

Série A : Exercices de récapitulation

Table des matières

Série A : Exercices de récapitulation.....	1
Exercice A.1 : Récapitulation des notions vues en 3 ^e	2
Exercice A.2 : Addition de deux fractions.....	14
Exercice A.3 : Voitures de sport.....	15
Exercice A.4 : Exercice de compréhension.....	16

Exercice A.1 : Récapitulation des notions vues en 3^e


Vous allez réaliser sous Unimozer la classe **Fraction** qui permet de représenter une fraction définie par un numérateur et un dénominateur.

Exemples: $\frac{1}{4}$, $\frac{2}{6}$, $\frac{-5}{7}$, $\frac{-3}{-9}$, $\frac{3}{1}$, $\frac{0}{6}$, $\frac{7}{0}$

Appeler le projet **Fraction**. Veuillez à indenter votre code correctement et à documenter la classe par l'utilisation de Javadoc.

Répondez sur papier aux questions marquées du symbole .

Classes et attributs

- 1)  Quelle est la différence entre la notion de classe et la notion d'objet (ou instance)? Donnez des exemples.

Une classe est une description formelle d'une collection d'objets de même identité. Il s'agit donc purement d'une description générale. Un objet est une instance d'une classe, c'est-à-dire que l'objet est créé dans la mémoire de l'ordinateur en suivant exactement les définitions contenues dans la classe. Tous les objets instanciés à partir d'une même classe possèdent le même fonctionnement.

- 2) Définissez sous Unimozer la classe **Fraction** et ajoutez les attributs nécessaires. Quelle visibilité définissez-vous pour ces attributs? Quels types de données attribuez-vous aux attributs de la classe **Fraction**?

```
public class Fraction
{
    private int numerator;
    private int denominator;
}
```

- 3)  Expliquez les visibilités **public** et **private**.

Un attribut ou une méthode de visibilité **public** peut être accédé directement (lu, modifié ou exécuté) aussi bien à l'intérieur de la classe qu'en dehors de la classe. Un attribut ou une méthode de visibilité **private** peut être accédé directement (lu, modifié ou exécuté) à l'intérieur de la classe mais pas en dehors de la classe.

- 4)  Remplissez le tableau avec les types de données que vous connaissez.

Type	Explication
byte	Nombre entier [-128, 127]
short	Nombre entier [-32768, 32767]
int	Nombre entier [-2147483648, 2147483647]
long	Nombre entier [-9223372036854775808, 9223372036854775807]
float	Nombre décimal

Type	Explication
double	Nombre décimal
boolean	Valeur logique vrai ou faux (<code>true</code> ou <code>false</code>)
String	Chaîne de caractères
void	vide (type des méthodes ne retournant pas de résultat)

Méthodes

Accesseurs (« Getters ») et manipulateurs (« Setters »)

- 5)  Pourquoi utilise-t-on des accesseurs et des manipulateurs ?

Les accesseurs sont des méthodes qui permettent d'accéder aux attributs à visibilité `private` d'une classe. Les « getters » permettent de lire la valeur d'un attribut, les « setters » de modifier la valeur d'un attribut.


- 6) Réalisez des accesseurs utiles pour la classe **Fraction**.

```
public int getNumerator()
{
    return numerator;
}

public int getDenominator()
{
    return denominator;
}
```

- 7) Réalisez un manipulateur utile pour la classe **Fraction**.


```
public void setFraction(int pNumerator, int pDenominator)
{
    numerator = pNumerator;
    denominator = pDenominator;
}
```

- 8)  Créez une instance de la classe **Fraction** représentant le quotient $\frac{1}{4}$.

Comment procédez-vous ? Qu'est-ce qui permettrait de simplifier cette tâche ?

- 1) Créer une instance de la classe
 - 2) Définir le numérateur et le dénominateur de la fraction (à l'aide de setters)
- La définition d'un constructeur pour la classe permet de simplifier cette tâche.

Constructeurs

- 9)  A quoi sert un constructeur ? Est-il nécessaire de définir un constructeur pour une classe donnée ? Une classe peut-elle avoir plusieurs constructeurs ?

Le « constructeur » est la méthode qui est appelée lors de la construction d'un objet. Il s'agit d'une méthode spéciale dont le rôle est d'initialiser les attributs et sous-objets de la classe afin de mettre l'objet dans son état initial.

Ce qu'il faut savoir à propos du constructeur :

- Une même classe peut posséder plusieurs constructeurs.
- Tout constructeur porte toujours le nom de la classe.
- Un constructeur n'a pas de type de retour.
- Un constructeur peut avoir des paramètres.
- La définition d'un constructeur n'est pas obligatoire – S'il n'y a pas de constructeur, alors Java utilise le constructeur par défaut (sans paramètres).

- 10) Réalisez un constructeur pour la classe **Fraction**.


```
public Fraction(int pNumerator, int pDenominator)
```

```

{
    numerator = pNumerator;
    denominator = pDenominator;
}

```

Conventions de noms

- 11)  Quelles conventions de noms sont utilisées en Java pour: les classes, les attributs, les méthodes, les paramètres et les variables locales ?


Classes: Les noms sont écrits en lettres minuscules. La première lettre est en majuscule. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules.

