

CS339: Abstractions and Paradigms for Programming

Overview of Object-Oriented Programming

Manas Thakur
CSE, IIT Bombay



Autumn 2024

What all do you associate with OOP?

➤ Objects!

- Ability to **group** multiple items into a single abstraction
- Ability to **create multiple** objects of a certain kind
- Ability to **perform operations** on the object abstraction
- Ability to say that some objects are **like others** in some sense but different in their own ways
- Let's handle these abilities one by one.



1. Ability to group multiple items into a single abstraction

- Structures in C; Classes in C++/Java
- What about in Scheme?

```
(define (make-rat x y)
  (lambda (select)
    (if (= select 0) x y)))
```

- We already know how to create classes in Scheme!



2. Ability to create multiple objects of a certain kind

- Constructors in C++/Java
- Yes sir, we've already got 'em!

```
(define (make-rat x y)
  (lambda (which)
    (if (= which 0) x y)))

(define n1 (make-rat 2 3))

(define n2 (make-rat 3 4))
```

- Reckon that both n1 and n2 hold the values of different “fields” (again due to the existence of **closures!**).



3. Ability to perform operations on the object abstraction

➤ Functions/Methods in C++/Java

➤ In Scheme?

➤ *Yeh to pehli class se padha rahe hain!*

```
(define (make-rat x y)
  (lambda (which)
    (if (= which 0) x y)))
```

```
(define (numer n) (n 0))
(define (denom n) (n 1))
```

```
(define (mult-rat n1 n2)
  (make-rat (* (numer n1) (numer n2))
            (* (denom n1) (denom n2))))
```



Let's compare...

```
(define (make-rat x y)
  (lambda (which)
    (if (= which 0) x y)))

(define (numer n) (n 0))
(define (denom n) (n 1))

(define (mult-rat n1 n2)
  (make-rat (* (numer n1)
               (numer n2))
            (* (denom n1)
               (denom n2))))

(define n1 (make-rat 2 3))
(define n2 (make-rat 3 4))
(define n3 (mult-rat n1 n2))
```

```
class Rational {
  int x; int y;

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

  int numer() { return x; }
  int denom() { return y; }

  Rational mult-rat(Rational other) {
    return new Rational(
      this.numer() * other.numer(),
      this.denom() * other.denom());
  }
}

Rational n1 = new Rational(2,3);
Rational n2 = new Rational(3,4);
Rational n3 = n1.mult-rat(n2);
```



What all does our Scheme version lack?

➤ Packaging

- Encapsulation of the defined functions/methods into a module

➤ Dispatch on objects

- Aka message passing

- And the complete miss on the 4th bullet from Slide 2!

- We'll learn how to address all of these and more — in Scheme!

```
(define (make-rational x y)
  (lambda (which)
    (if (= which 0) x y)))
(define (numerator n) (n 0))
(define (denominator n) (n 1))
(define (mult-rat n1 n2)
  (make-rat (* (numerator n1)
               (numerator n2))
            (* (denominator n1)
               (denominator n2))))
(define n1 (make-rat 2 3))
(define n2 (make-rat 3 4))
(define n3 (mult-rat n1 n2))
```

```
class Rational {
  int x; int y;
  Rational(int x, int y) {
    this.x = x; this.y = y;
  }
  int numerator() { return x; }
  int denominator() { return y; }
  Rational mult-rat(Rational other) {
    return new Rational(
      this.numerator() * other.numerator(),
      this.denominator() * other.denominator());
  }
}
Rational n1 = new Rational(2,3);
Rational n2 = new Rational(3,4);
Rational n3 = n1.mult-rat(n2);
```



The rabbit asked the king, “Where shall I begin, please your Majesty?”.
The king replied gravely, “Begin at the beginning.”



Let's begin with complex numbers

- Two representations:

- Rectangular (real and imaginary)
- Polar (magnitude and angle)

