IMMUTABILITY & PURE FUNCTIONS

Malte Neuss

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

```
  # assert prices[0] == 5?
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

```
  # assert prices[0] == 5?
```

```
def minimum(values)
   values.sort()                                  # surprise
   return values[0]
```

CONTENT

CONTENT

- Immutability

CONTENT

- Immutability
- Pure Functions

CONTENT

- Immutability
- Pure Functions
- Apply to OO

CONTENT

- Immutability
- Pure Functions
- Apply to OO

Easier reasoning, testing, debugging..

IMMUTABILITY

No reassign of variables:

```
  value = 1                                    # Assign
# value = 2                                    # Reassign
  other = 2
```

IMMUTABILITY

No reassign of variables:

```
  value = 1                                    # Assign
# value = 2                                    # Reassign
  other = 2
```

No mutation on objects:

```
def minimum(values)
   # values.sort()                             # mutation
   other = sorted(values)                       # no mutation
   return other[0]
```

BENEFIT: REASONING

```
def main()
  prices = immutable([5,2,7])                    # if supported
  # assert prices[0] == 5
  minPrice = minimum(prices)
  # assert prices[0] == 5!
```

BENEFIT: REASONING

```
def main()
  prices = immutable([5,2,7])                    # if supported
  # assert prices[0] == 5
  minPrice = minimum(prices)
  # assert prices[0] == 5!
```

```
def minimum(values)
   values.sort()                                 # error
   return other[0]
```

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \textbf{Type}$$

$$f(x) = x + \pi \qquad \textbf{Body}$$

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```python
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output
- No side-effects

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```python
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output
- No side-effects
- External immutable values or constants ok

SIDE-EFFECT

SIDE-EFFECT

- File read/write

SIDE-EFFECT

- File read/write
- Network access

SIDE-EFFECT

- File read/write
- Network access
- I/O

SIDE-EFFECT

- File read/write
- Network access
- I/O
- Throwing exception

SIDE-EFFECT

- File read/write
- Network access
- I/O
- Throwing exception
- Argument mutation

SIDE-EFFECT

- File read/write
- Network access
- I/O
- Throwing exception
- Argument mutation
- ...

SIDE-EFFECT

- File read/write
- Network access
- I/O
- Throwing exception
- Argument mutation
- ...

Any state change outside of return value

```
def f(x: float) -> float
```

BENEFIT: REASONING

```
def minimum(values)                                    # pure
    # values.sort()
    other = sorted(values)                             # no mutation
    return other[0]
```

BENEFIT: REASONING

```
def minimum(values)                                    # pure
    # values.sort()
    other = sorted(values)                             # no mutation
    return other[0]
```

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
  minPrice = minimum(prices)                           # pure
  # assert prices[0] == 5!
```

BENEFIT: TESTING

```
def test()
   values   = [5,2,7]                          # data
   expected = 2

   out = minimum(values)                        # pure

   assert out == expected
```

BENEFIT: TESTING

```
def test()
   values   = [5,2,7]                              # data
   expected = 2

   out = minimum(values)                            # pure

   assert out == expected
```

cost(data) < cost(classes)

BENEFIT: REFACTORING

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

BENEFIT: REFACTORING

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

=

```
def main():
  x = compute(5,2)                    # Factored out
  y = x             + 7
  z = x             + 1
```

BENEFIT: REFACTORING

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

=

```
def main():
  x = compute(5,2)                    # Factored out
  y = x             + 7
  z = x             + 1
```

Fearless refactoring, also by compiler

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = randomNumber() + 7
  z = randomNumber() + 1
```

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = randomNumber() + 7
  z = randomNumber() + 1
```

!=

```
def main():                                    # 1 side-effect
  x = randomNumber()                           # Factored out
  y = x              + 7
  z = x              + 1
```

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = randomNumber() + 7
  z = randomNumber() + 1
```

!=

```
def main():                                    # 1 side-effect
  x = randomNumber()                           # Factored out
  y = x              + 7
  z = x              + 1
```

Careful refactoring, often manual

BENEFIT: ABSTRACTION

```
def f(_: Char) -> Int
```

BENEFIT: ABSTRACTION

```
def f(_: Char) -> Int
```

```
def toAscii(c: Char) -> Int
```

BENEFIT: ABSTRACTION

```
def f(_: Char) -> Int
```

```
def toAscii(c: Char) -> Int
```

```
def f(_: [Any]) -> Int
```

BENEFIT: ABSTRACTION

```
def f(_: Char) -> Int
```

```
def toAscii(c: Char) -> Int
```

```
def f(_: [Any]) -> Int
```

```
def length(list: [Any]) -> Int
```

Types often enough for understanding.

IMPURE FUNCTIONS

