



Alma Mater Studiorum-Università di Bologna

Scuola di Ingegneria

Fondamenti di Informatica T2

Lab04 – Da frazioni a insiemi di frazioni

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ROBERTA CALEGARI

Prof. AMBRA MOLESINI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Agenda

1. Da singole frazioni ad array di frazioni

- array sfruttato come supporto fisico per *insiemi* di frazioni
- prima, **Frazione** come ADT + **FrazLib** come libreria

2. **Frazione** come entità "double face"

- ADT+ libreria

3. Da "semplici array" a *collezioni di frazioni*

- oltre l'array come supporto fisico: i danni della trasparenza della rappresentazione interna
- un nuovo ADT **FractionCollection** per rappresentare *compiutamente* gli insiemi di frazioni con le loro proprietà (indipendentemente dall'uso sottostante di array...)

Agenda

1. Da singole frazioni ad array di frazioni

- array sfruttando i *insiemi di frazioni*
- prima, **Fr** **Lab04a-FrazioniBase** come libreria

2. **Frazione** come entità "double face"

- ADT+ libreria **Lab04b-FrazioniDoubleFace**

3. Da "semplici array" a *collezioni di frazioni*

- oltre l'array come supporto fisico: i danni della trasparenza della rapp **Lab04c-FractionCollection**
- un nuovo **per rappresentare** *computatamente* gli insiemi di frazioni con le loro proprietà (indipendentemente dall'uso sottostante di array...)



Oltre la classe Frazione

- La **classe Frazione** sviluppata finora offre metodi per operare su ***una singola frazione***
 - eventualmente su due (la seconda, come argomento)
- Possono però essere necessarie operazioni che lavorino su ***un insieme di frazioni***
 - Somma di un insieme di frazioni
 - Prodotto di un insieme di frazioni



Come operare su insiemi?

- **Cosa intendiamo per *insiemi* di Frazioni?**
 - per ora, *insiemi* = **array**
- **Come e dove** mettere queste operazioni?
 - potrebbero essere **invocati su una frazione** e **prendere in ingresso le altre N-1** da sommare/moltiplicare, ma:
 - sarebbero funzioni scomode e innaturali da usare
 - peggio: sarebbe un **approccio lontano dalla realtà dei fatti**
 - in un insieme, tutti gli elementi sono "alla pari"
 - con che criterio/diritto sceglierne uno come "destinatario" ?
 - *romperebbe l'uniformità del mondo reale*
 - oltre tutto, per farlo dovremmo **passare un array più corto, che dovremmo creare e copiare apposta: follia!**

© Ufficio Complicazione Cose Semplici



Ufficio complicazione cose semplici

- **Come verrebbe?**

- due **metodi** come questi:

```
public Frazione sum(Frazione[] altre) {  
    return new frazione somma di this e altre  
}  
  
public Frazione mul(Frazione[] altre) {  
    return new frazione prodotto di this e altre  
}
```

- da usare così (per sommare 1/2, 2/3, 1/5, ..):

```
Frazione f1 = new Frazione(1,2);  
Frazione res = f1.sum(array con solo 2/3, 1/5, ...)
```

ANCHE NO, grazie 😊

Chi fa cosa?

- Un metodo è la scelta giusta **quando è chiaro CHI debba svolgere un'operazione**
 - per sommare/moltiplicare una frazione con un'altra, è chiaro che puoi rivolgerti a una passando l'altra
 - ma per sommare/moltiplicare N frazioni in un array, *non c'è un "destinatario evidente"*
- Quando invece un'operazione coinvolge più entità e non è ovvio chi debba farla, **probabilmente non è nessuna di loro**
 - il "destinatario" reale è *un ente "terzo" che gestisca le N entità in modo uniforme*
 - "manager", libreria, etc.



Operare su insiemi – *revised*

- Nel nostro caso:
 - l'array contiene tante frazioni.. perché una dovrebbe essere "la prescelta"? Quale, poi?
 - *oltre tutto, tecnicamente ci complica le cose..*
- Molto meglio **far fare quell'operazione a qualcun altro**
 - in effetti, **il destinatario sarebbe "l'insieme di frazioni"** (cioè, qui, l'array), non *una specifica* frazione!
 - **MA non possiamo aggiungere un metodo alla classe []**
 - quindi, la soluzione è scegliere come **ente "terzo" una libreria**
→ **funzioni statiche** (come **Math** per **sin**, **cos**, **exp**, ...)



Libreria FrazLib

- Somma e Prodotto come metodi di libreria

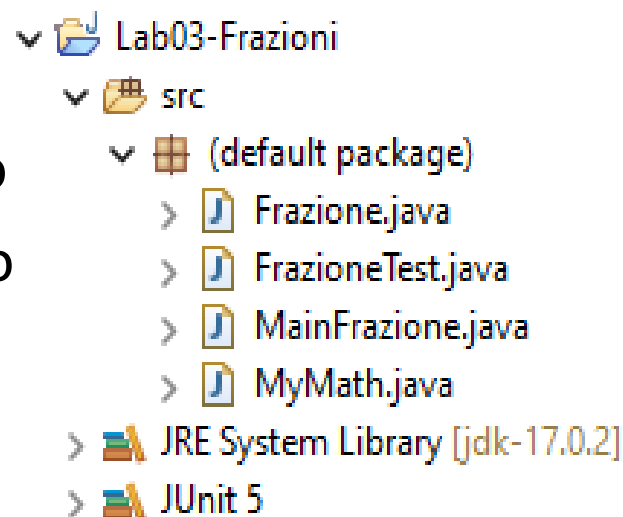
```
public class FrazLib {  
    public static Frazione sum(Frazione[] tutte) {  
        return new frazione somma di tutte  
    }  
    public static Frazione mul(Frazione[] tutte) {  
        return new frazione prodotto di tutte  
    }  
}
```

USO:

```
Frazione sum = FrazLib.sum(array di frazioni)  
Frazione prod = FrazLib.mul(array di frazioni)
```

Strutturazione di Applicazioni

- Nella precedente esercitazione abbiamo inserito tutte le nostre classi in un unico «pacchetto» che Eclipse chiama **«(default package) »**
- Ci era parsa una scelta sensata al momento
- MA **non è una scelta vincente** se dobbiamo lavorare con progetti di medie / grandi dimensioni (> 10 classi)
- Abbiamo bisogno **di iniziare a strutturare** le applicazioni per semplificare il lavoro e renderlo **più pulito e chiaro** sia per noi sia per chi in futuro dovrà lavorare sul codice





Strutturazione di Applicazioni

- Una applicazione complessa è tipicamente composta di *molte classi e librerie*
 - rischio di **conflitti di nome** (*name clash*)
 - necessità di *caratterizzare* gruppi di classi che costituiscono concettualmente un "*pacchetto software*"
- Necessità di uno **spazio di nomi strutturato**
 - ingestibilità di un insieme "piatto" di nomi
 - stesso problema dei nomi di file in un file system
- Costrutto **package** in Java

La teoria completa relativa ai **package** l'avete vista a lezione: qui richiamiamo solo alcuni concetti per essere operativi in Eclipse.

