Java Streams and Functional Programming

What is Functional Programming

- Functional Programming in Java means **writing code by using “functions” like building blocks**, and focusing on **what to do**, instead of **how to do it step-by-step**.

```
irror_mod = modifier_ob.
 mirror object to mirror
irror_mod.mirror_object
peration == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
 operation == "MIRROR_Y"
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
 operation == "MIRROR_Z"
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

election at the end -add
_ob.select= 1
er_ob.select=1
ntext.scene.objects.acti
("Selected" + str(modifie
irror_ob.select = 0
 bpy.context.selected_ob
ata.objects[one.name].se

int("please select exactl

- OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
rror X"

ontext):
ext.active_object is not
```

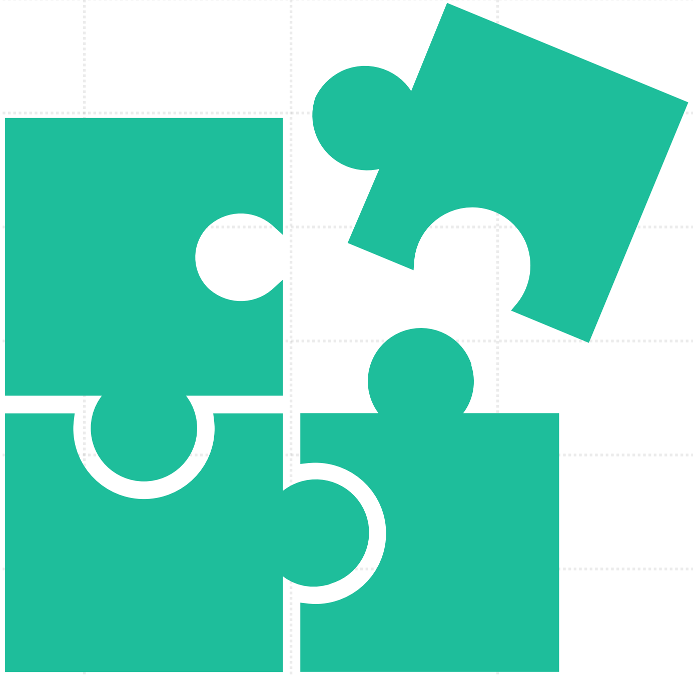
Simple idea

- **Normal Java style (step-by-step):**
  You tell Java every small step:
  - Take the list
  - Loop through it
  - Check each item
  - Store the result
- **Functional style:**
  You just say:
  - From this list, keep only even numbers
  - Then double them
  - Then print them

So, it feels more like giving **instructions**, not writing long loops.

Why is it called "Functional"?

- Because it uses **functions** (small pieces of logic) like:
- filtering (remove unwanted items)
- mapping (convert/change items)
- reducing (combine items)



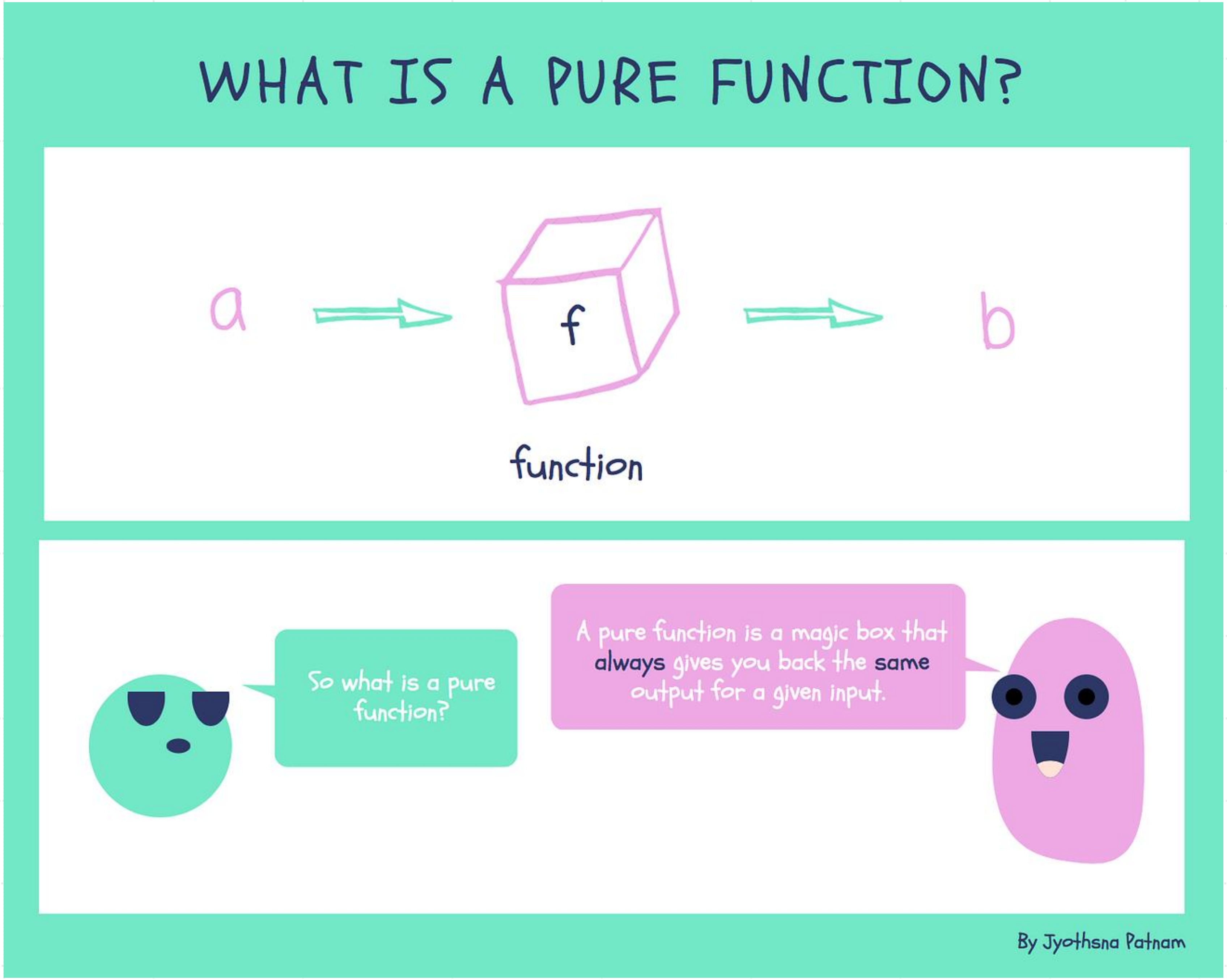
Core Concepts of Java Functional Programming