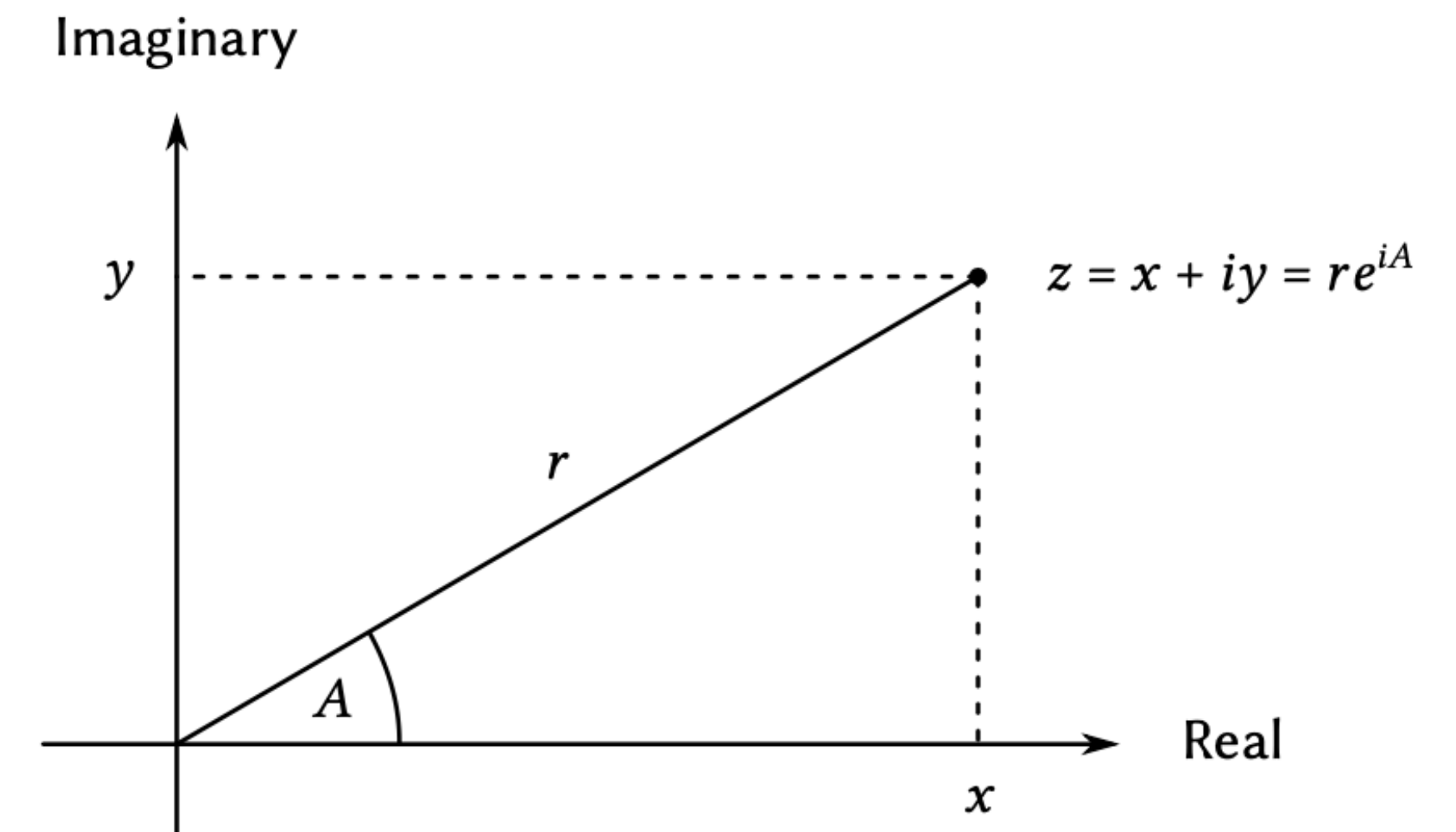
- Which one to choose?

- Addition/subtraction easier using rectangular representation:

$$\begin{aligned}\text{real-part}(z_1+z_2) &= \text{real-part}(z_1) + \text{real-part}(z_2) \\ \text{img-part}(z_1+z_2) &= \text{img-part}(z_1) + \text{img-part}(z_2)\end{aligned}$$

- Multiplication/division easier using polar representation:

$$\begin{aligned}\text{magnitude}(z_1*z_2) &= \text{magnitude}(z_1) * \text{magnitude}(z_2) \\ \text{angle}(z_1*z_2) &= \text{angle}(z_1) + \text{angle}(z_2)\end{aligned}$$



Let's choose both the representations

► Rectangular complex numbers:

```
(define (make-from-real-imag x y) (cons x y))
```

```
(define (make-from-mag-ang r a)  
  (cons (* r (cos a)) (* r (sin a))))
```

```
(define (real-part z) (car z))  
(define (imag-part z) (cdr z))
```

```
(define (magnitude z)  
  (sqrt (+ (square (real-part z))  
            (square (imag-part z)))))  
(define (angle z)  
  (atan (imag-part z) (real-part z)))
```



Let's choose both the representations

► Polar complex numbers:

```
(define (make-from-mag-ang r a) (cons r a))
```

```
(define (make-from-real-imag x y)  
  (cons (sqrt (+ (square x) (square y)))  
        (atan y x)))
```

```
(define (magnitude z) (car z))  
(define (angle z) (cdr z))
```

```
(define (real-part z) (* (magnitude z) (cos (angle z))))  
(define (imag-part z) (* (magnitude z) (sin (angle z))))
```



What about the operations?

- Do not depend on the representation!

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                        (+ (imag-part z1) (imag-part z2))))

(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                        (- (imag-part z1) (imag-part z2))))

(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))

(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

- Principle of **data abstraction**: Separate usage from representation.



But now we have a problem!

- We have two different selectors with the same name:

```
(define (real-part z) (car z))  
  
(define (real-part z) (* (magnitude z) (cos (angle z))))
```

- Which one should be called?

```
(define (add-complex z1 z2)  
  (make-from-real-imag (+ (real-part z1) (real-part z2))  
                        (+ (imag-part z1) (imag-part z2))))
```

- Topic for the next class!