Package

- Un package introduce uno *spazio di nomi strutturato*, che può comprendere classi definite su file separati
- Convenzione Java: i package hanno nomi *minuscoli*
 - ESEMPLI: `fractionCollection`, `frazione`, etc
- Cosa significa *spazio di nomi strutturato*?
 - significa che per referenziare una classe si deve usare il suo *nome assoluto (strutturato)*, non più solo il nome *relativo*
 - ad esempio, se la classe `Frazione` (*nome relativo*) viene messa *nel package `frazione`* ha come *nome assoluto `frazione.Frazione`*

Package

- Come si usa un nome di classe strutturato?
 - semplicemente, scrivendolo per esteso:

```
it.unibo.utilities.Point p;  
p = new it.unibo.utilities.Point(x,y);
```
- Se non si specifica alcun nome di package, una classe appartiene al **default package**
 - è il caso delle classi che abbiamo definito fino ad oggi
 - il default package va *evitato il più possibile in pratica, perché le sue classi non hanno nome assoluto*
 - di conseguenza, è *impossibile usare tali classi da un altro package* perché sono "innominabili"

Package in Eclipse

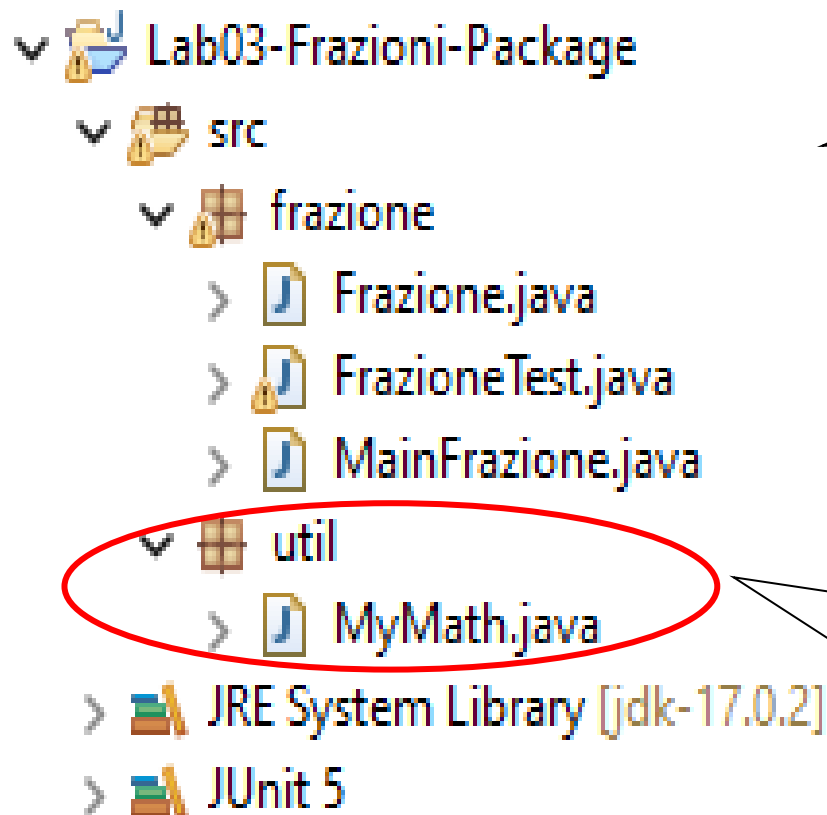
- ▼ Lab03-Frazioni
 - ▼ src
 - ▼ (default package)
 - > Frazione.java
 - > FrazioneTest.java
 - > MainFrazione.java
 - > MyMath.java
 - > JRE System Library [jdk-17.0.2]
 - > JUnit 5

Progetto organizzato sul
solo package di default

Nuovo progetto organizzato
su *due package, senza più
usare il default package*

- ▼ Lab03-Frazioni-Package
 - ▼ src
 - > frazione
 - > util
 - > JRE System Library [jdk-17.0.2]
 - > JUnit 5

Package in Eclipse



La nostra applicazione ora è ben strutturata e tutte le classi sono «*nominabili*»

Perché **MyMath** è in un package separato?

Perché non è legata alla nozione di **Frazione**!

È solo una classe con *funzioni di utilità* (mcd, etc.) *indipendenti* da frazione



Importazione di nomi (1)

- Però, i nomi strutturati (molto lunghi) sono *scomodi se la classe è usata spesso*.
- Si rimedia *importando i nomi pubblici* di un dato package o namespace nell'applicazione corrente
 - in Java: **direttiva *import***
- Ciò permette di *scrivere il nome relativo (corto) della classe invece del nome completo (lungo)*
 - ovviamente, si può fare solo se non ci sono omonimie
 - NB: la classe da importare *non dev'essere nel default package*, perché, dato che esso non ha nome, le sue classi sono "innominabili" e quindi non sono importabili altrove.



Importazione di nomi (2)

Per importare tutte le classi pubbliche di un package:

- in JAVA: import **frazione.*;**

Se serve una sola classe, si può importare solo quella:

- in Java: import **frazione.Frazione;**

In caso di omonimie:

Java permette una sola import: import **java.awt.Point**

L'altra classe si referenzierà usando il nome assoluto, ad esempio:

it.unibo.utilities.Point

Importazione di nomi (3)

Frazione.java X

```
1 package frazione;
2
3 import util.MyMath;
4
5 /**
6  * Frazione come tipo di dato astratto (ADT)
7  *
8  * @author Fondamenti di Informatica T-2
9  * @version MArch 2022
10 */
11 public class Frazione {
12     private int num, den;
13 }
```

