🧠🔥 ASYNC PROGRAMMING — MASTER REVISION SHEET

(From ZERO → Production Thinking)

> **Purpose:** “When I’m stuck, confused, or designing async code — what should I think, what should I use, and why?”

0️⃣ THE ONE-LINE TRUTH (Never Forget)

> **Async ka matlab hai: Waiting time waste mat karo — us waqt dusra kaam karao.**

Async ≠ faster CPU Async = **waiting overlap**

1️⃣ CORE MENTAL MODELS (THE WHY)

🧠 Sync vs Async

Sync (one by one)

```
Task A → wait → done
Task B → wait → done
Task C → wait → done
```

⏱ Total time = A + B + C

Async (overlap waiting)

```
Start A → waiting
Start B → waiting
Start C → waiting
Resume A → Resume B → Resume C
```

⏱ Total time ≈ max(A, B, C)

🔒 **Async speed comes ONLY from overlapping wait time**

🧠 Blocking vs Non-Blocking

❌ Blocking

- Poora program ruk jaata hai
- Event loop freeze

Examples:

```
time.sleep(2)      # ❌
requests.get()     # ❌ inside async
```

✅ Non-Blocking

- Sirf current task rukta hai
- Event loop free

Examples:

```
await asyncio.sleep(2)    # ✅
aiohttp request           # ✅
```

🚨 Rule

> Async code + blocking call = system destroyer

🧠 Event Loop (Traffic Police Model)

Event loop:

- Ek **manager / scheduler**

- Decide karta hai:

  - kaunsa task chale
  - kaunsa wait kare
  - kaunsa resume ho

🔑 Golden Rule

> Event loop ko control **sirf `await` pe milta hai**

Agar `await` nahi:

- ❌ no switching
- ❌ starvation
- ❌ cancellation fail

2️⃣ ASYNC BUILDING BLOCKS (THE WHAT)

◇ `async def` — Coroutine Definition

```python
async def fetch(url):
    ...
```

- Run nahi hota
- Sirf **coroutine object** banata hai

❌ Ye galat soch:

> “Function call se run ho jaata hai”

☑ Sahi soch:

> “Coroutine ban gaya, run baad mein hoga”

◇ Coroutine

> **Coroutine = pause / resume hone wala function**

- Execution tab hoti hai jab:
  - `await` mile
  - event loop mile

◇ `await` — Pause + Handover

```python
await fetch(url)
```

- Coroutine ko run karta hai
- Pause point deta hai

- Event loop ko control deta hai

❌ `await` outside `async def` → error ❌ `await` CPU-heavy code → useless

◇ `asyncio.run()` — Engine Start

```
asyncio.run(main())
```

- Event loop start karta hai
- Async program ka **entry gate**

🔒 **Rule**

> Async program hamesha `asyncio.run()` se start hota hai

3️⃣ CONCURRENCY KA REAL ENGINE

⚡ `asyncio.gather()` — REAL ASYNC POWER

❌ Galat (serial async)

```
for url in urls:
    await fetch(url)
```

Why slow?

- Event loop ke paas **ek hi task** hota hai

☑ Sahi (concurrent async)

```
tasks = [fetch(url) for url in urls]
results = await asyncio.gather(*tasks)
```

Why fast?

- Event loop ke paas **multiple tasks**
- Waiting overlap hoti hai

🔐 Golden Rule

> Loop + await = SERIAL Tasks + gather = CONCURRENT

🛑 CRITICAL `gather()` WARNING (Production Level)

❌ Default Behavior

```
await asyncio.gather(*tasks)
```

- Ek task fail → **sab fail**
- Program crash
- Baaki tasks cancel

☑ SAFE VERSION (Always for Scraping / APIs)

```
results = await asyncio.gather(
    *tasks,
    return_exceptions=True
)
```

Now:

- Ek fail ho sakta hai
- Baaki continue
- Errors list mein milte hain

🔐 Rule

> Scraping / APIs → always `return_exceptions=True`

4️⃣ PAUSING THE RIGHT WAY

💤 `asyncio.sleep()`

```
await asyncio.sleep(2)
```

- Non-blocking pause
- Event loop dusre tasks chala sakta hai
- CPU idle

❌ Kabhi mat use karo:

```
time.sleep(2)
```

5️⃣ TASK LIFECYCLE (1.2.3 — VERY IMPORTANT)

Har async task in states se guzarta hai:

```
CREATED → RUNNING → WAITING → DONE
                     ↓
                 CANCELLED
```

Why important?

