

*Introduction to
Async/Sync
programming,
Promises*



JS



Express JS



sebinbenjamin

What we'll do learn today

- Async vs Sync
- Blocking vs non-blocking
- Javascript Promises

Processes and threads

A **process** is a collection of **code**, memory, data and other resources.

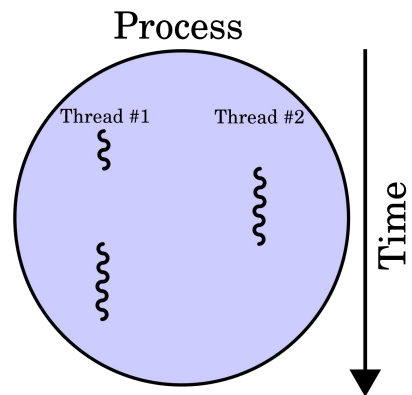
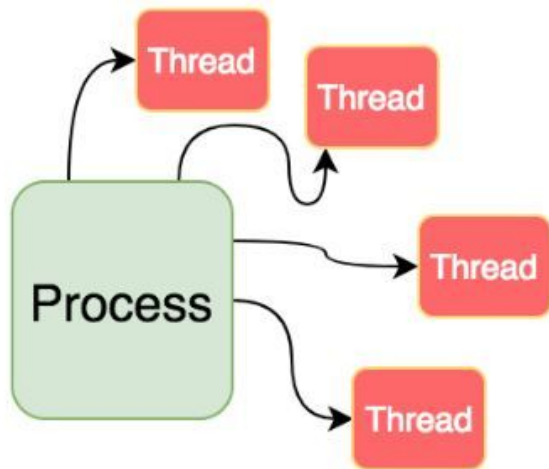
- An application consists of one or more processes.
- A process, in the simplest terms, is an executing program.

A **thread** is a sequence of **code that is executed** within the scope of the process.

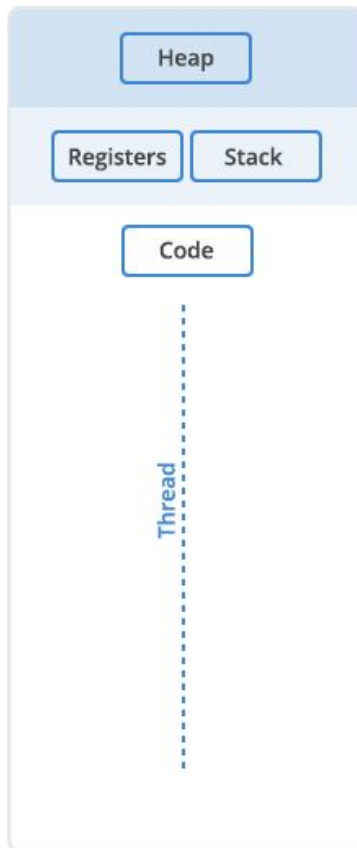
Threads share the process's resources. Each thread can only do a single task at once.

- Each task will be run sequentially; a task has to complete before the next one can be started.
- JavaScript is traditionally single-threaded.