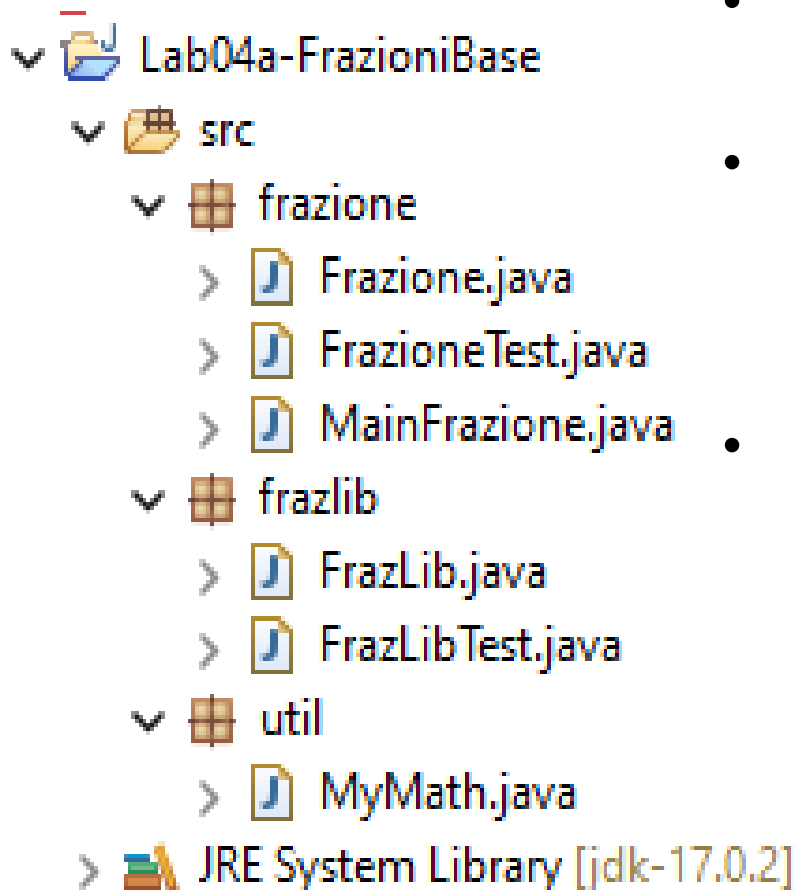
Package di cui **Frazione** fa parte

Il package importato

```
/**
 * Calcola la funzione ridotta ai minimi termini.
 *
 * @return Una nuova funzione equivalente all'attuale, ridotta ai minimi
 * termini.
 */
public Frazione minTerm() {
    if (getNum()==0) return new Frazione(getNum(), getDen());
    int mcd = MyMath.mcd(Math.abs(getNum()), getDen());
    int n = getNum() / mcd;
    int d = getDen() / mcd;
    return new Frazione(n, d);
}
```

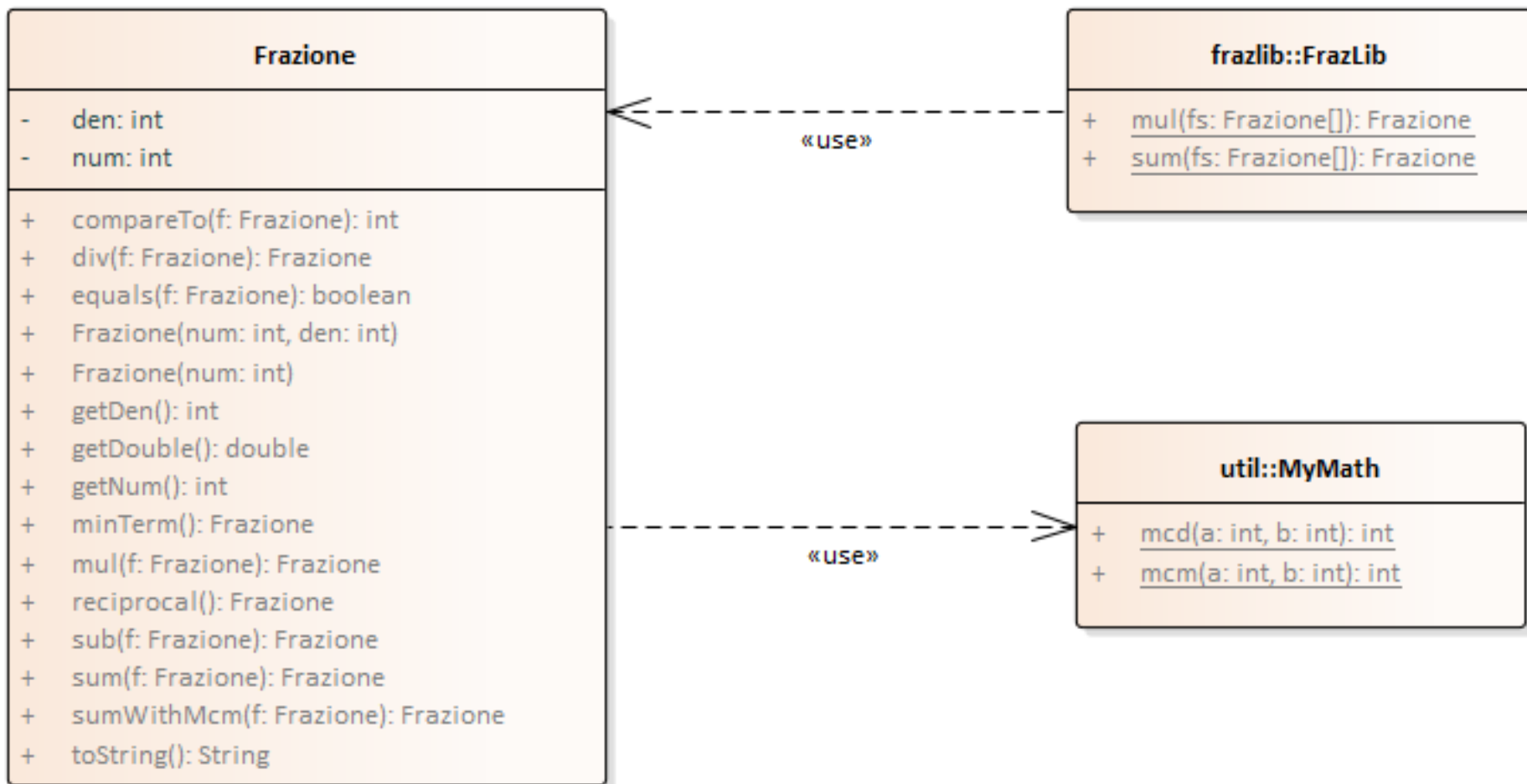
Non cambia rispetto alla versione della precedente esercitazione

La struttura di Lab04a



- Package **util**
 - contiene **MyMath**
- Package **frazione**
 - contiene le classi **Frazione**, **FrazioneTest**, **MainFrazione** sviluppate nella precedente esercitazione
- Package **frazlib**
 - contiene la libreria **FrazLib** con i metodi statici per somma e moltiplicazione di array di **Frazione** e la relativa classe di test **FrazLibTest**
 - Attenzione: **FrazLib** usa la classe **Frazione** del package **frazione** → necessaria direttiva **import**

Il modello





Primo Step Esercitazione

- Implementare la libreria **FrazLib**
 - `public static Frazione sum(Frazione[] tutte)`
 - `public static Frazione mul(Frazione[] tutte)`
- Progettare e scrivere la classe **FrazLibTest** per il collaudo della libreria
- Nessuno startkit fornito, strutturate il progetto come da esempio copiando la parti svolte dalla precedente esercitazione

Tempo a disposizione: 20 minuti



Un primo bilancio

Tutto a posto...?

