CSE216
Programming Abstraction

Instructor: Zhoulai Fu

State University of New York, Korea

Course materials and Info available here:
https://github.com/zhoulaifu/23_cse216_spring

Some slides are taken from SBU. Thanks!
Some writings and code are generated by ChatGDP. Wow :-)

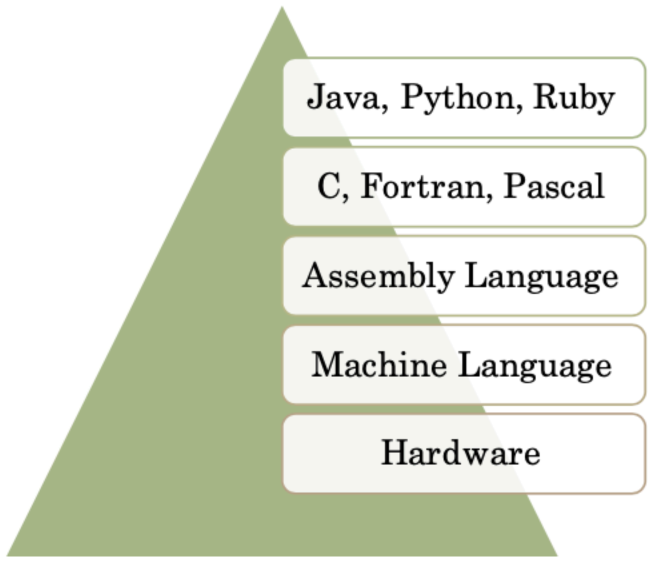
What is Programming Abstraction

- Programming abstraction is a technique used in computer programming **to simplify the complexity** of writing and understanding code. It involves creating higher-level abstractions that **hide low-level implementation details**, allowing programmers to focus on solving problems at a higher level of abstraction.

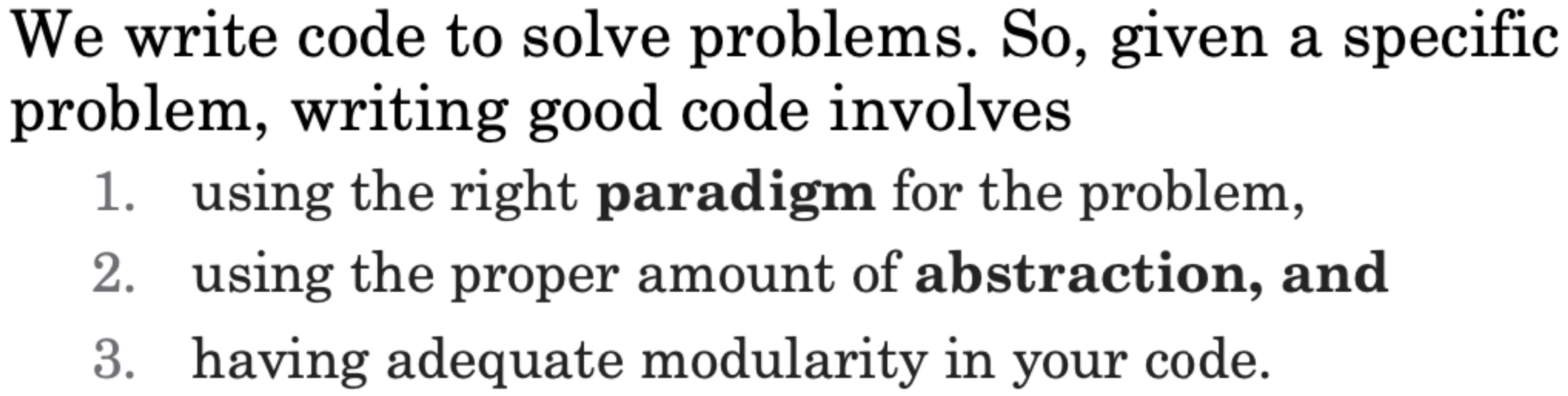
- Abstractions in programming can take many forms, including **data structures, functions, and classes.** For example, a programmer can use a class in object-oriented programming to group together related functions and data, creating a higher-level abstraction that makes it easier to work with the program.

