-Programming Paradigm-

Programming Paradigms

Programming paradigms are different approaches to solve problems using various programming languages. Each paradigm represents a distinct way of thinking about structure, data, and flow within a program.

---

1. Imperative Programming

- **Definition**: Instructs the machine on how to change its state step by step.
- **Key Features**: Explicit control flow, mutable state.
- **Languages**: C, Java, Python (can be imperative), Go, Fortran.
- **Example**:

```python
x = 0
for i in range(5):
    x += i
print(x)
```

---

2. Declarative Programming

- **Definition**: Focuses on what needs to be done, rather than how it is done.
- **Key Features**: Emphasizes expressions over statements, immutability.
- **Languages**: SQL, HTML, CSS, Prolog, Haskell.
- **Example (SQL)**:

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

---

3. Functional Programming

- **Definition**: Treats computation as the evaluation of mathematical functions without changing state or data.
- **Key Features**: First-class functions, immutability, no side effects.
- **Languages**: Haskell, Lisp, Scala, F#, Clojure.
- **Example**:

```
square x = x * x
map square [1, 2, 3, 4]
```

---

4. Object-Oriented Programming (OOP)

- **Definition**: Organizes code around objects, which bundle data and methods together.
- **Key Features**: Encapsulation, inheritance, polymorphism, abstraction.
- **Languages**: Java, C++, Python, Ruby, Smalltalk, C#.
- **Example**:

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

    def speak(self):
        print(f"{self.name} speaks")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks")

dog = Dog("Rex")
dog.speak()
```

---

5. Procedural Programming

- **Definition**: A subset of imperative programming focusing on procedure calls and routines (functions).
- **Key Features**: Code is divided into procedures that operate on data.
- **Languages**: C, Pascal, Fortran, Ada.
- **Example**:

```
int sum(int a, int b) {
    return a + b;
}
```

---

6. Logic Programming

- **Definition**: Focuses on rules and facts; computation derives conclusions based on rules.
- **Key Features**: Rule-based, query solving.
- **Languages**: Prolog, Datalog.
- **Example (Prolog)**:

```
parent(john, mary).
parent(mary, susan).
ancestor(X, Y) :- parent(X, Y).
```

7. Event-Driven Programming

- **Definition**: The flow of the program is determined by events like user inputs or messages.
- **Key Features**: Event loops, handlers, callback functions.
- **Languages**: JavaScript, C#, VB.NET, Swift.
- **Example (JavaScript)**:

```
document.getElementById("button").addEventListener("click", function() {
    alert("Button clicked!");
});
```

8. Concurrent Programming

- **Definition**: Allows multiple computations to occur in overlapping time periods.
- **Key Features**: Threads, synchronization, parallel execution.
- **Languages**: Go, Erlang, Python (with threads), Java (with threads), Rust.
- **Example (Python)**:

```
import threading

def print_hello():
    print("Hello from thread")

thread = threading.Thread(target=print_hello)
thread.start()
```

9. Parallel Programming

- **Definition**: Executes multiple computations simultaneously on multiple processors.
- **Key Features**: Multi-core execution, SIMD, distributed computing.
- **Languages**: CUDA, OpenMP, MPI, Go, Python (multiprocessing).
- **Example (Python)**:

```
from multiprocessing import Pool

def square(x):
    return x * x
```

```python
with Pool(4) as p:
    print(p.map(square, [1, 2, 3, 4]))
```

---

10. Aspect-Oriented Programming (AOP)

- **Definition**: Separates concerns, especially cross-cutting concerns like logging, security, or transactions.
- **Key Features**: Modularity, separation of concerns.
- **Languages**: AspectJ (Java), PostSharp (C#), Python (via decorators).
- **Example (Python)**:

```python
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log
def greet(name):
    print(f"Hello, {name}")

greet("Alice")
```

---

11. Reactive Programming

- **Definition**: Programming with asynchronous data streams; reacts to data as it arrives.
- **Key Features**: Data flow, propagation of change, async events.
- **Languages**: RxJava, RxJS, Akka, Dart (Flutter), Swift (Combine).
- **Example (RxJS)**:

```javascript
const { fromEvent } = rxjs;
const { map } = rxjs.operators;

fromEvent(document, 'click')
    .pipe(map(event => `Clicked at ${event.clientX}, ${event.clientY}`))
    .subscribe(console.log);
```

---

12. Query-Based Programming

- **Definition**: Specifies what to achieve rather than how, often using queries.
- **Key Features**: High-level abstractions, often used in databases and configuration.
- **Languages**: SQL, XQuery.

- **Example (SQL)**:

```sql
SELECT * FROM employees WHERE salary > 50000;
```

13. Dataflow Programming

- **Definition**: Models programs as directed graphs of the data flowing between operations.
- **Key Features**: No explicit control flow, data-driven execution.
- **Languages**: LabVIEW, TensorFlow, Spark.
- **Example (TensorFlow)**:

```python
import tensorflow as tf
x = tf.constant([1, 2, 3])
y = tf.square(x)
print(y)
```

14. Metaprogramming

- **Definition**: Writing programs that write or manipulate other programs.
- **Key Features**: Code generation, reflection.
- **Languages**: Lisp (macros), C++ (templates), Python (metaclasses), Ruby.
- **Example (Python)**:

```python
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass
```

15. Template Metaprogramming

- **Definition**: A technique used in languages like C++ to perform computations at compile-time.
- **Key Features**: Compile-time logic, type manipulations.
- **Languages**: C++, D.
- **Example (C++ template)**:

```cpp
template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static const int value = 1;
};
```

Each paradigm has its own strengths and is suited for different types of problems. Many modern languages support multiple paradigms, allowing developers to choose the best approach based on their needs.