- **Functions are “First-Class Citizens”**
  - Means: **Functions can be treated like values**
    - You can store them in variables
    - Pass them as arguments
    - Return them from other functions
  - Like passing a “rule” or “instruction” to another method.

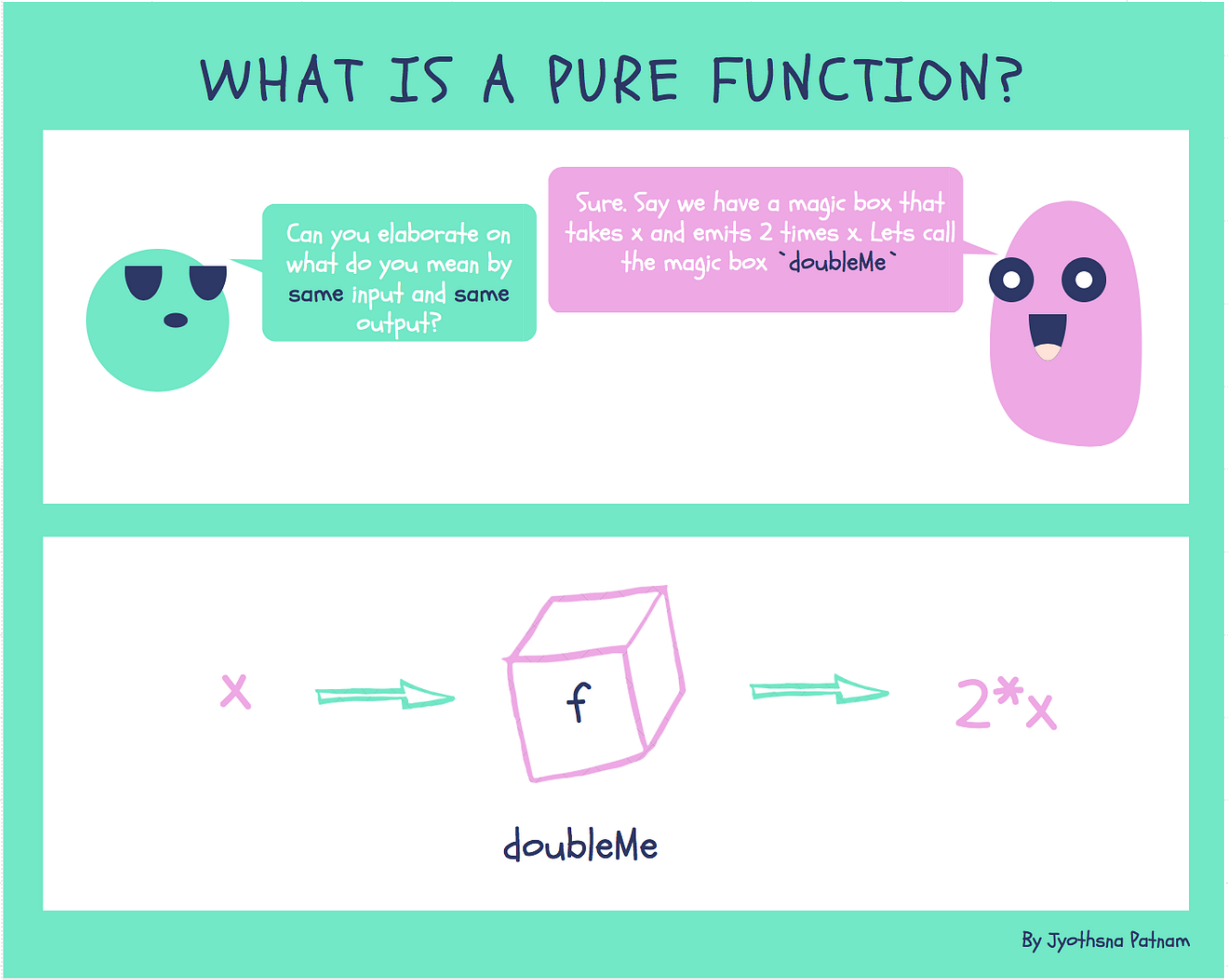
Pure Functions

- A **pure function** always gives the **same output for the same input**.
- No dependence on outside things
- No changing global variables
- No printing inside the function

