A Retrospect on Object-Oriented Calculator

김명호
enkiluv@cau.ac.kr

Object-Orientation

Key Traits or Aims

- Reuse
- Additive (Non-destructive) Extension

Of WHAT?

- Types and Implementations
- Frameworks (class libraries)
- Programs!

The HOW-TO's of O-O: Data Abstraciton + …

- Message-passing
- Polymorphism and Dynamic Binding
- Inheritance (Type, Implementation, Both)

O-O Calculator

A calculator in the spirit of O-O

- What to reuse and extend? The Whole Program!
  - Even the “main” program is to be reused without any changes

How can we achieve it using O-O?

- Just languages alone are not enough!
- In addition, an elaborate design is also needed

Things to Ponder

Math types vs. Calc types

- Is an Integer also a Complex? Y vs. N
  - Does an integer require an imaginary part? Probably not.
- Is a Rational also a Real(Float)? Y vs. N
  - Can an inexact float number represent an exact ratio? Absolutely not!

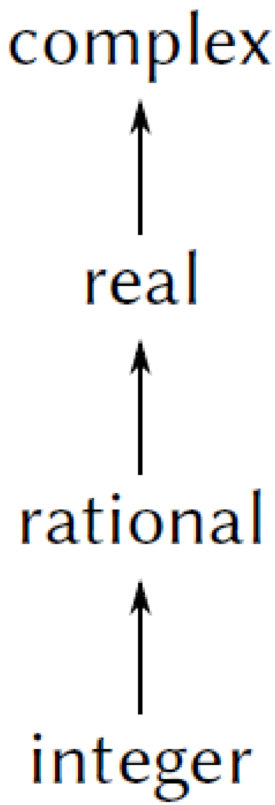
Rationale

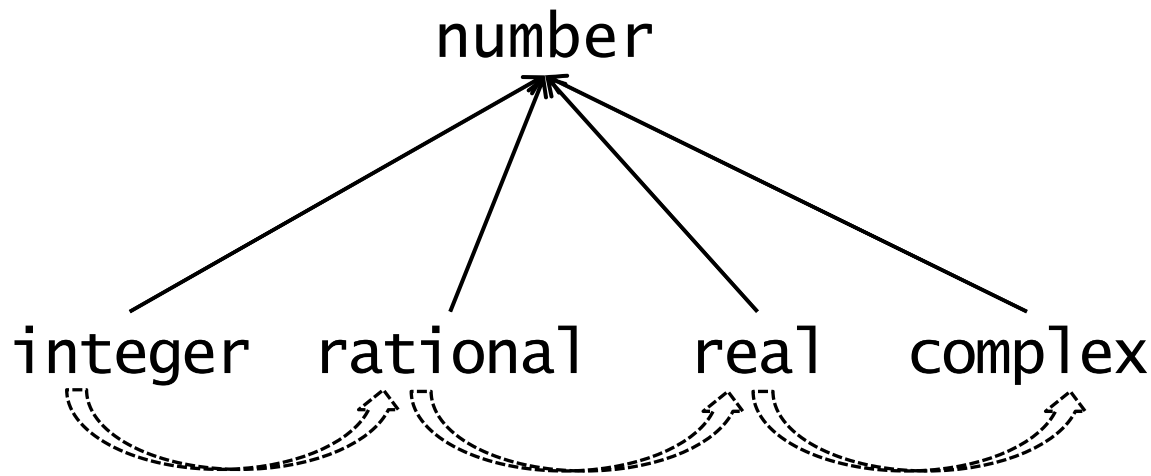
- Calc types do not faithfully reflect Math types!
- Still, they are all Numbers

A Remedy?