```
def f() -> None
```

IMPURE FUNCTIONS

```
def f() -> None
```

```
def performTask() -> None
  inputs = loadInputs()
  computeResult(inputs)
```

IMPURE FUNCTIONS

```
def f() -> None
```

```
def performTask() -> None
  inputs = loadInputs()
  computeResult(inputs)
```

```
def launchMissiles() -> None
  coord = loadCoordinates()
  ...
```

Need to look at body to be sure.

CLASSICAL OO

```
class MyClass
  state: LotsOfState                                    # private
```

CLASSICAL OO

```
class MyClass
  state: LotsOfState                                  # private
```

```
  def do() -> None                                    # public
    state.do_a()
    state.do_b()
    this._do()
```

CLASSICAL OO

```
class MyClass
  state: LotsOfState                                  # private
```

```
  def do() -> None                                    # public
    state.do_a()
    state.do_b()
    this._do()
```

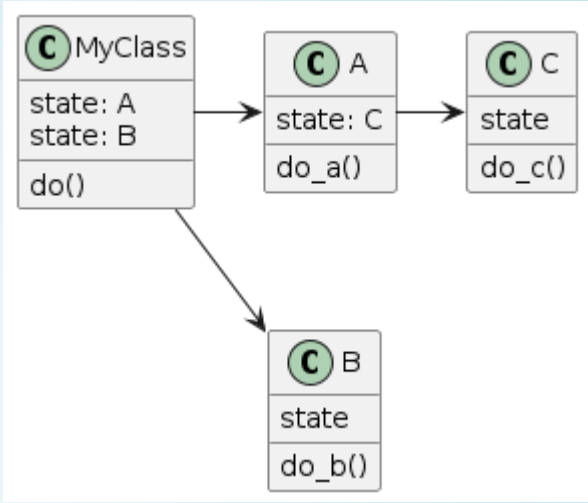
```
  def _do()                                           # private
    state.do_c()
    ...
```

CLASSICAL OO

```
class MyClass
  state: LotsOfState

  def do() -> None
    state.do_a()
    state.do_b()
    this._do()

  def _do()
    state.do_c()
    ...
```

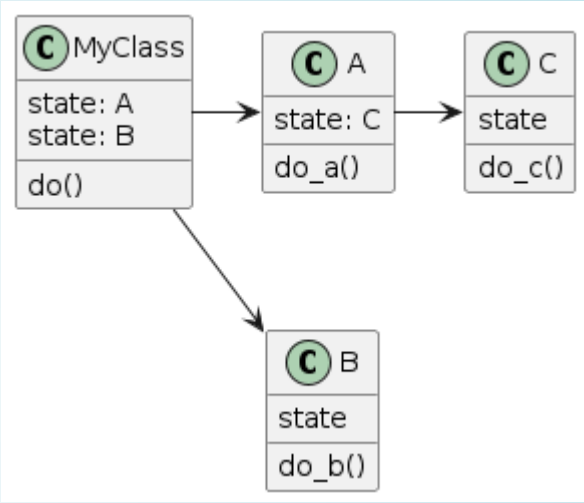


CLASSICAL OO

```
class MyClass
  state: LotsOfState

  def do() -> None
    state.do_a()
    state.do_b()
    this._do()

  def _do()
    state.do_c()
    ...
```




```
...
myClass.do()
...
```

FP-ISH OO

```
class MyClass
  constants
```

FP-ISH OO

```
class MyClass
  constants
```

```
  def do(start)                                  # Pure
    x = do_a(start)                              # No mutation
    y = do_b(x, constants)                       # No mutation
    z = this._do(y)                              # No mutation
    return z
```

FP-ISH OO

```
class MyClass
  constants

  def do(start)                                    # Pure
    x = do_a(start)                                # No mutation
    y = do_b(x, constants)                         # No mutation
    z = this._do(y)                                # No mutation
    return z


  def _do(y)                                       # Pure
    return do_c(y, constants)                      # No mutation
```

GLOBAL MUTATION

```
state: LotsOfState                                    # Global variable
```

GLOBAL MUTATION

```
state: LotsOfState                                  # Global variable
```

```
def someOperation()
   state.do_a()                                     # Global mutation
```

GLOBAL MUTATION

