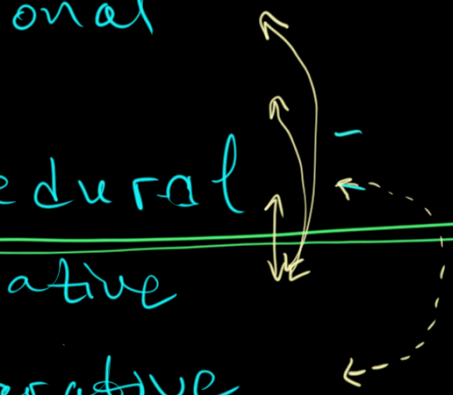
Programming Paradigms

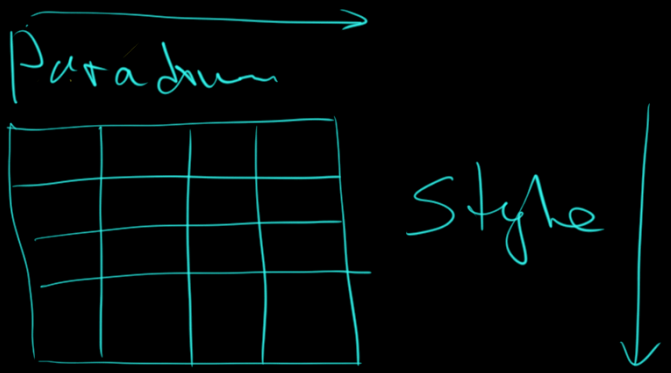
- functional
- OOP
- procedural

⊕ imperative

⊕ declarative



Another Category / Style




```java
/** imperative */
public int min(int[] as) {

  int min = as[0];

  for (int i = 1; i <= as.length-1 ; i++) {
    if (as[i] < min) min = as[i];
  }

  return min;
}
```

boilerplate

business value

we exactly tell compiler what to do

many places to introduce a mistake

```java
public int min2(int[] as) {

  int min = as[0];

  for (int a : as) {
    if (a < min) min = a;
  }

  return min;
}
```

compiler knows everything

```java
public int min(int[] as) {

  int min = as[0];

  for (int i = 1; i < as.length ; i++) {
    if (as[i] < min) min = as[i];
  }

  return min;
}
```

state

declarative

→ traverse / iterate over all elements in the structure

more human understandable

there is no way to introduce a mistake

imperative

- take index 1
- apply <
- check the length of array

knowledge about structure

"consequences" of this approach

```
public int min2(int[] as) {
  int min = as[0];

  for (int a : as) {
    if (a < min) min = a;
  }

  return min;
}
```

places for potential mistakes

you can easily reassign wrong variable

imperative

declarative

```
public int justMin(int a, int b) {
  return a <= b ? a : b;
}
```

imperative

```
public int min3(Stream<Integer> as) {

  int result = as.reduce(
      identity: 0,
      (a, b) -> justMin(a, b)
  );
  return result;
}
```

declarative

reduce ≡ iterate + apply function

iteration and function application to eliminate possibility of mistake.

we don't have =

=> no way to brake the things

the ultimate goal is ELIMINATE points where we can introduce a mistake

```java
public int min3(Stream<Integer> as) {

  int result = as.reduce(
      identity: 0,
      (a, b) -> justMin(a, b)
  );
  return result;
}
```

initial value to start from

```java
int result = as.reduce(
    Integer.MAX_VALUE,
    (a, b) -> justMin(a, b)
);
```

PAIN

[1,2,3]

(0,1) → 0
(0,2) → 0
(0,3) → 0

(Max,1) → 1
(1,2) → 1
(1,3) → 1

if dataset is empty ?

result INT.MAX_VALUE
2.000.000.000 ???

int[] → int wrong signature

what is the result type if dataset (as) is Empty we CAN NOT provide a result

```java
as.reduce(
    (a, b) -> justMin(a, b)
);
```