- Make each Calc type distinct
- Promote a Calc type to another according to the Math type hierarchy

Type Hierarches





Doing Arithmetic with Numbers:

1. Match (dynamic) types of operands through promotion, and then
2. Apply (dispatch to) an appropriate operation using dynamic binding

The Calculator

Maintains a stack of Numbers

Performs calculations expressed in the RPN

- Fetch two operands from the stack
- Compute result by performing the operation
- Push back the result into the stack

Number(s)

Provides generic operations +, -, … that can be invoked by the calculator's add(), sub(), … requests

- They call appropriate add, sub, mul, div methods according to the actual types of operands

Appropriate?

- Promote operand types if necessary (into type *t*)
- Dispatch to the corresponding implementation of the operation using dynamic binding on *t*
- e.g. An example: 1+2.5 (by the commands "1", "2.5", then "add")
  - Promotes 1 to Float as *t*
  - Calls(dispatches to) *t*.add(Float(1), Float(2.5)) to get Float(2.5)

Maintains a table of allowed promotions (coercions)

Integer

Define the type by extending Number

Provide implementations of add, sub, mul, div

Provide promotion rules if necessary

Making Additive Extensions

Define new types (e.g. Float, Rational, Complex) by extending Number

Implement operations add, sub, mul, div for the type

- Shouldn’t rely on the internal representations of existing types
  - They are generally extremely prone to change anytime in the future

Define promotion rules for the relevant types

Nothing else is needed!

Isn’t it an Egg of Columbus?

Even More Extensions

Inexact Numbers

- Inexact or uncertain values can often be regarded as an interval, say [a, b]
- We can also define arithmetic operations on intervals
  - https://mitpress.mit.edu/sites/default/files/sicp/full-text/sicp/book/node31.html
- Add Inexact as a new kind of Number, and allow mixed-mode operations with other Number types with modest efforts

Why is This an O-O Calculator?

All the operations are generic (reusable) by nature because their operands are polymorphic.

Calculations are performed by message passing based-on dynamic types.

Number is inherited to all different number types.

- Implementation inheritance is kept to a minimum
- We just provide add, sub, … and coercions instead of redefining (overriding) inherited operations +, -, *, and /.

Fosters extensions with modest (if not minimal) efforts!

More Thoughts

Typical Anti-Patterns

- Ever growing chained IF's in overriding methods (static binding)
- Ever growing list of multi-methods (painfully additive)

May result in combinatorial explosion of mixed types

Can this be done differently? Yes, of course!

- Implementing generic operations using data-directed programming with tagged data

  https://inst.eecs.berkeley.edu/~cs61a/su10/resources/sp11-Jordy/ddp.html
- …

The Pythonic Way: Calc

```python
from Number import *
import operator

stack = [] # List[Number]

def out():
    print(stack[-1])

def _op(op):
    num2 = stack.pop()
    num1 = stack.pop()
    stack.append(op(num1, num2))
    out()
```

```python
def add(): _op(operator.add)  # +
def sub(): _op(operator.sub)  # -
def mul(): _op(operator.mul)  # ...
def div(): _op(operator.truediv)

def num(val):
    stack.append(val); out()
def clr():
    stack.clear(); print()
```

The Pythonic Way: Number

```
# The ONE and ONLY supertype of all numbers
class Number(object):
    def __add__(n1, n2):      # +
        return dispatch("add", n1, n2)
    def __sub__(n1, n2):      # -
        return dispatch("sub", n1, n2)
    def __mul__(n1, n2):      # *
        return dispatch("mul", n1, n2)
    def __truediv__(n1, n2):  # /
        return dispatch("div", n1, n2)

# The GENERIC calculation engine:
# Simulates multiple dispatch using coercions
def dispatch(op, n1, n2):
    types = type(n1), type(n2)                   # Check types
    coercer = Coercer.get(types, types[0])   # Look up coercer
    # Perform coercer(n1).op(coercer(n2))    # Single dispatch
    #      or op(coercer(n1), coercer(n2))   # Multiple dispatch
    return getattr(coercer(n1), op)(coercer(n2)) # Pythonic

# Coercers
Coercer = dict()
```

