CSE216
Programming Abstractions

State University of New York, Korea

Agenda

- Programming paradigms: imperative, procedural, reactive, declarative, object-oriented, functional

- Demo

Programming Paradigms

Programming Paradigms

- A programming paradigm is a style or approach to programming that provides **a framework for building a software system.** It is a **set of principles, concepts, and practices** that define the way of thinking and organizing the code to solve a specific problem.

There are several programming paradigms, and each has its unique way of approaching problem-solving, code organization, and design. Some of the popular programming paradigms include:

- **Imperative Programming:** This paradigm focuses on the sequence of **statements that modify the state of the program**, and how to control that sequence using conditional statements, loops, and other control flow constructs.

- **Object-Oriented Programming (OOP)**: This paradigm is based on the idea of **organizing software systems as a collection of objects** that interact with each other to solve a problem.

- **Functional Programming:** This paradigm emphasizes **the use of functions** to solve problems and avoid changing the state of the program.

- **Declarative Programming:** This paradigm focuses on **describing the problem** to be solved, rather than how to solve it.

Choosing a programming paradigm for your work

- The choice of a programming paradigm depends on the problem domain, the team's expertise, and the software requirements. **Each paradigm has its strengths and weaknesses, and the choice of the paradigm can have a significant impact** on the design, maintainability, and performance of the software system.

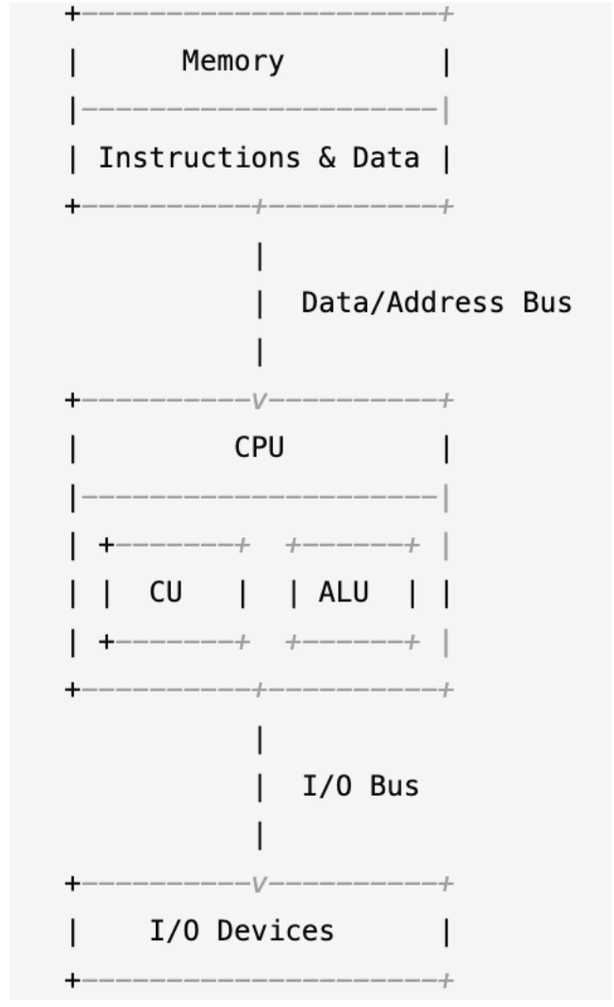
Imperative programming: All about states

- Imperative programming is a programming paradigm that emphasizes the sequence of statements that **modify the state of the program.** In imperative programming, the program consists of a set of commands or statements that change the program's state.

- What can be a state?

