# Strategy Pattern – Pluggable Schedulers (C# / Python / JS)

Below are OOP, strategy-based schedulers where the **Scheduler** depends only on an **interface**. You can swap **FCFS**, **SJF**, or **RR** without changing the `Scheduler` class.

---

## C# Implementation

```
using System;
using System.Collections.Generic;

public class Process
{
    public string Pid { get; }
    public int Remaining { get; set; }

    public Process(string pid, int remaining)
    {
        Pid = pid ?? throw new
        ArgumentNullException(nameof(pid));
        Remaining = remaining >= 0 ? remaining : throw new
        ArgumentException("Remaining must be >= 0");
    }
}

public interface ISchedulingStrategy
{
    // Choose the next process to run from the ready queue.
    Process DequeueNext(Queue<Process> readyQueue);
}
```

```
public sealed class FcfsStrategy : ISchedulingStrategy
{
    public Process DequeueNext(Queue<Process> readyQueue) =>
        readyQueue.Dequeue();
}

public sealed class SjfStrategy : ISchedulingStrategy
{
    public Process DequeueNext(Queue<Process> readyQueue)
    {
        // Find the process with the smallest Remaining and
        remove it from the queue.
        if (readyQueue.Count == 0) throw new
        InvalidOperationException("Queue is empty");
        Process best = null!;
        int n = readyQueue.Count;
        for (int i = 0; i < n; i++)
        {
            var p = readyQueue.Dequeue();
            if (best == null || p.Remaining < best.Remaining)
            {
                // put the previous best back when replaced
                if (best != null) readyQueue.Enqueue(best);
                best = p;
            }
            else
            {
                readyQueue.Enqueue(p);
            }
        }
        return best; // not in queue anymore
    }
}
```

```
public sealed class RrStrategy : ISchedulingStrategy
{
    public Process DequeueNext(Queue<Process> readyQueue) =>
        readyQueue.Dequeue();
}

public class Scheduler
{
    private readonly Queue<Process> _ready;
    private readonly ISchedulingStrategy _strategy;
    private readonly int _quantum; // for FCFS/SJF use
        int.MaxValue; for RR use finite value

    public Scheduler(Queue<Process> ready,
        ISchedulingStrategy strategy, int quantum)
    {
        _ready = ready ?? throw new
        ArgumentNullException(nameof(ready));
        _strategy = strategy ?? throw new
        ArgumentNullException(nameof(strategy));
        _quantum = quantum > 0 ? quantum : throw new
        ArgumentException("Quantum must be positive.");
    }

    public void Run()
    {
        while (_ready.Count > 0)
        {
            var p = _strategy.DequeueNext(_ready);
            int run = Math.Min(_quantum, p.Remaining);
            Console.WriteLine($"Running {p.Pid} for {run}
        units");
            p.Remaining -= run;
            if (p.Remaining > 0)
            {
                _ready.Enqueue(p); // preempted; goes to end
        (RR semantics); FCFS/SJF use large quantum to finish
```

```
            }
        }
    }
}

public static class Demo
{
    public static void Main()
    {
        var ready = new Queue<Process>(new[]
        {
            new Process("P1", 5),
            new Process("P2", 3),
            new Process("P3", 8)
        });

        // Swap strategy without touching Scheduler
        ISchedulingStrategy fcfs = new FcfsStrategy();
        ISchedulingStrategy sjf  = new SjfStrategy();
        ISchedulingStrategy rr   = new RrStrategy();

        Console.WriteLine("-- FCFS (runs to completion) --");
        new Scheduler(new Queue<Process>(ready), fcfs, int.MaxValue).Run();

        Console.WriteLine("\n-- SJF (runs shortest to completion) --");
        new Scheduler(new Queue<Process>(ready), sjf, int.MaxValue).Run();

        Console.WriteLine("\n-- RR (quantum = 2) --");
        new Scheduler(new Queue<Process>(ready), rr, 2).Run();
    }
}
```

**Key idea:** The **only** difference between FCFS/SJF and RR is *selection* vs *time slicing*. We model that by:

- Strategies decide **which process** to run next.
- The `Scheduler` executes for `min(quantum, remaining)` and requeues if needed. Set `quantum = int.MaxValue` to model non-preemptive FCFS/SJF.

