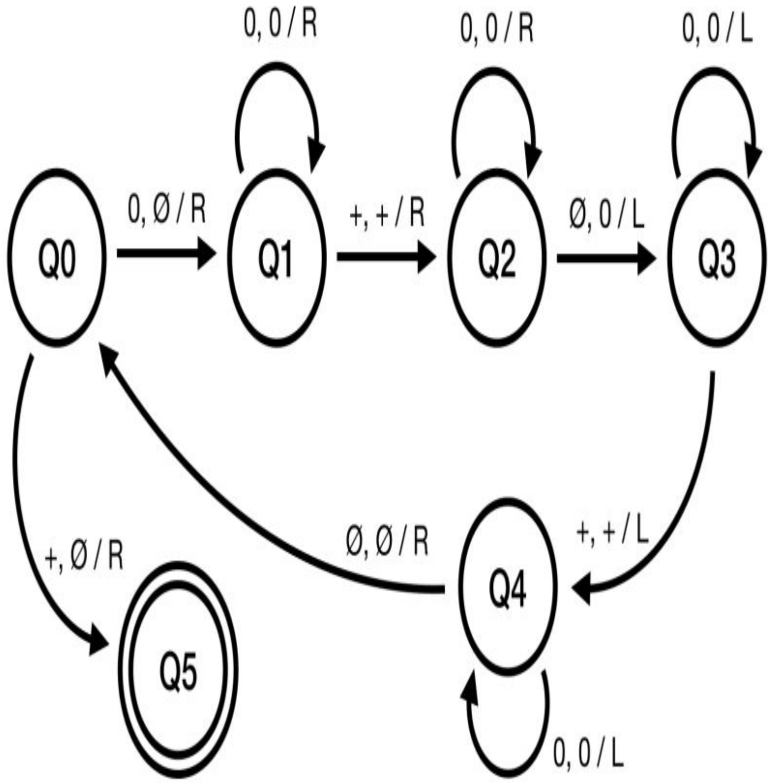
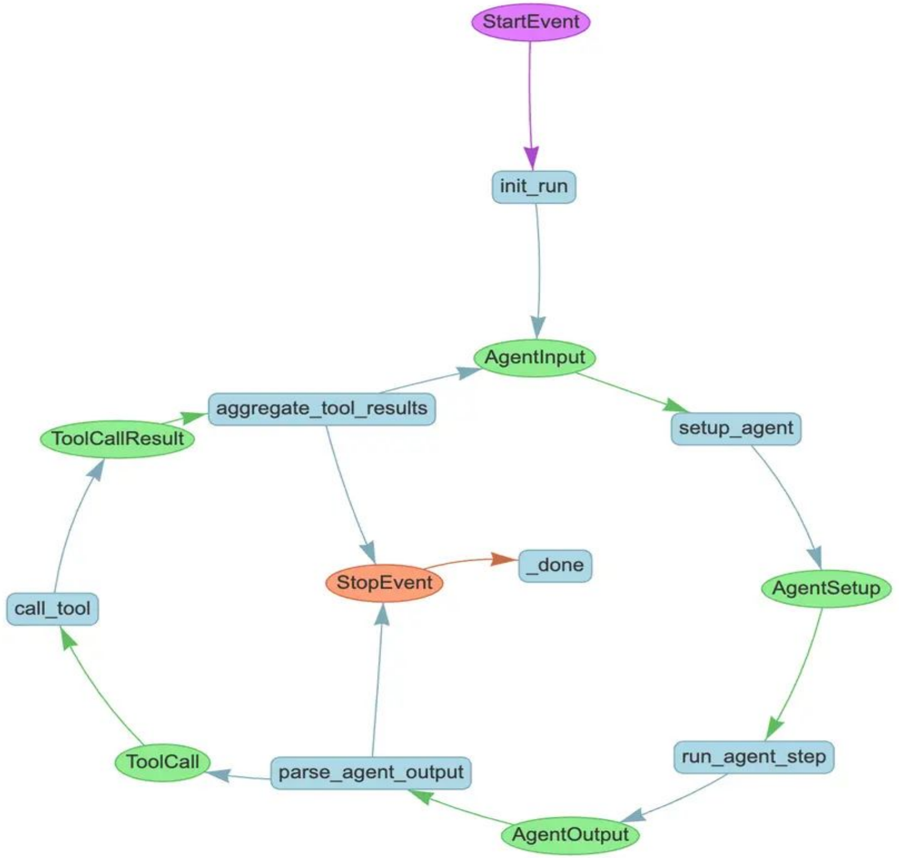
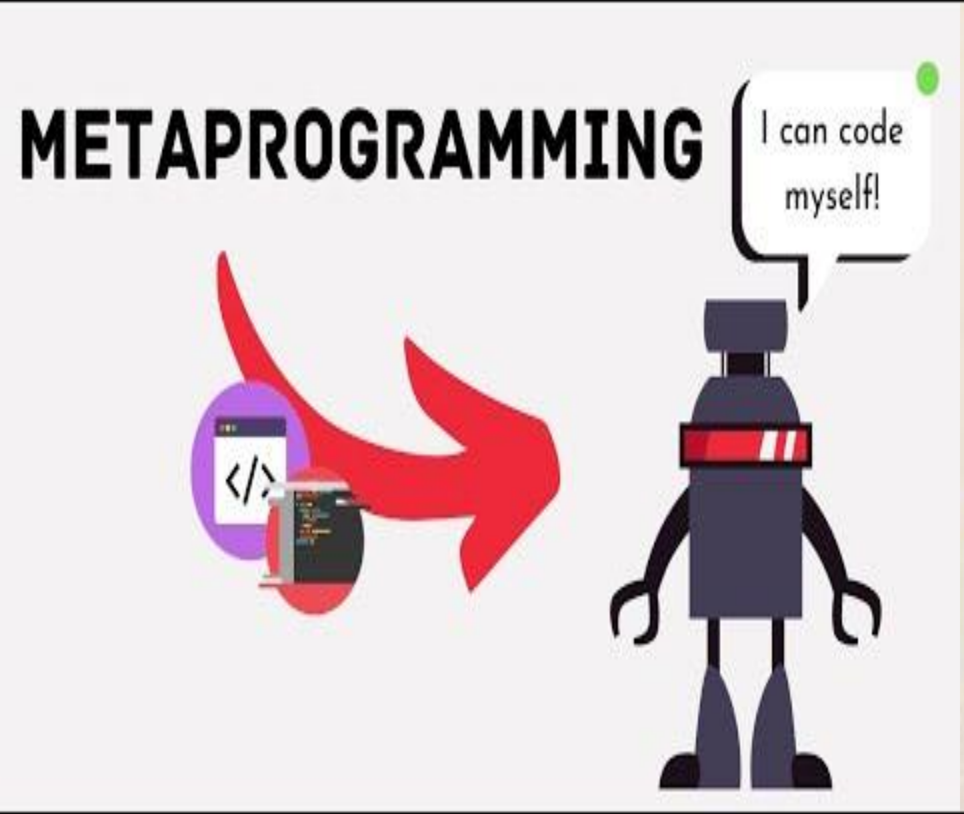
Lesson 3 - MetaProgramming and the JVM