Optional<Int>
if (nowEmpty) int Optional.of
if (empty) None

```
Optional<Integer> reduced = as.reduce(
    (a, b) -> justMin(a, b)
);
```

good	(can be) bad
• we don't need to provide any default value	• we need to deal with Optional

we deferred the place of taking final decision for default value

```
a <= b ? a : b
```

imperative

how to pass Optional[int] to UI

int[] — Optional.empty

int[x] = Optional.of(x)

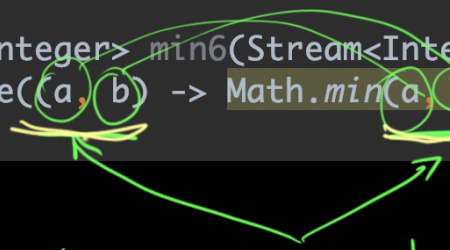
~~number? : int~~

number : int[] — .empty / num[0]

```
public Optional<Integer> min5(Stream<Integer> as) {
  return as.reduce(Math::min);
}
```

declarative

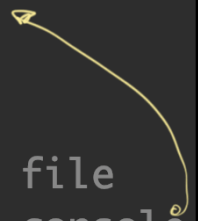
```java
public Optional<Integer> min5(Stream<Integer> as) {
  return as.reduce(Math::min);
}

public Optional<Integer> min6(Stream<Integer> as) {
  return as.reduce((a, b) -> Math.min(a, b));
}
```



because we have EXACTLY the SAME
parameter number
parameter types

and we used once

```
// open console
// read the number
// read the file
// do the logic
// write some result to file
// write some result to console
// end
```



```
int[] data = {1,2,3};
int i = 0;

while (i < data.length) {
  System.out.println(data[i]);
}
```

- infinite loop.

problems

- code duplication
- complexity
- easy to brake the things

```
int[] data = {1,2,3};
int i = 0;

while (i < data.length) {
  System.out.println(data[i]);
  i++;
}

System.out.println(i);
```

this var. is still accessible

```
int[] data = {1,2,3};
int i = 0;

while (i < data.length) {
  System.out.println(data[i]);
  i++;
}

data = new int[]{1,2,3,4};
while (i < data.length) {
  System.out.println(data[i]);
  i++;
}

System.out.println(i);
```

global variable

implied things but required

this variable isn't connected to real place where it is required

we must somehow (?) wire i and use case variable

does it contain any useful ? → NO

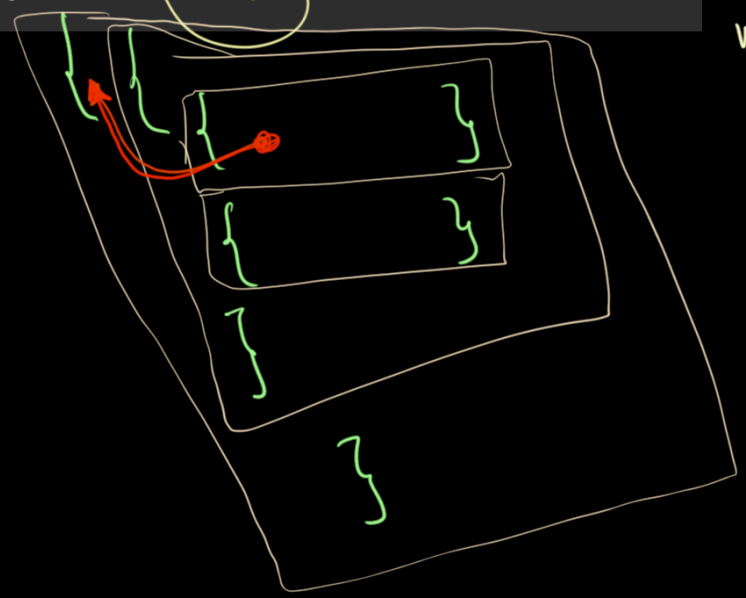
should we be able to refer it ? → NO

```java
int[] data = {1,2,3};
for (int i = 0; i< data.length; i++) {
  System.out.println(data[i]);
}

data = new int[]{1,2,3,4};
for (int i = 0; i< data.length; i++) {
  System.out.println(data[i]);
}

System.out.println(i);
```