- **Sì e no:**

- il progetto precedente è corretto, ma..
- *...relega le operazioni "collettive" in una libreria accessoria*

- **Bisognerebbe riconciliare le due esigenze**

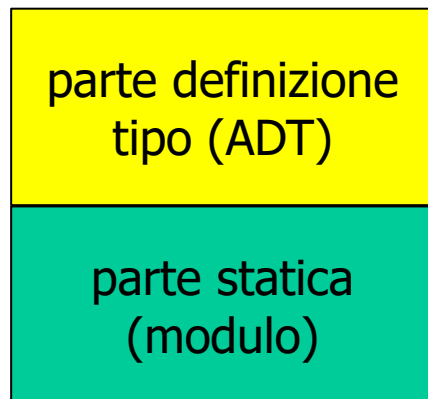
- operazioni standard come metodi della classe **Frazione**
- operazioni *su insiemi* come funzioni statiche di un ente terzo (qui, la classe accessoria **FrazLib**)

Vi ricorda niente..?

- c'era lo stesso problema con le stringhe!
- la classe **String** lo ha risolto *incorporando* la libreria
 - ADT coi suoi metodi + **libreria statica omonima con tante `valueOf`**

Riconciliare le esigenze

- Per superare questa "separazione" si può pensare di **riusare come "ente terzo" la stessa classe Frazione**, mettendoci dentro
 - sia le operazioni standard (metodi)
 - **sia le operazioni statiche su insiemi**



Finora, una classe svolgeva
uno solo dei due ruoli

ORA LI SFRUTTIAMO ENTRAMBI

La stessa classe:

- **definisce un tipo (Frazione)**
- **funge anche da componente statico "libreria per le frazioni"**



Un componente "bifronte"

- **Dai "separati in casa"...**
 - classe **Frazione** come tipo, definisce i metodi standard
 - classe accessoria **FrazLib** per i metodi statici su insiemi
- **..a un contenitore unico con doppio ruolo**
 - la **parte ADT** della classe **Frazione** definisce il tipo e fornisce i metodi standard
 - la **parte statica** della classe **Frazione** svolge il ruolo della libreria, definendo i metodi statici che operano su insiemi *senza bisogno di introdurre una classe extra*

Unitarietà nella diversità 😊



Nuova implementazione

```
public class Frazione {  
  
    private int num, den;  
    ...  
    public Frazione sum(Frazione f)  
    {  
        int n = this.num * f.den + this.den * f.num;  
        int d = this.den * f.den;  
        return new Frazione(n, d);  
    }  
  
    public Frazione mul(Frazione f)  
    {  
        int n = this.num * f.num;  
        int d = this.den * f.den;  
        return new Frazione(n, d);  
    }  
    ...  
}
```

In questa parte,
nessuna modifica



Nuova implementazione

```
...  
public static Frazione sum(Frazione[] tutte)  
{  
    ...  
}  
  
public static Frazione mul(Frazione[] tutte)  
{  
    ...  
}  
}
```

ex FrazLib



Nuove funzioni statiche (1/4)

- Oltre alle funzioni statiche già predisposte, che operano su più frazioni per calcolare **una** frazione-risultato:
 - `static Frazione sum(Frazione[] tutte)`
 - `static Frazione mul(Frazione[] tutte)`
- Potrebbe aver senso introdurne altre, come:
 - `static String convertToString(Frazione[] tutte)`
- nonché operazioni che operino su **due insiemi** di frazioni, generando come **risultato** un **nuovo insieme**:
 - `static Frazione[] sum(Frazione[] setA, Frazione[] setB)`
 - `static Frazione[] mul(Frazione[] setA, Frazione[] setB)`
 - NB: la dimensione dei due insiemi dev'essere identica, altrimenti...

Nuove funzioni statiche (2/4)

Versione molto inefficiente
(più tardi vedremo perché e come risolvere)

```
public static String convertToString(Frazione[] tutte) {  
    String res = "[";  
    for (int k=0; k<tutte.length && tutte[k] != null; k++){  
        res += tutte[k].toString() + ", ";  
    }  
    res += "];"  
    return res;  
}
```

Bruttino – stampa la virgola
anche alla fine..

```
public static Frazione[] sum(Frazione[] setA, Frazione[] setB) {  
    // 1. verificare consistenza dimensione logica setA e setB  
    // 2. creare nuova collezione risultato di equal dimensione  
    // 3. popolare tale nuova collezione sommando setA e setB  
}
```

idem per la moltiplicazione

MA... Come verificare uguaglianza dimensione logica?
Serve una *funzione ausiliaria* → **size**



Nuove funzioni statiche (3/4)

```
public static String convertToString(Frazione[] tutte) {
    String res = "[";
    for (int k=0; k<tutte.length && tutte[k]!= null; k++){
        res += tutte[k].toString() + ", ";
    }
    res += "]";
    return res;
}

public static Frazione[] sum(Frazione[] setA, Frazione[] setB) {
    // 1. verificare consistenza dimensione logica setA e setB
    // 2. creare nuova collezione risultato di equal dimensione
    // 3. popolare tale nuova collezione sommando setA e setB
}

public static int size(Frazione[] tutte) {
    

Dover scorrere l'array per conoscere  
la dimensione logica è inefficiente!


}
```



Nuove funzioni statiche (4/4)

```
public static String convertToString(Frazione[] tutte) {  
    String res = "[";  
    for (int k=0; k<tutte.length && tutte[k]!= null; k++){  
        res += tutte[k].toString() + ", ";  
    }  
    res += "];"  
    return res;  
}
```

```
public static Frazione[] sum(Frazione[] setA, Frazione[] setB) {  
    if (size(setA) != size(setB)) return null;  
    Frazione[] result = new Frazione[size(setB)];  
    for (int k=0; k<result.length; k++){  
        result[k] = setA[k].sum(setB[k]);  
    }  
    return result;  
}
```

Il solo modo che
abbiamo per lanciare
un allarme (per ora..)