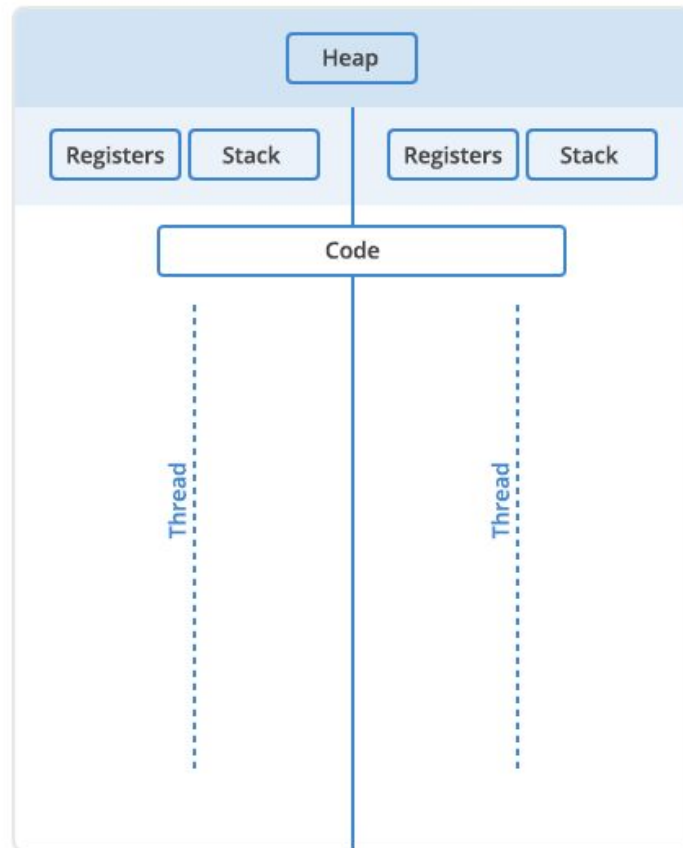
	Program	Process
<i>What</i>	Program is a set of instruction.	When a program is executed , it is known as process.
<i>Nature</i>	Passive	Active
<i>Required resources</i>	Program is stored on disk in some file and does NOT require any other resources .	Process holds resources such as CPU, memory address, disk, I/O etc.



Single Thread

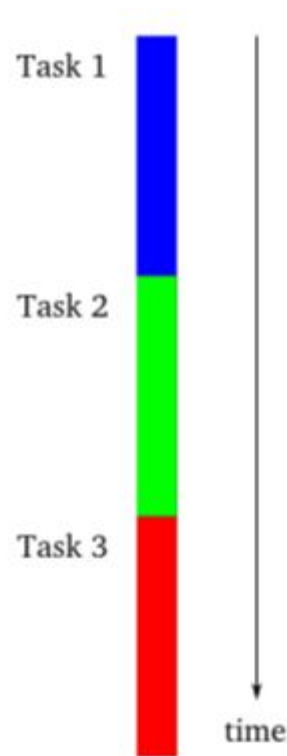


Multi Threaded



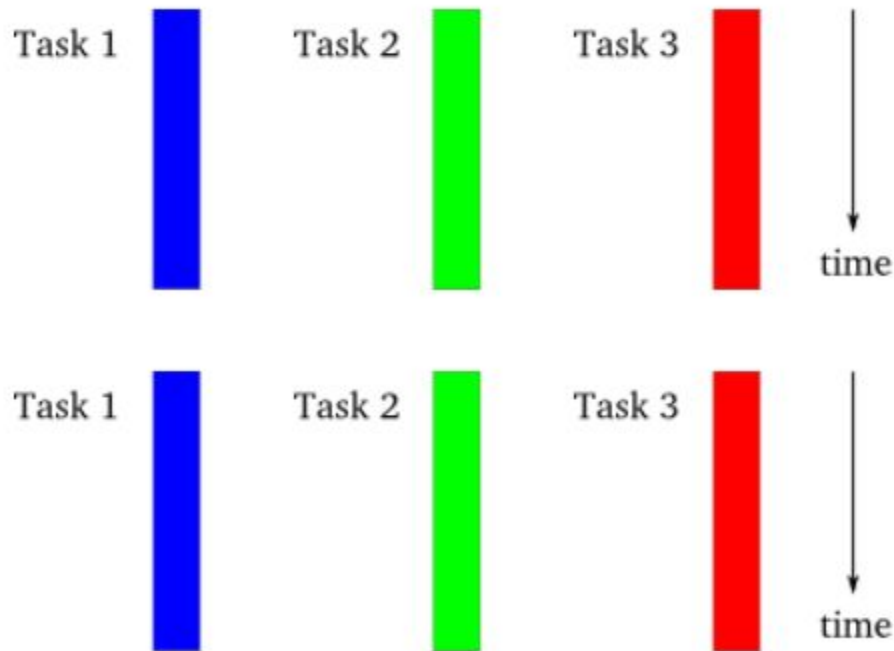
Synchronous model

- Simplest style of programming
- Each task is performed **one at a time**, with one finishing completely before another is started.
- We can assume that all earlier tasks have finished without errors, with all their output available for use.
- Single Threaded & Multi threaded synchronuous models.



