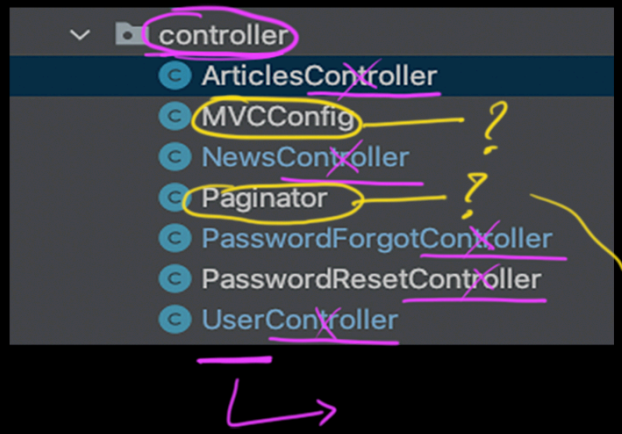
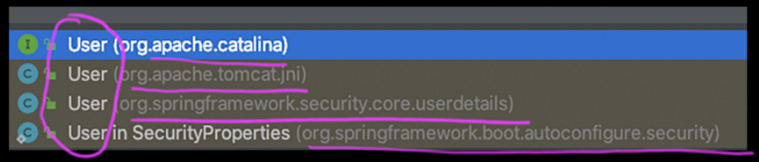
Clean Code / Clean Architecture

1. file location
   - proper folder name — extra inform. about domain
   - having many folders for different purposes
2. file size ≈ 200 lines
3. file name — make it short — no more than 2 words





put them to different folders

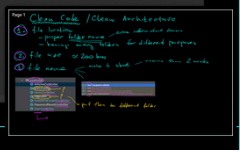
Page 2

```
List<Integer> getPaginator(int pageNumber, int totalPages){
    List<Integer> head = (pageNumber > 4)? new ArrayList<>(Arrays.asList(1,-1)) :new ArrayList<>(Arrays.asList(1,2,3));
    List<Integer> tail = (pageNumber < totalPages - 3)?new ArrayList<>(Arrays.asList(-1,totalPages)):new ArrayList<>(Arrays.asList(totalPages-2,totalPages-1,totalP
    List<Integer> bodyBefore = (pageNumber > 4 && pageNumber < totalPages - 1)?new ArrayList<>(Arrays.asList(pageNumber - 2, pageNumber - 1)):new ArrayList<>();
    List<Integer> bodyAfter = (pageNumber > 2 && pageNumber < totalPages - 3)?new ArrayList<>(Arrays.asList(pageNumber + 1, pageNumber + 2)):new ArrayList<>();


    Optional.ofNullable(bodyBefore).ifPresent(head::addAll);
    head.addAll((pageNumber > 3 && pageNumber < totalPages - 2)?new ArrayList<>(Arrays.asList(pageNumber)):new ArrayList<>());
    Optional.ofNullable(bodyAfter).ifPresent(head::addAll);
    Optional.ofNullable(tail).ifPresent(head::addAll);

    return head;
```

bad

good

```
public List<Integer> getPaginator(int pageNumber, int totalPages){
    List<Integer> head = (pageNumber > 4) ?
        new ArrayList<>(Arrays.asList(1,-1)) :
        new ArrayList<>(Arrays.asList(1,2,3));

    List<Integer> tail = (pageNumber < totalPages - 3) ?
        new ArrayList<>(Arrays.asList(-1,totalPages)):
        new ArrayList<>(Arrays.asList(totalPages-2,totalPages-1,totalPages));

    List<Integer> bodyBefore = (pageNumber > 4 && pageNumber < totalPages - 1) ?
        new ArrayList<>(Arrays.asList(pageNumber - 2, pageNumber - 1)):
        new ArrayList<>();

    List<Integer> bodyAfter = (pageNumber > 2 && pageNumber < totalPages - 3) ?
        new ArrayList<>(Arrays.asList(pageNumber + 1, pageNumber + 2)) :
        new ArrayList<>();

    Optional.ofNullable(bodyBefore).ifPresent(head::addAll);

    head.addAll(
        (pageNumber > 3 && pageNumber < totalPages - 2) ?
            new ArrayList<>(Arrays.asList(pageNumber)) :
            new ArrayList<>()
    );

    Optional.ofNullable(bodyAfter)
        .ifPresent(head::addAll);

    Optional.ofNullable(tail)
        .ifPresent(head::addAll);

    return head;
}
```



