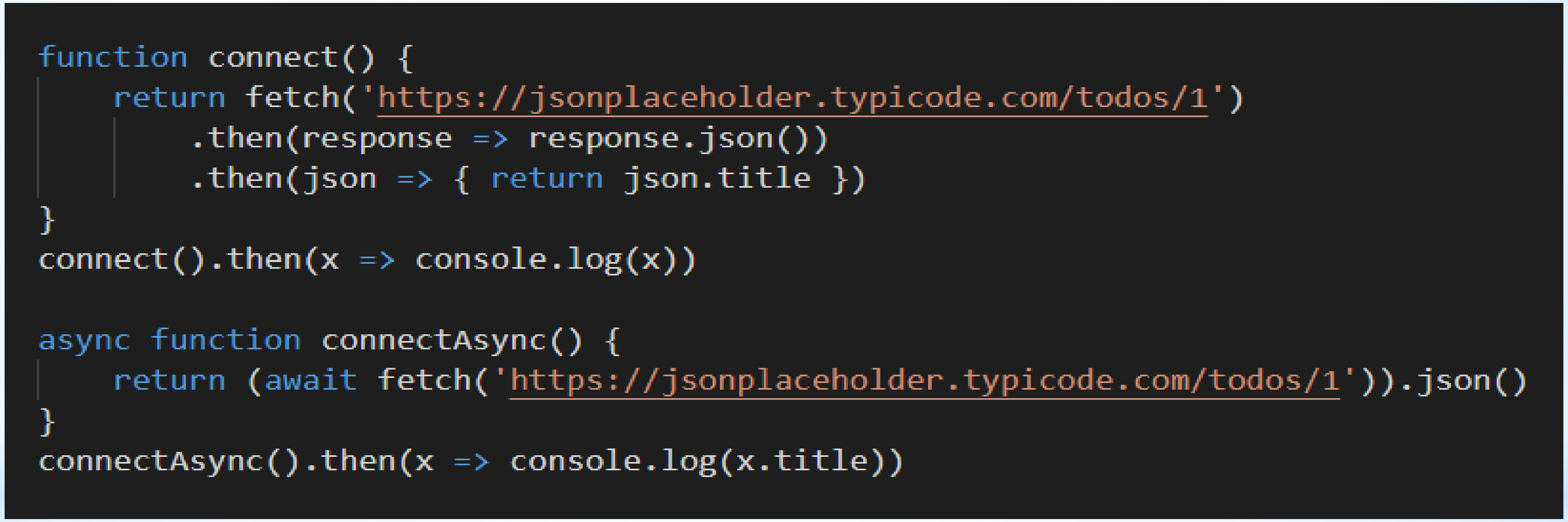
Async/ await



```
async function viewDate() {
  try {
    let data = await getData();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

*Promise

* As a quick overview, async and await keywords allow you to use them and try/catch blocks to make functions behave asynchronously.

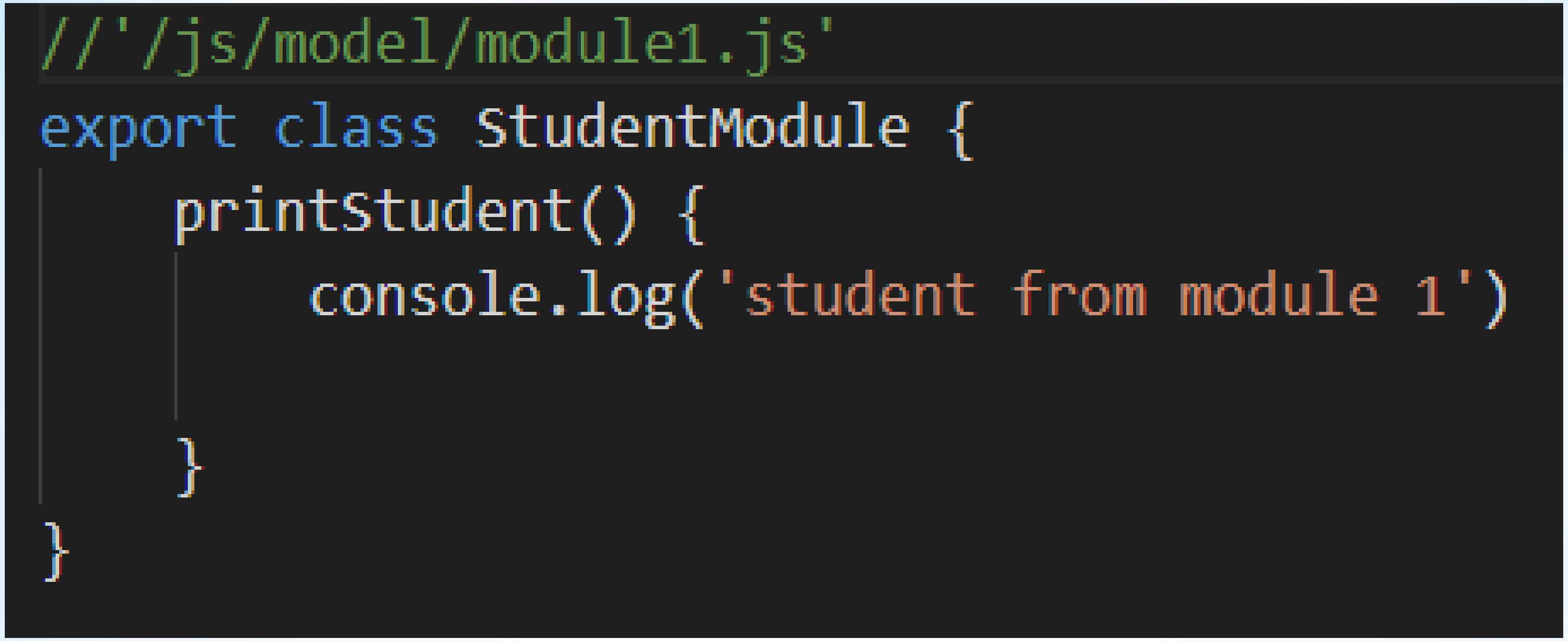