Imperative programming: Memory-cpu-io

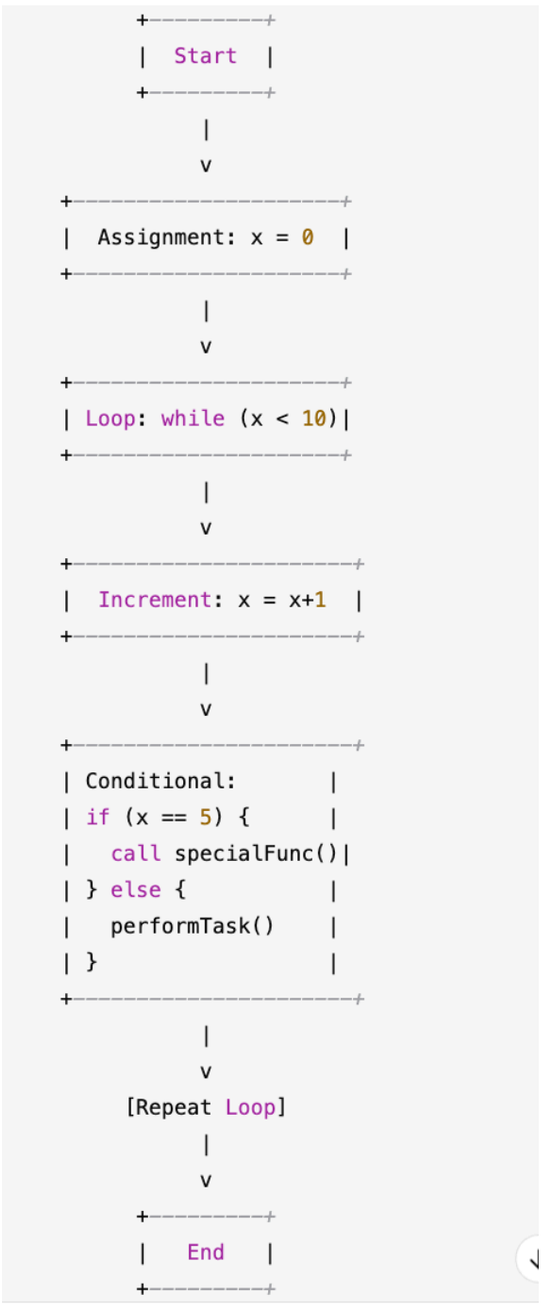
- Imperative programming is **based on the Von Neumann architecture**, which describes a computer as a machine that stores data and instructions in memory, fetches instructions from memory, and executes them in a sequential manner.



Imperative programming: Command Your Code!

- In imperative programming, the programmer specifies how the program should perform its tasks, by **giving a sequence of commands** to be executed by the computer. These commands can include assignments, loops, conditionals, and function calls.

- Be the **micro-management Director:** Think of it as directing a performance where every command you write choreographs the computer's actions.



Imperative programming: Pro and cons

- +: **a great deal of control** over the program's behavior.

- -: challenging to write, debug, and maintain large imperative programs

Imperative Programming

- A **statement** is a syntactic unit of an imperative programming language that expresses some action to be carried out.
- The program in such a language thus becomes a sequence of statements.

```
assert(x == 7);       /* assertion statement; programmer assumes the
                         value of x to be 7 after this line */
x = 2 + 3;            /* assignment statement */
2 + 3;                /* has no effect; smart compilers will discard
                         this line */
puts(“hello world”); /* a function call */
goto label;           /* a goto statement */
return 0;             /* a return statement */

x = sqrt(2); /* an assignment statement containing a function call */
```

Simple Statements

Imperative Programming

- An **assignment statement** performs an operation on information located in memory and stores the results in memory for later use.
  - Higher-level imperative languages permit evaluation of complex expressions that may consist of a combination of arithmetic operations and function evaluations.

```c
assert(x == 7);        /* assertion statement; programmer assumes the
                          value of x to be 7 after this line */
x = 2 + 3;             /* assignment statement */
2 + 3;                 /* has no effect; smart compilers will discard
                          this line */
puts("hello world"); /* a function call */
goto label;            /* a goto statement */
return 0;              /* a return statement */

x = sqrt(2); /* an assignment statement containing a function call */
```

