```java
public class SpaghettiDrivenDesign {

  public static void main(String[] args) {
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
    //..
  }
}
```

Procedural

```java
public class ProcedureDesign {

  public static void do1() {
    //..
    //..
    //..
  }

  public static void do2() {
    //..
    //..
    //..
  }

  public static void do3() {
    //..
    //..
    //..
  }

  public static void main(String[] args) {
    //..
    //..
    //..
    do1();
    //..
    //..
    //..
    //..
    do2();
    //..
    //..
    //..do1();
    //..
    //..
    do3();
  }

}
```

Procedural Evolution

```java
public class ProcedureDesign2 {

  public static void do1(int i) {
    //..
    //..
    //..
  }

  public static void do2(String name) {
    //..
    //..
    //..
  }

  public static void do3(int index1, double k) {
    //..
    //..
    //..
  }

  public static void main(String[] args) {
    //..
    //..
    //..
    do1( i: 1);
    //..
    //..
    //..
    //..
    do2( name: "Jim");
    //..
    //..
    //..
    //..
    //..
    do3( index1: 2, k: 3);
  }
```

```java
public class ProcedureDesign2 {

  public static List<String> do1(File file) {
    //..
    //..
    //..
  }

  public static double calculate(int index1, double k) {
    //..
    //..
    //..
  }

  public static void store(File name, List<String> data) {
    //..
    //..
    //..
  }

  public static void main(String[] args) {
    //..
    //..
    //..
    List<String> data = do1(new File( pathname: "input"));
    //..
    //..
    //..
    //..
    double result = calculate( index1: 2,  k: 3);
    //..
    //..
    //..
    //..
    //..
    store(new File( pathname: "output"),???);
  }
```

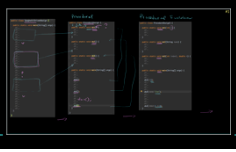
File => List<String>

(File, List<String>) => void

=>

once we have void
we can't predict, understand
without reading the code

seeing void we need to understand
we modify something outside



OOP

we encapsulate data and code together and call them a class

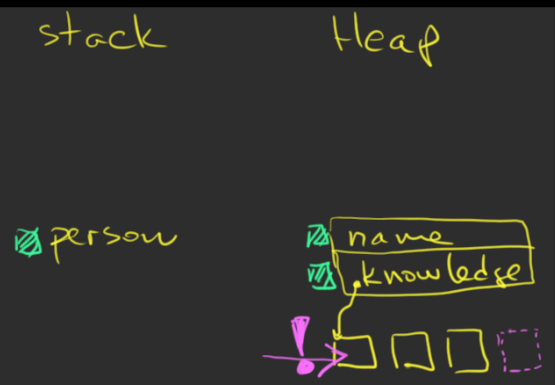
```java
class Point2D {
  int x;
  int y;

  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }

  void move(int dx, int dy) {
    x += dx;
    y += dy;
  }

  void show() {
    //...
  }

  void hide() {
    //...
  }

}


public class ObjectOrientedDesign {

  public static void main(String[] args) {
    Point2D p1 = new Point2D( x: 10,  y: 20);
    p1.show();
    p1.hide();
    p1.move( dx: 5,  dy: 6); // 15, 26
    p1.x -= 10;
    p1.show();
  }

}
```

we are modifying this point

problem

```java
class Person {
  private final String name;
  private final List<String> knowledge;

  Person(String name, List<String> knowledge) {
    this.name = name;
    this.knowledge = knowledge;
  }

  public String getName() {
    return name;
  }

  public List<String> getKnowledge() {
    return knowledge;
  }
}

public class ObjectOrientedDesign {

  public static void main1(String[] args) {...}

  public static void main(String[] args) {
    Person person = new Person( name: "Jim", new ArrayList<>(Arrays.asList("Java", "JavaScript")));
    String name = person.getName();
    List<String> knowledge = person.getKnowledge();
    knowledge.add("PHP");
    System.out.println(person.getKnowledge());
  }
```

stack heap



[Java, JavaScript, PHP]

problem → absence of proper data type

not a problem of OOD

Functional Programming

```
public class FunctionalDesign {

  int add(int x, int y) {
    return x + y;
  }

}
```

not to have any state
any global variable

pros

100% data is coming as param. list

(x, y) -> int

```
void printLine(String line) {
  System.out.println(line);
}
```

```
public String readLine() {
  Scanner s = new Scanner(System.in);
  String line = s.nextLine();
  return line;
}
```

you can not write any tests for such kind of code

global variable (indirect, implicit dependency)

verbose

```
void printLine(String line, PrintStream out) {
  out.println(line);
}
```

```
public String readLine(InputStream in) {
  Scanner s = new Scanner(in);
  String line = s.nextLine();
  return line;
}
```

you can write any test for that code

you can provide custom impl (mocked)

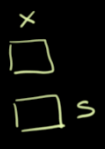
because 100% dependencies in parameters !!!