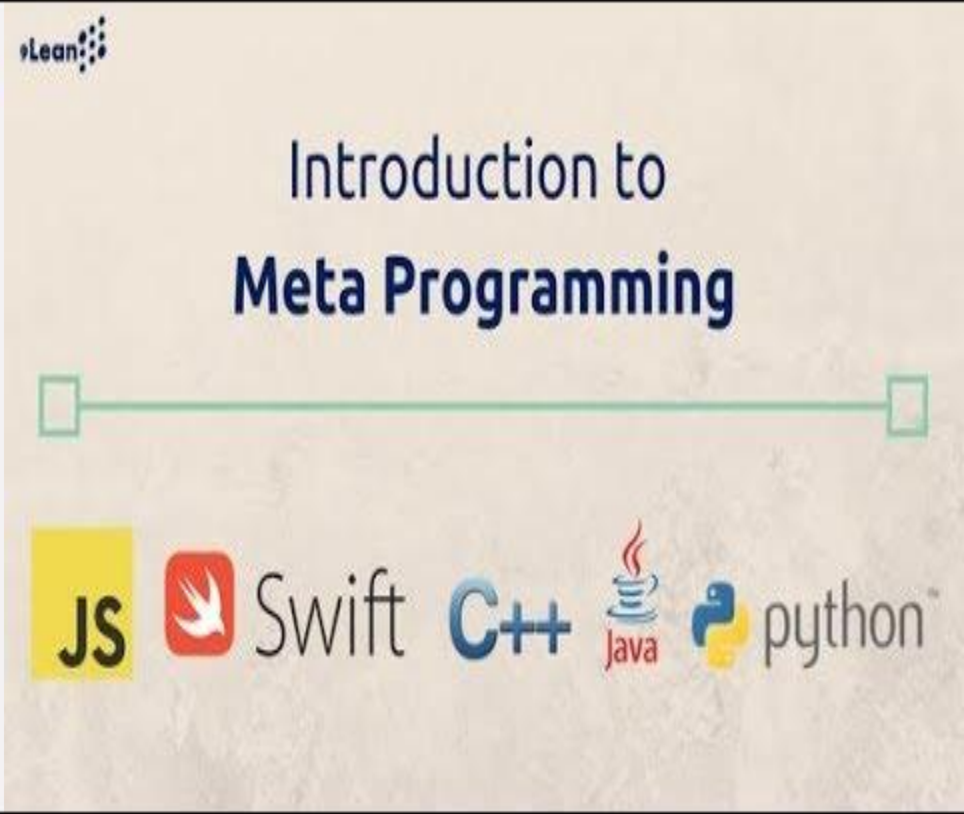
Turing Machine



Agent Workflow

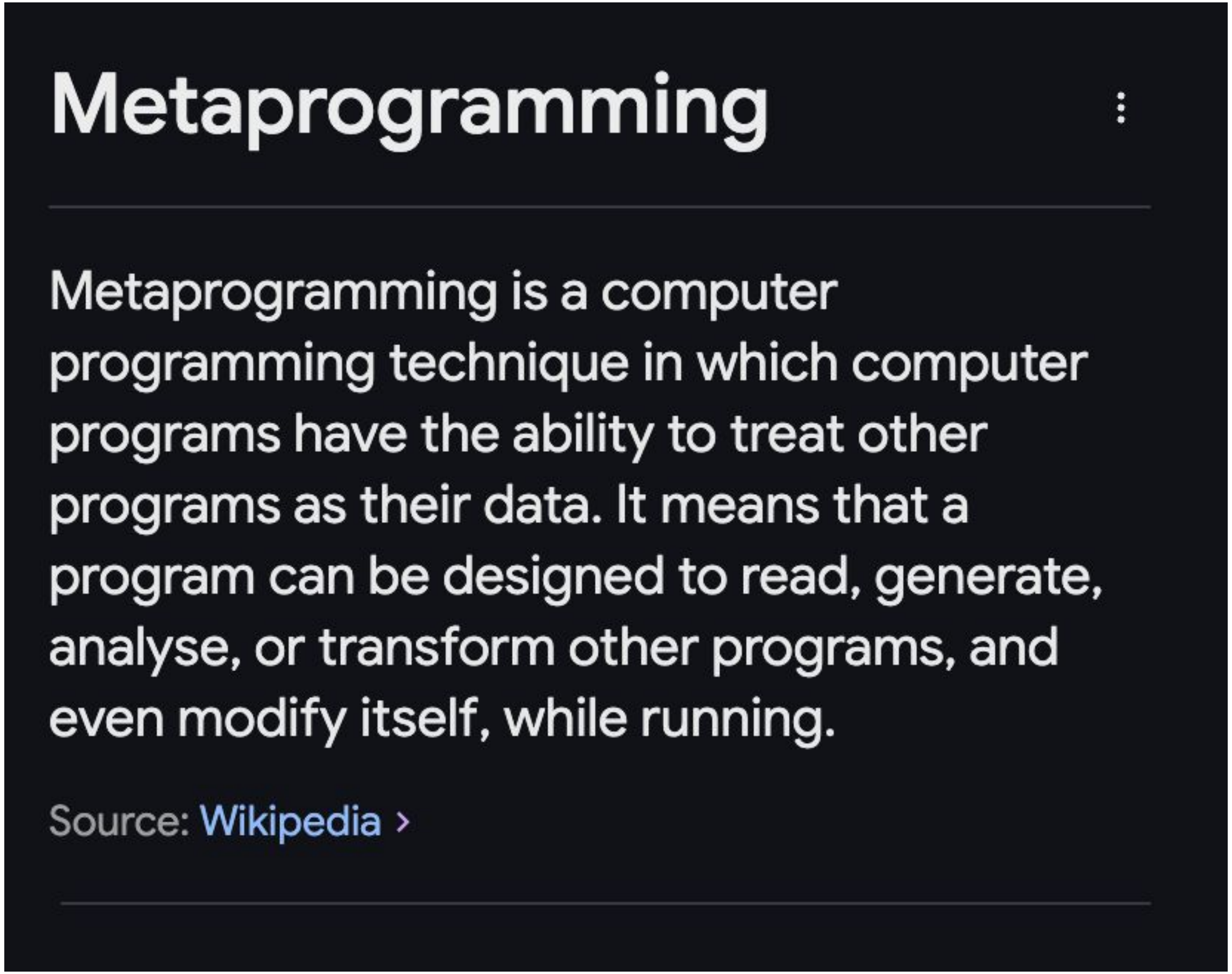





Lean

Introduction to
Meta Programming

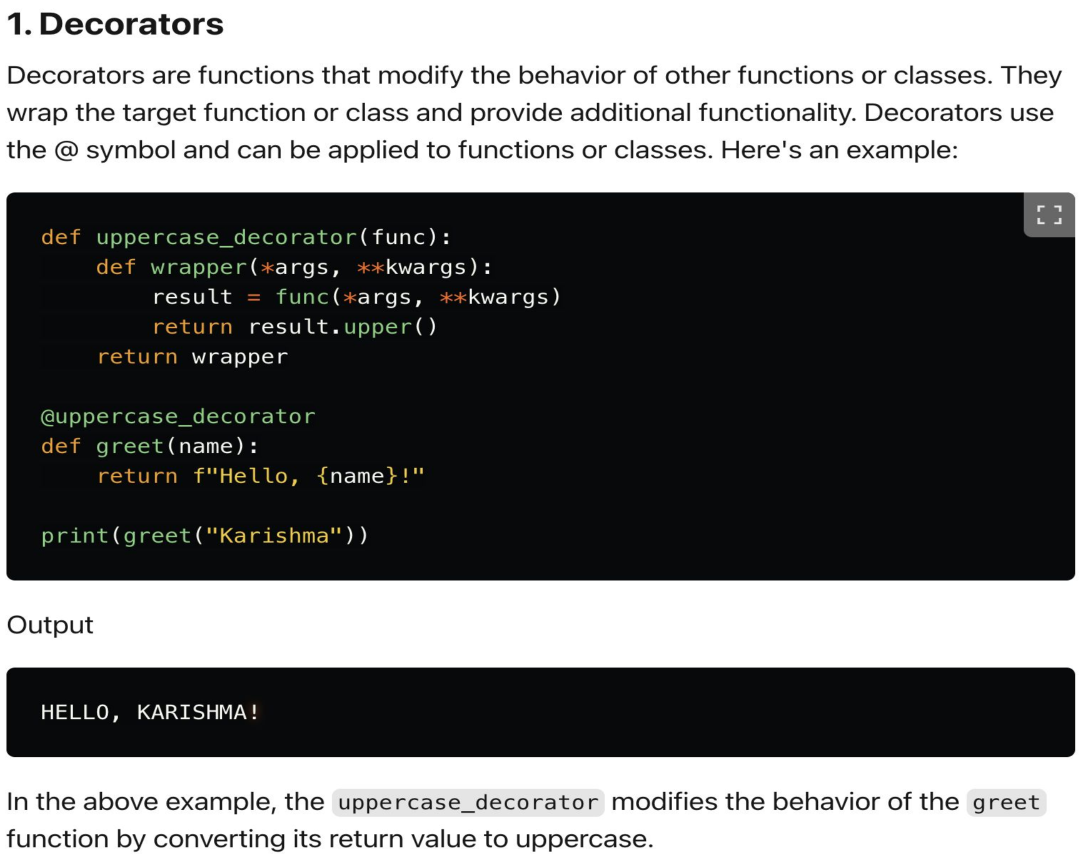
JS
Swift
C++
Java
python



Metaprogramming

Metaprogramming is a computer programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyse, or transform other programs, and even modify itself, while running.