The Pythonic Way: Integer

```
class Integer(Number):
    def __init__(n1, n2):     # Constructor as conversion
        if type(n2) == str:
            n1.value = int(n2)
        elif type(n2) == int:
            n1.value = n2
        else:
            raise Exception("Bizarre integer")
    def add(n1, n2):
        return Integer(n1.value + n2.value)
    ...
    def lt(n1, n2):
        return n1.value < n2.value
    def gcd(n1, n2):
        return Integer(math.gcd(n1.value, n2.value))
    def __str__(n1):
        return str(n1.value)
```

The Pythonic Way: Float

```
class Float(Number):
    def __init__(n1, n2):
        if type(n2) == str:
            n1.value = float(n2)
        elif type(n2) in [int, float]:
            n1.value = n2
        else:
            raise Exception("Bizarre Float")
        n1.value = round(n1.value, Number.digits)
    def add(n1, n2):
        return Float(n1.value + n2.value)
    ...
    def __str__(n1):
        value = n1.value
        if value.is_integer(): value = int(value)
        return str(value)

Coercer[(Integer, Float)] = Float
```

The Pythonic Way: Rational (1)

```
class Rational(Number):
    def __init__(n1, n2):
        if type(n2) == str:                 # "a/b"
            npair = n2.split("/")
            n1.numer, n1.denom = \
                abbrev(Integer(npair[0].lstrip().rstrip()),
                       Integer(npair[1].lstrip().rstrip()))
        elif type(n2) == tuple:             # (Integer, Integer)
            n1.numer, n1.denom = abbrev(n2[0], n2[1])
        else:
            raise Exception("Bizarre rational number")
    def add(n1, n2):
        return make_rat(n1.numer*n2.denom + n2.numer*n1.denom,
                        n1.denom*n2.denom)
    ...
    def __str__(n1):
        return str(n1.numer) + "/" + str(n1.denom)
```

The Pythonic Way: Rational (2)

```
def make_rat(numer, denom):   # (Integer, Integer) => Rational
    numer, denom = abbrev(numer, denom)
    if denom.eq(Integer("1")):
        return numer          # Reduce to an Integer
    return Rational((numer, denom))

def abbrev(numer, denom):     # Reduce numer & denom by their GCDs
    v_gcd = numer.gcd(denom)
    return numer.div(v_gcd), denom.div(v_gcd)

# Rational => Float
# Defined this because we can’t change implementation of Float
# to accommodate this promotion
def RatFloat(num):
    if type(num) == Rational:
        return Float(num.numer).div(Float(num.denom))
    return num

Coercer[(Integer, Rational)] = Rational
Coercer[(Rational, Float)]   = RatFloat
```

Using The Extended Calculator

```
# Computing 2/3 - 10 + 10.5
clr()
num(Integer("10"))              # 10
num(Rational("2/3"))            # 2/3
sub()                           # 28/3
num(Float("10.5"))              # 10.5
add()                           # 19.8333
```

