```
java.lang.Object
  └── java.io.File   (represents pathnames)

java.io.InputStream  (abstract)
  ├── FileInputStream
  ├── BufferedInputStream

java.io.OutputStream (abstract)
  ├── FileOutputStream
  ├── BufferedOutputStream

java.io.Reader (abstract)
  ├── FileReader
  ├── BufferedReader

java.io.Writer (abstract)
  ├── FileWriter
  ├── BufferedWriter
```

I want to know about the common names here, Like what is File, then OutputStream, InputStream, Reader, Writer..? and if we go within any one of them what is the fundamental task they are performing that makes there name to be what it is. And also some of the most frequently used methods with a simple standard example of it demonstrating a use of it

Absolutely! Let's break this down carefully and clearly so you can **understand the naming and the fundamental roles** of each of these Java I/O classes. I'll also give some simple examples.

---

1. java.io.File

What it is:

- Represents a **pathname**, i.e., a file or directory in the file system.
- **It does NOT read or write content**. Its job is to let Java interact with files in terms of existence, path, creation, deletion, etc.

Common methods:

java Copy code

```java
File file = new File("example.txt");
file.exists();      // checks if file exists
file.createNewFile(); // creates the file if it doesn't exist
file.delete();      // deletes the file
file.isDirectory(); // checks if it's a directory
file.getName();     // returns the name of the file
file.getPath();     // returns the path
```

Example:

java Copy code

```java
import java.io.File;
import java.io.IOException;

public class FileExample {
    public static void main(String[] args) throws IOException {
        File file = new File("example.txt");
        if (!file.exists()) {
            file.createNewFile();  // create file if it doesn't exist
            System.out.println("File created: " + file.getName());
        } else {
            System.out.println("File already exists.");
        }
    }
}
```

Summary: File = **represents a file or folder**, mainly for metadata, paths, and file operations.

---

2. InputStream and OutputStream

What they are:

- `InputStream` → abstract class for reading **bytes** from a source (file, network, memory).
- `OutputStream` → abstract class for writing **bytes** to a destination.

Think: InputStream = "source of data", OutputStream = "sink for data".

Common subclasses:

- `FileInputStream` → reads bytes from a file
- `FileOutputStream` → writes bytes to a file
- `BufferedInputStream / BufferedOutputStream` → add **buffering** for efficiency

Most common methods:

java Copy code

```java
// InputStream
int read() throws IOException;       // reads one byte
int read(byte[] b) throws IOException; // reads into array
void close() throws IOException;      // close the stream

// OutputStream
void write(int b) throws IOException;     // writes one byte
void write(byte[] b) throws IOException;  // writes array of bytes
void flush() throws IOException;          // flush buffered bytes
void close() throws IOException;          // close the stream
```

Example: Copy a file using byte streams

java Copy code

```java
import java.io.*;
```

```java
public class CopyFileExample {
    public static void main(String[] args) throws IOException {
        try (FileInputStream fis = new FileInputStream("source.txt");
             FileOutputStream fos = new FileOutputStream("dest.txt")) {

            byte[] buffer = new byte[1024];
            int length;
            while ((length = fis.read(buffer)) > 0) {
                fos.write(buffer, 0, length);
            }
        }
        System.out.println("File copied successfully.");
    }
}
```

Summary: InputStream/OutputStream = **deal with raw bytes**.

---

3. Reader and Writer

What they are:

- `Reader` → abstract class for reading **characters** (text)
- `Writer` → abstract class for writing **characters** (text)

Think: Reader/Writer = **text-oriented I/O**, whereas InputStream/OutputStream = **binary-oriented I/O**.

Common subclasses:

- `FileReader` → reads characters from a file
- `FileWriter` → writes characters to a file
- `BufferedReader` → reads text **efficiently**, also provides `readLine()`
- `BufferedWriter` → writes text efficiently

Most common methods:

java Copy code

```
// Reader
int read() throws IOException;        // reads one character
int read(char[] cbuf) throws IOException; // reads into char array
void close() throws IOException;      // close the reader

// Writer
void write(int c) throws IOException;       // write one char
void write(char[] cbuf) throws IOException;// write char array
void write(String str) throws IOException; // write string
void flush() throws IOException;           // flush
void close() throws IOException;
```

Example: Read a file line by line using BufferedReader

java Copy code

```
import java.io.*;

public class ReadTextExample {
    public static void main(String[] args) throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}
```

Summary: Reader/Writer = **deal with characters and text**, more convenient for strings, not raw bytes.

Quick Naming Logic

Class/Interface	What it represents / does	Use-case summary
`File`	Path, metadata, existence of file/folder	File operations, not content
`InputStream`	Abstract byte input source	Read raw bytes from file/network/memory
`OutputStream`	Abstract byte output destination	Write raw bytes
`Reader`	Abstract character input source	Read text/characters
`Writer`	Abstract character output destination	Write text/characters

Key distinction:

- Streams = bytes
- Readers/Writers = characters
- Buffered versions = more efficient (store in memory buffer first)

If you want, I can make a **small visual diagram showing the whole hierarchy with what each class "does"**, which makes it extremely easy to memorize.

Do you want me to do that?