Source: Wikipedia ›



1. Decorators

Decorators are functions that modify the behavior of other functions or classes. They wrap the target function or class and provide additional functionality. Decorators use the @ symbol and can be applied to functions or classes. Here's an example:

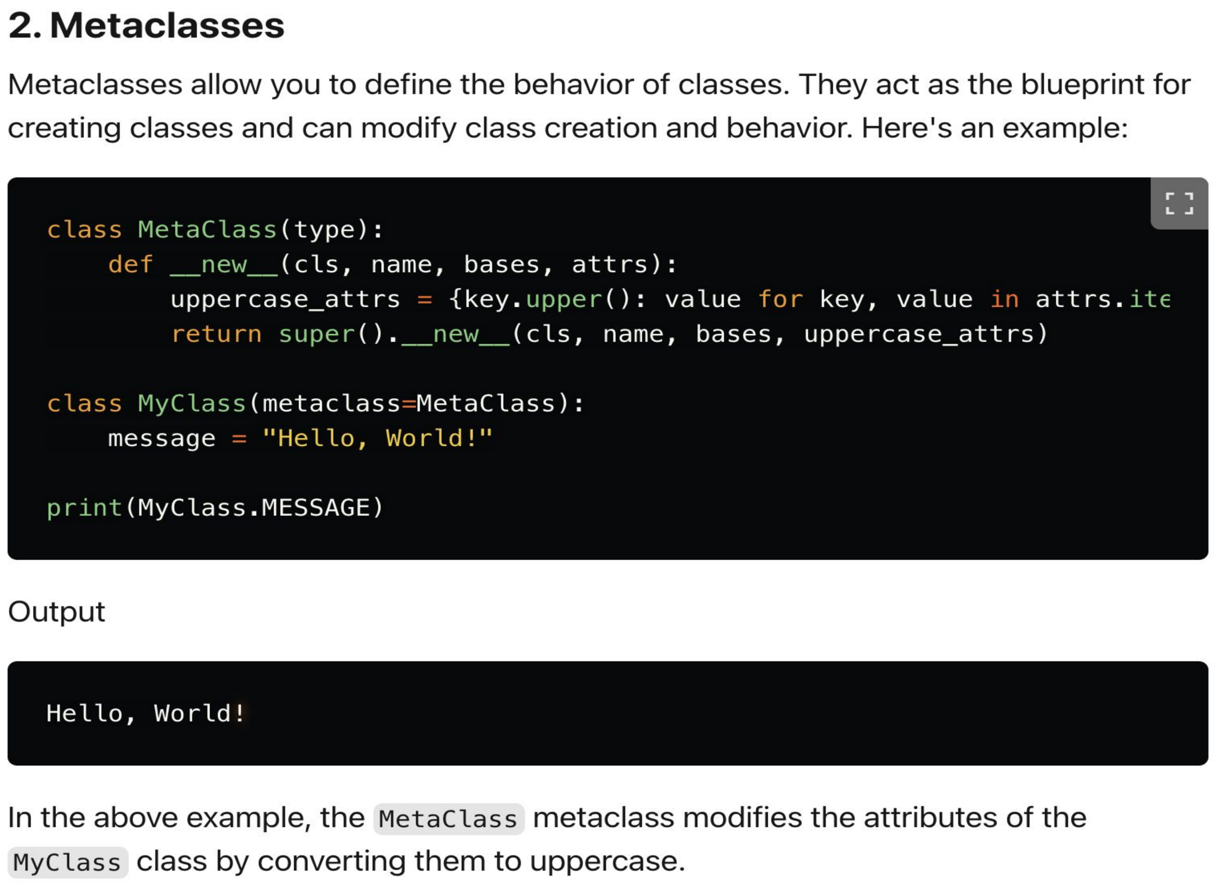
```python
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

@uppercase_decorator
def greet(name):
    return f"Hello, {name}!"

print(greet("Karishma"))
```

Output

```
HELLO, KARISHMA!
```

In the above example, the `uppercase_decorator` modifies the behavior of the `greet` function by converting its return value to uppercase.

https://dev.to/karishmashukla/a-practical-guide-to-metaprogramming-in-python-691



2. Metaclasses

Metaclasses allow you to define the behavior of classes. They act as the blueprint for creating classes and can modify class creation and behavior. Here's an example:

```python
class MetaClass(type):
    def __new__(cls, name, bases, attrs):
        uppercase_attrs = {key.upper(): value for key, value in attrs.ite
        return super().__new__(cls, name, bases, uppercase_attrs)

class MyClass(metaclass=MetaClass):
    message = "Hello, World!"

print(MyClass.MESSAGE)
```

Output

```
Hello, World!
```

In the above example, the `MetaClass` metaclass modifies the attributes of the `MyClass` class by converting them to uppercase.

