require vs import
with require --> will load all code, loading is synchronous
with import --> can selectively load only the pieces we need saves memory, can be asynchronous (can perform little better than require)

The require('fs') call returns an object that contains various methods and properties related to file system operations, such as reading
files, writing files, watching for changes, and so on.

The file system is a component of an operating system that manages how data is stored and retrieved on storage devices, such as hard
drives, SSDs, or removable media. Here are some key aspects of a file system.

Organization of Data: The file system organizes data into files and directories (or folders). It provides a hierarchical structure,
allowing users to create, delete, and navigate through directories.

File Management: It handles operations such as creating, reading, updating, and deleting files. Each file has attributes, like its
name, size, and timestamps (e.g., creation and modification dates).

Storage Allocation: The file system manages how data is physically stored on the disk. It determines where files are saved and keeps
track of free and used space.

Access Control: File systems often include security features, such as permissions that control who can read, write, or execute a file.

Types of File Systems: Different operating systems use different types of file systems, such as NTFS (Windows), HFS+ (macOS), ext4
(Linux), and FAT32 (used across various systems).

Performance: The design of a file system can affect the performance of file operations, such as how quickly files can be accessed or how efficiently space is utilized.

Create Files

The File System module has methods for creating new files:

- fs.appendFile()
- fs.open()
- fs.writeFile()

filehandle.readFile(options)#
Added in: v10.0.0
options <Object> | <string>
encoding <string> | <null> Default: null
signal <AbortSignal> allows aborting an in-progress readFile
Returns: <Promise> Fulfills upon a successful read with the contents of the file. If no encoding is specified (using options.encoding),
the data is returned as a <Buffer> object. Otherwise, the data will be a string.

In Node.js, when reading or writing files, you can specify various character encodings. Here are some common encoding types you might use with or :

1. **UTF-8**: (string)
   - A widely used encoding that supports all characters in the Unicode standard.
2. **ASCII**:
   - An encoding that supports the first 128 Unicode characters (0-127). Useful for English text without special characters.
3. **UTF-16**:
   - A Unicode encoding that supports all characters but uses more space than UTF-8.
4. **ISO-8859-1**:
   - Supports Western European languages and uses a single byte for each character.
5. **Windows-1252**:
   - A superset of ISO-8859-1, commonly used in Windows environments.
6. **Base64**:
   - An encoding that converts binary data into a string format using 64 characters (A-Z, a-z, 0-9, +, /).
7. **Hex**:
   - Represents binary data as a hexadecimal string.
8. **UTF-32**:
   - A Unicode encoding that supports all characters and uses four bytes for each character.

fsPromises.appendFile(path, data[, options])

- data <string> | <Buffer>
- options <Object> | <string>
  - encoding <string> | <null> Default: 'utf8'
  - mode <integer> Default: 0o666

    - flag <string> See support of file system flags. Default: 'a'.
    - flush <boolean> If true, the underlying file descriptor is flushed prior to closing it. Default: false.
- Returns: <Promise> Fulfills with undefined upon success.

When you use fs.open in Node.js, it opens a file and returns a file descriptor. The file descriptor is a non-negative integer that uniquely identifies the opened file within the process.

Why It Might Print 3

The specific value you see (like 3) is the file descriptor assigned to that file. Here's why it might be 3:

1. Standard File Descriptors: When a Node.js process starts, it typically has three standard file descriptors open by default:
    - 0: Standard Input (stdin)
    - 1: Standard Output (stdout)
    - 2: Standard Error (stderr)
2. First Available Descriptor: The file descriptors are assigned in order, so when you open a new file, it uses the next available number. If the first three are in use, the next file descriptor will be 3, and so on for subsequent file openings.

Reasons for Clearing Content

1. Overwrite Behavior: The 'w+' mode is designed to open a file for reading and writing but also truncates it to zero length. This means that if the file already exists, its content is removed, allowing you to start fresh with new data.
2. Use Case: This mode is useful when you want to create a new file or reset an existing file. It's commonly used in scenarios where the previous data is no longer needed, and you want to replace it with new content.
3. Safety and Clarity: By truncating the file, it ensures that any old data that might confuse or interfere with your new writes is removed.

fs.rename(oldPath, newPath, callback)

oldPath <string> | <Buffer> | <URL>

- newPath <string> | <Buffer> | <URL>
- callback <Function>
    - err <Error>

Asynchronously rename file at oldPath to the pathname provided as newPath. In the case that newPath already exists, it will be overwritten. If there is a directory at newPath, an error will be raised instead. No arguments other than a possible exception are given to the completion callback.