Cons:

- code starts being verbose

```
public static void main(String[] args) {
  int x = 5;
  x = x + 10;
  String s = Integer.toString(x);
  System.out.println(s);
}
```

```
public static void main(String[] args) {
  int x = 5;
  String s = Integer.toString(x);
  x = x + 10;
  System.out.println(s);
}
```




```
Optional.of(5) Optional<Integer>
  .map(z -> z + 10)
  .map(z -> Integer.toString(z)) Optional<String>
  .ifPresent(z -> System.out.println(z));
```

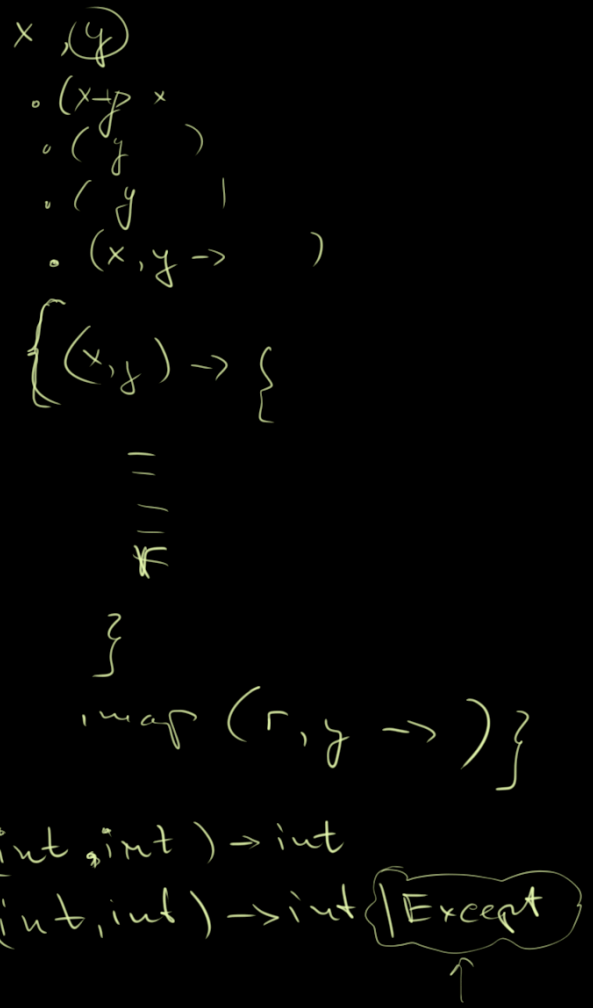
Pros

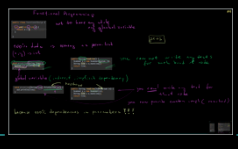
1) the scope is limited
2) variable is passing automatically

5
15
"15"

the longer chain you have
the more memory you consume
the memory is cheaper than developer's time

your code is times robust
times less fragile





Object Oriented Programming

Classes

- Methods

A lot of Patterns ~30

variable mutation (x = x+5)

loops

easy ~~simple~~ complexity is high

focusing on process

hard to reuse

easy to start

Fragile

Functional Programming

Data Types (Classes w.o. methods)
struct

- function

function composition

immutable => new variable creation

recursion

simple not easy
not complex
not complicated

focus on data

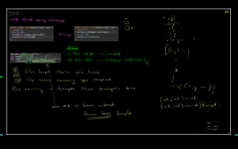
easy to reuse

A => B
B => C
A => C

hard to start

consumes more memory

Robust

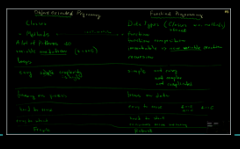


```java
static BiFunction<Integer, Integer, Integer> makeSafe(
  BiFunction<Integer, Integer, Integer> unSafe,
  Function<Exception, Integer> handler
) {
  return (a, b) -> {
    try {
      return unSafe.apply(a, b);
    } catch (Exception ex) {
      return handler.apply(ex);
    }
  };
}
```

```java
static <A, B, C> BiFunction<A, B, C> makeSafeGen(
  BiFunction<A, B, C> unSafe,
  Function<Exception, C> handler
) {
  return (a, b) -> {
    try {
      return unSafe.apply(a, b);
    } catch (Exception ex) {
      return handler.apply(ex);
    }
  };
}
```

```java
BiFunction<Integer, Integer, Integer> safeDiv =
  makeSafe(FunctionalTryCatch::div, ex -> 13);

Integer applied = safeDiv.apply(t: 20, u: 0); // 13
```

(A, B) => C | Exception



modularity

/project

/project
/common/ABC
/m1
/m2