Abstraction in Language Evolution

- Most programming languages are *high-level* languages, where the phrase “high-level” indicates a higher degree of abstraction.
- The second word of this course’s name – **abstraction** – is among the most important ideas in programming.
- It refers to the degree to which the language’s features are separated away from the details of a particular computer’s architecture and/or implementation at *lower* levels.



Why abstraction?



We write code to solve problems. So, given a specific problem, writing good code involves

1. using the right **paradigm** for the problem,
2. using the proper amount of **abstraction, and**
3. having adequate modularity in your code.

- Programming abstraction is essential in software engineering because it allows programmers to manage the complexity of large software projects. By providing high-level abstractions, it makes it easier for developers to work collaboratively, and it allows for easier maintenance and testing of software code.

This course

- In this course, we work with three programming languages: **OCaml, Java, and Python**. However, the course **does not solely focus on these individual programming languages**. If you approach the course with a narrow focus on the syntax of each language, you may find it more challenging than necessary.

- Instead, the course **emphasizes the underlying concepts that are common to all programming languages.** We examine the programming paradigms that have emerged. Each paradigm has its own strengths and weaknesses, and our goal is to **gain a deep understanding of the various ways of thinking about programming**. This will enable us to determine, based on a given scenario, which language and paradigm to use to write efficient and effective code.

Course outcomes

An understanding of programming paradigms and tradeoffs.

An understanding of functional techniques to identify, formulate, and solve problems.

An ability to apply techniques of object-oriented programming in the context of software development.

Meet the Instructor

Education

- B.Sc, M.Sc, Ecole Polytechnique, France
- M.Eng. Telecom Paris, France
- Ph.D. INRIA (National CS Lab), France

Teaching & Research

- University of California Davis, United States
- IT University of Copenhagen, Denmark
- SUNY Korea

TA

- Unknown 1
- Unknown 2

Team

You	TA	Instructor	ChatGPT
Lectures	Office hours	Office hours	Not do homework
Homework	Answer questions	Lectures	Answer questions
Ask questions		Grading	
		Answer questions	

Practical matters

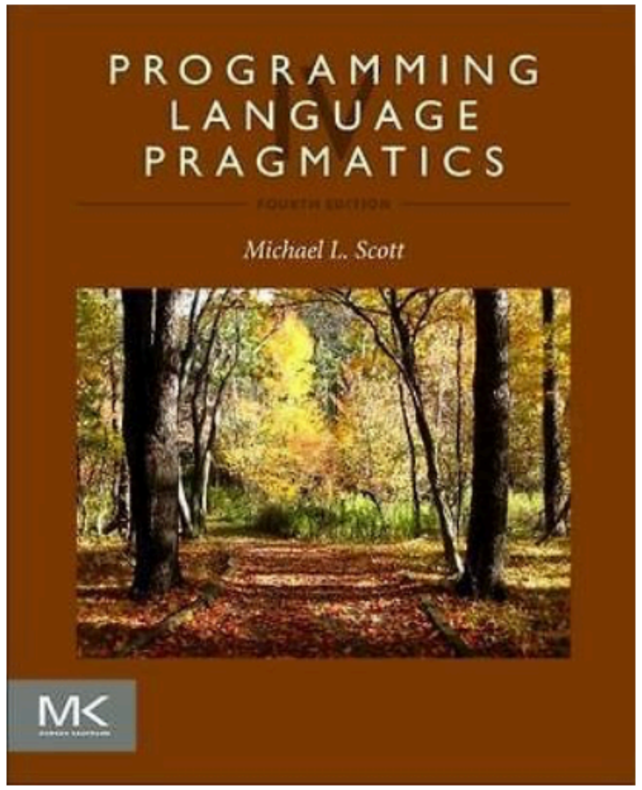
- COVID
- Reference books
- Schedule
- Exams and grading