Nota: la dimensione *logica e fisica*
sono identiche

Uso diretto di array di frazioni (1/5)

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[2] = new Frazione(-1, 2);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[4] = ...;  
        ...  
    }  
}
```

**Sono i tipici problemi di
*trasparenza della
rappresentazione***

Ci pensiamo dopo..

- È necessario riempire **in sequenza**?
- È opportuno riempire **in sequenza**?
- Hanno senso elementi «vuoti»?
- Come distinguere la dimensione «logica» da quella fisica?
- Cosa succede se si eccede l'indice massimo (qui, 9) ?
- Cosa succede se l'indice è negativo?

Uso diretto di array di frazioni (2/5)

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[2] = new Frazione(-1,2);  
        collezioneA[3] = new Frazione(1,6);  
        ...  
    }  
}
```

Dimensione logica = 4
Indici = 0..3 (forse...)

Scelte pratiche (per ora)

- riempimento in sequenza
- Il primo elemento vuoto (==null) indica la *dimensione logica*
- Gli indici effettivamente usati non usciranno da quel range (speriamo..)

Uso diretto di array di frazioni (3/5)

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[2] = new Frazione(-1,2);  
        collezioneA[3] = new Frazione(1,6);  
  
        Frazione[] collezioneB = new Frazione[10]; // iniz. vuoto  
        collezioneB[0] = new Frazione(1,5);  
        collezioneB[1] = new Frazione(2,8);  
        collezioneB[2] = new Frazione(1,7);  
        collezioneB[3] = new Frazione(-1,6);  
        collezioneB[4] = new Frazione(2,9);  
        ...  
    }  
}
```

Dimensione logica = 4

Dimensione logica = 5

E le operazioni?

Ha senso sommare/sottrarre.. *elemento per elemento*
collezioni **di dimensioni diverse?**



Uso diretto di array di frazioni (4/5)

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[2] = new Frazione(-1,2);  
        collezioneA[3] = new Frazione(1,6);  
  
        Frazione[] collezioneB = new Frazione[10]; // iniz. vuoto  
        collezioneB[0] = new Frazione(1,5);  
        collezioneB[1] = new Frazione(2,8);  
        collezioneB[2] = new Frazione(1,7);  
        collezioneB[3] = new Frazione(-1,6);  
        ...  
    }  
}
```

Dimensione logica = 4

Dimensione logica = 4

VINCOLO DI CONSISTENZA: stessa dimensione logica

(MA.. chi la controlla? A chi compete?)

Uso diretto di array di frazioni (5/5)

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[2] = new Frazione(-1,2);  
        collezioneA[3] = new Frazione(1,6);  
  
        Frazione[] collezioneB = new Frazione[10]; // iniz. vuoto  
        collezioneB[0] = new Frazione(1,5);  
        collezioneB[1] = new Frazione(2,8);  
        collezioneB[2] = new Frazione(1,7);  
        collezioneB[3] = new Frazione(-1,6);  
  
        Frazione[] somma =  
            Frazione.sum(collezioneA,collezioneB);  
        System.out.println(Frazione.convertToString(somma));  
    }  
}
```

Dimensione logica = 4

Dimensione logica = 4

Dimensione logica e fisica = 4



E il rispetto dei vincoli?

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
        collezioneA[2] = new Frazione(-1,2);  
        collezioneA[3] = new Frazione(1,6);  
  
        Frazione[] collezioneC = new Frazione[10]; // iniz. vuoto  
        collezioneC[0] = new Frazione(1,5);  
        collezioneC[1] = new Frazione(2,8);  
  
        Frazione[] somma =  
            Frazione.sum(collezioneA,collezioneB);  
        System.out.println(  
            Frazione.convertToString(somma));  
    }  
}
```

Dimensione logica = 4

Dimensione logica = 2

Oggetto nullo! OCCHIO!

Esplosione a run-time: *NullPointerException*
Si tenta di usare un oggetto che non esiste!

E il rispetto dei vincoli?

```
public class MyMain {  
    public static void main(String[] args) {  
        Frazione[] collezioneA = new Frazione[10]; // iniz. vuoto  
        collezioneA[0] = new Frazione(1,3);  
        collezioneA[1] = new Frazione(2,3);  
  
        Frazione[] collezioneB = new Frazione[10]; // iniz. vuoto  
        collezioneB[0] = new Frazione(1,3);  
        collezioneB[1] = new Frazione(2,8);  
  
        Frazione[] somma =  
            Frazione.sum(collezioneA,collezioneB);  
        System.out.println(  
            Frazione.convertToString(somma));  
    }  
}
```

CONTRATTO D'USO: è il cliente *che deve garantire la consistenza degli argomenti*. Se non lo fa → esplosione










Dimensione logica = 4

Dimensione logica = 2

Oggetto nullo! OCCHIO!

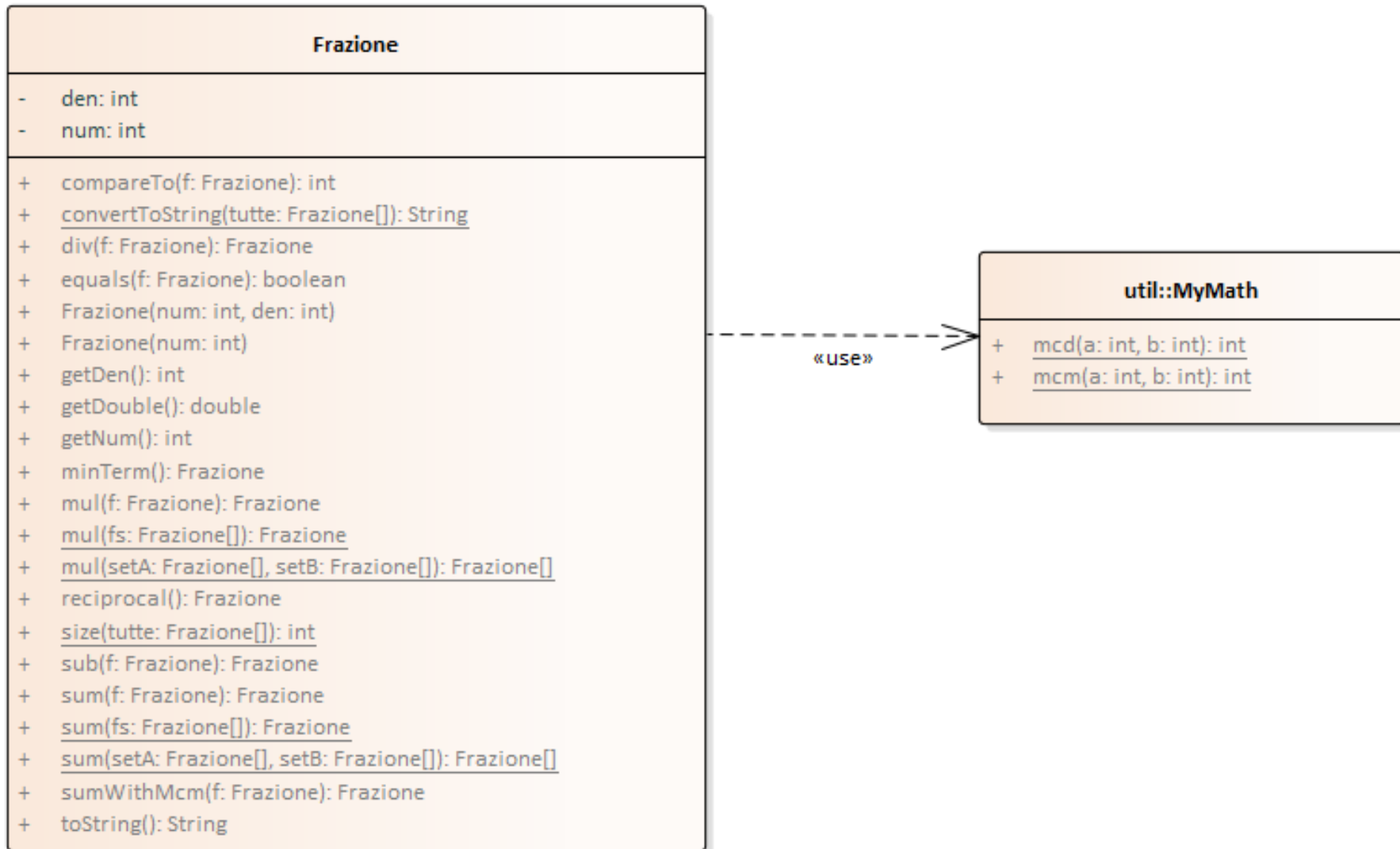
Esplosione a run-time: *NullPointerException*
Si tenta di usare un oggetto che non esiste!