Synchronous model - multi threaded

- Each task is performed in a separate thread of control.
- Threads are managed by the operating system and may, on a system with **multiple processors** or **multiple cores**, run truly concurrently, or may be interleaved together on a single processor.



Asynchronous model - single threaded

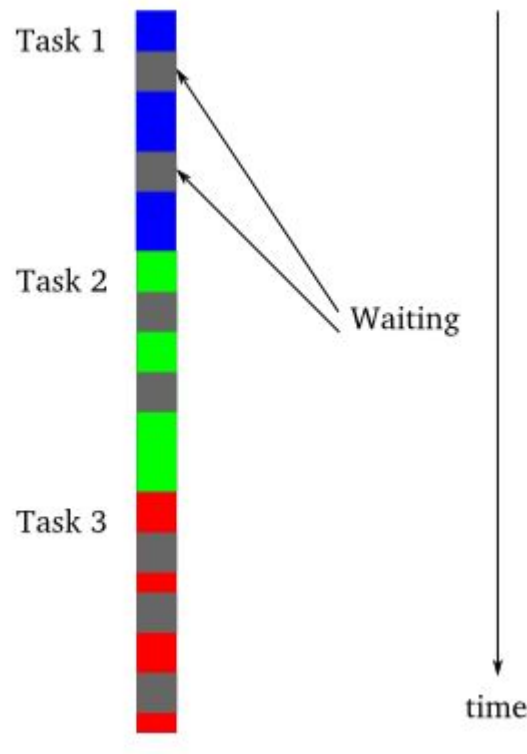
- *Tasks are interleaved* with one another, but in a single thread of control.
- Programmer always knows that when one task is executing, another task is not.
- Asynchronous system can outperform a synchronous one when certain tasks are forced to wait, or blocked.



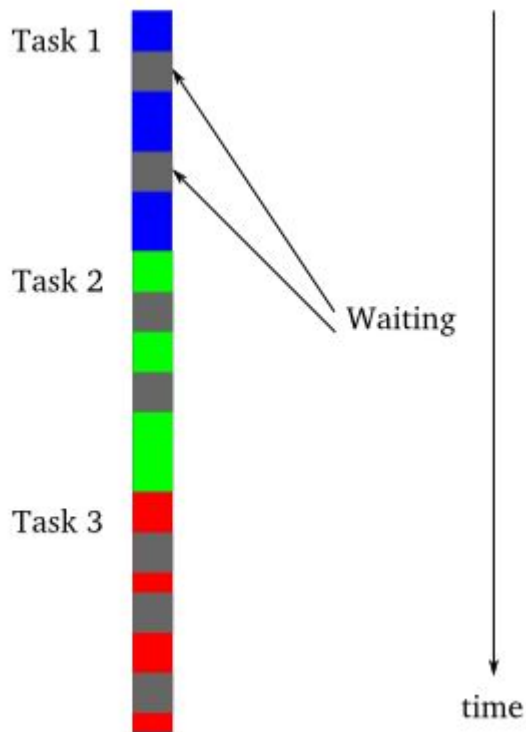
Blocking - Synchronous programs

Tasks are blocked when they could be waiting to perform I/O, to transfer data to or from an external device.

A synchronous program that is doing lots of I/O will spend much of its time blocked while a disk or network catches up.

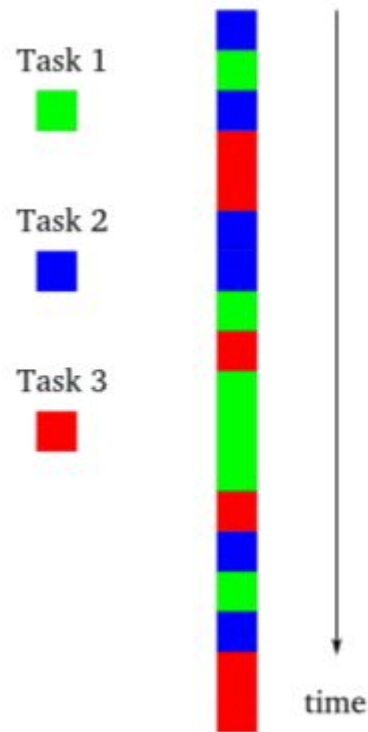


Non-blocking - Asynchronous programs



Synchronous

Idea behind the asynchronous model is that an asynchronous program, when faced with a task that would normally block in a synchronous program, will instead execute some other task that can *still make progress*.