4. Method Size
- 1/2 screen
- NEVER Ever more 1 screen

→ brake down

Architecture

Single Responsibility

SOLID
- Dependency Inversion
- Interface Segregation
- Liskov Substitution
- Open / Close
- Single Responsibility

```java
Scanner scanner = new Scanner(System.in);
System.out.print("Enter x:");
String x = scanner.nextLine();
System.out.print("Enter y:");
String y = scanner.nextLine();
System.out.print("Enter op:");
String op = scanner.nextLine();
int r;
switch (op) {
  case "+" : r = Integer.parseInt(x) + Integer.parseInt(y); break;
  case "-" : r = Integer.parseInt(x) - Integer.parseInt(y); break;
  case "/" : r = Integer.parseInt(x) / Integer.parseInt(y); break;
  case "*" : r = Integer.parseInt(x) * Integer.parseInt(y); break;
}
System.out.println("result is:");
System.out.println(r);
```

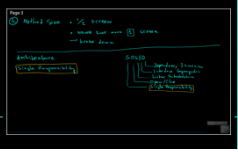
② Operation can be different

code duplication

① can fail

input tightly wired with output & logic

there is no error handling



```java
public static int expectsInt(Scanner scanner) {
  try {
    return Integer.parseInt(scanner.next());
  } catch (NumberFormatException x) {
    System.out.println("Integer number is expected, repeat please");
    return expectsInt(scanner);
  }
}

public static void main(String[] args) {

  Scanner scanner = new Scanner(System.in);

  System.out.print("Enter x:");
  int x = expectsInt(scanner);

  System.out.print("Enter y:");
  int y = expectsInt(scanner);

  System.out.print("Enter op:");
  String op = scanner.nextLine();
  int r = 0;
  switch (op) {
    case "+" : r = x + y; break;
    case "-" : r = x - y; break;
    case "/" : r = x / y; break;
    case "*" : r = x * y; break;
  }
  System.out.println("result is:");
  System.out.println(r);
```

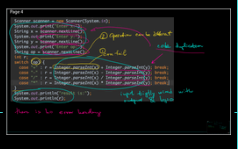
error handling here

input

no code duplication

logic

output



```java
public static int doOp(int x, int y, BiFunction<Integer, Integer, Integer> op) {
  return op.apply(x, y);
}

public static Optional<BiFunction<Integer, Integer, Integer>> validateOp(String op) {
  switch (op) {
    case "+" : return Optional.of((x, y) -> x + y);
    case "-" : return Optional.of((x, y) -> x - y);
    case "/" : return Optional.of((x, y) -> x / y);
    case "*" : return Optional.of((x, y) -> x * y);
    default: return Optional.empty();
  }
}

public static void main(String[] args) {

  Scanner scanner = new Scanner(System.in);

  System.out.print("Enter x:");
  int x = expectsInt(scanner);

  System.out.print("Enter y:");
  int y = expectsInt(scanner);

  System.out.print("Enter op:");
  String op = scanner.nextLine();

  validateOp(op) Optional<BiFunction<Integer, Integer, Integer>>
      .map(operation -> doOp(x, y, operation)) Optional<Integer>
      .ifPresent(r -> System.out.printf("Result is: %d", r));
```

no any relation with the whole code!!!