La struttura di Lab04b

- ▼  Lab04b-FrazioniDoubleFace
 - ▼  src
 - ▼  frazione
 - >  Frazione.java
 - >  FrazioneTest.java
 - >  MyMain.java
 - ▼  util
 - >  MyMath.java
 - >  JRE System Library [jdk-17.0.2]

- Package **util**
 - contiene **MyMath**
- Package **frazione**
 - **Frazione** a cui sono stati aggiunti i metodi statici
 - **MyMain** (fornita) con un main di prova
 - **FrazioneTest** (da realizzare) che rappresenta la classe di test

Il modello





Secondo Step

- Aggiungere alla classe **Frazione** i metodi statici
 - `String convertToString(Frazione[] tutte)`
 - `int size(Frazione[] tutte)`
 - `Frazione[] sum(Frazione[] setA, Frazione[] setB)`
 - `Frazione[] mul(Frazione[] setA, Frazione[] setB)`
- Scrivere la classe di test **FrazioneTest** per il collaudo della classe
- La classe **MyMain** per provare le stampe la trovate pronta nello startkit

Tempo a disposizione: 30 minuti

Un primo bilancio

- **L'uso diretto di array non è un'idea meravigliosa**
 - certo, è immediato da scrivere: *tutte le operazioni su insiemi di frazioni diventano funzioni statiche di una qualche libreria*
 - **ma ingegneristicamente è pessimo:** *la trasparenza della rappresentazione* permette a chiunque di violare o abusare del contratto d'uso
 - il risultato è *software fragile*, che «si rompe» con poco
- **Noi vogliamo invece *software robusto***
 - ciò richiede di **non far trasparire all'esterno** le scelte interne, in modo che sia *impossibile «rompere» la consistenza del dato*

Richiede
incapsulamento

Verso l'ADT

"Collezione di frazioni"

Un nuovo obiettivo

Perché un nuovo progetto?

- Con le (pseudo-)collezioni di Frazioni sviluppate finora, si opera su insiemi di frazioni ma in modo *innaturale, contorto, non protetto*
 - array manipolati direttamente: *le "collezioni" esistono solo nella nostra mente*
 - inevitabile usare funzioni statiche: `fc3 = Frazione.sum(fc1, fc2)`
 - impossibile usarle in stile OOP: `fc3 = fc1.sum(fc2)`

Cosa vorremmo, invece?

- **Collezioni di Frazioni come ADT**
 - obiettivo: usarle in stile OOP: `fc3 = fc1.sum(fc2)`
 - per farlo, incapsuleremo l'array dentro a un nuovo tipo **FractionCollection**
 - l'array rimane, ma viene "degradato" a mero supporto fisico: non è più lui a dettare le scelte esterne, a stabilire come lo dovrà vedere e usare il cliente!
 - **FractionCollection** *modellare esternamente l'astrazione come desiderato*



Una nuova classe (1/2)

- **IOTESI: «collezione» = una nuova classe**
 - creare una nuova collezione → **costruttore**
(internamente creerà l'array di supporto)
 - aggiungere un elemento alla collezione → **metodo** apposito
(aggiunge all'array di supporto, se è il caso, crea un array più grande)
 - rimuovere un elemento dalla collezione → **metodo** apposito
(elimina un elemento dalla posizione specificata e ricompatta l'array)
 - ottenere la dimensione della collezione → **metodo** apposito
 - ottenere il valore dell'elemento i-esimo → **metodo** apposito
 - stampare la collezione (*come?*) → **metodo** apposito
 - sommare/sottrarre/moltiplicare/dividere → **metodi** appositi



Una nuova classe (2/2)

- **IOTESI: «collezione» = una nuova classe**

- creare una nuova collezione → **costruttore**
(internamente creerà l'array di supporto)
- aggiungere un elemento alla collezione → **metodo** apposito
(aggiunge all'array di supporto, se è il caso, crea un array più grande)
- rimuovere **Tutte le operazioni sulle collezioni diventano ora veri metodi (non funzioni statiche!) di questa classe** (elimina un array)
- ottenere la dimensione della collezione → **metodo** apposito
- ottenere il valore dell'elemento i-esimo → **metodo** apposito
- stampare la collezione (come?) → **metodo** apposito
- sommare/sottrarre/moltiplicare/dividere → **metodi** appositi



Obiettivo: da ex metodi statici....

```
public class Frazione {
```

```
    public static String convertToString(Frazione[] collection) {
```

```
        ...
```

```
    }
```

L'oggetto su cui si opera è passato come argomento alla funzione statica

```
    public static Frazione[] sum(Frazione[] coll1, Frazione[] coll2) {
```

```
        ...
```

```
    }
```

I due oggetti su cui si opera sono passati **entrambi** come argomenti alla funzione statica

```
    public static int size(Frazione[] collection) {
```

```
        ...
```

```
    }
```

L'oggetto su cui si opera è passato come argomento alla funzione statica



Obiettivo: ...a nuovo ADT

```
public class FractionCollection {  
    public String toString() {
```

