Major Programming Paradigms

There's more than just OOP

Natalie Agus

Procedural Programming

Linear, step-by-step execution with procedures/functions

- **Example Languages**: C, Pascal, or Python used **without** classes, like your CTD homework

```python
def get_total(price, qty):
    return price * qty

def apply_discount(total, rate):
    return total * (1 - rate)

total = get_total(50, 3)
final = apply_discount(total, 0.1)
print(f"Final amount: ${final}")
```

Object Oriented Programming (OOP)

Models real-world entities

- Supports **encapsulation**, **inheritance**, **polymorphism**.
- **Example Languages**: Java, Python, C++, C#

```python
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} says woof!"

d = Dog("Buddy")
print(d.speak())
```

Functional Programming

Focuses on pure functions, immutability, and no side effects.

- **Example Languages**: Haskell, Elixir, Scala, F#, Python (partial), JavaScript

```python
def is_even(n):
    return n % 2 == 0

def square(n):
    return n * n

nums = [1, 2, 3, 4, 5]
evens = filter(is_even, nums)
squared = map(square, evens)
print(list(squared))  # [4, 16]
```

Logic Programming

Uses facts and rules to derive answers via inference

- **Example Languages**: Prolog, Datalog

```
parent(john, mary).
parent(mary, alice).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

- John is parent of Mary, Mary is parent of Alice
- Query: ?-ancestor(john, alice) --> returns **TRUE**

Declarative Programming

Focuses on what should be done, not how to do it.

- **Example Languages**: SQL, HTML, CSS

```sql
SELECT name FROM users WHERE age > 18;
```

```html
<h1>Welcome</h1>
```

Event-Driven Programming

Control flow is driven by events such as user input, timers, or messages.

- **Example Languages**: Javascript (browser), Python (Tkinter)

```javascript
document.getElementById("btn").addEventListener("click", () => {
  alert("Button clicked!");
});
```

```python
import tkinter as tk
def on_click():
    print("Clicked!")
tk.Button(text="Click", command=on_click).pack()
tk.mainloop()
```

Reactive Programming

Deals with asynchronous data streams that react to changes over time.

- **Example Languages**: Javascript (RxJS), Dart (Flutter), Kotlin (Flow)

```
Stream<int> counterStream() async* {
  int count = 0;
  while (true) {
    await Future.delayed(Duration(seconds: 1));
    yield count++;
  }
}
```

Aspect-Oriented Programming (AOP)

Modularizes cross-cutting concerns (e.g., logging, security) using aspects.

- Goal: **cleanly separate** cross-cutting concerns like logging, security, transactions, from your **core business logic**
- **Example Languages**: Java (Spring AOP), AspectJ, .NET (PostSharp)
- Example without AOP:
  - Imagine we have 50 functions
  - Each one needs logging, access control, performance timing
  - Are we copy pasting that logging/ security/timing in EVERY function??

```python
def create_user(data):
    print("LOG: create_user called")   # Logging
    if not user_is_admin():            # Security check
        raise PermissionError
    start = time.time()
    ...  # actual user creation logic
    print("Execution time:", time.time() - start)
```

Aspect-Oriented Programming (AOP)

Modularizes cross-cutting concerns (e.g., logging, security) using aspects.

- With AOP:
  - Define separate "aspect"
  - AOP tools (like Spring AOP or AspectJ) **intercept** function/ method calls behind the scenes.
- "Before calling any method in any class inside com.myapp.service package, **run my logBefore() code**."

```
@Aspect
public class LoggingAspect {

    @Before("execution(* com.myapp.service.*.*(..))")
    public void logBefore(JoinPoint jp) {
        System.out.println("Calling: " + jp.getSignature().getName());
    }
}
```

Aspect-Oriented Programming (AOP)

Modularizes cross-cutting concerns (e.g., logging, security) using aspects.

- Python achieve the same effect with **decorators**

- In this example: Every call to functions decorated **now logs automatically**, no code duplicated

```python

def log_function(fn) -> _Wrapped[Callable[..., Any], Any, Calla…:
    """Decorator that logs when the function starts and finishes."""
    @wraps(fn)
    def wrapper(*args, **kwargs) -> Any:
        print(f"[LOG] Calling {fn.__name__}()")
        start: float = time.time()
        result: Any = fn(*args, **kwargs)       # run the real work
        duration: float = time.time() - start
        print(f"[LOG] {fn.__name__}() returned {result!r} in {duration:.4f}s")
        return result
    return wrapper


# Any function you decorate now gets automatic logging

@log_function
def add(a, b) -> Any:
    return a + b

@log_function
def greet(name) -> str:
    return f"Hello, {name}!"

@log_function
def slow_task() -> Literal['done']:
    time.sleep(0.3)
    return "done"

```