Asynchronous

Blocking refers to operations that **block further execution** until that operation finishes

Node.js uses an
event-driven,
asynchronous
non-blocking I/O model

I/O (input/output)

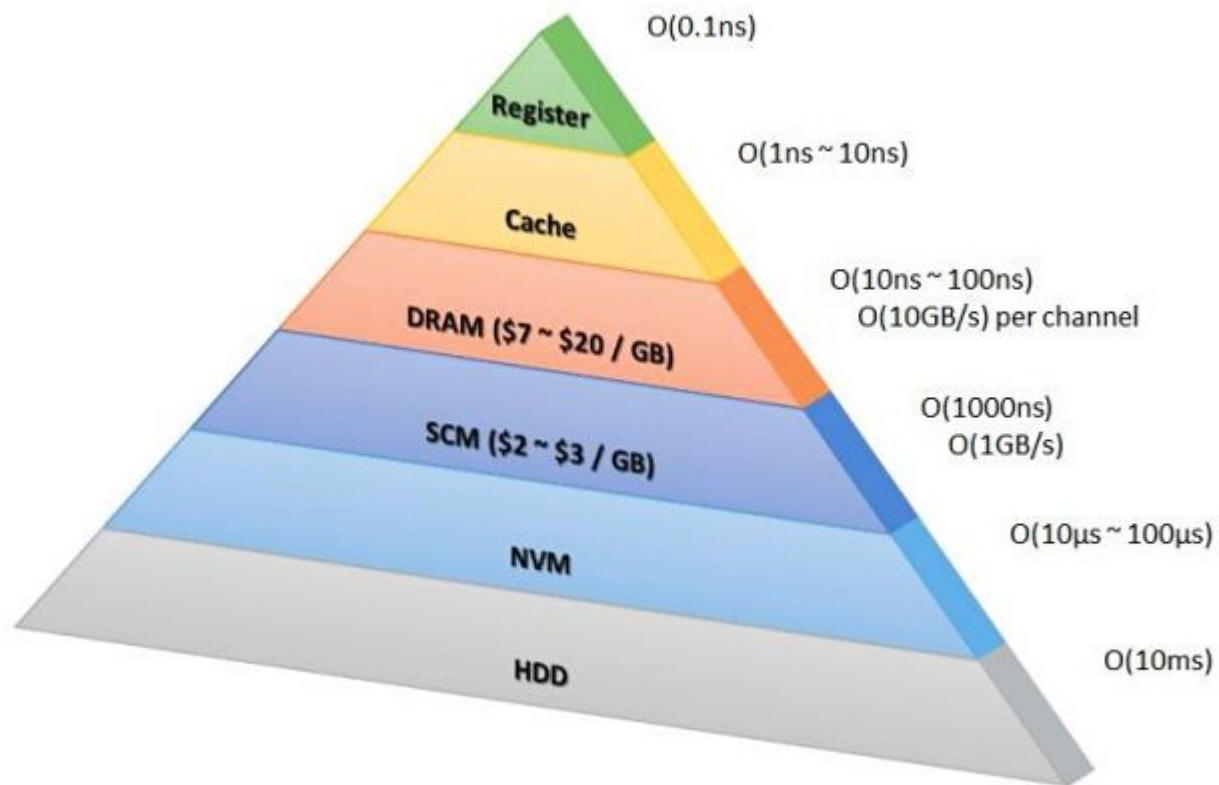
- Short for input/output, I/O refers primarily to the program's **interaction** with the system's **disk** and **network**.
- Examples of I/O operations include reading/writing data from/to a disk, making HTTP requests, and talking to databases.
 - They are very slow compared to accessing memory (RAM) or doing work on the CPU.