Simple Statements

C / C++

```c
int x = 10;
```

JavaScript

```javascript
let x = 10;
```

Imperative Programming

- A conditional statement allows a sequence of statements (known as a *block* or a *code block*) to be executed only if some condition is met.
- Otherwise, the statements are skipped, and the execution sequence continues from the statement following them.

```
if (happy) {
    smile();
} else if (sad) {
    frown();
} else {
    stoic();
}
```

```
switch (i % 2) {
    case 0:
        type = EVEN;
        break;
    default:          /* equiv. to case 1 */
        type = ODD;
        break;
}
```

Compound Statements

Imperative Programming

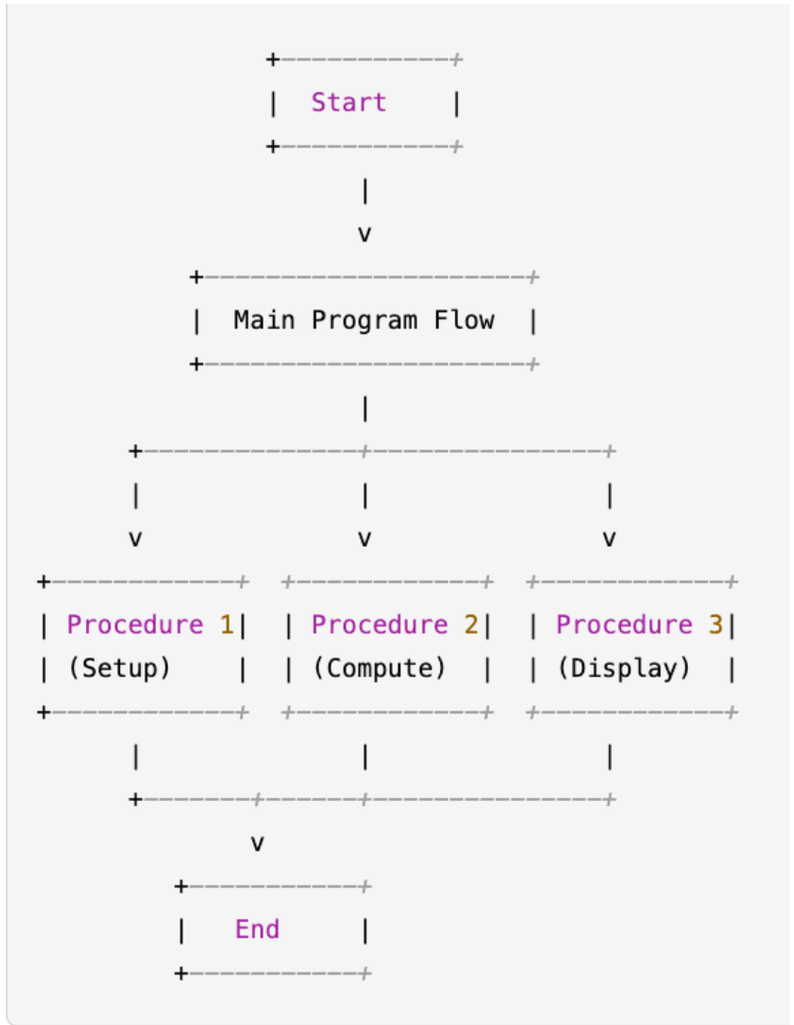
- Looping statements allow a block to be executed multiple times.
- Loops can execute a block a predefined number of times, or they can execute them repeatedly until some condition changes.

```c
while ((c = getchar()) != EOF) {
    putchar(c);
}

do {
    computation(i);
    ++i;
} while (i < 10);

for (i = 1; i < n; i *= 2) {
    printf("%d\n", i);
}
```

Compound Statements

Procedural Programming

- Break It Down: split your program into smaller, manageable procedures. Each procedure is like a mini-task that does one specific job.
- Sequence Matters
- Focus on Procedures: Procedures execute a series of instructions using loops, conditionals, and other control structures like in imperative programming.



Procedural Programming

- Procedural programming is particularly **useful for developing programs that perform a series of operations on data,** as the data can be passed between functions to perform various operations.