Pure Functions

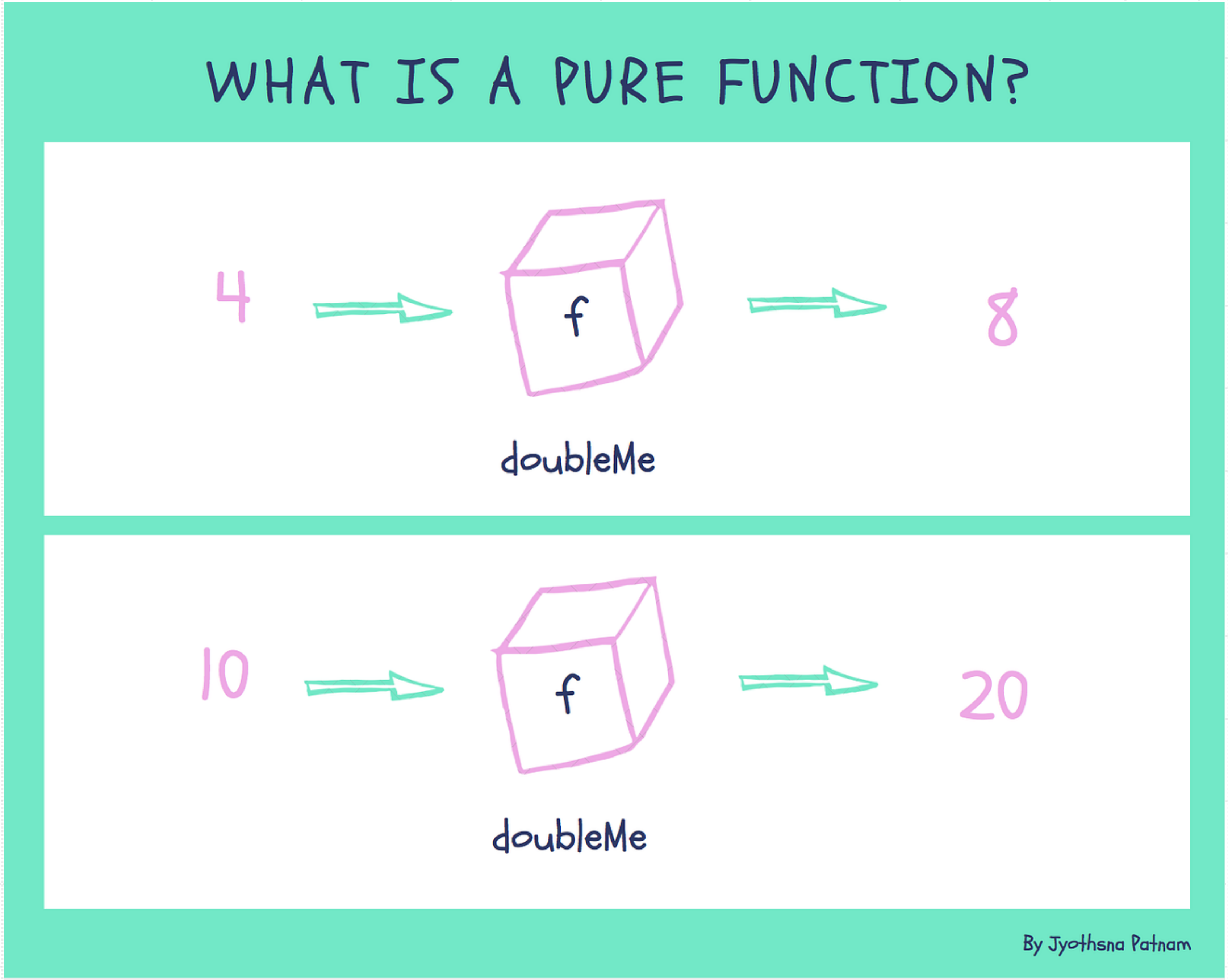

WHAT IS A PURE FUNCTION?
a
f
b
function
So what is a pure function?
A pure function is a magic box that always gives you back the same output for a given input.
By Jyothsna Patnam

Pure Functions


WHAT IS A PURE FUNCTION?
Can you elaborate on what do you mean by same input and same output?
Sure. Say we have a magic box that takes x and emits 2 times x. Lets call the magic box `doubleMe`
x
f
2*x
doubleMe