L'oggetto su cui si opera **non è più** passato come argomento perché ora è il **target del metodo**

```
    }  
    public FractionCollection sum(FractionCollection coll) {
```

Dei due oggetti su cui si opera **uno solo è ancora passato esplicitamente** come argomento perché il primo ora è il **target del metodo**

Inoltre, sia l'argomento sia il risultato sono ora di tipo **FractionCollection**

```
    public int size() {  
    }
```

L'oggetto su cui si opera **non è più** passato come argomento perché ora è il **target del metodo**



FractionCollection

Specifiche di Dettaglio (1/2)

- **Costruttore 1** (un parametro intero)
Costruisce una collezione (logicamente vuota) data la dimensione fisica iniziale dell'array interno (parametro)
- **Costruttore 2** (nessun parametro)
Costruisce una collezione (logicamente vuota) con dimensione fisica iniziale dell'array di default (ad es. 10) *hardcoded* nel codice
- **Costruttore 3** (un parametro array di Frazione)
Costruisce una collezione dato un array di frazioni (parametro) che sarà **copiato** in quello interno
- **Metodo *size***
Restituisce la dimensione logica della collezione
- **Metodo *get***
Restituisce l'elemento i-esimo (parametro) della collezione



FractionCollection

Specifiche di Dettaglio (2/2)

- **Metodo *put***

Aggiunge un elemento in coda alla collezione e incrementa la dimensione logica; **se non c'è posto**: (1) crea un nuovo array grande il doppio del corrente, (2) vi copia tutte le frazioni del corrente (3) rende corrente il nuovo array

- **Metodo *remove***

Elimina un elemento dalla posizione i -esima (parametro) della collezione e, se necessario, compatta la collezione eliminando il "buco"

- **Metodo *toString***

Restituisce una stringa che rappresenta il contenuto della collezione; gli elementi sono racchiusi da parentesi quadre e separati con un carattere ',' (es.: "[1/3, 2/5, 3/2]")

- **Metodi *sum/sub/mul/div***

Eseguono somma/sottrazione/moltiplicazione/divisione con gli elementi di un'altra collezione di pari dimensione (parametro) e restituiscono una nuova collezione contenente i risultati.




Rappresentazione Interna

- Rappresentazione interna:
 - Un **array contenitore**
 - Un intero che rappresenta sia la **dimensione logica**, sia la **prima posizione libera** all'interno dell'array
- Metodo **put**
 - Se la **dimensione fisica** dell'array è **superiore** alla **dimensione logica**:
 - Aggiunge la frazione ricevuta nella **prima posizione libera**
 - Incrementa la **dimensione logica**
 - **Altrimenti**:
 - crea un nuovo array grande il doppio del corrente
 - vi copia tutte le frazioni contenute nel corrente
 - rende corrente il nuovo array
 - Aggiunge la frazione ricevuta nella **prima posizione libera**
 - Incrementa la **dimensione logica**

Attenzione a
fattorizzare il codice
in modo corretto!

Rappresentazione Interna

```
public class FractionCollection {  
    private Frazione[] innerContainer;  
    private int size;  
  
    public int size() {  
        return size;  
    }  
  
    public Frazione get(int k) {  
        return innerContainer[k];  
    }  
}
```



Chi controlla k? Se k è
negativo o «troppo
grande»?

- L'**accesso** alla rappresentazione interna è **mediato** da opportuni metodi.
- La **rappresentazione interna** è **completamente nascosta**: l'incapsulamento è completo!

FractionCollection.toString


- La **toString** della collezione può (deve) basarsi sulla **toString** di **Frazione**

```
@Override
public String toString() {
    String str = "";
    int num = getNum();
    int den = getDen();
    str += getDen() == 1 ? num : num + "/" + den;
    return str;
}
```



toString di Frazione

- Per la **toString** della collezione, la prima implementazione che viene in mente è quella che prevede di concatenare tutte le stringhe
 - Se la collezione è piccola, non c'è problema
 - Se la collezione è grande, la JVM **s'inginocchia**
- Il problema è che la stringa è un oggetto immutabile
 - la concatenazione produce sempre nuove stringhe**
 - le stringhe non più usate vengono recuperate dal *garbage collector*
 - Oltre un certo punto, il lavoro che il *garbage collector* deve compiere per tenere "pulita" la memoria diventa superiore al lavoro efficace (quello che il programma deve effettivamente compiere)
- Per risparmiare inutile lavoro al *garbage collector* si può usare lo **StringBuilder**



Gli oggetti costanti
non hanno solo
pregi...



Ladies and Gentlemen

The StringBuilder!

- Estratto da JavaDoc
 - A **mutable** sequence of characters. (...)
 - The principal operations on a **StringBuilder** are the **append** and **insert** methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string builder.
The append method always adds these characters at the end of the builder; the insert method adds the characters at a specified point.
- Si tratta di un oggetto che rappresenta una sequenza di caratteri **mutable**
- **Non è un oggetto costante** come le stringhe
- **Inserimenti e cancellazioni di caratteri sono ottimizzati**
- Da usare quando si compongono stringhe per **evitare di sovraccaricare il garbage collector**
- **Per ottenere una stringa (costante) contenente la sequenza di caratteri contenuta nello StringBuilder, usare il metodo toString()**

Andate a vedere come si usa!



FractionCollection

Come si usa?

```
public class CollectionMain {  
    public static void main(String[] args) {  
        Frazione[] array = new Frazione[10]; // iniz. vuoto  
        array[0] = new Frazione(1,3);  
        array[1] = new Frazione(2,3);  
        array[2] = new Frazione(-1,2);  
        array[3] = new Frazione(1,6);  
        FractionCollection collezioneA =  
            new FractionCollection(array);  
        array[0] = new Frazione(1,5);  
        array[1] = new Frazione(2,8);  
        array[2] = new Frazione(1,7);  
        array[3] = new Frazione(-1,6);  
        FractionCollection collezioneB =  
            new FractionCollection(array);  
        FractionCollection somma = collezioneA.sum(collezioneB);  
        System.out.println(somma.toString());  
    }  
}
```

È solo un supporto fisico temporaneo...