- How to make a Korean chatbot if you only speak English?

Procedural Programming: Pro and Cons

- +: **easy to understand and debug,** as each function is responsible for a specific task.

- -: (like imperative programming) challenging to write and maintain large procedural programs, as the functions can become complex and difficult to manage.

An example in C

```c
#include <stdio.h>

// Define a function that prints the Fibonacci sequence up to n
void fibonacci(int n) {
    int a = 0, b = 1, c, i;
    printf("%d %d ", a, b);
    for(i = 2; i < n; i++) {
        c = a + b;
        printf("%d ", c);
        a = b;
        b = c;
    }
}

// Call the function to print the Fibonacci sequence up to 10
fibonacci(10);
```

An example in Python

```python
# Function to calculate the area of a rectangle
def calculate_area(length, width):
    area = length * width
    return area

# Function to calculate the perimeter of a rectangle
def calculate_perimeter(length, width):
    perimeter = 2 * (length + width)
    return perimeter

# Main program
length = float(input("Enter the length of the rectangle: "))
width = float(input("Enter the width of the rectangle: "))

area = calculate_area(length, width)
perimeter = calculate_perimeter(length, width)

print("The area of the rectangle is:", area)
print("The perimeter of the rectangle is:", perimeter)
```

Object-oriented Programming

- Object-oriented programming (OOP) is a programming paradigm or a style of programming that is **based on the concept of "objects." An object is** a self-contained unit that consists of both **data and the methods** that operate on that data. In OOP, everything is treated as an object, and the code is organized around these objects.

- OOP is widely used in software development because it provides an **advantageous** way of **creating complex programs,** making it easier to write, test, and maintain code. Some of the most popular programming languages that use OOP include Java, C++, Python, and Ruby.

Core concepts in OOP

- **Encapsulation**: It means that the data and behavior of an object are **hidden** from the outside world, and only the methods that the object exposes can be used to interact with it.

- **Inheritance**: It allows for the creation of new classes by inheriting properties and methods **from existing ones**.

- **Polymorphism**: It means that objects can take on different forms and **exhibit different behaviors, depending on the context** in which they are used.

