GUVI | HCL
Skill Up. Level Up

# File I/O & Concurrency

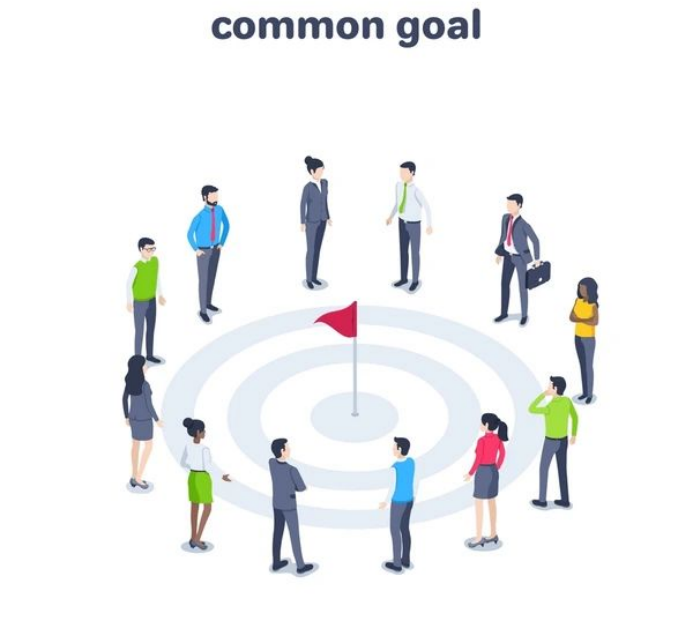
Programming Foundation

Presented by
**Nikhil Nair**

Website
**www.guvi.com**

# Objectives

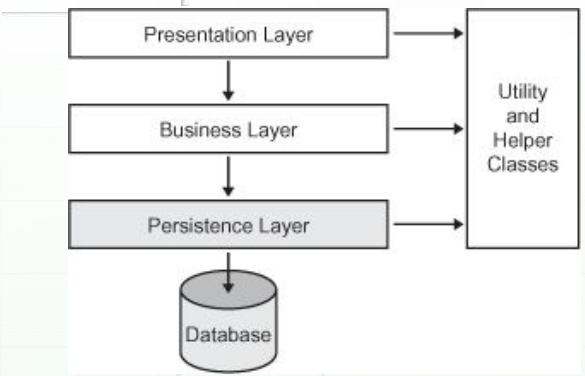
- What you will learn today
    - Persistence in the context of files
    - Introduction to Concurrency
    - Threads, Processes, Runnable
    - A brief on Thread Pool



# Persistence & Files

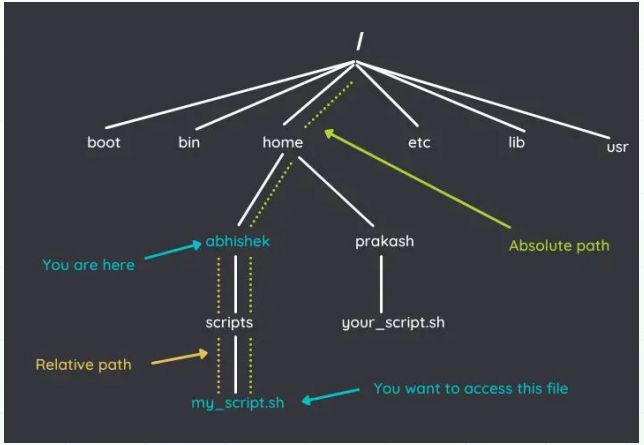
# Persistent data

- Persistent data are values kept across program restarts
- Contrast transient memory with durable storage used after exit
- Common forms of persistence you'll encounter in practice
    - Text files for small settings, logs
    - Databases for structured, concurrent, query heavy data
- Benefits: recover state after crashes and customize behavior easily
- We will work with simple text files as our first form of persistence
    - Easy to inspect and share, perfect for tiny settings and lists
    - We will use Java Path and Files to locate and store data



# Paths and directories hygiene

- When using files, we need reliable locations on disk before any read or write
- In Java, a Path is as an object representing filesystem locations
- Relative paths resolve against the current working directory
    - Example: data/todo.txt inside the project folder
    - Example: ../logs/app.txt one folder above current
- Absolute paths start at root, independent of process location
    - Example: C:\Users\Asha\data\todo.txt on Windows
    - Example: /home/asha/data/todo.txt on Linux or macOS
- Create parent directories before writing any new file
    - Use Files.exists to check if a File exists
    - Use Files.createDirectories to create directory given a path