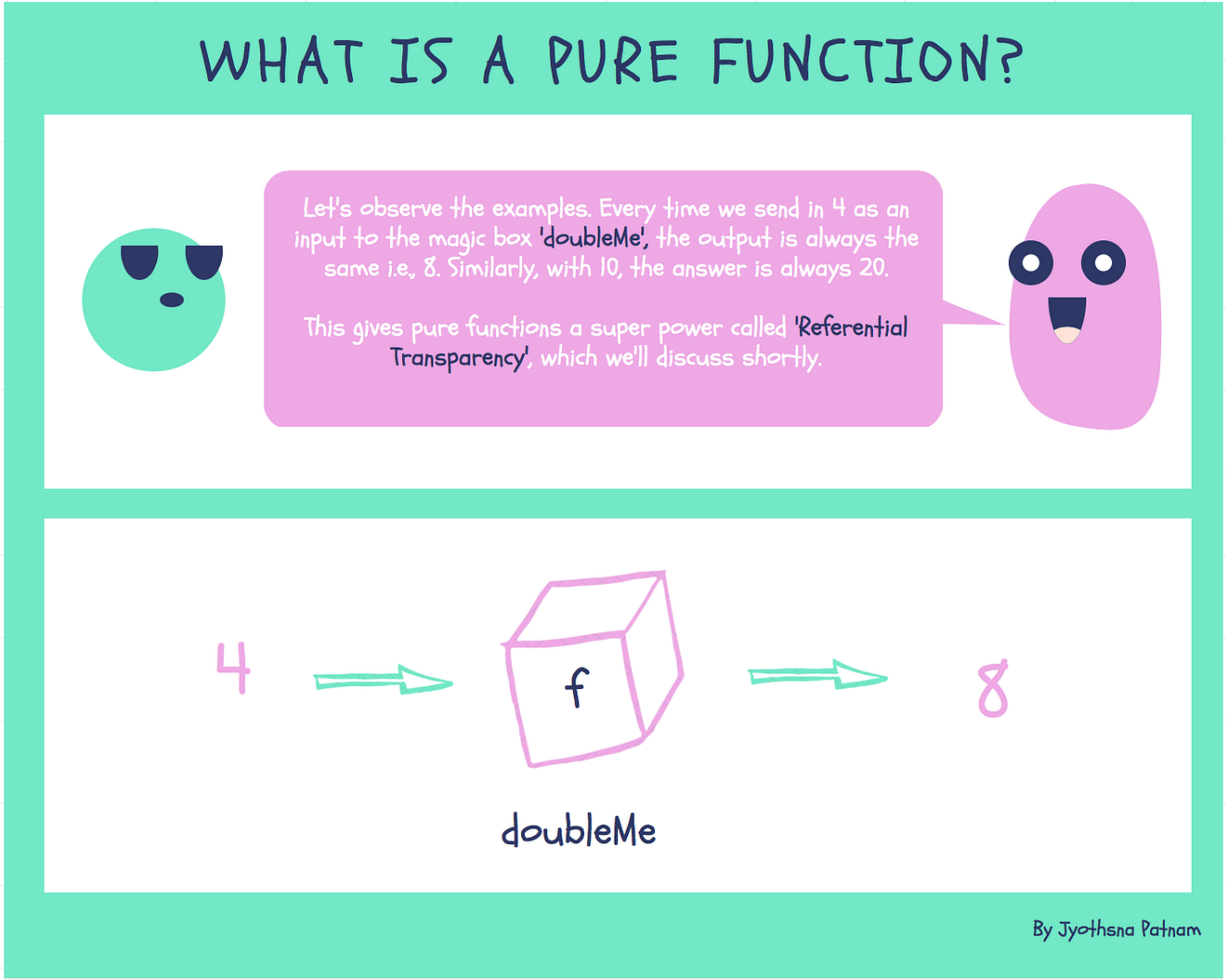
By Jyothsna Patnam

Pure Functions



By Jyothsna Patnam

Pure Functions



WHAT IS A PURE FUNCTION?

Let's observe the examples. Every time we send in 4 as an input to the magic box 'doubleMe', the output is always the same i.e., 8. Similarly, with 10, the answer is always 20.

This gives pure functions a super power called 'Referential Transparency', which we'll discuss shortly.