https://dev.to/karishmashukla/a-practical-guide-to-metaprogramming-in-python-691

3. Function and Class Decorators

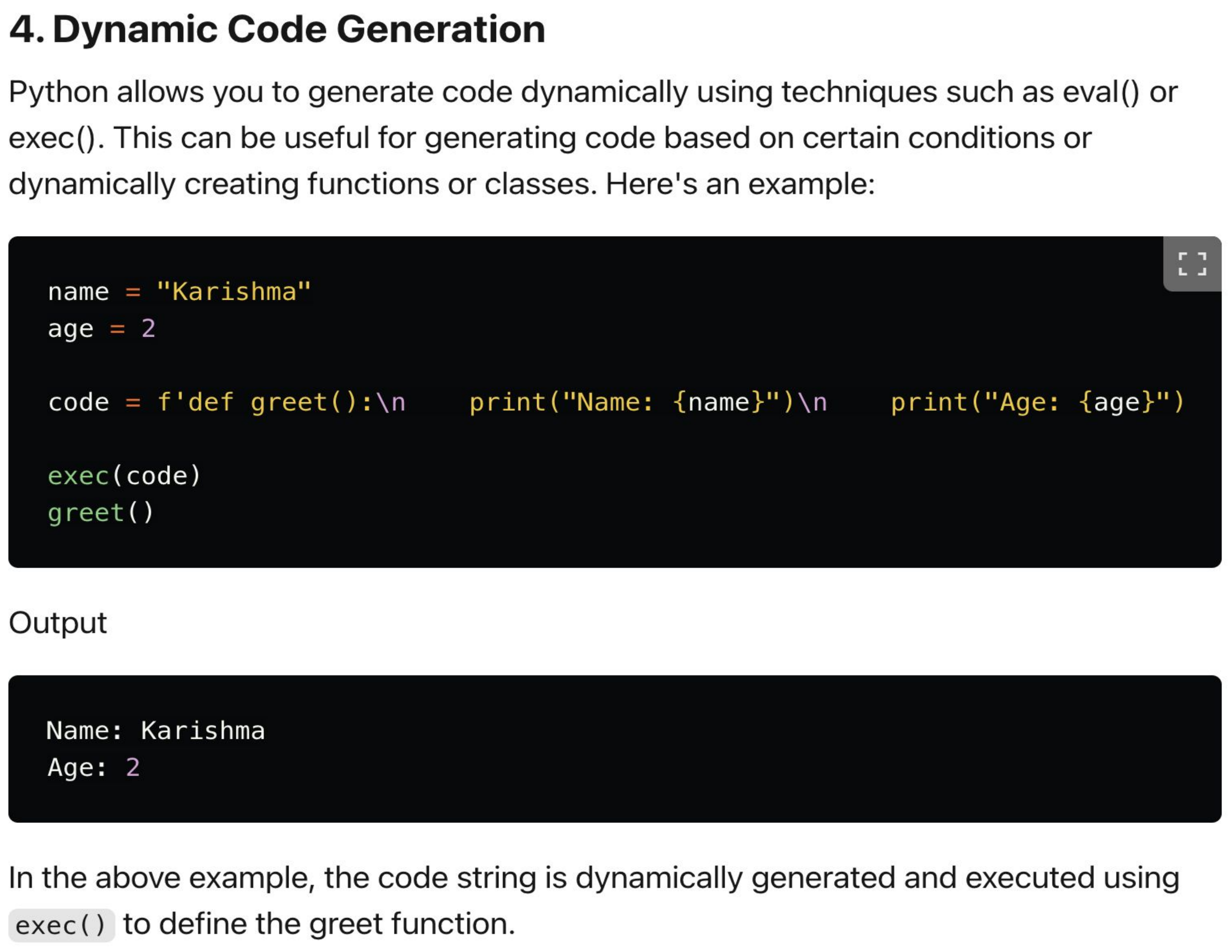
Besides using decorators with the @ symbol, you can also apply decorators using the function or class syntax. Here's an example:

```
class CubeCalculator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):

        # before function
        result = self.function(*args, **kwargs)

        # after function
        return result

# adding class decorator to the function
@CubeCalculator
def get_cube(n):
    print("Input number:", n)
    return n * n * n

print("Cube of number:", get_cube(100))
```

Output

```
Input number: 100
Cube of number: 1000000
```

In the above example, we define a class decorator `CubeCalculator` that wraps the function `get_cube` to perform additional actions before and after its execution, and then applies the decorator to the `get_cube` function. When `get_cube` is called with an



4. Dynamic Code Generation

Python allows you to generate code dynamically using techniques such as eval() or exec(). This can be useful for generating code based on certain conditions or dynamically creating functions or classes. Here's an example:

```
name = "Karishma"
age = 2

code = f'def greet():\n    print("Name: {name}")\n    print("Age: {age}")

exec(code)
greet()
```

Output

```
Name: Karishma
Age: 2
```

In the above example, the code string is dynamically generated and executed using `exec()` to define the greet function.

https://dev.to/karishmashukla/a-practical-guide-to-metaprogramming-in-python-691

The Magic Behind Metaprogramming

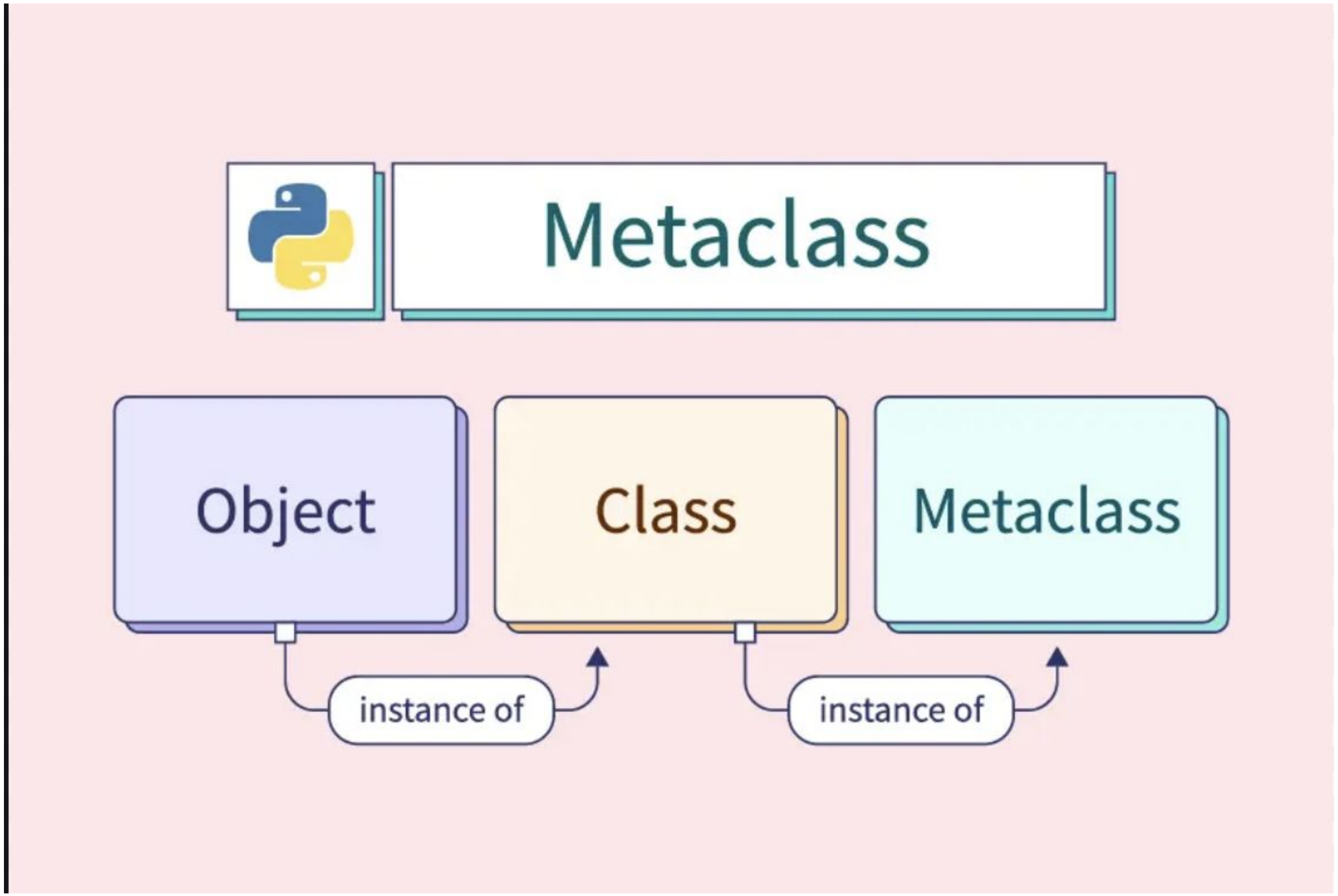
The core idea behind metaprogramming is to increase the flexibility and efficiency of programs by allowing them to be self-aware and capable of modifying their behavior based on certain conditions or inputs.