# Text file read and write

- Goal: read a small text file and write it back
- Files.readString for reading entire textual file content
    - Returns full file as String using default UTF-8
- Files.writeString for writing textual content to files
    - Overwrites by default, suitable for regenerating small files
- Why these APIs suit small text content today
    - They read or write whole files in one operation
    - Keep examples simple, predictable, and easy to reason about
- Ensure folder exists and handle first run gracefully

```
public class App {
  public static void main(String[] a) throws
java.io.IOException {
    java.nio.file.Path f = java.nio.file.Path.of("data", "todo.txt");
// /a/b/c/
// data/todo.txt -> /a/b/c/data/todo.txt
    java.nio.file.Files.createDirectories(f.getParent());
    String content = java.nio.file.Files.exists(f)
        ? java.nio.file.Files.readString(f) : "";
    String updated = content + "Buy milk\n";
    java.nio.file.Files.writeString(f, updated);
  }
}
```

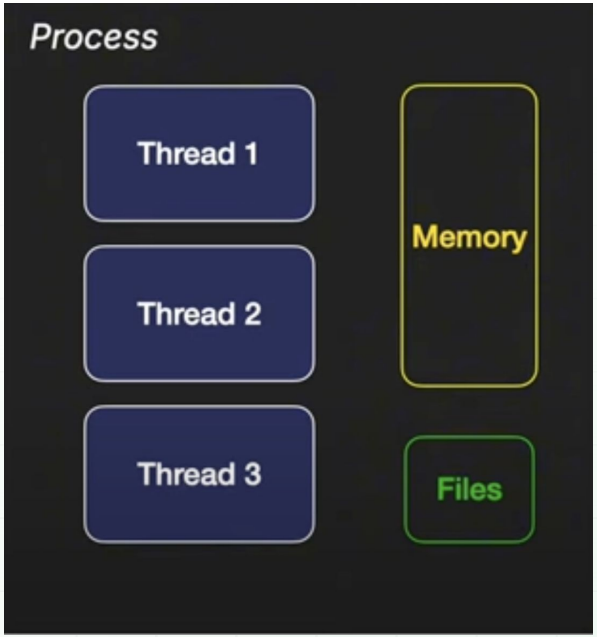
# Activity: persist a todo list

- Clean and persist a todo.txt into a data folder
    - Input may contain duplicates, blanks, and leading or trailing spaces
    - The data folder may not exist yet; create it before writing
- Enforce output invariants that make the file dependable
    - Trim whitespace, drop empty lines, preserve first seen order
    - Remove exact duplicates after trimming, do not append
- Use the APIs introduced so far
    - Read with readString if present, write back with writeString
    - Ensure folders exist using createDirectories before any write
- Make the operation idempotent across runs
    - Running the program twice produces identical file content

# Concurrency

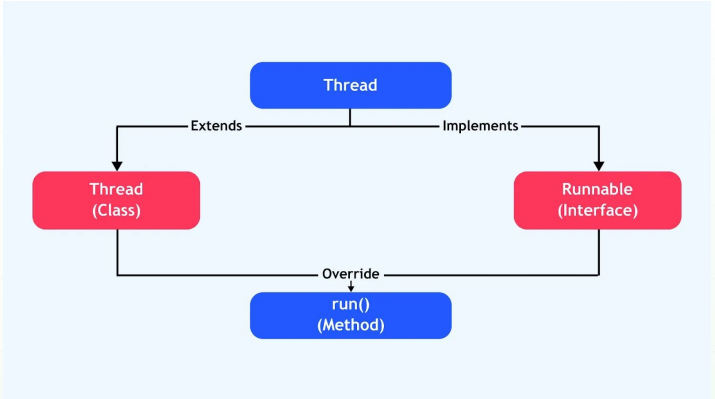
# Concurrency foundations

- Concurrency refers to the ability of a program to execute multiple tasks simultaneously, improving performance and resource utilization
- Processes and threads are two fundamental components of concurrency
    - A process is a running program with its own memory
    - A thread is a path of execution inside the same process
- The main thread drives program startup and primary user tasks
- Background threads handle independent work without blocking the main thread
- Threads within a single process share the same heap memory and CPU resources
- Context switches let the CPU alternate between threads quickly



# Thread vs Runnable

- Java provides two primary ways to define tasks for concurrent execution
    - Extending the Thread class or implementing the Runnable
- Extending Thread ties the task directly to a specific thread instance
- Implementing Runnable decouples the task logic from the thread that executes it
- Use Runnable when you want to reuse the same task across multiple threads
- Extend Thread only if you need to customize thread behavior beyond task execution



# Example: two workers using Runnable

- Start two independent tasks while the main thread continues
- Use Runnable to highlight task reuse and portability
- Expect interleaved output because line ordering is not guaranteed

```
public class App {
  public static void main(String[] args) {
    Runnable a = () -> System.out.println("task A");
    Runnable b = () -> System.out.println("task B");
    new Thread(a, "worker-A").start();
    new Thread(b, "worker-B").start();
    System.out.println("main continues");
  }
}
```