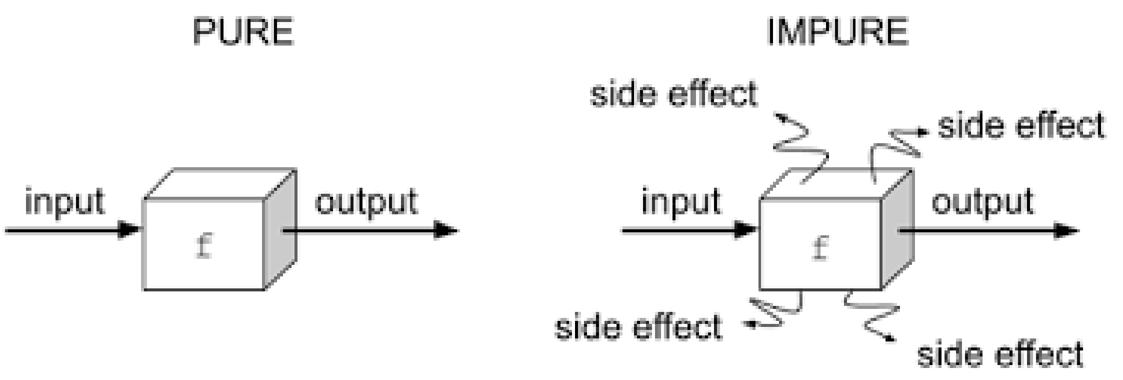
By Jyothsna Patnam

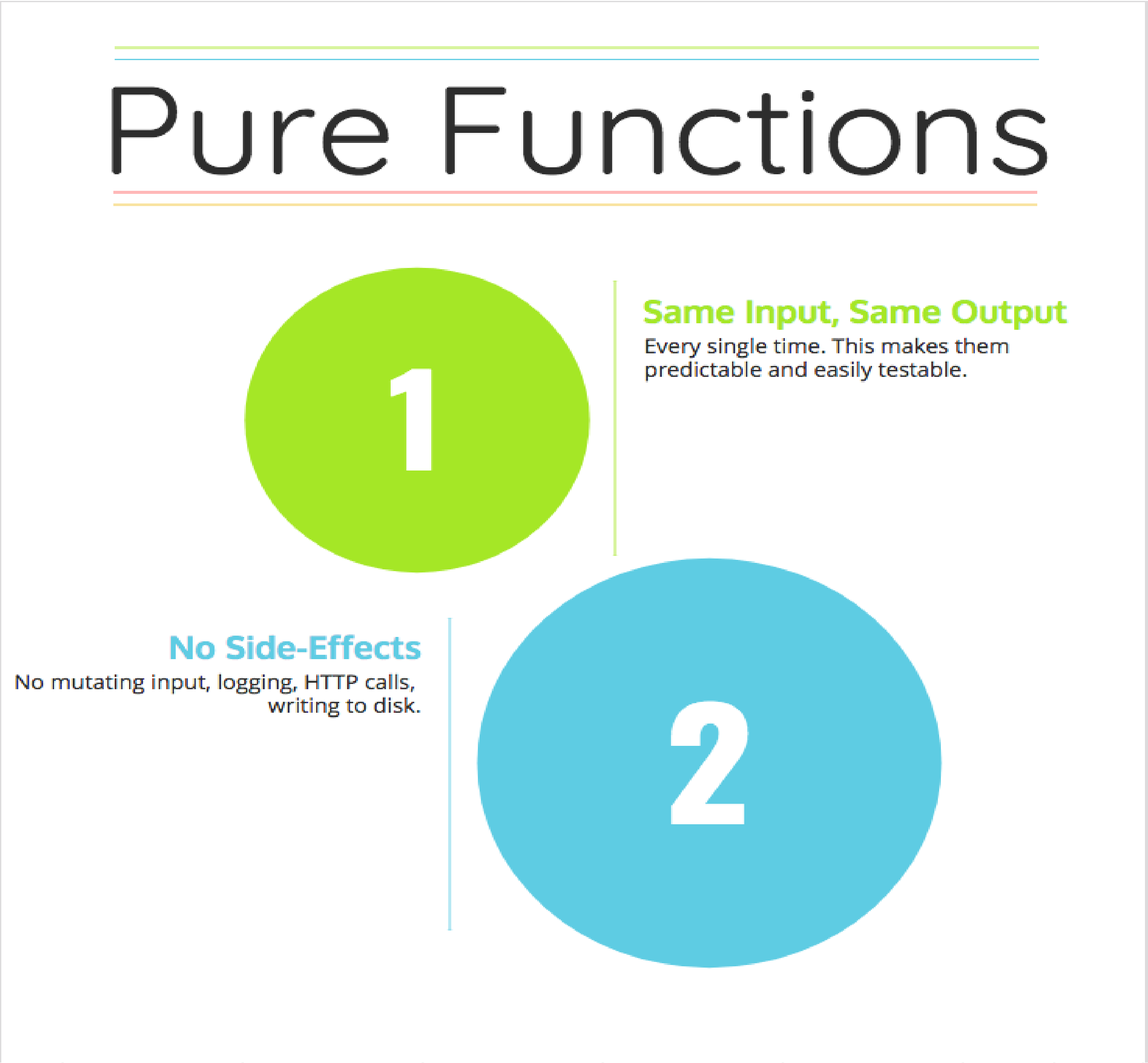
No Side Effects

- Side effect means the function **changes something outside itself**, like:
  - modifying a global variable
  - changing an object directly
  - printing to console
  - writing to a file

  Functional programming tries to **avoid side effects**.

Pure Functions





Pure Functions

1

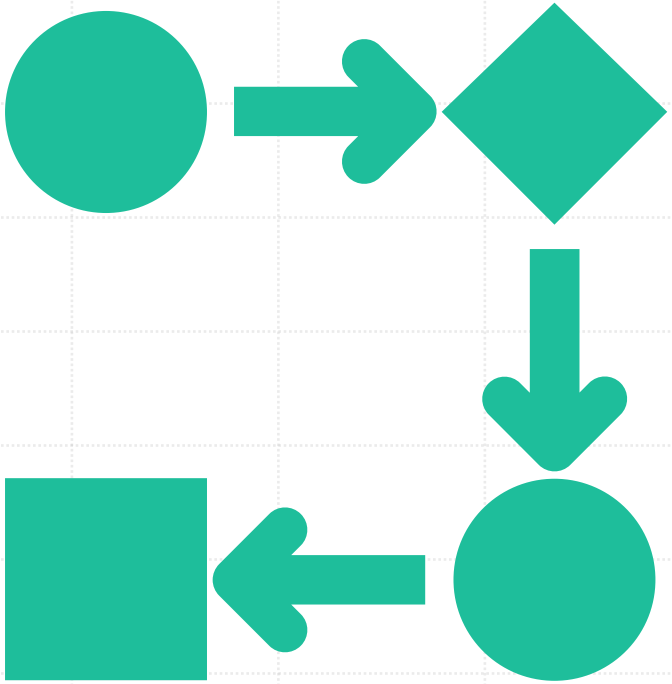
Same Input, Same Output

Every single time. This makes them predictable and easily testable.

2

No Side-Effects

No mutating input, logging, HTTP calls, writing to disk.