---

## Python (same design)

```python
from collections import deque
from dataclasses import dataclass

@dataclass
class Process:
    pid: str
    remaining: int

class SchedulingStrategy:
    def dequeue_next(self, ready: deque[Process]) -> Process:
        raise NotImplementedError

class Fcfs(SchedulingStrategy):
    def dequeue_next(self, ready: deque[Process]) -> Process:
        return ready.popleft()

class Sjf(SchedulingStrategy):
    def dequeue_next(self, ready: deque[Process]) -> Process:
        # find shortest remaining and remove it
        best = None
        for _ in range(len(ready)):
            p = ready.popleft()
            if best is None or p.remaining < best.remaining:
```

```python
                if best is not None:
                    ready.append(best)
                best = p
            else:
                ready.append(p)
        return best

class Rr(SchedulingStrategy):
    def dequeue_next(self, ready: deque[Process]) -> Process:
        return ready.popleft()

class Scheduler:
    def __init__(self, ready: deque[Process], strategy:
        SchedulingStrategy, quantum: int):
        assert quantum > 0
        self.ready, self.strategy, self.quantum = ready,
        strategy, quantum

    def run(self):
        while self.ready:
            p = self.strategy.dequeue_next(self.ready)
            run = min(self.quantum, p.remaining)
            print(f"Running {p.pid} for {run} units")
            p.remaining -= run
            if p.remaining > 0:
                self.ready.append(p)

if __name__ == "__main__":
    base = deque([Process("P1",5), Process("P2",3),
        Process("P3",8)])
    Scheduler(deque(base), Fcfs(), 10**9).run()
    print("\nSJF:")
    Scheduler(deque(base), Sjf(), 10**9).run()
    print("\nRR:")
    Scheduler(deque(base), Rr(), 2).run()
```

## JavaScript (same design)

```
class Process {
  constructor(pid, remaining) {
    this.pid = pid;
    this.remaining = remaining;
  }
}

class FcfsStrategy { dequeueNext(q) { return q.shift(); } }

class SjfStrategy {
  dequeueNext(q) {
    if (q.length === 0) throw new Error("empty");
    let bestIdx = 0;
    for (let i = 1; i < q.length; i++) {
      if (q[i].remaining < q[bestIdx].remaining) bestIdx = i;
    }
    return q.splice(bestIdx, 1)[0];
  }
}

class RrStrategy { dequeueNext(q) { return q.shift(); } }

class Scheduler {
  constructor(ready, strategy, quantum) {
    if (quantum <= 0) throw new Error("quantum must be > 0");
    this.ready = [...ready];
    this.strategy = strategy;
    this.quantum = quantum;
  }
  run() {
    while (this.ready.length > 0) {
```

```
      const p = this.strategy.dequeueNext(this.ready);
      const run = Math.min(this.quantum, p.remaining);
      console.log(`Running ${p.pid} for ${run} units`);
      p.remaining -= run;
      if (p.remaining > 0) this.ready.push(p);
    }
  }
}

const base = [new Process("P1",5), new Process("P2",3), new
        Process("P3",8)];
new Scheduler(base, new FcfsStrategy(),
        Number.MAX_SAFE_INTEGER).run();
console.log("\nSJF:");
new Scheduler(base, new SjfStrategy(),
        Number.MAX_SAFE_INTEGER).run();
console.log("\nRR (q=2):");
new Scheduler(base, new RrStrategy(), 2).run();
```

---

**Swap strategies, not the scheduler.** That's the Strategy Pattern in action – and why you can add new algorithms later *without touching* the `Scheduler` class.