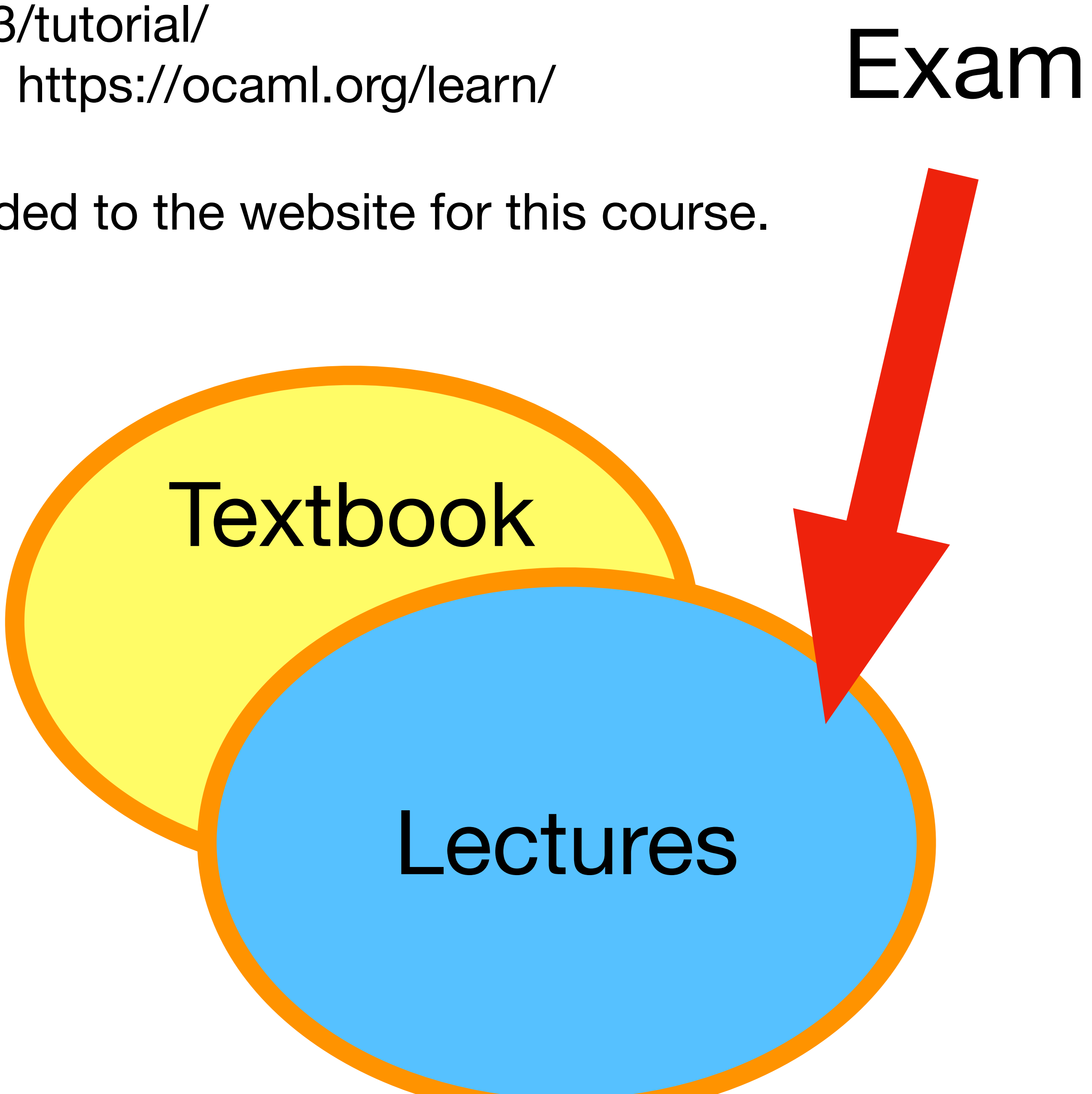
Covid is gone, but

- Inform instructor immediately of the date of a positive test. Your absences will be excused
- Follow government guidelines including a 7-day quarantine.
- Return to the class after quarantine. Negative test not needed.

Reference books and reading material

- Michael L. Scott. Programming Language Pragmatics.
- For details pertaining to specific programming languages, the recommended material will mostly be from the following:
  Python tutorial: https://docs.python.org/3/tutorial/
  The official OCaml learning material from https://ocaml.org/learn/
- Other reading material (if used) will be added to the website for this course.