RAM Latency	83 nanoseconds
F-18 Hornet	1,1190 MPH



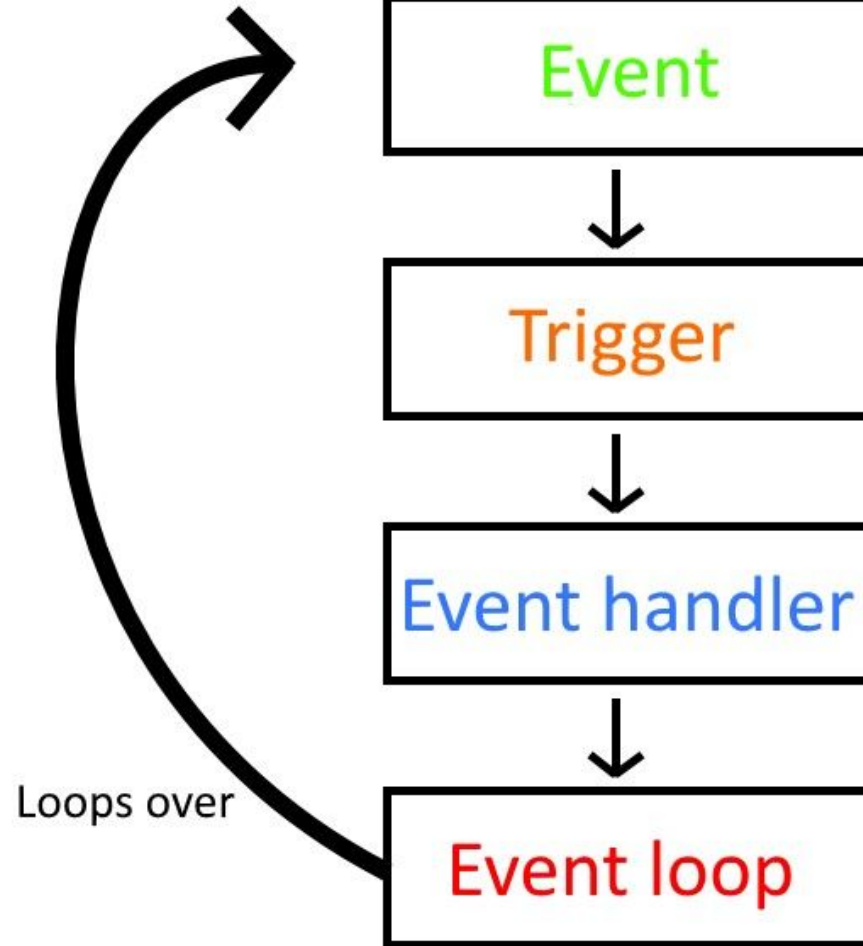
Disk Latency	13 milliseconds
Banana Slug	0.007 mph




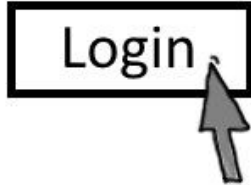
Events and event-driven programming

- Events are actions generated by the user or the system, like a click, a completed file download, or a hardware or software error.
- Event-driven programming is a programming paradigm in which the flow of the program is determined by events.
 - An event-driven program performs actions in response to events. When an event occurs it triggers a callback function.

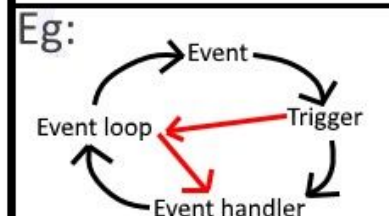
In an event driven program flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs or threads.



Eg: User interaction


Eg: Button trigger


Eg: Private Sub btnLogin_Click
Relevant login code goes here
And the Event handler executes it
End Sub