```
function connect() {
    return fetch('https://jsonplaceholder.typicode.com/todos/1')
        .then(response => response.json())
        .then(json => { return json.title })
}
connect().then(x => console.log(x))

async function connectAsync() {
    return (await fetch('https://jsonplaceholder.typicode.com/todos/1')).json()
}
connectAsync().then(x => console.log(x.title))
```

* Async Functions

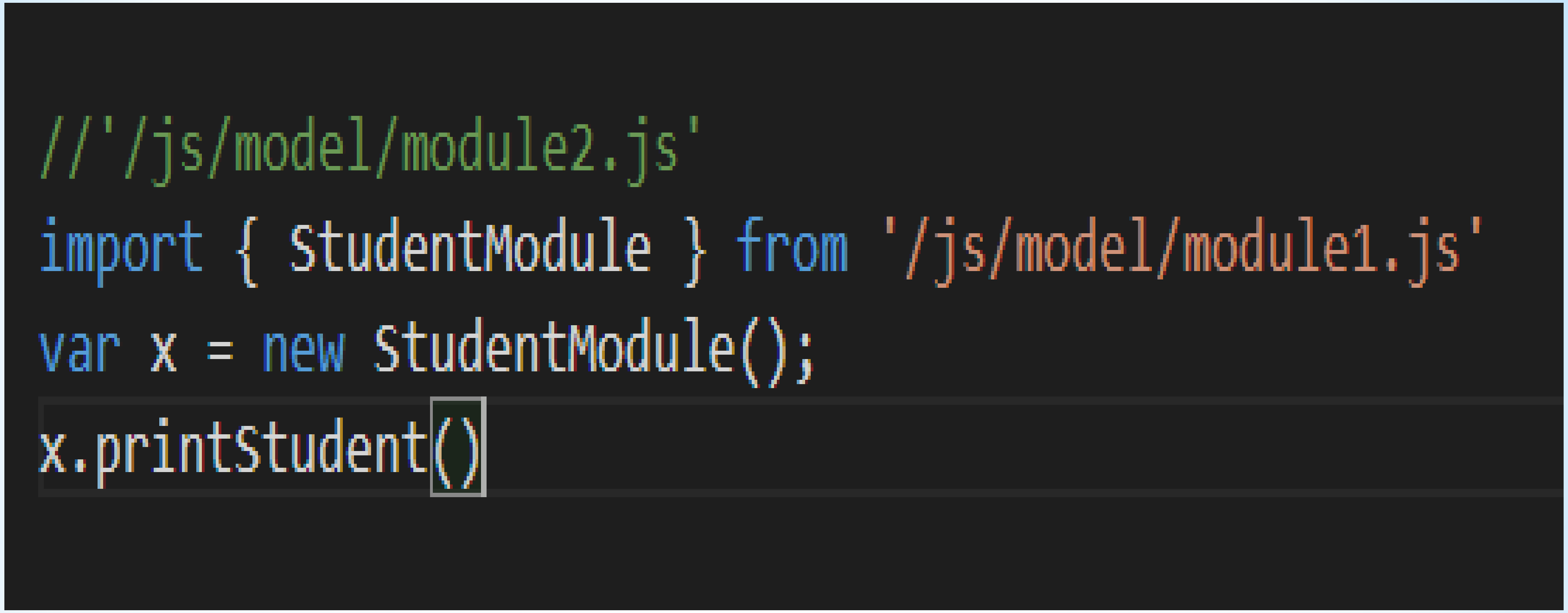
* It has therefore made sense in recent years to start thinking about providing mechanisms for splitting JavaScript programs up into separate modules that can be imported when needed.

*MODULES



```
//'/js/model/module1.js'
export class StudentModule {
    printStudent() {
        console.log('student from module 1')

    }
}
```

*export



```
//'/js/model/module2.js'
import { StudentModule } from '/js/model/module1.js'
var x = new StudentModule();
x.printStudent()
```

*import