Immutability

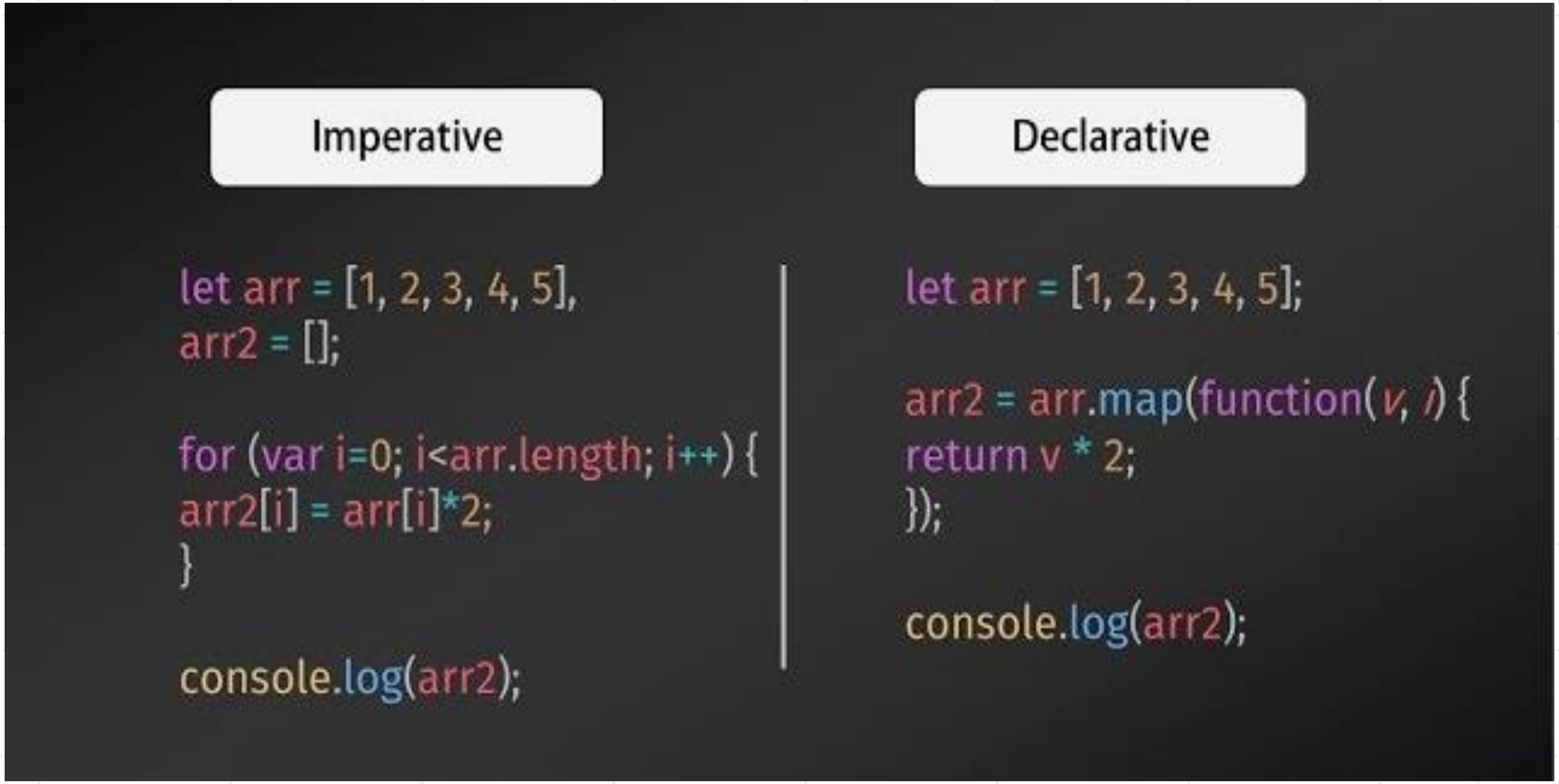
- Means: **Don’t change data once created**

  Instead of modifying existing data, create **new data**.

- Example idea:
  - Instead of Change the old list
  - Create a new updated list

Declarative Style

- Focuses on *what* to do rather than *how* to do it, which leads to more readable and concise code.



Imperative

```
let arr = [1, 2, 3, 4, 5],
arr2 = [];

for (var i=0; i<arr.length; i++) {
arr2[i] = arr[i]*2;
}

console.log(arr2);
```

Declarative

```
let arr = [1, 2, 3, 4, 5];

arr2 = arr.map(function(v, i) {
return v * 2;
});

console.log(arr2);
```

Why do people like Functional Programming?

Less code

- Instead of writing long loops and if-statements, you can write short and clear code using filter, map, etc.

Cleaner and easier to read

Functional code looks like a **set of instructions**:

- “filter this → change this → print this”
  So beginners and teams can understand faster.

No long loops

Easy to modify

- If you want to add a new condition later, you can easily add another step.

Useful for modern Java development

Why do people like Functional Programming?