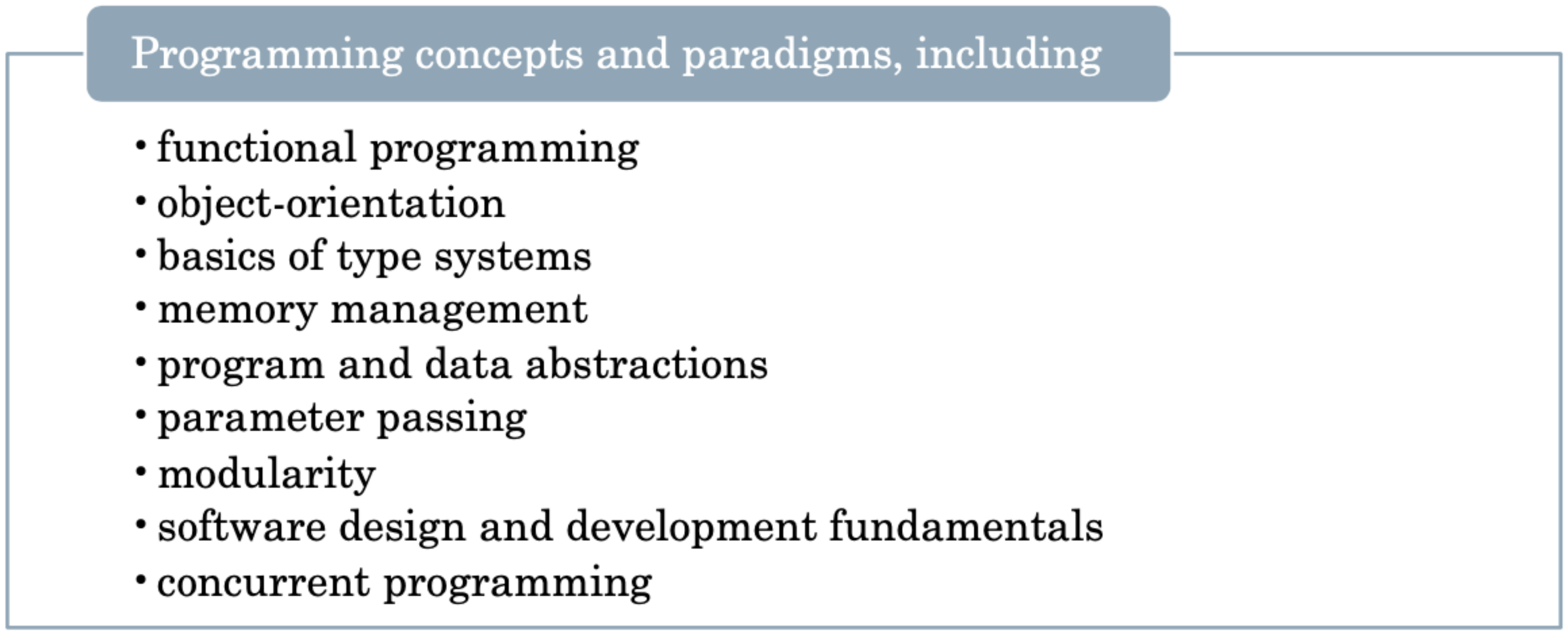




Schedule

- Lectures: Monday and Wednesday 2:00 - 3:20pm, at B203
- Recitation: Wednesday 3:30 - 4:25pm, at B203
- Office hours: Monday 5:00 - 6:00pm and Wednesday 8:00 - 9:00pm, at B424
- TA office hours: TBA
- course website: https://github.com/zhoulaifu/23_cse216_spring

Course outline



Programming concepts and paradigms, including

- functional programming
- object-orientation
- basics of type systems
- memory management
- program and data abstractions
- parameter passing
- modularity
- software design and development fundamentals
- concurrent programming

Grading

- Attendance: 5%
- take-home assignment (homework): 25%
- In-class assignments (quiz): 10%
- Midterms: 30%
- Final exam: 30%
- Students with regular participation get 1% bonus

Cont.

