# Template Method Pattern

## 1. Intent

- **Define** the skeleton of an algorithm in a “template” method.
- **Defer** some steps (the “varying parts”) to subclasses without changing the overall sequence.

## 2. Key Concepts

- **Template Method**: a `final` method in an abstract class that outlines the algorithm’s steps in order.
- **Concrete Steps**: fully implemented methods in the abstract class (shared behavior).
- **Primitive Operations (Hooks)**: abstract or default-no-op methods that subclasses **must** or **may** override.
- **Hook**: a no-op/default method you can override for optional behavior, or ignore entirely.

## 3. Participants

1. **AbstractClass** (`CaffeineBeverage`)
   - Declares the `final prepare()` template method.
   - Implements concrete steps (e.g., `boilWater()`, `pourInCup()`).
   - Declares abstract hooks (`brewOrSteep()`, `addCondiments()`) and any default hooks.
2. **ConcreteClass** (`Tea`, `Coffee`)
   - Extends `AbstractClass`.
   - Overrides only the primitive operations/hook methods to provide drink-specific behavior.

## 4. Structure (simplified UML)

```
┌───────────────────────────────┐
│ AbstractClass                 │
│───────────────────────────────│
│ + final prepare()             │
│ # boilWater()                 │ ← concrete step
│ # brewOrSteep() «abstract»    │ ← primitive operation
│ # pourInCup()                 │ ← concrete step
│ # addCondiments() «hook»      │ ← optional hook
└───────────────────────────────┘
           │
┌─────────────────────┐    ┌─────────────────────┐
│   Tea               │    │   Coffee            │
│─────────────────────│    │─────────────────────│
│ + brewOrSteep()     │    │ + brewOrSteep()     │
│ + addCondiments()   │    │ + addCondiments()   │
└─────────────────────┘    └─────────────────────┘
```

## 5. Example Code Sketch

```
abstract class CaffeineBeverage {
  // Template method
  public final void prepare() {
    boilWater();
    brewOrSteep();               // subclass-specific
    pourInCup();
    if (wantsCondiments())        // optional hook
      addCondiments();           // subclass-specific
  }

  // Concrete steps
  private void boilWater() { System.out.println("Boiling water"); }
  private void pourInCup()  { System.out.println("Pouring into cup"); }

  // Primitive operations (hooks)
  protected abstract void brewOrSteep();
  protected abstract void addCondiments();

  // Optional hook with default
  protected boolean wantsCondiments() { return true; }
}

class Tea extends CaffeineBeverage {
  protected void brewOrSteep() { System.out.println("Steeping tea"); }
  protected void addCondiments() { System.out.println("Adding lemon"); }
}

class Coffee extends CaffeineBeverage {
  protected void brewOrSteep() { System.out.println("Brewing coffee"); }
  protected void addCondiments() { System.out.println("Adding sugar and
milk"); }
}
```

## 6. Relation to the Hollywood Principle

**“Don’t call us, we’ll call you.”**

- The template method **calls** subclass hooks at the right time—subclasses **do not** invoke the template.

## 7. When to Use

- You have multiple classes that share the same broad algorithm but differ in some steps.
- You want to enforce a fixed sequence while letting subclasses customize specific parts.
- You need optional steps (use hooks) that subclasses can override or skip.

## 8. Benefits

- **Code reuse**: common code lives in the abstract class.
- **Control**: template method is `final`, so sequence cannot be altered by subclasses.
- **Flexibility**: subclasses override only what they need.
- **Extensibility**: add new variants by creating new subclasses.

## 9. Drawbacks

- **Inheritance**: binds you to a class hierarchy; you cannot choose at runtime between different behaviors unless you introduce Strategy.
- **Complexity**: many small methods and classes can clutter simple use-cases.
- **Hook proliferation**: too many optional hooks can lead to unclear extension points.

## 10. Template vs. Strategy

| Aspect | Template Method | Strategy |
|---|---|---|
| **Who defines flow** | Abstract class defines algorithm sequence | Client/context assembles steps |
| **Extension mechanism** | Subclass overrides primitive steps/hooks | Pass interchangeable strategy objects |
| **Runtime choice** | Static—subclass chosen at compile time | Dynamic—swap strategy at runtime |
| **Use when** | Sequence is fixed but steps vary | Algorithm steps can be reordered or entirely swapped |