Benefits of Metaprogramming

- **Code Reusability**: Allows for the creation of generic code that can work with various data types and structures.
- **Reduced Boilerplate**: Helps in automating repetitive code patterns and tasks, leading to cleaner and more maintainable code.
- **Dynamic Behavior**: Enables programs to adapt their behavior dynamically at runtime based on user input or other external conditions.

Challenges and Considerations

While metaprogramming offers significant advantages, it also introduces complexity and can make code harder to understand and debug. It's essential to use it judiciously and ensure that the code remains readable and maintainable.


Metaclass
Object
Class
Metaclass
instance of
instance of

Is LLM prompting MetaProgramming?

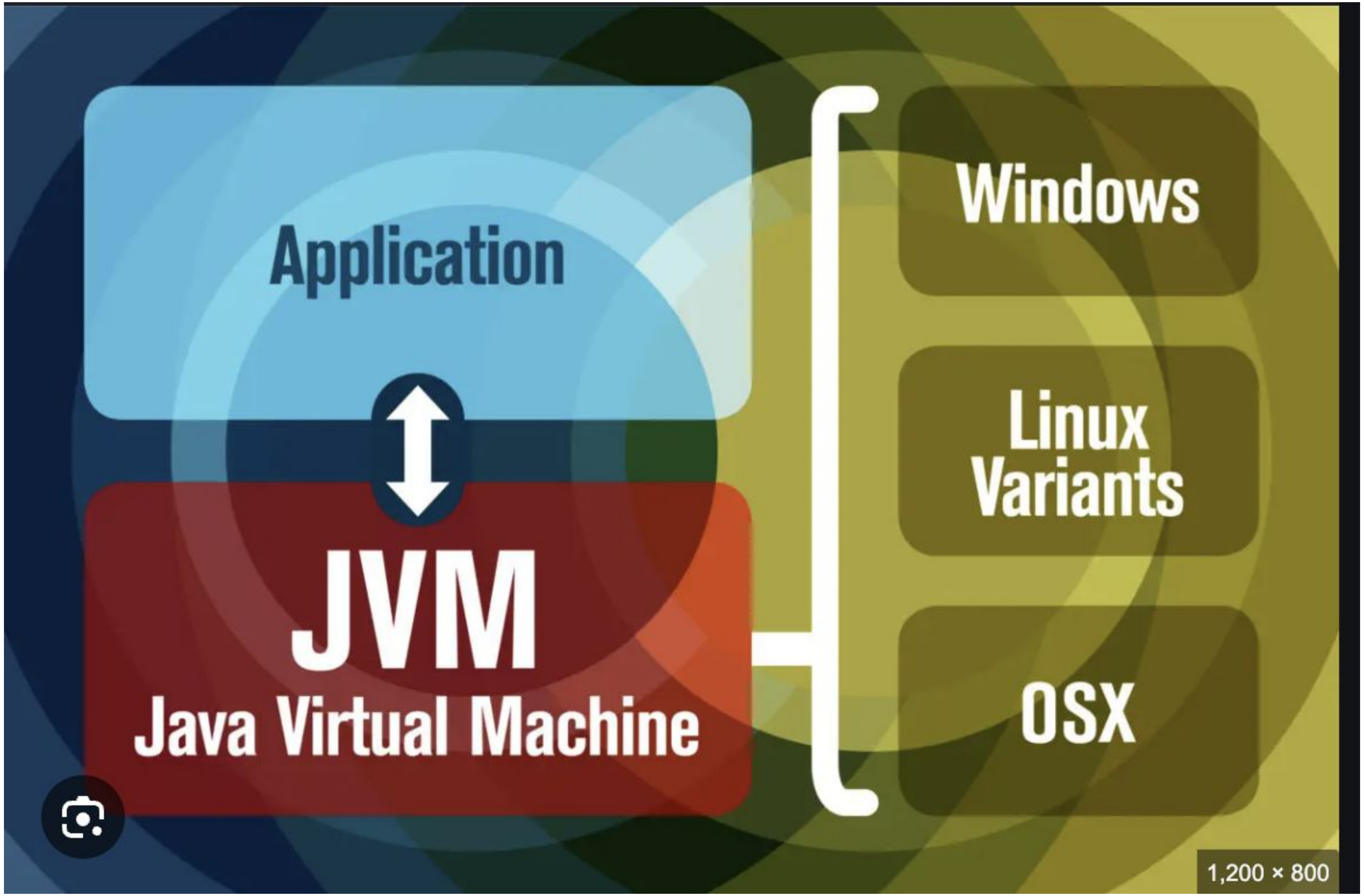
🔷 Is an LLM a "True" Metaprogram?