- In-class assignments take place in recitation classes (Wednesday). Format: Paper-based, open-notes. No Internet.
- In-class assignments are basic exercises.
- Take-home assignments take place at the end of each “chapter”, usually on Thursday
- Take-home assignments are harder and may need reflection, but you can ask for help.
- Exam format = In-class assignment format.

Questions so far?

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.

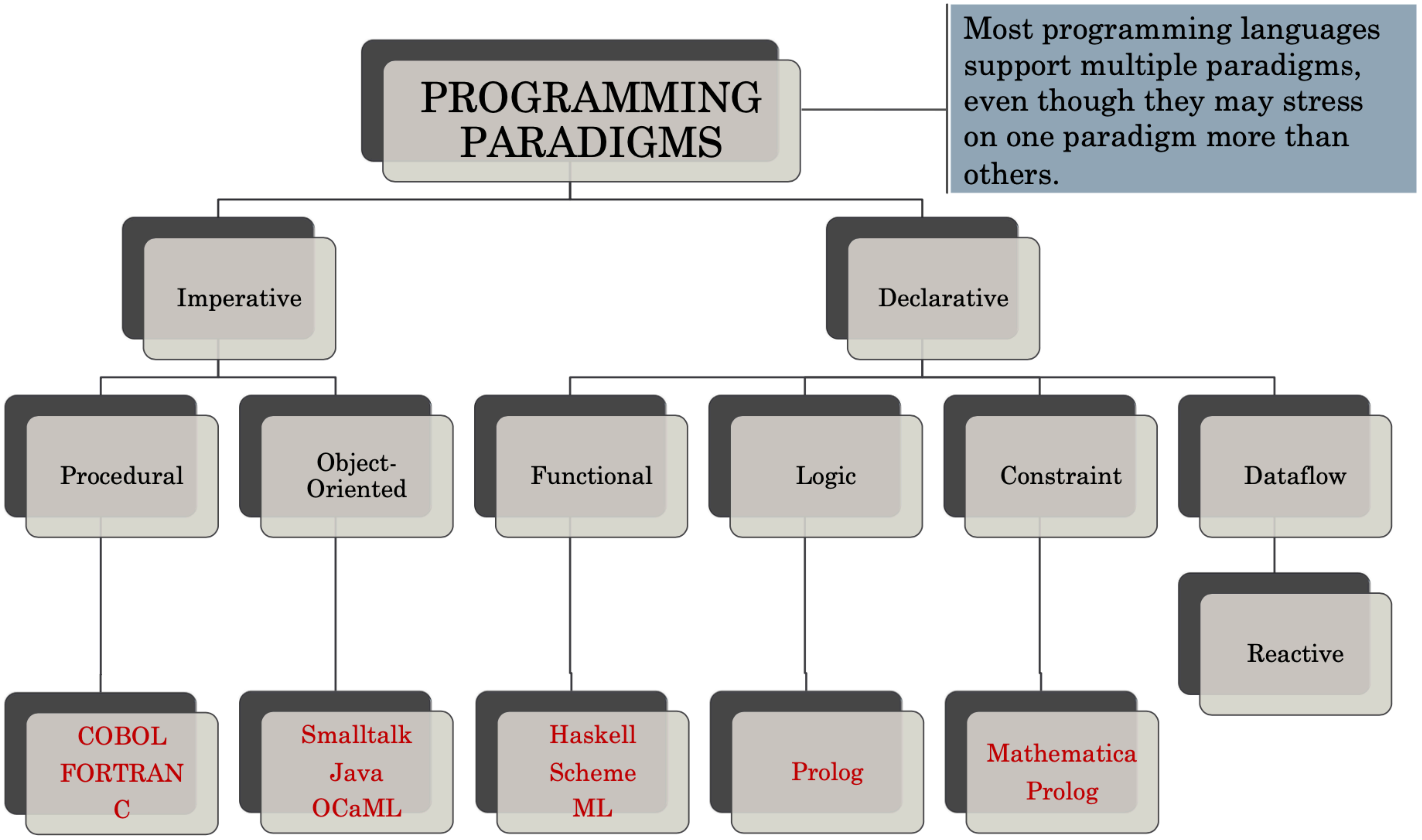
There are several programming paradigms, and each has its unique way of approaching problem-solving, code organization, and design. Some of the popular programming paradigms include:

- **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.