...quindi, se serve si può riusare

Metodi invocati su un oggetto target



FractionCollection

Come si usa?

```
public class CollectionMain {  
    public static void main(String[] args) {  
        FractionCollection collezioneA =  
            new FractionCollection(10);  
        collezioneA.put(new Frazione(1,3));  
        collezioneA.put(new Frazione(2,3));  
        collezioneA.put(new Frazione(-1,2));  
        collezioneA.put(new Frazione(1,6));  
        FractionCollection collezioneB =  
            new FractionCollection(12);  
        collezioneB.put(new Frazione(1,5));  
        collezioneB.put(new Frazione(2,8));  
        collezioneB.put(new Frazione(1,7));  
        collezioneB.put(new Frazione(-1,6));  
        FractionCollection somma = collezioneA.sum(collezioneB);  
        System.out.println(somma.toString());  
    }  
}
```

Da fuori, non si vedono
più gli array

**La dimensione fisica
iniziale è irrilevante!**
Si potrebbe fornire il
costruttore di default
che crea un array
interno di "dimensione
standard"

Collaudo

- Il collaudo è un'ottima forma di documentazione
 - Si tocca con mano il "come si usa l'oggetto target del test"
 - Si vedono esempi d'uso funzionanti
 - Si vedono esempi sul modo in cui l'oggetto target del test risponde a sollecitazioni errate o non previste (es. passaggio di parametri errati)
 - Non si può basare su uso indiscriminato di stampe a console ma va opportunamente ingegnerizzato ed automatizzato (JUnit)

Dove sono le mie stampe?!

Le tue stampe non ci sono più!!!



- Le stampe indiscriminate su console sono il **male**:
 - Andiamo verso un mondo modulare in cui le nostre classi possono vivere all'interno di applicazioni/contexti diversi
 - Applicazioni *window based*, *web based*, *mobile*, *servizi senza UI*, ...
- Le stampe vanno fatte nelle classi opportune, *non in quelle che rappresentano DATI come **Frazione** o **FractionCollection***
 - Queste classi dovrebbero limitarsi a fornire delle rappresentazioni in stringa che siano stampabili (**toString**)
 - ...poi qualcun altro sceglierà eventualmente dove stampare:
 - Una *text box* in una *app mobile*, la risposta di un *web service*, una *web page*, un *log file*, ecc..

La struttura di Lab04c

▼ Lab04c-FractionCollection

▼ src

▼ fractioncollection

> CollectionMain.java

> FractionCollection.java

> FractionCollectionTests.java

▼ frazione

> Frazione.java

> FrazioneTest.java

> MainFrazione.java

▼ util

> MyMath.java

> JRE System Library [jdk-17.0.2]

- Package **util**

- contiene **MyMath**

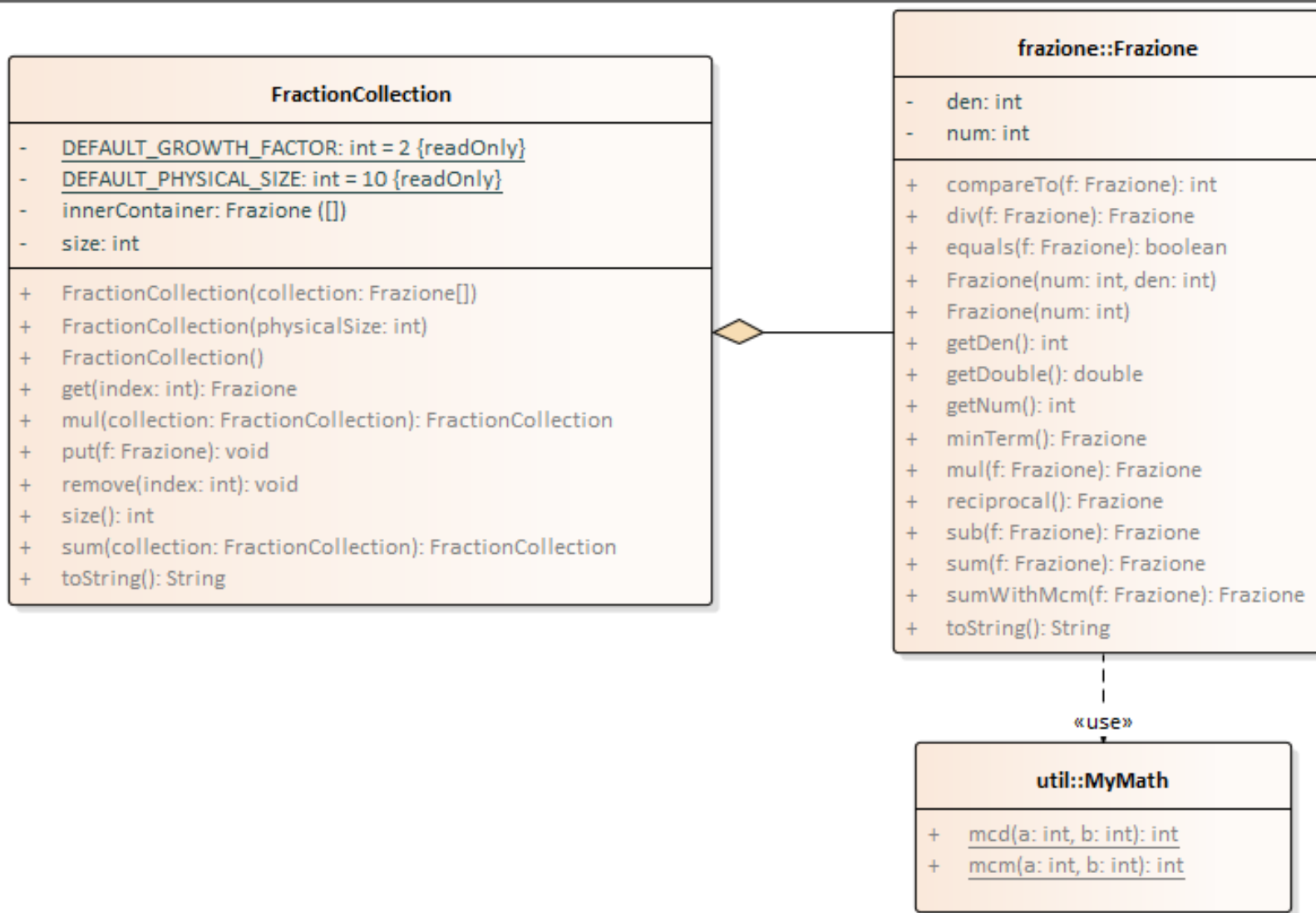
- Package **frazione**

- contiene **Frazione**, **FrazioneTest**, **MainFrazione** sviluppate nella precedente esercitazione

- Package **fractionCollection**

- **FractionCollection** il nuovo ADT da creare
- **CollectionMain** (fornita) con un main di prova
- **FractionCollectionTests** (fornita) che rappresenta la classe di test

Il modello





Terzo Step

- Realizzare collezioni di frazioni sfruttando gli array incapsulati
 - Scrivere la classe **FractionCollection** che realizza una collezione di Frazioni incapsulando l'array
 - Verificare il corretto funzionamento facendo girare sia la classe che contiene il main **CollectionMain** sia la classe di test **FractionCollectionTests** già pronte nello startkit

Tempo a disposizione: 1 h