input

validation

busines idea

output

input → validate → doOp → output

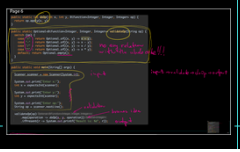
```java
class InputData {
  public final int x;
  public final int y;
  public final String op;

  InputData(int x, int y, String op) {
    this.x = x;
    this.y = y;
    this.op = op;
  }
}

public class CalculatorV4 {

  public static int expectsInt(Scanner scanner) {...}

  public static InputData getInputData(Scanner scanner) {
    System.out.print("Enter x:");
    int x = expectsInt(scanner);

    System.out.print("Enter y:");
    int y = expectsInt(scanner);

    System.out.print("Enter op:");
    String op = scanner.nextLine();

    return new InputData(x, y, op);
  }

  public static int doOp(int x, int y, BiFunction<Integer, Integer, Integer> op) { return op.apply(x, y); }

  public static Optional<BiFunction<Integer, Integer, Integer>> validateOp(String op) {...}

  public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);
    InputData data = getInputData(scanner);

    validateOp(data.op) Optional<BiFunction<Integer, Integer, Integer>>
        .map(operation -> doOp(data.x, data.y, operation)) Optional<Integer>
        .ifPresent(r -> System.out.printf("Result is: %d", r));
```

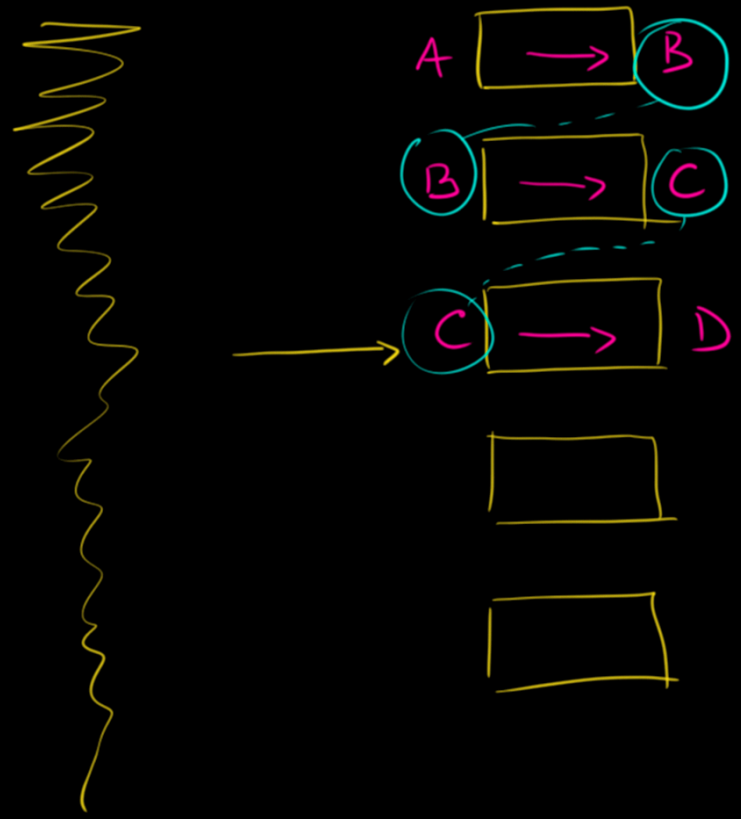
1 data representation

A => B

InputData => Input

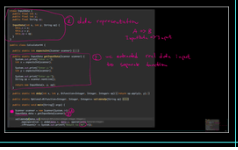
2 we extracted real data input to separate function





this approach
makes impossible to combine
things in a wrong way

1. structurize your code
2. define input / output types
3. chain them



```
int div(int x, int y) {
  return x / y;
}
```