Most programming languages support multiple paradigms, even though they may stress on one paradigm more than others.

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

- 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.
- 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.
- 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.
- Examples of imperative programming languages include **C, Java, Python**, and many others. These languages offer a rich set of control flow constructs and operators that enable the programmer to manipulate the program's state in a variety of ways.
- One of the advantages of imperative programming is that it provides the programmer with **a great deal of control** over the program's behavior. However, it can be challenging to write, debug, and maintain large imperative programs, as they can quickly become complex and difficult to understand.

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

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

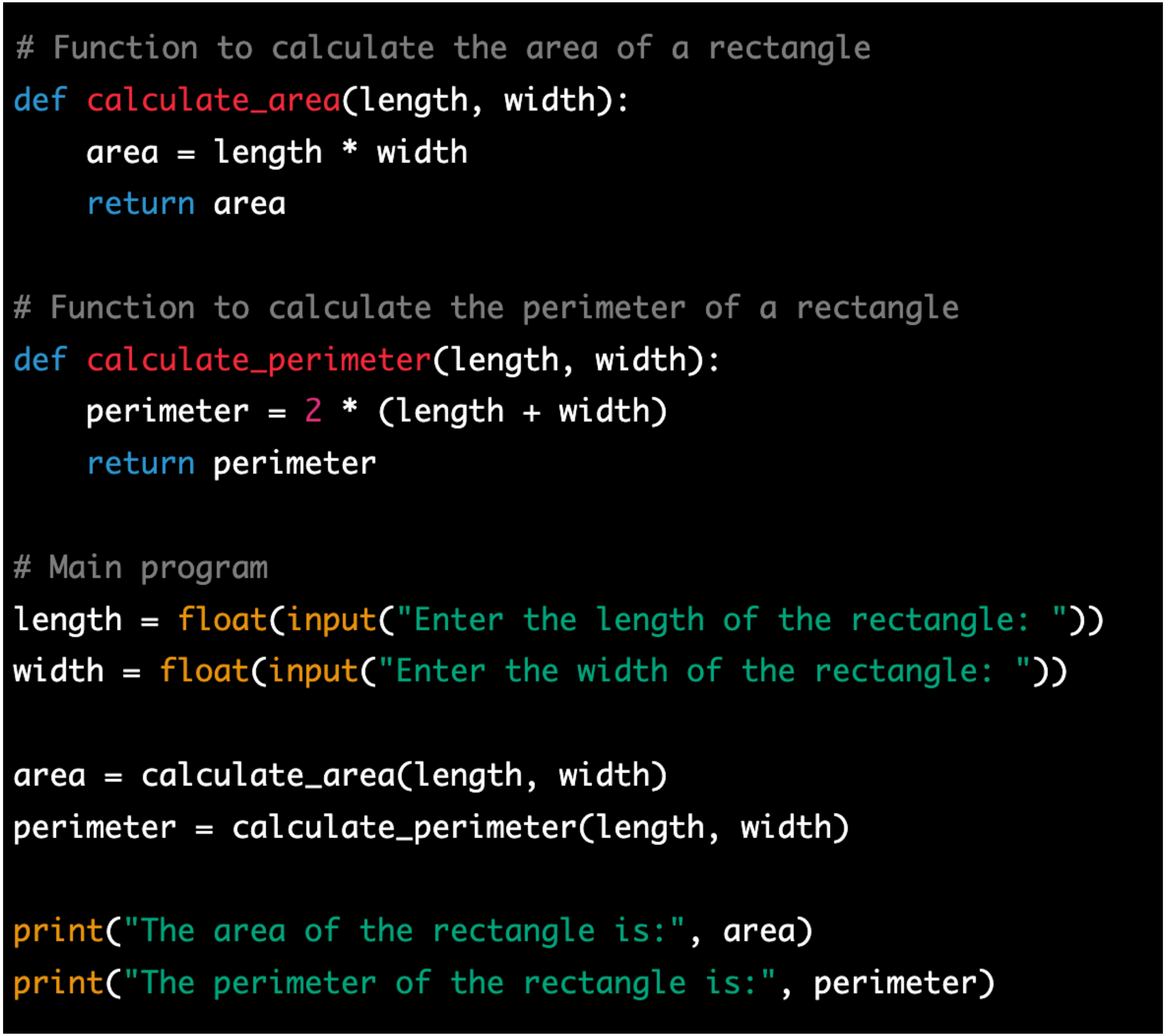
- Procedural programming is a programming paradigm that is based on the idea of **breaking down a program into smaller, reusable procedures**. In procedural programming, the program consists of a collection of procedures that operate on data, and the procedures are called in a specified sequence to perform a particular task.

