Oggetti Immutabili

Programmazione

Corso di Laurea in Informatica
Corso di Laurea in Informatica per la Comunicazione Digitale

Università di Camerino

Prof. Michele Loreti

Anno Accademico 2024/25

Caso di studio

Progettare una classe `Rational` per rappresentare **numeri razionali** come rapporto tra due numeri interi. Dotare la classe di metodi per realizzare le seguenti operazioni:

- leggere **numeratore** e **denominatore** di un numero razionale
- calcolare la **negazione** ed il **reciproco** di un numero razionale
- **semplificare** un numero razionale ai minimi termini
- calcolare **somma**, **differenza**, **moltiplicazione** e **divisione** di due numeri razionali
- convertire un numero razionale in `double` e in `String`

Osservazioni

- quando sommiamo $\frac{1}{2}$ e $\frac{2}{3}$ otteniamo $\frac{7}{6}$, ma i numeri sommati **continuano ad esistere**. Tutte le operazioni sopraelencate sono **non distruttive**
- notare la differenza con **oggetti mutabili** come recipienti, conti correnti, ecc.

Struttura della classe

```
public class Rational {
    private int numerator;
    private int denominator; // denominator > 0

    ... // da completare
}
```

Nota

- l'invariante di classe `denominator > 0` non è strettamente necessario, ma semplifica la realizzazione di alcune operazioni

Costruttori overloaded (1/2)

```
public Rational() {
    numerator = 0;
    denominator = 1;
}

public Rational(int numerator) {
    this.numerator = numerator;
    this.denominator = 1;
}
```

- definiamo **più costruttori** che differiscono per il numero degli argomenti. Questo meccanismo prende il nome di **overloading** e dà maggiore flessibilità in fase di creazione di un numero razionale
- Java eseguirà il costruttore corrispondente al numero di argomenti forniti al momento della creazione
- nel secondo costruttore il nome `numerator` è quello più appropriato per l'argomento, ma la scelta di questo nome "nasconde" il campo omonimo. Usiamo `this.numerator` per riferirci al campo `numerator` in presenza di ambiguità

Costruttori overloaded (2/2)

```
public Rational(int numerator, int denominator) {
    // denominator != 0;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
    this.numerator = numerator;
    this.denominator = denominator;
}
```

- il costruttore più generale accetta sia un numeratore che un denominatore
- nel caso in cui il denominatore fornito sia negativo, numeratore e denominatore vengono entrambi negati in modo da far valere l'invariante di classe preservando al tempo stesso il valore del numero razionale

Metodi getter e conversioni

```
    public int get_numerator() { return numerator; }
    public int get_denominator() { return denominator; }

    public double to_double() {
        return (double) numerator / denominator;
    }

    public String toString() {
        if (denominator == 1)
            return Integer.toString(numerator);
        else
            return numerator + "/" + denominator;
    }
```

- la conversione del numeratore in `double` evita il troncamento
- se il denominatore è 1 il numero è intero e lo convertiamo in stringa usando il metodo `Integer.toString` definito nella libreria standard di Java
- se il denominatore è diverso da 1 costruiamo una stringa della forma `m/n`. Siccome solo il numeratore può essere negativo, la stringa ottenuta ha la forma giusta

Operazioni unarie

```
public Rational negate() {
    return new Rational(-numerator, denominator);
}

public Rational reciprocal() {
    return new Rational(denominator, numerator);
}
```

- ogni operazione crea un **nuovo** numero razionale
- il costruttore generale (che gestisce il caso in cui il denominatore fornito è negativo) aiuta a semplificare `reciprocal`
- `reciprocal` può fallire se `numerator` è `0`

Somma e moltiplicazione

```
public Rational add(Rational x) {
    return new Rational(numerator * x.denominator +
                        denominator * x.numerator,
                        denominator * x.denominator);
}
public Rational mul(Rational x) {
    return new Rational(numerator * x.numerator,
                        denominator * x.denominator);
}
public Rational sub(Rational x) {
    return this.add(x.negate());
}
public Rational div(Rational x) {
    return this.mul(x.reciprocal());
}
```

- `this.add` invoca il metodo `add` sull'oggetto ricevente
- anziché riproporre codice simile a quello in `add` e `mul` riformuliamo sottrazione e divisione in termini di somma e moltiplicazione

Un piccolo programma di prova

```
public class TestRational {
    public static void main(String[] args) {
  Rational a = new Rational(1, 2); // 1/2
  Rational b = new Rational(2, 3); // 2/3
  System.out.println("A + B = " + a.add(b));
  System.out.println("A - B = " + a.sub(b));
  System.out.println("A * B = " + a.mul(b));
  System.out.println("A / B = " + a.div(b));
    }
}
```

```
A + B = 7/6
A - B = -1/6
A * B = 2/6
A / B = 3/4
```

Nota

- laddove l'operatore + è applicato a una stringa e a un oggetto, Java invoca implicitamente il metodo `toString` per ottenere la stringa corrispondente

Esercizi

1. Aggiungere un metodo `int compare(Rational y)` alla classe `Rational` tale che `x.compare(y)` restituisca `-1`, `0` oppure `1` a seconda che `x` sia rispettivamente più piccolo di, uguale a o più grande di `y`.
2. Aggiungere un metodo `Rational simplify()` alla classe `Rational` che semplifica la frazione con cui il numero razione è rappresentato ai minimi termini. Suggerimento: usare il metodo `mcd` definito in precedenza per ottenere il massimo comun divisore di numeratore e denominatore.
3. Progettare una classe `Complex` per rappresentare **numeri complessi** e dotarla di metodi per leggere parte reale e immaginaria e per eseguire le seguenti operazioni: somma, sottrazione, negazione, moltiplicazione, divisione, modulo, reciproco, coniugato.