~~(int, int) => int~~

this signature lies

this function is not TOTAL

=> (int, int) -> Int | Exception

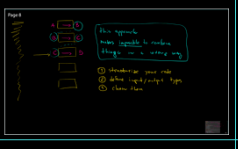
not to throw Exceptions

but REPRESENT wrong states

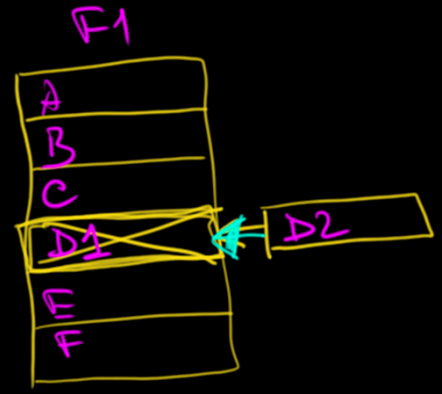
Option[A]

Either[E, A]

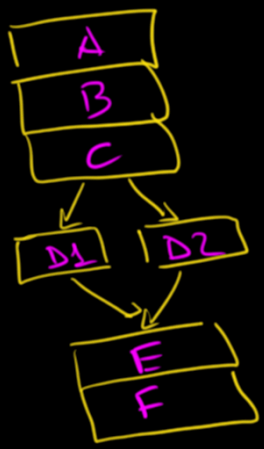
The Function Should Be Total



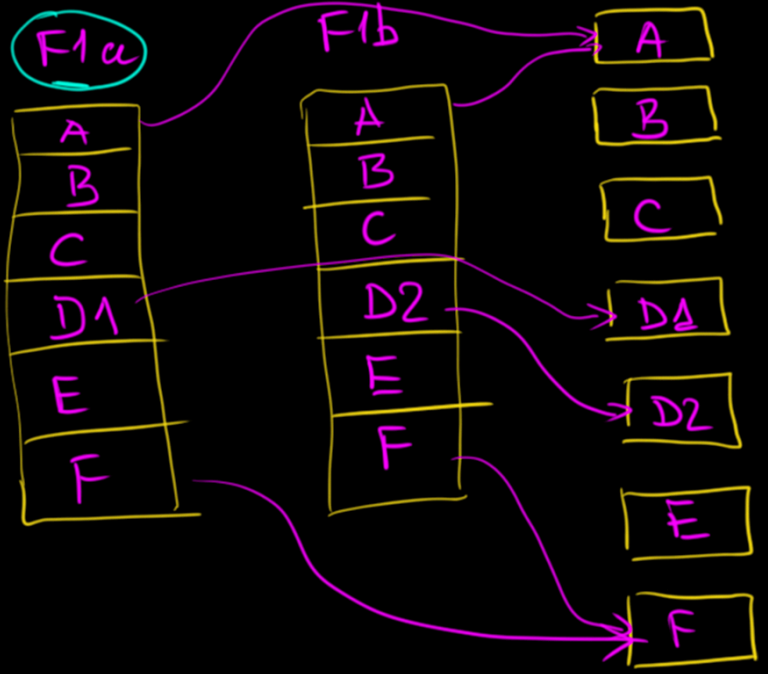
S - Single Responsibility
O - Open/Close Principle
L
I
D