```
state: LotsOfState                                  # Global variable
```

```
def someOperation()
   state.do_a()                                     # Global mutation
```

```
def main()
   ...
   someOperation()
   ...
```

CLASS MUTATION

```
class MyClass
  state: LotsOfState                                    # Class variable
```

CLASS MUTATION

```
class MyClass
  state: LotsOfState                          # Class variable
```

```
  def someOperation()
    state.do_a()                              # Class mutation
```

CLASS MUTATION

```
class MyClass
  state: LotsOfState                          # Class variable
```

```
  def someOperation()
    state.do_a()                              # Class mutation
```

```
def main()
   ...
   myClass.someOperation()
   ...
```

OUTPUT ARGUMENTS

Not ok:

```
def do(state):
  # ... modify argument 'state'
```

OUTPUT ARGUMENTS

Not ok:

```
def do(state):
  # ... modify argument 'state'
```

```
x = do(state)                                    # 'state' mutated
```

OUTPUT ARGUMENTS

Not ok:

```
def do(state):
  # ... modify argument 'state'
```

```
x = do(state)                                    # 'state' mutated
```

So why should this?

```
x = state.do()                                   # 'state' mutated
```

OUTPUT ARGUMENTS

Not ok:

```
def do(state):
  # ... modify argument 'state'
```

```
x = do(state)                                    # 'state' mutated
```

So why should this?

```
x = state.do()                                   # 'state' mutated
```

```
class State:
```

```
def do(self):                                   # 'self' is
    'state'
  # ... modify argument self
```

HOW TO SIDE-EFFECTS

```
def performTask()
  inputs = fetchInputs()                    # IO
  result = 2*inputs
  publish(result)                           # IO
```

HOW TO SIDE-EFFECTS

```
def performTask()
  inputs = fetchInputs()                    # IO
  result = 2*inputs
  publish(result)                           # IO
```

```
# Infrastructure/Application layer        # Impure
def performTask()
  inputs = fetchInputs()
  result = domainLogic(inputs)
  publish(result)

# Domain layer
def domainLogic(inputs)                     # Pure
  return 2*inputs
```

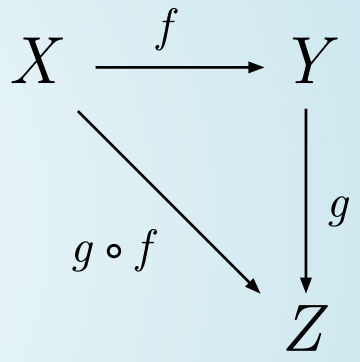
Separate IO and logic

FURTHER TOPICS

Try

- Scala
- Rust
- Haskell

Category Theory



- Algebraic Data Types
- Functor (map)
- Monad (flatMap)
- Lens ...

QUESTIONS?

Thanks

TYPE CHECKER SUPPORT

Make illegal state (more) unrepresentable

IMMUTABLE BUILT-IN CLASSES

Immutable interface:

```
mySet = frozenset([1, 2, 3])                    # Python
mySet.add(4)                                    # type error
```

IMMUTABLE BUILT-IN CLASSES

Immutable interface:

```
mySet = frozenset([1, 2, 3])                    # Python
mySet.add(4)                                    # type error
```

Copy on change:

```
val myList = immutable.List(1, 2, 3)            // Scala
val other  = myList.appended(4)                 // other list
```

IMMUTABLE CUSTOM CLASSES

Immutable interface around mutable data:

```
class MyClass:
   _value: int                                  # hide mutables

   def getValue()                               # no setters
     return _value

   def calcSth()                                # _value read only
     return _value*2

   def incremented()                            # Copy on change
     return MyClass(_value+1)
```

IMMUTABLE CUSTOM CLASSES

With extra language support:

```python
@dataclass(frozen=True)                    # Python
class MyClass:
   value: int

object       = MyClass(1)
object.value = 2                           # compile error!
```

IMMUTABLE CUSTOM CLASSES

With extra language support:

```python
@dataclass(frozen=True)                    # Python
class MyClass:
   value: int

object       = MyClass(1)
object.value = 2                           # compile error!
```

```typescript
interface MyInterface {                    // Typescript
  readonly value: int;
}
```

IMMUTABLE VARIABLES

Mutable:

```
var value = 1                                   // Typescript
value = 2                                       // ok
```

IMMUTABLE VARIABLES

Mutable:

```
var value = 1                                   // Typescript
value = 2                                       // ok
```

Immutable:

```
const value = 1                                 // Typescript
value = 2                                       // type error!
```