- We need functional programming because it helps us write cleaner, shorter, safer, and easier-to-maintain code, especially when working with collections and modern Java applications.

```
irror_mod = modifier_ob.
set mirror object to mirror
mirror_mod.mirror_object
operation == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
 operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
 operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

selection at the end -add
 _ob.select= 1
ier_ob.select=1
ntext.scene.objects.active
("Selected" + str(modifier
irror_ob.select = 0
 bpy.context.selected_obj
data.objects[one.name].se

print("please select exactly

- OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
rror X"

context):
ext.active_object is not
```

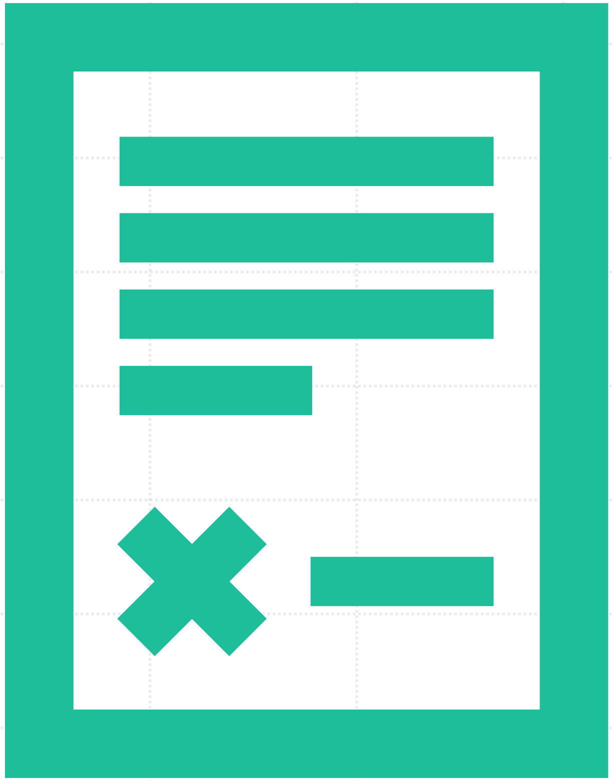
Why Do We Need Functional Programming?

- Functional Programming (FP) is a programming style where we solve problems using **functions**, and we focus more on:
- **“What to do”** (result)
  instead of
  “How to do it step-by-step” (loops + manual logic)
- In Java, Functional Programming became popular mainly after **Java 8** because Java introduced:
  - **Lambda expressions (->)**
  - **Streams (stream())**
  - **Functional interfaces (Predicate, Function, Consumer, Supplier)**

What are functional Interfaces

- **Functional Interfaces are the "bridge" that connects Functional Programming to Java.**
  Because Java is an **Object-Oriented language**, it needed a way to represent "functions" inside Java.
  That's exactly what **Functional Interfaces** do

- A **Functional Interface** is an interface that has **only ONE abstract method**.
- Because it has only one method, Java can treat it like a **function type**..



What is an Interface in Java?

- An **interface** is like a **rule book / contract**.
- It tells a class:
- **“If you want to use me, you must follow these rules and implement these methods.”**

Why do we need Interfaces?

- Interfaces are used to achieve:
  - **Abstraction (Hide details, show only required features)**
    - You only show *what* a thing can do, not *how* it does it.
    - Example:
      - A car has **start()**
      - You don’t care how the engine works internally
  - **Loose Coupling (Flexible code)**
    - Your code becomes flexible because it depends on an interface, not a specific class.
    - So, you can change the class later without breaking the whole program.

Why do we need Interfaces?

- **Multiple Inheritance (Java allows it through interfaces)**
  - Java does NOT allow multiple inheritance using classes, but it allows it using interfaces.
  - A class can implement **many interfaces**.
- **Standardization**
  - Interfaces create a standard way of doing things.
  - Example:
    - Runnable
    - Comparable
    - List
    - Map

Key Benefits

- **Interfaces help in:**
  - ✓ Writing flexible code
  - ✓ Achieving abstraction
  - ✓ Supporting polymorphism
  - ✓ Creating reusable systems
  - ✓ Allowing multiple inheritance
  - ✓ Making code easy to test and maintain
- An **interface** in Java is a blueprint/contract that defines **what methods a class must implement**. It helps in **abstraction, flexibility, and standardization**, and supports **multiple inheritance**.

What can an Interface contain?