required variable is linked to the place it is used

we have NO ACCESS to it after

Scope



we introduced a mistake because INNER / NESTED scope depends on the OUTER scope

the ultimate goal of Prog is to reduce the scope

```java
public int min(int[] xs) {
  // real min calculation
  int result = ; // I don't care for this impl
  System.out.println("min:" + result);
  return result;
}

public void example3() {
  // ...
  // ...
  int x = min(new int[]{3, 5});
  // ...
  // ...
}
```

we have the signature

1. input param(s) w/ types
2. output type
3. meaningful name

global variable

Scope

we are w/o intention brought the global variable

the ultimate goal is - not to have/use any global variables

OOP
- procedural
- functional

different way to manage the scope

we started:

procedural
- we take the code duplication
- we pull out from our code
- we give them a meaningful name
- we reuse it
- PROFIT.

state?

can not be handled

Object oriented

we combine state with behavior

class is definition

```
static class Point {
  int x;
  int y;

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

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

}

public static void objectOriented() {
  Point point1 = new Point( x: 10,  y: 10);
  point1.move( dx: 3, dy: 3); // 13, 13

  Point point2 = new Point( x: 20,  y: 20);
  point2.move( dx: 2, dy: 2); // 22, 22
}
```

- it's easier to deal with state because state
  - is hidden inside the object

object is instance of the class

- we don't expose this state
- we just operate with state in the scope of object

procedural:

int x
int y
point

move (Δx, Δy)

point is moved

int[] xs

sort

data is sorted

```java
public class ProcedureWay {

  static class Point {
    int x;
    int y;

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

  /** move implementation can be reused */
  public static void move(Point p) {
    ///
  }

  public static void procedural(String[] args) {
    Point point = new Point( x: 10,  y: 10);
    move(point);
  }
```

combination state and behaivor => pass less parameters

=> the less way to intro. mistake

```java
static class Point {
  int x;
  int y;

  // 1. point creation
  public Point(int x, int y) {...}

  // 2. move operation
  public void move(int dx, int dy) {...}

  // 3. show operation
  public void show() {}

  // 3. hide operation
  public void hide() {}

}

public static void objectOriented() {
  Point point1 = new Point( x: 10,  y: 10);
  point1.hide();
  point1.move( dx: 3, dy: 3); // 13, 13
  point1.show();

  Point point2 = new Point( x: 20,  y: 20);
  point2.move( dx: 2, dy: 2); // 22, 22
}
```

```
move(point);
```

primary focus

secondary focus

```
point1.hide();
point1.move( dx: 3, dy: 3);
point1.show();
```

primary focus
it's kind of guaranty
that you picked proper state

OOP > procedural

problems:

- there is NO way to implement SINGLE RESPONSIBILITY point (x, y)
- show point on different devices
  - show <=> mac, win, linux ...

1. you do what you need
2. you are limited what you can do.

Function Programming

1. absence of global context (state, vars, ...)
   everything you need -> you have in f(.....)
2. we don't have a state
   we don't mutate any data
3. No exceptions
4. No SideEffects like sout("hello")

```
static class PointOOP {
  int x;
  int y;
  public PointOOP(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
```

```
static class PointFP {
  int x;
  int y;
  public PointFP(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public PointFP move(int dx, int dy) {
    return new PointFP(x:x + dx, y:y + dy);
  }
}
```

```java
/** step 0 */
public void print0(String message) {
  System.out.println(message);
}

/** step 1 */
public void print1(String message, PrintStream device) {
  device.println(message);
}

/** step 3 */
public Supplier<void> print3(String message, PrintStream device) {
  return () -> device.println(message);
}
```

we make all deps explicit.

we eliminate side effects