Promises

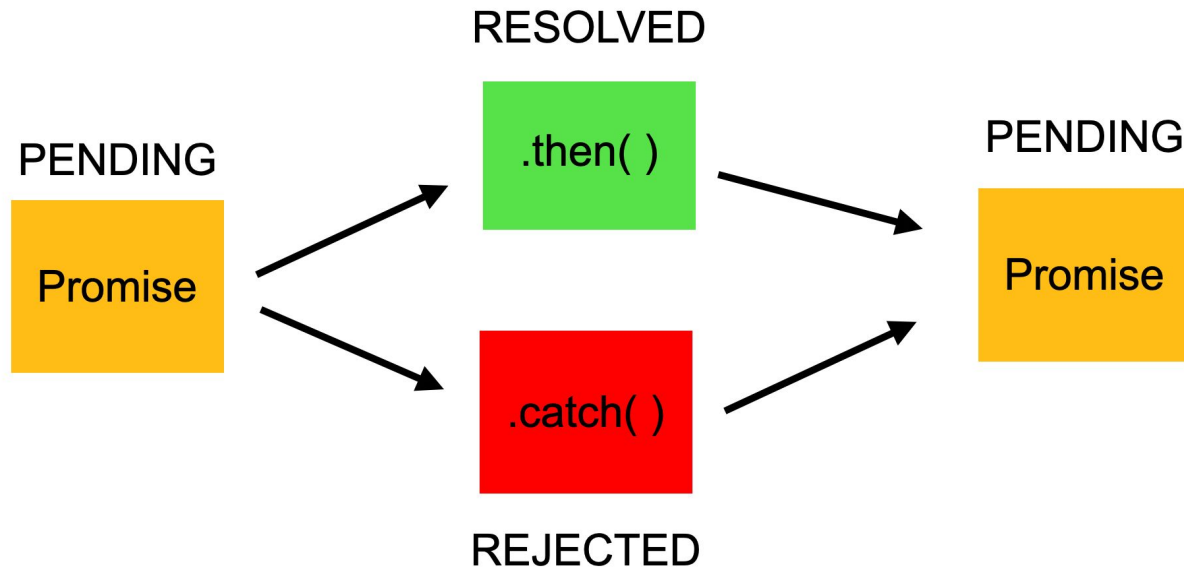
- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
 - A **proxy** for a value that will eventually become available.
- As promises in real life are either kept or broken, JavaScript Promises get either **resolved** or **rejected**.
- A promise can ***succeed or fail only once***. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.

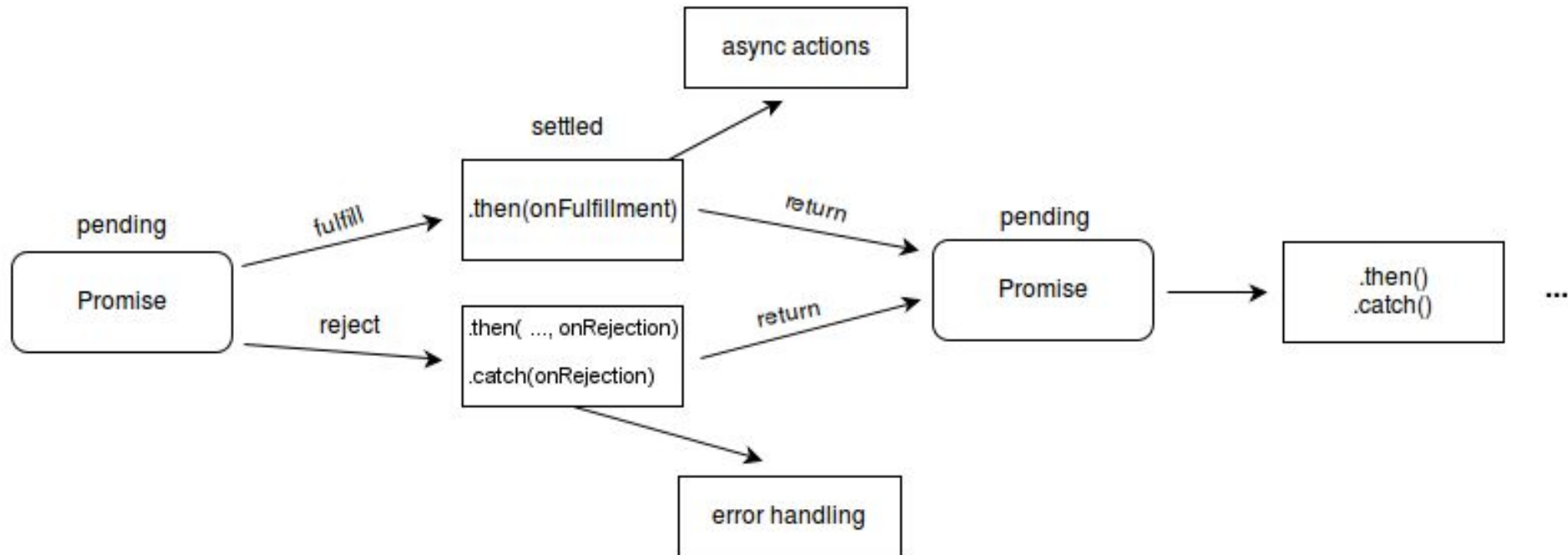
Promises

A promise is created using a constructor that takes a call back function with two arguments.