- In procedural programming, the **focus is on the procedures** that are executed, rather than the data that is manipulated. The procedures are defined using a set of instructions that are executed in a sequence, with control structures such as loops and conditionals used to control the flow of the program.

- 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. It is widely used for developing programs that involve input/output operations, data processing, and mathematical calculations.

- Some popular languages that support procedural programming include C, Pascal, and Fortran. However, many modern programming languages, such as Python and Ruby, also support procedural programming in addition to other programming paradigms such as object-oriented programming and functional programming.

- One of the **advantages** of procedural programming is that it is relatively **easy to understand and debug,** as each function is responsible for a specific task. However, it can be challenging to write and maintain large procedural programs, as the functions can become complex and difficult to manage.



```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)
```

- In this example, we have two functions: calculate_area and calculate_perimeter. These functions take the length and width of a rectangle as inputs and return the area and perimeter, respectively.

- The main program prompts the user to enter the length and width of a rectangle, calls the calculate_area and calculate_perimeter functions, and then prints the results.

- This code demonstrates the basic principles of procedural programming, which involve breaking down a program into smaller, reusable procedures, and calling these procedures in a specific sequence to perform a particular task.

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.

Encapsulation

OOP languages usually provide ways to do the following:

a) restrict access to some of the object's components (attributes and/or methods)

b) bundle the data with the methods operating on that data

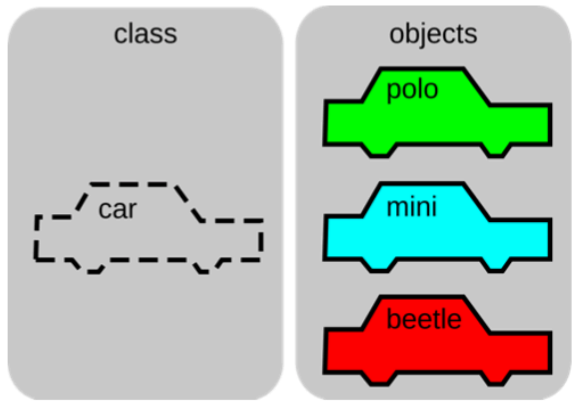
- Either (a) or (b) or a combination of the two is known as **encapsulation**.
- Often, (a) just by itself is known as **information hiding**.

```java
public class Person {
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

What could happen if `name` is not **`private`**?

Inheritance

- An object is an instance of a class.



Source: Pluke [CC0], from Wikimedia Commons

- In class-based OOP languages, classes are defined beforehand, and the objects are instantiated based on the classes.
- Given an object, a programmer can safely assume all the properties of the class to which the object belongs.

Inheritance is the mechanism of basing an object (or class) upon another object (or class) retaining similar implementation.

- In some OOP languages, subclasses inherit the features of one superclass. This is known as **single inheritance**.
- If one class can have more than one superclass and inherit features from multiple parent classes, it is known as **multiple inheritance**.

Polymorphism

Polymorphism is a concept that refers to the ability of an entity to take on multiple forms.