How to cope with Changes

SOLID Design Principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

SOLID Principles
Open/Closed Principle

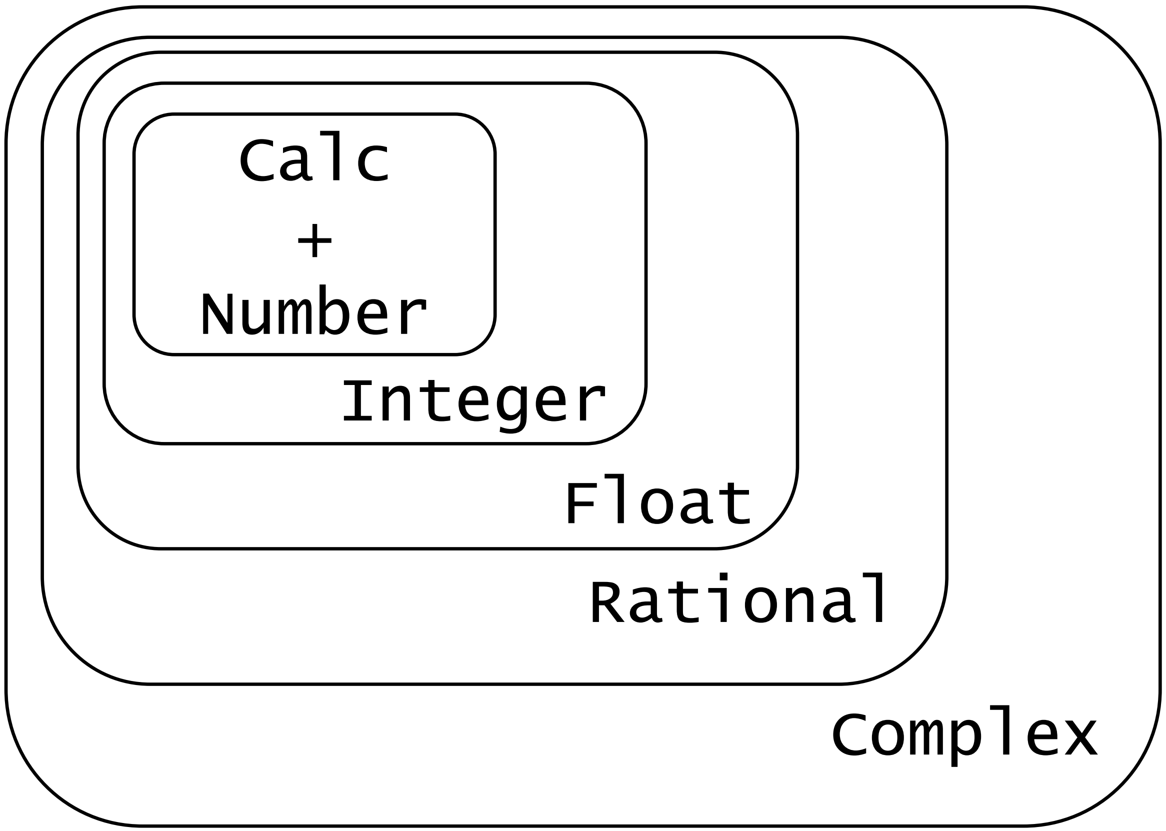
Every class should be open for extension, but closed for modification

- Write your classes in a generic way so that whenever you feel the need to extend the behavior of the class, then you shouldn't have to change the class itself. Rather, a simple extension of the class should help you build the new behavior.

Implications

- The chances of regression are less
- It also helps maintain backward compatibility

Example of Open/Closed Principle



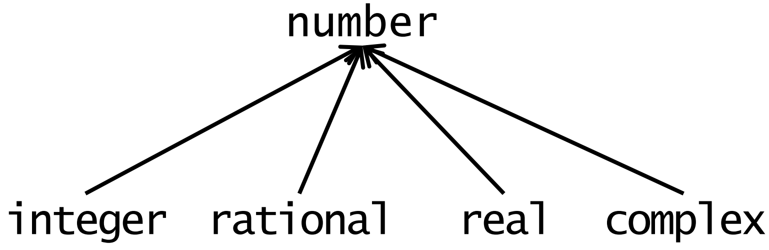
SOLID Principles
Liskov Substitution Principle

Derived classes must be able to completely substitute the base classes

- Every implementation of an interface needs to fully comply with the requirements of this interface
- Any algorithm that works on the interface, should continue to work for any substitute implementation

Implications

- Derived classes should just extend the base classes.



SOLID Principles
Dependency Inversion Principle

High-level modules shouldn't be dependent on low-level modules

- they should both be dependent on abstractions.

Rationale

- Low-level modules are more likely to change
- High-level modules are more likely to remain stable: they implement the business policies, which is the purpose of the system and is unlikely to change

Example of Dependency Inversion Principle