Promise takes in two callbacks:

- Resolve
 - ***then*** function will trigger only if the Promise gets resolved, ie code needed to perform the promised task is written.
- Reject
 - ***catch*** function will only trigger only if the Promise gets rejected. Called if the promise is rejected or if there was an error during the code execution





A promise can be:

- **pending** - Hasn't fulfilled or rejected yet
- **settled** - Has fulfilled or rejected
 - a. **fulfilled** - The action relating to the promise succeeded
 - b. **rejected** - The action relating to the promise failed

Promises Syntax

```
const myPromise = new Promise((resolve, reject) => {  
    if(everythingOK) resolve(successValue)  
    else reject(errorValue)  
});  
myPromise.then((successValue) => {  
    // do something  
}).catch((failureInfo) => {  
    // do something to handle errors  
});
```

Why promises ?

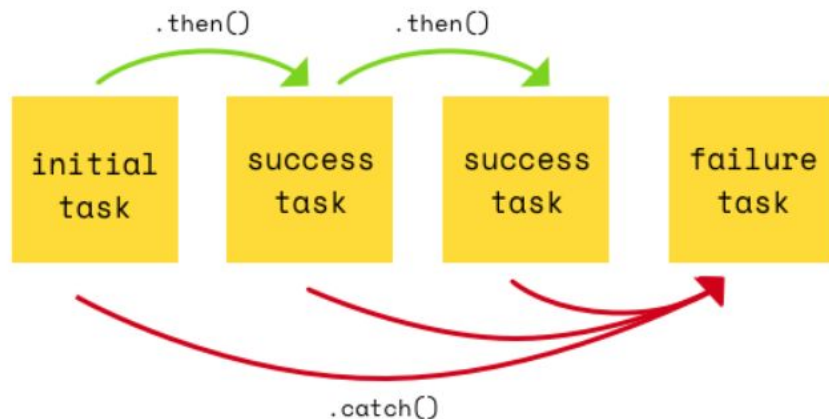
*JavaScript is **single threaded***, meaning that two bits of script cannot run at the same time; they have to run one after another.

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

Promises give us a way to *wait* for our *asynchronous code to complete*, capture some values from it, and pass those values on to other parts of our program.

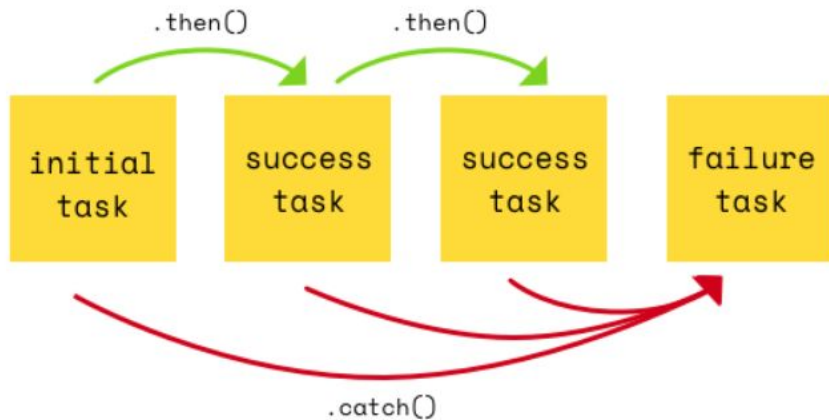
Chaining Promises

- A promise can be returned to another promise, creating a chain of promises.
- Promise chains provide precise control over how and where errors are handled.
- Promises allow you to mimic normal synchronous code's try/catch behavior.



Chaining Promises

- When anything in the chain of promises fails and raises an error or rejects the promise, the control goes to the nearest `catch()` statement down the chain





Thank you