```
class Animal {
  public void move() { /* no implementation */ }
}

class Horse extends Animal {
  public void move() { gallop(); }
}

class Bird extends Animal {
  public void move() { fly(); }
}

class AnimalPolymorphism {
  public static void main(String[] args) {
    Animal a1 = new Animal();
    Animal a2 = new Bird();
    Animal a3 = new Horse();

    a1.move();
    a2.move();
    a3.move();
  }
}
```

Reactive Programming

- Reactive programming is a programming paradigm that is **focused on reacting to changes in data or events**, rather than relying on a fixed flow of control. It involves using streams of data and functional programming concepts to react to changes in the data, rather than using imperative programming constructs.

- The fundamental idea behind reactive programming is to model data and events as streams that flow through the system, and then to use operators to transform and manipulate these streams in real-time.

- Reactive programming is **commonly used in web development**, where it's used to build reactive user interfaces that respond to user input and server-side events. It's also used in mobile and desktop development, gaming, and other areas where real-time data processing is necessary.

- As an overly simplistic idea, consider a statement such as `x = y + z`
  - In traditional imperative programming, the x is assigned the sum of the values of y and z. If the value(s) of y and/or z changes later, the value of x is not affected.
  - In reactive programming, the value of x is automatically updated whenever y and/or z change.

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.

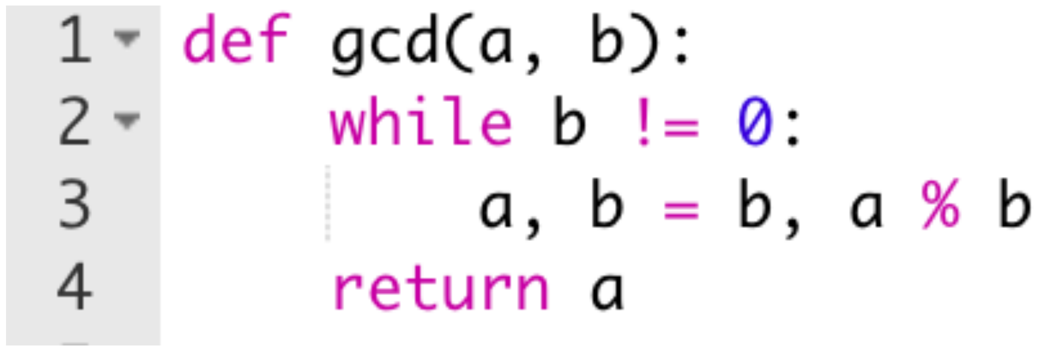
- We will see many examples of functional programming in multiple languages (e.g., Java, Python, OCaML).

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

The Pseudocode

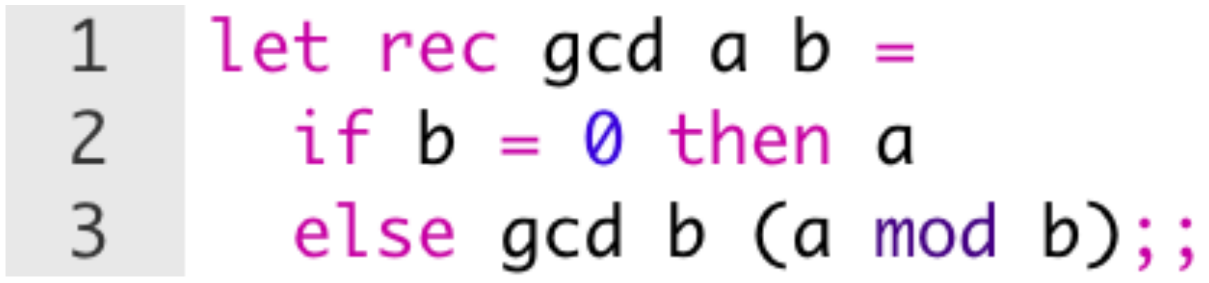
```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

Paradigm 1: Imperative Programming in Python



```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Paradigm 2:
Functional Programming in Ocaml



```
let rec gcd a b =
  if b = 0 then a
  else gcd b (a mod b);;
```

Paradigm 3: Object-Oriented Programming in Java

```java
public class GCD {
    private int a;
    private int b;

    public GCD(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public int compute() {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
}

```

To use:

```java
GCD gcd = new GCD(12, 18);
int result = gcd.compute(); // result is 6
```

Summary for the day

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