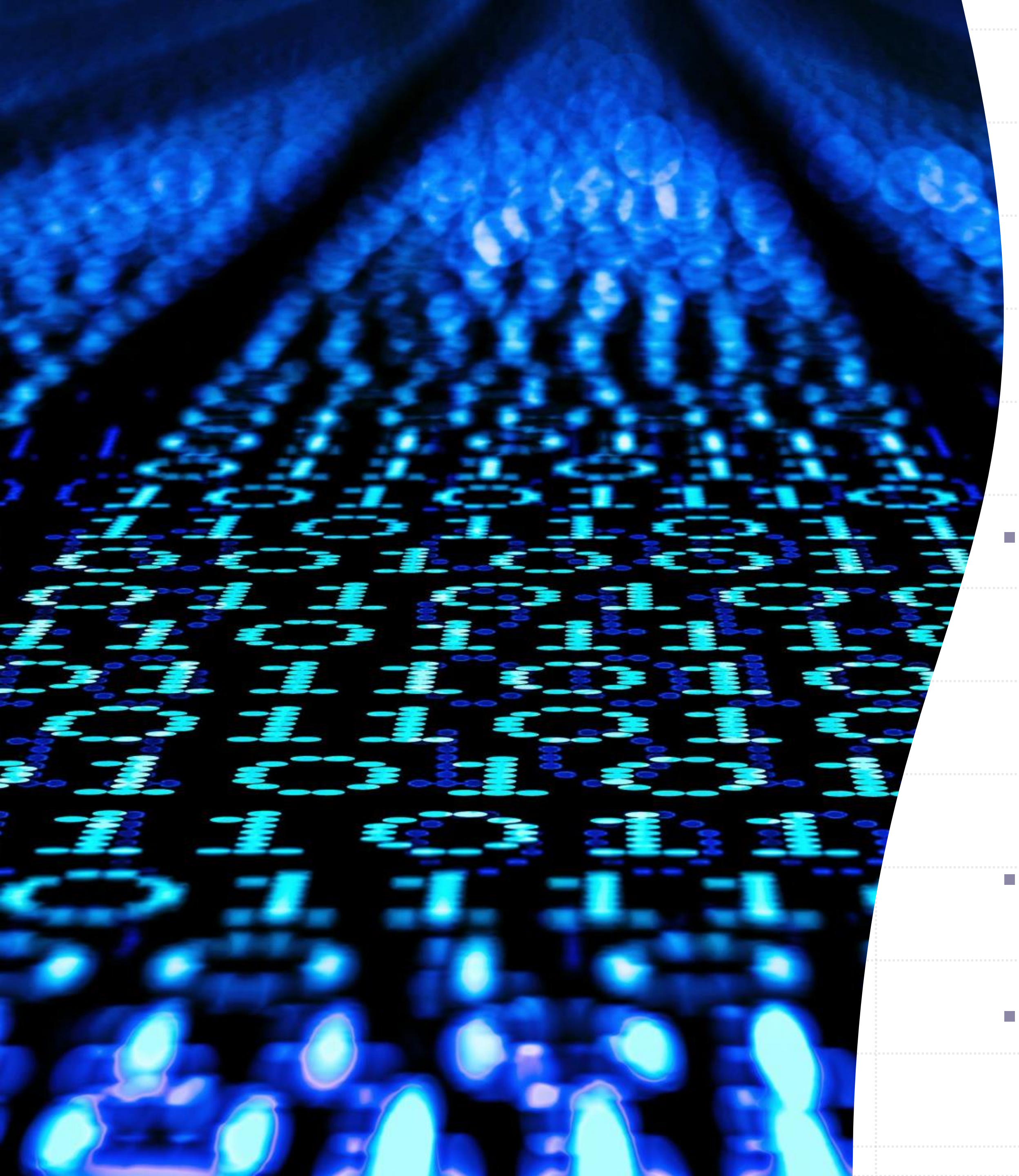
- 1. Abstract methods (main purpose)
- 2. Default methods (Java 8+)
- 3. Static methods
- 4. Variables (they are always public static final)

Types of Interfaces

Normal Interface: Contains multiple methods to be implemented by a class.

Functional Interface: Contains exactly **one abstract method**. These are the foundation for Lambda Expressions in Java.

Marker Interface: An interface with no methods or fields
(e.g., Serializable or Cloneable). It simply "marks" a class as having a certain property



Why Functional Interfaces are important in Functional Programming?

- Functional Programming needs:
  - functions that can be passed around
  - functions that can be stored in variables
  - functions that can be given to methods (like filter, map)
- But in Java, you can’t directly pass “functions” like in Python/JavaScript.
- So, Java uses:
  Functional Interface + Lambda Expression

How Lambda fits into Functional Interface

01

A **lambda expression** is basically an **implementation of the functional interface's single method**.

02

A **lambda** (lambda expression) in Java is a **short way to write a function** without creating a full method.

03

It is mainly used to write **small logic quickly**, especially with **functional interfaces**.

04

Lambda works only with **Functional Interfaces** (interfaces having only **one abstract method**).

Method Reference

- A **method reference** in Java is a **shorter way to write a lambda expression** when you are just calling an existing method.
- It uses :: operator.
- Example (Lambda vs Method Reference)
  - **Lambda:**
    - list.forEach(x -> System.out.println(x));
  - **Method Reference:**
    - list.forEach(System.out::println);

Streams

- **Java Streams** are used to process collections/arrays in a clean **functional style** (filter, map, sort, reduce) without writing loops.
- **Key Characteristics:**
  - **No Storage:** Streams do not store data; they carry values from a source (like a List) through a pipeline.
  - **Non-Mutating:** Operations on a stream do not modify the original collection.
- Basic Stream Flow:
  - Source → Intermediate operations → Terminal operation

Streams

- list.stream() // source
- .filter(...) // intermediate
- .map(...) // intermediate
- .collect(...) // terminal
- **Common Operations:**
  - **Intermediate:** filter() (removes elements), map() (transforms elements), sorted(), and distinct().
  - **Terminal:** collect() (turns stream back into a List/Set), forEach(), reduce(), and count().

Terminal operations

Terminal operations in Java Streams are the operations that **end the stream** and **produce a result**.

After a terminal operation, the stream is **closed** (you cannot reuse it).

Common Terminal Operations:

- **forEach()** → perform action on each element (print, update)
- **collect()** → convert stream to **List / Set / Map**
- **reduce()** → combine elements into **single value** (sum, max, etc.)
- **count()** → returns total number of elements
- **min() / max()** → returns smallest / largest element
- **anyMatch() / allMatch() / noneMatch()** → returns boolean (true/false)
- **findFirst() / findAny()** → returns an element (Optional)
- **toArray()** → converts stream into an array

Summary

- **Functional Interfaces make Functional Programming possible in Java.**
  They provide a **standard way to represent functions** so that Java can use:
  - **lambdas**
  - **streams**
  - **method references**
- Without functional interfaces, Java could not pass behavior (logic) as data.