OOP concept 1: Encapsulation

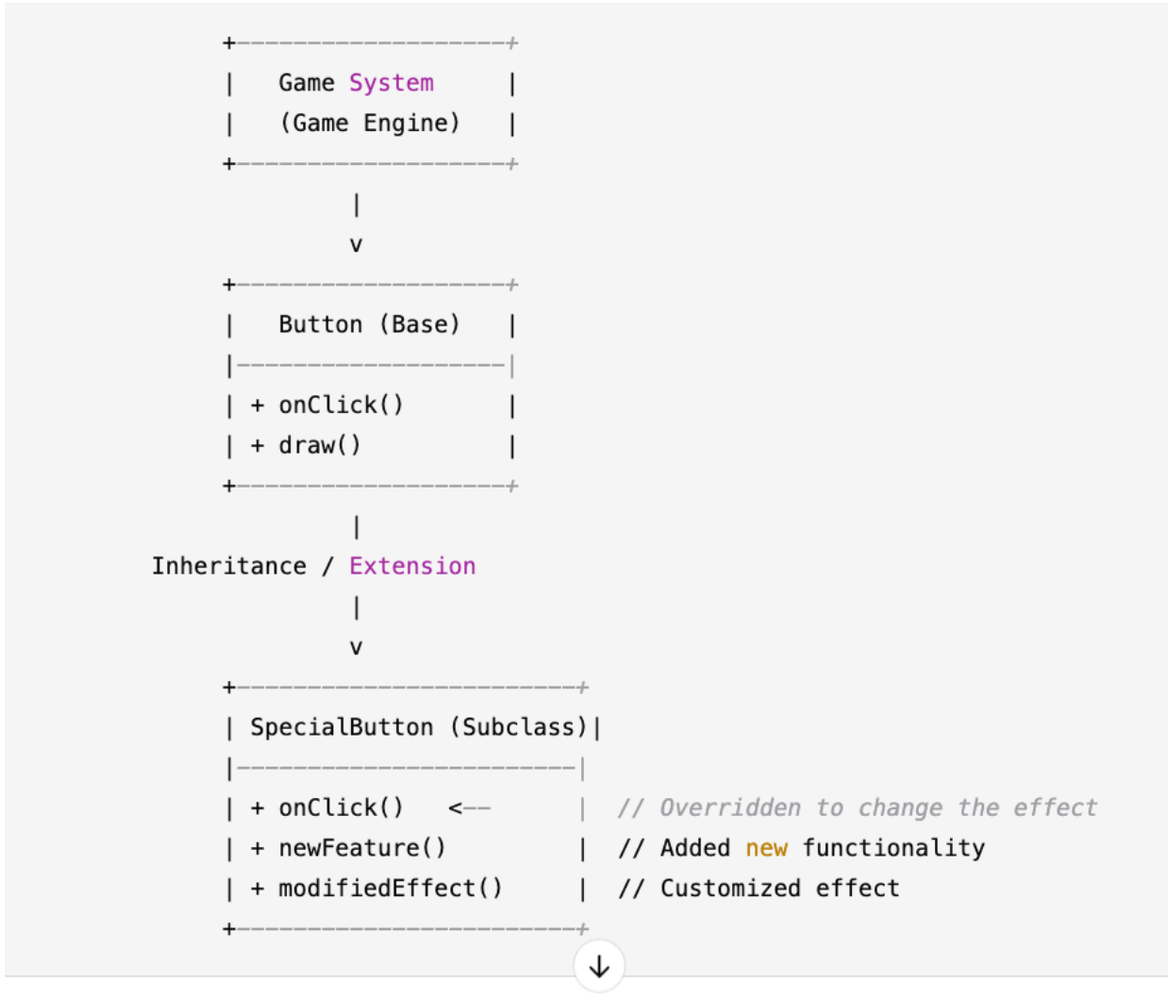
What is the benefit of using a private balance?

```java
public class BankAccount {
    private int balance;

    public BankAccount(int balance) {
        this.balance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds");
        }
    }
}
```

OOP concept 2: Inheritance

```java
class Vehicle {
    protected String make;
    protected String model;
    protected int year;

    public Vehicle(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}

class Car extends Vehicle {
    private String country;

    public KoreanCar(String make, String model, int year, String country) {
        super(make, model, year);
        this.country = country;
    }

    public String getCountry() {
        return country;
    }
}

class Main {
    public static void main(String[] args) {
        Car car = new Car("Kia", "Seltos", 2022, "South Korea");
        System.out.println(car.make); // prints "Kia"
        System.out.println(car.model); // prints "Seltos"
        System.out.println(car.year); // prints 2022
        System.out.println(car.getCountry()); // prints "South Korea"
    }
}
```

- How to reimplement a button if you don’t like it?



OOP concept 3: Polymorphism

```java
// Base class
class Shape {
    public void draw() {
        System.out.println("Drawing a shape...");
    }
}

// Derived class 1
class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a circle...");
    }
}

// Derived class 2
class Square extends Shape {
    public void draw() {
        System.out.println("Drawing a square...");
    }
}

// Main class
class Main {
    public static void main(String[] args) {
        // Create an array of shapes
        Shape[] shapes = { new Circle(), new Square() };

        // Draw all the shapes in the array
        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}
```

- Thanks to polymorphism, we can treat each object in the array as if it were a Shape object, even though they are actually instances of different classes. This allows us to write code that is flexible and reusable — we can easily add new shapes to the array without having to change the for loop or any other code.

Declarative programming languages

- In declarative programming, the program **describes the desired result, rather than specifying how to achieve it.** This is in contrast to procedural programming, where the program specifies a series of steps to accomplish a task.

- One example of declarative programming is SQL (Structured Query Language), which is used to query relational databases. In SQL, a programmer specifies the criteria for selecting records from a database, and the database management system determines how to retrieve the data.