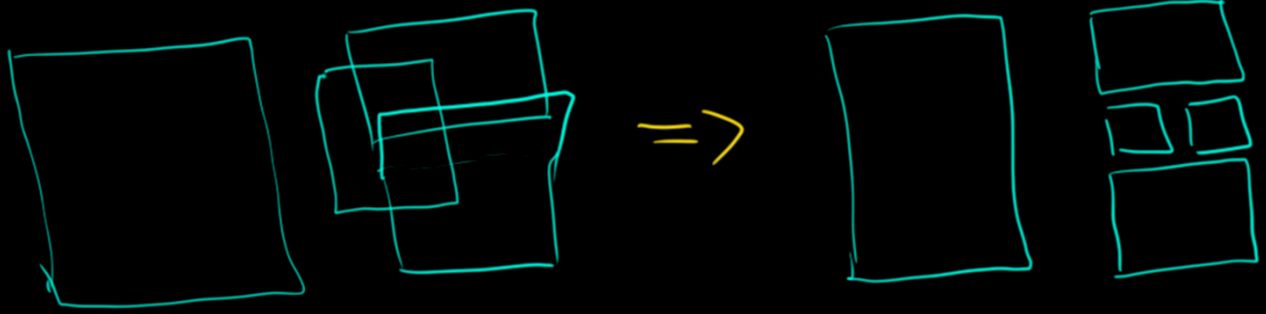
=>

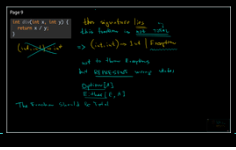


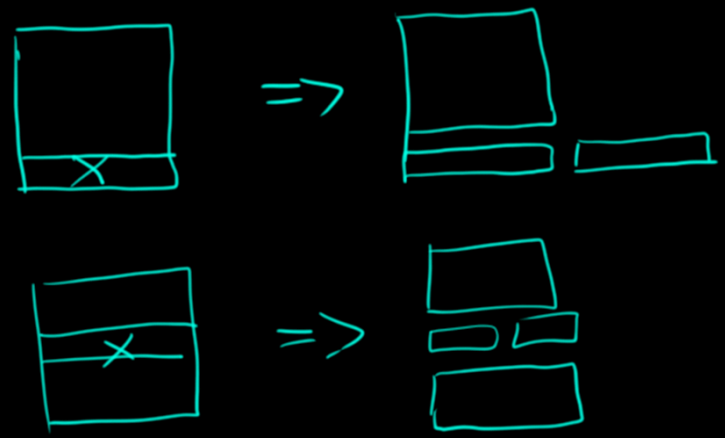
=>



never modify => create smaller blocks and combine them differently







A → C ⇏ B → D

A => B => C

B => C => D

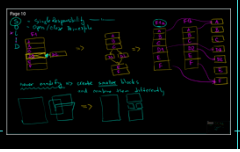
A => B => C => D

A => B = > C => D

common parts need to be factored out and pulled to separate function

we need to establish common joint points

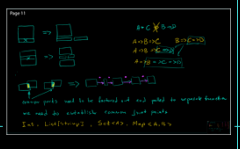
Int , List[String] , Set<A> , Map<A, B>



S ✓
O ✓
L ✓
I Interface Segre
D ✓

```java
abstract class Car {
  abstract void drive();
  abstract void autopilotDrive();
}

class Ford extends Car {

  @Override
  void drive() { System.out.println("Ford is driving"); }

  @Override
  void autopilotDrive() { throw new IllegalArgumentException("not supported"); }
}
class AppleCar extends Car {

  @Override
  void drive() { throw new IllegalArgumentException("not supported"); }

  @Override
  void autopilotDrive() { System.out.println("AppleCar is auto-driving"); }
}
class Tesla extends Car {

  @Override
  void drive() { System.out.println("Tesla is driving"); }

  @Override
  void autopilotDrive() { System.out.println("Tesla is auto-driving"); }
}

public class SOLID {

  public void drive(Car car) {
```

implementation will be hard



```java
abstract class Car {
  abstract void drive();
  abstract void autopilotDrive();
}
```

Interface Segregation Principle

```java
interface Driveable {
  void drive();
}

interface HasAutoPilot {
  void autopilotDrive();
}

class Ford1 implements Driveable {
  @Override
  public void drive() { System.out.println("Ford is driving"); }

}
class AppleCar2 implements HasAutoPilot {
  @Override
  public void autopilotDrive() { System.out.println("AppleCar is auto-driving"); }
}

class Tesla2 implements Driveable, HasAutoPilot {
  @Override
  public void drive() { System.out.println("Tesla is driving"); }
  @Override
  public void autopilotDrive() { System.out.println("Tesla is auto-driving"); }
}

public class SOLID2 {

  public void drive(Driveable car) {
  }
  public void autoDrive(HasAutoPilot car) {
  }
```

we don't need to write if

```
class File {
  int size
  byte[] content
}
```

=>

```
class ZippedFile {
  File file
  int size
  byte[] content
}
```

```
class ZippedFile extends File {
  override int size
  override byte[] content
}
```

- zipped/non-zipped
- zipped/non-zipped

???