- Debugging
- Cancellation
- Memory leaks samajhne ke liye

🛑 Cancellation (`task.cancel()`)

```
task.cancel()
```

Truth:

- Cancel **instant nahi hota**
- Cancel tab hota hai jab task **next `await` pe aaye**

❌ Agar task mein `await` hi nahi:

- Cancel fail
- Task zombie ban jaata hai

🔒 Rule

> Cancellation = cooperative

6️⃣ BACKGROUND TASKS & MEMORY LEAKS

❌ Dangerous Pattern (Orphan Task)

```
asyncio.create_task(job())
```

Problems:

- Reference nahi
- Cancel nahi
- Infinite background task

Result:

- RAM leak
- Long-running app crash
- Telegram bot 2 din baad dead

✅ Safe Thinking

Use `create_task()` only when:

- Task track ho
- Task cancel ho
- Background ka reason ho

🔒 Rule

> Fire-and-forget = future crash

7️⃣ ASYNC DESIGN PATTERNS (SYSTEM THINKING)

🧩 Pattern 1: Async Scraping (URLs)

Mental Flow:

1. URLs list
2. Async fetch (ONE url)
3. Tasks create
4. Gather execute
5. Results process

```
tasks = [fetch(url) for url in urls]
results = await asyncio.gather(*tasks, return_exceptions=True)
```

🧩 Pattern 2: API Fan-Out

Problem: Ek request → multiple APIs

❌ Slow:

```
a = await api1()
b = await api2()
```

☑ Fast:

```
tasks = [api1(), api2()]
a, b = await asyncio.gather(*tasks)
```

🧩 Pattern 3: Rate Limiting (Semaphore)

```
sem = asyncio.Semaphore(5)

async def safe_fetch(url):
    async with sem:
        return await fetch(url)
```

Why?

- Ban avoid
- 429 avoid

- Controlled concurrency

🔐 Rule

> Async ≠ unlimited speed Async = controlled speed

8️⃣ STARVATION (HIDDEN BUG)

Cause:

- Long CPU loop
- No `await`

Effect:

- Event loop ko control nahi milta
- Baaki tasks freeze

Solution:

- Frequent `await`
- CPU work async ke bahar

9️⃣ BLANK SCREEN DECISION FLOW (MOST IMPORTANT)

When stuck, ask:

❓ Situation → ✅ Tool

- Multiple URLs / APIs? → `asyncio.gather()`

- Delay chahiye? → `await asyncio.sleep()`

- Too many requests? → `asyncio.Semaphore()`

- Background kaam? → `asyncio.create_task()` (carefully)

- Loop ke andar `await` likhne ka mann? → ❌ STOP — redesign

- Cancel kaam nahi kar raha? → Check: next `await` hai?

- App long time baad crash? → Orphan tasks

🔑 KEY TERMS QUICK MAP

- Coroutine → pause/resume function
- Task → coroutine under event loop
- Event Loop → async manager
- Blocking → freezes loop
- Non-Blocking → allows switching
- Concurrency → multiple waiting tasks
- Parallelism → multi-core (not async)
- Gather → concurrent execution
- Semaphore → concurrency limiter
- Orphan Task → unowned background task
- Starvation → no await, no switching

---

10 PROTECTION LAYERS (SURVIVAL)

asyncio.wait_for() - THE DEADLINE MANAGER

X Galat (Blind Trust)

```
await fetch(url)
```

- Agar server hang hua?
- Task infinite wait karega
- Worker blocked forever

Sahi (Time Limit)

```
try:
   await asyncio.wait_for(fetch(url), timeout=5)
except asyncio.TimeoutError:
   print("Too slow, skip...")
```

- 5 second wait karega
- Agar response nahi aaya Error (TimeoutError)
- Worker free ho jayega

Why?

- Server hamesha fail nahi hote, kabhi kabhi bas "chup" ho jate hain (Hang).
- Bina timeout ke scraper 1 URL pe atak jayega.

Rule Network call = Always limited time. Unlimited wait = Suicide.

🔒 FINAL MASTER MENTAL MODEL (WRITE THIS)

> **Async programming ka flow:**
>
> 1. Kaam define karo (coroutines)
> 2. Sab kaam event loop ko do (tasks)
> 3. Waiting overlap hone do (gather)
> 4. Speed control karo (semaphore)
> 5. Failure aur cancellation ko design karo