🔶 YES, in a broader sense

- LLMs **generate** code, which is a key trait of metaprogramming.
- They **analyze and transform** existing programs.
- They can act as **interpreters** or **code assistants**, aiding in metaprogramming tasks.

🔶 NO, if strictly defined

- LLMs don’t modify themselves **at runtime** like a JIT compiler.
- They don’t possess **true reflection** (they don't "know" their own architecture dynamically).
- They lack **direct execution control** over generated code.


Application
JVM
Java Virtual Machine
Windows
Linux Variants
OSX
1,200 × 800

Which JVM language to choose for Enterprise?

BELLSOFT

Java

Java is for large enterprises that want stable and reliable software.

Kotlin

For fast-growing companies establishing a productive programming method, **Kotlin** would be a better choice.

Groovy

Smaller projects, where development speed and fast delivery are key, should pick **Groovy.**

Scala

Companies working with data-intensive applications would find **Scala** ideal for their purposes.

Clojure

If a startup plans a massive greenfield project or a highly concurrent system, **Clojure** is superb.

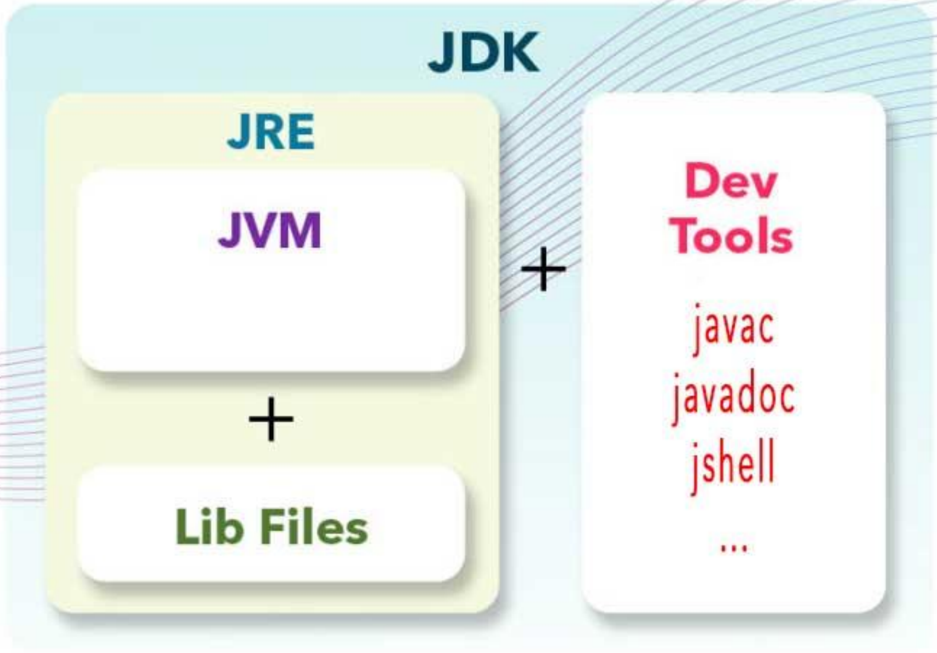
JDK

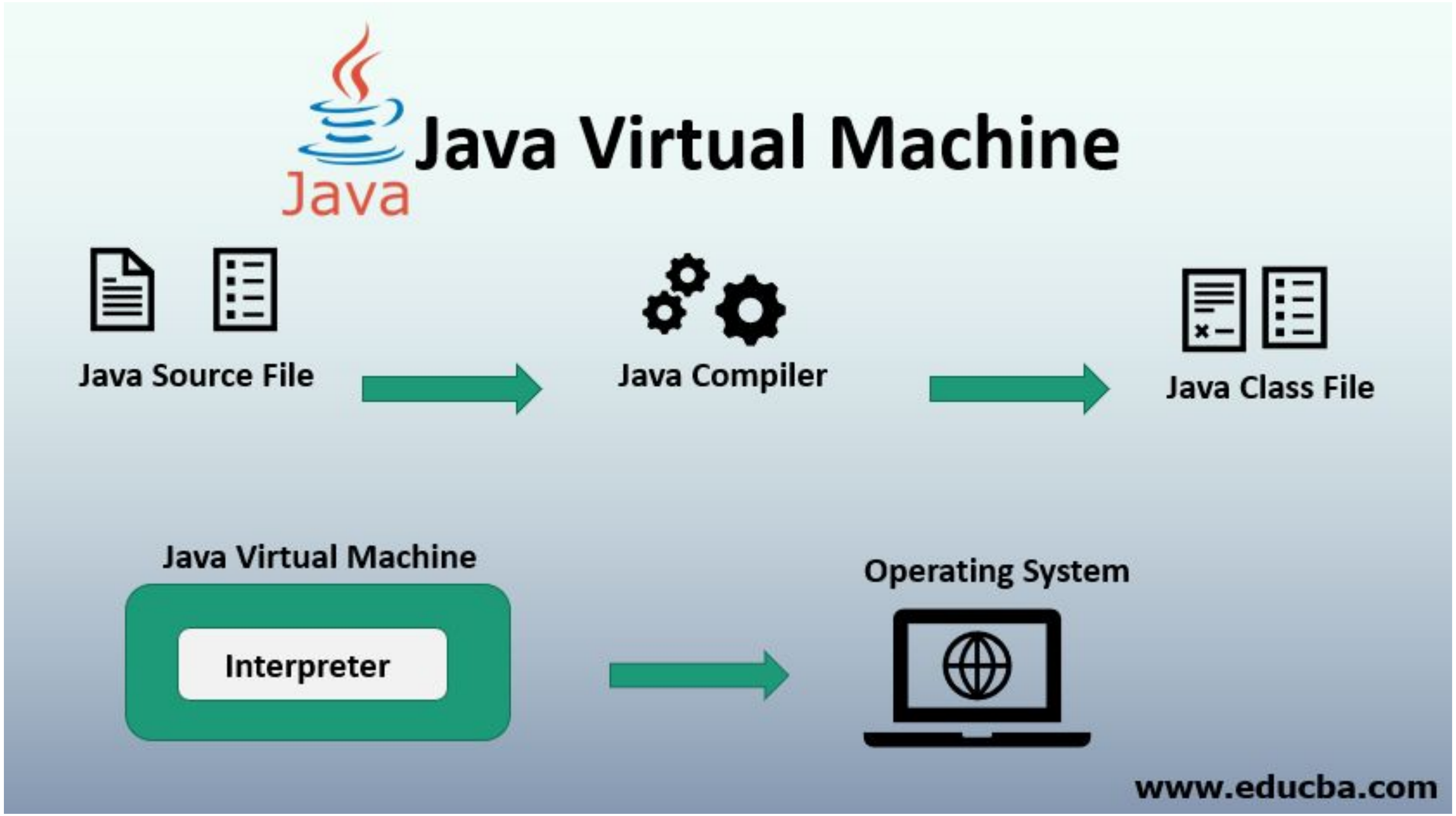
vs

JRE

vs

JVM




Java
Java Virtual Machine
Java Source File
Java Compiler
Java Class File
Java Virtual Machine
Interpreter
Operating System
www.educba.com

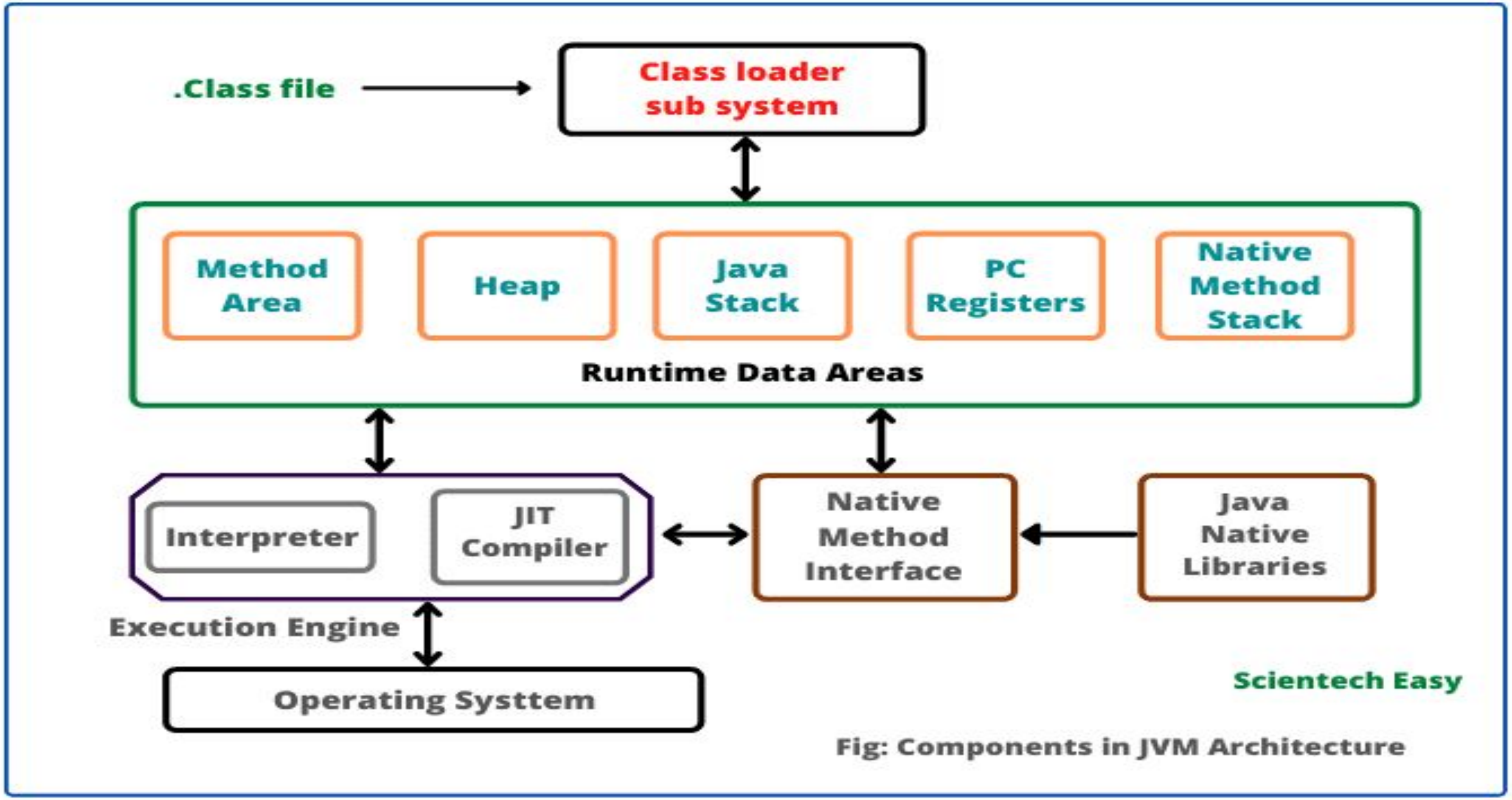


Fig: Components in JVM Architecture