File

File

ZippedFile

ZippedFile

size

content

← Underlying file

size

content

???

more questions than solutions

You don't need to use inheritance AT ALL

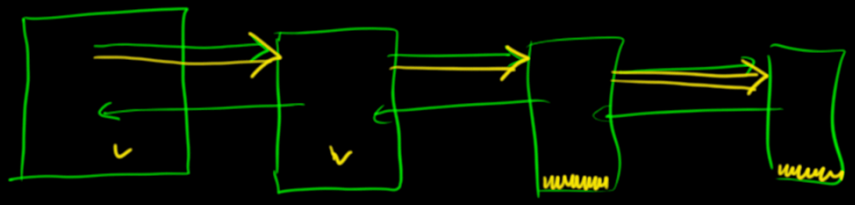
you can be abstract (incomplete, interface)

you can be final Integer, String final

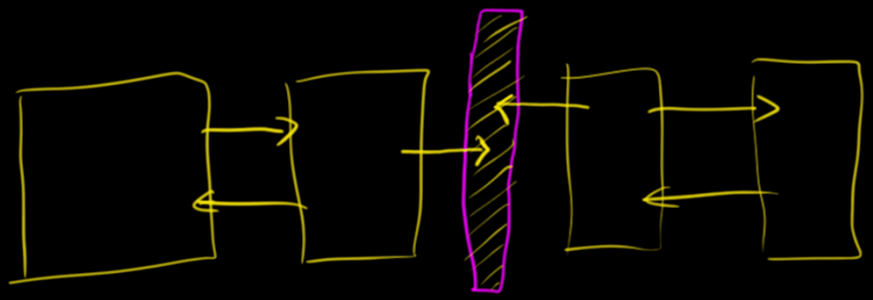
⇓

Use composition instead of inheritance 99%

(D) Dependency Inversion



all functions are dependent

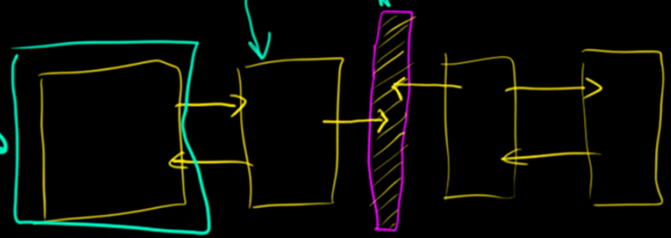


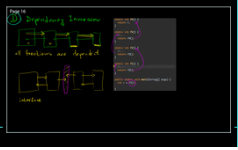
interface

```
static int f4() {
  return 1;
}

static int f3() {
  //...
  return f4();
}

static int f2() {
  //...
  return f3();
}

static int f1() {
  //...
  return f2();
}

public static void main(String[] args) {
  int x = f1();

}
```

```java
interface CanFormat {
  String format(String origin);
}

static class FormatterToUpper implements CanFormat {
  @Override
  public String format(String origin) { return origin.toUpperCase(); }
}

static class FormatterToLower implements CanFormat {...}

static class FormatterWithBraces implements CanFormat {...}

public static String format1(String s) {
  CanFormat fmt = new FormatterToUpper();
  return fmt.format(s);
}

public static void main(String[] args) {
  System.out.println(format1( s: "hello"));
}
```

is it aware of implementation?





```java
public static String format1(String s, CanFormat fmt) {
  return fmt.format(s);
}
```

```
format1(String s, String => String fmt) {
  return fmt(s)
}
```

All design principles
All patterns are about different composition ways