- Another example is HTML. Programmers specify web structure and content using tags, and the web browser determines how to render the page based on those tags.

Declarative programming in SQL

```sql
-- Define a table of students
CREATE TABLE students (
  id INT,
  name VARCHAR(255),
  major VARCHAR(255),
  gpa FLOAT
);

-- Select all students with a GPA greater than 3.0
SELECT id, name, major
FROM students
WHERE gpa > 3.0;
```

- Note that we're not specifying how to retrieve the data or iterating over it step-by-step, but rather describing the desired result in a declarative way. SQL takes care of the details of how to retrieve the data and return the desired result.

- So, this is declarative programming - we declare what we want to happen, and the programming language takes care of the details.

Declarative programming in HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <h1>Welcome to My Web Page</h1>
    <p>This is a paragraph of text.</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

- Note that we're not specifying how the page should be rendered, but rather describing the structure and content of the page in a declarative way. The web browser takes care of the details of how to render the page.

Functional Programming

1

Based on recursive definitions.

- They are inspired by a computational model called **lambda calculus**, developed by Alonzo Church in the 1930s.

2

A program is viewed as a mathematical function that transforms an input to an output. It is often defined in terms of simpler functions.

- We will see many examples of functional programming in multiple languages (e.g., Java, Python, OCaML).

Functional Programming in Python

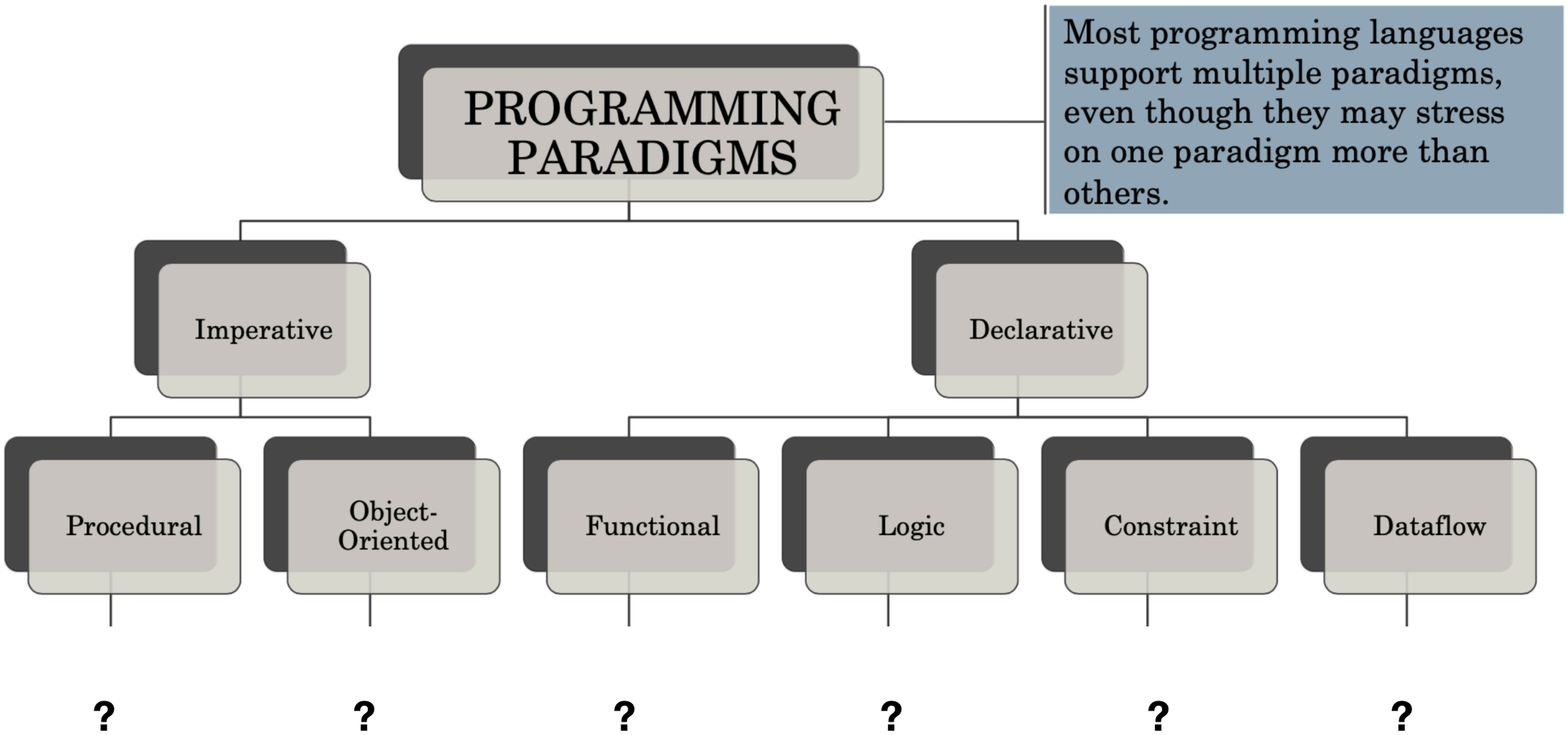
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