# Activity: start two workers and compare forms

- Create two workers that print their task names and then exit
- Implement one as a Thread subclass and the other as a Runnable

# Thread lifecycle and ordering

# Thread lifecycle

- Use a four state mental model for threads
    - NEW, RUNNABLE, RUNNING, TERMINATED in this simplified view
- start moves a thread from NEW to RUNNABLE to RUNNING
- run is the work a thread performs when scheduled
- join makes the current thread wait until another thread finishes
- sleep pauses the current thread without releasing shared memory
- Note: Java defines more states officially, we keep it simple today

# Example: visualize ordering with start join sleep

- Use join to guarantee a thread finishes before the next step
- Use sleep only to delay; it does not control completion order
- Contrast start order, sleep timing, and join-based ordering

```
public static void main(String[] args) throws InterruptedException {
  Thread t1 = new Thread(() -> {
    try { Thread.sleep(100); }
    catch (InterruptedException e) {}
    System.out.println("A");
  }, "A");

  Thread t2 = new Thread(() -> {
    try {
      Thread.sleep(50);
    } catch (InterruptedException e) {}
    System.out.println("B");
  }, "B");

  t1.start();
  t2.start();

  t2.join();
  System.out.println("after B");
}
```

# Activity: predict prints with partial joins

- Start three workers named alpha, beta, gamma with short sleeps
- Join beta and gamma, then print from the main thread
- What is the most likely print order?
- What is the smallest change to guarantee main prints last?

# Thread ergonomics and interruption

# Naming threads and simple logs

- Name threads to make logs readable and debugging faster
    - Examples: worker-A, worker-B, flusher, report
- Set names at creation or with setName before calling start
- Print timestamps with names to trace execution clearly
- Keep one responsibility per thread for clear, consistent messages

```
Runnable task = () ->
System.out.println(Thread.currentThread().getName() + " started");

Thread t1 = new Thread(task, "worker-A");
Thread t2 = new Thread(task);
t2.setName("worker-B");

t1.start();
t2.start();
```

# Daemon and user threads

- A daemon thread does not keep the process alive at shutdown.
    - Example: A background thread that auto-saves your notes every few seconds while you're typing
- A user thread keeps the process alive until its work completes
    - Example: A thread that uploads a photo you just selected; your app only waits until the upload is done.
- Set daemon status to true/false before start using setDaemon method
- Check the role at runtime with isDaemon for clear logging
- Use daemons for best effort housekeeping; use users for must finish tasks

```
Runnable show = () ->
System.out.println(Thread.currentThread().getName() + "
isDaemon=" + Thread.currentThread().isDaemon());

Thread flusher = new Thread(show, "flusher");
flusher.setDaemon(true);   // must be called before start

Thread report = new Thread(show, "report");
report.setDaemon(false);

flusher.start();
report.start();
```

# Interruption basics with sleep

- interrupt() signals a thread to stop cooperatively and promptly
- Blocking methods like sleep() throw InterruptedException to deliver the signal
- On interrupt, log the event and exit the run() method cleanly
- Do not swallow interrupts; do not continue silently after catching
- Use case: ideal for stopping background tasks when the user exits the app

```
Thread flusher = new Thread(() -> {
  try { Thread.sleep(10_000); }
  catch (InterruptedException e) {
    System.out.println(Thread.currentThread().getName() + " stopping");
  }
}, "flusher");

flusher.start();
Thread.sleep(200);
flusher.interrupt();
```

# Activity: name, daemon, interrupt policy design

- Create class LogFlusher that implements Runnable and prints timestamped messages
- Start it in a thread named flusher; decide daemon or user and explain
- Create a short report task thread; sleep briefly, then print completion
- Ensure the program exits cleanly; stop flusher cooperatively using interrupt
- Include readable logging: ISO timestamps and thread names in every line

# Beyond Threads

# Tasks are work, threads are workers

- A task describes the work to do, not who does it
- A thread is the worker that runs a task to completion
- Keep tasks reusable so any thread can run them
- This separation makes testing and scheduling much simpler
    - Examples of tasks: resize an image, write a log line
    - Examples of workers: worker 1, worker 2 in the same process

# Why starting a thread for everything breaks down

- Each thread consumes memory and CPU scheduling time
- Too many threads compete and slow each other down
- You end up hand rolling queues, ordering, and shutdown
- Bursts create spikes: a thousand tasks can mean a thousand threads

# How pools and futures help, then where Java takes you

- A pool is a small team of threads reused for many tasks
- A queue holds incoming tasks until a worker is free
- A future is a simple handle to get a result later
- In Java, the Executor family bundles these ideas together
- Today is concept only; actual APIs come in a later lecture

**That’s for today!**
**Any questions?**