even_numbers = filter(lambda x: x % 2 == 0, numbers)
squared_numbers = map(lambda x: x ** 2, even_numbers)
sum_of_squares = sum(squared_numbers)
```

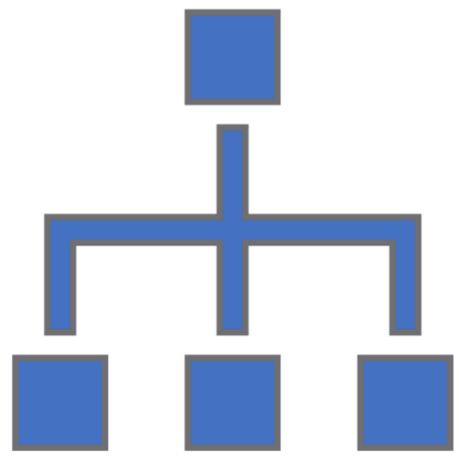
Functional Programming in Java

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sumOfSquares = numbers.stream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

Functional Programming in Ocaml

```
let numbers = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;


let is_even x = x mod 2 = 0;;

let sum_of_squares =
  List.filter is_even numbers
  |> List.map (fun x -> x * x)
  |> List.fold_left (+) 0;;
```


PROGRAMMING PARADIGMS
Most programming languages support multiple paradigms, even though they may stress on one paradigm more than others.
Imperative
Declarative
Procedural
Object-Oriented
Functional
Logic
Constraint
Dataflow
?
?
?
?
?
?



- One more example. Three paradigms.

- Implementing the greatest common divisor (GCD) solution in different paradigms and different languages.

The GCD problem: pseudocode

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

Paradigm 1:
Imperative Programming in Python

```
1  def gcd (x, y):
2      while (x!=y):
3          if (x>y): x = x-y
4          else: y = y -x
5      return x
6
7  x = 15
8  y = 40
9  print (gcd(x,y))
```

Paradigm 2:
Object-Oriented Programming in Java

/MyClass.java

```java
public class MyClass {



    static int gcd (int x, int y){
        while (x!=y){
            if (x>y) x = x-y;
            else y = y -x;
        }
        return x;
    }



    public static void main(String args[]) {
      int x=10;
      int y=25;


      System.out.println("GCD of x+y = "+ gcd(x,y));
    }
}
```

Paradigm 3:
Functional Programming in Ocaml

```
1  let rec gcd (x, y) = if x> y then gcd (x-y, y) else if x < y then gcd (x, y -x) else x ;;
2
3  let r= gcd(15, 40);;
4
5  print_int r;;
6
```

Summary

- Imperative, functional, object-oriented paradigms

- Instead of learning languages by languages, it is much more efficient to learn programming language paradigms.