Attributs, méthodes, paramètres et variables locales : Les noms sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot dont la première lettre est en minuscule. Il est recommandé de faire précéder les paramètres de la lettre "p".

La structure alternative

- 12)  Donnez un exemple d'une expression booléenne (condition).


```
a < 0
```

- 13)  Indiquez la condition qui permet de vérifier que la variable **x** contient un nombre compris dans l'intervalle [1, 60].

```
(x >= 1) && (x <= 60)
```

Remarque: En Java les opérateurs booléens && et || ont une priorité inférieure aux opérateurs de comparaison (==, !=, <, >, ...). Par conséquent, l'expression **x > 1 && x <= 60**

est correcte mais l'utilisation de parenthèses est préférable pour bien délimiter les conditions simples et ainsi rendre le code plus lisible.

- 14)  Indiquez la valeur de la variable **x** après l'exécution des lignes de code suivantes :

```

int x=-5;
if (x < 0)
    x = -x;
else;
    x = 2 * x;

```

x vaut 10.

Attention: le point-virgule après le else termine la structure alternative

- 15) Réalisez la méthode **toString** qui retourne la représentation d'une fraction sous forme de chaînes de caractères.

Exemples :

Pour la fraction $\frac{4}{9}$ la méthode retourne "4 / 9"

Pour la fraction $\frac{6}{1}$ la méthode retourne "6"

```
public String toString()
{
    if (denominator !=1)
        return numerator + " / "+denominator;
    else
        return numerator +""; //forcer le type string
}
```

- 16) Réalisez la méthode **getDecimal** qui retourne la représentation décimale de la fraction.

```
public double getDecimal()
{
    return (double)numerator / denominator;
}
```


- 17) Réalisez la méthode **isNegative** qui indique si la fraction représente un nombre négatif ou non. Veillez à définir le type de données adéquat pour le résultat de la méthode !

```
public boolean isNegative()
{
    if (getDecimal() < 0)
        return true;
    else
        return false;
}
```


ou plus simplement :

```
public boolean isNegative()
{
    return getDecimal() < 0;
}
```

La structure répétitive (boucles)

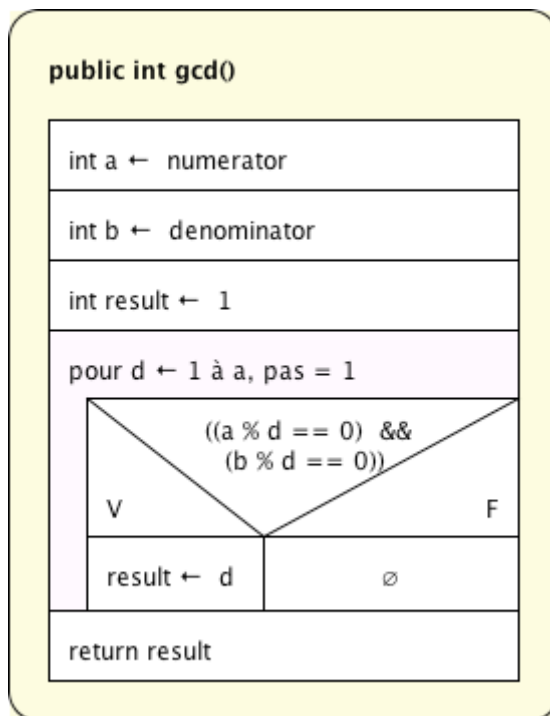
- 18)  Indiquez le code qui permet d'afficher à la console les nombres de 1 à 10 (par ordre croissant). Utilisez une boucle **for** !

```
for (int i=1; i<=10; i++)
    System.out.println(i+"");
```

- 19)  Indiquez le code qui permet d'afficher à la console les nombres de 10 à 1 (par ordre décroissant). Utilisez une boucle **for** !

```
for (int i=10; i>=1; i--)
    System.out.println(i+"");
```

- 20) Réalisez la méthode **gcd** (pour *greatest common divisor*) qui calcule le **plus grand commun diviseur** (ou PGCD) du numérateur et du dénominateur d'une fraction en traduisant le structogramme suivant :



solution: voir point 22

- 21) Dessinez un tableau des variables (tableau d'exécution) pour l'appel à **gcd** pour chacune des fractions suivantes :

i. 4 / 6

a	b	result	d	(a % d == 0) && (b % d == 0)
4	6	1		
		1	1	true
		2	2	true
		2	3	false
		2	4	false
		<u>2</u>	5	

ii. 6 / 4

a	b	result	d	(a % d == 0) && (b % d == 0)
6	4	1		
		1	1	true
		2	2	true
		2	3	false
		2	4	false
		2	5	false
		2	6	false
		<u>2</u>	7	

iii. -4 / 6

a	b	result	d	(a % d == 0) && (b % d == 0)
-4	6	<u>1</u>		

- 22) Modifiez la méthode **gcd** pour qu'elle fonctionne avec des nombres négatifs.

```
public int gcd()
{
    // Adaptation pour que la méthode fonctionne
}
```



```
..... // avec des nombres négatifs
.....
..... int a = Math.abs(numerator);
..... int b = denominator;
..... int result = 1;
..... for (int d = 1; d <= a; d++)
.....     if ((a % d == 0) && (b % d == 0))
.....         result = d;
..... return result;
..... }
```

- 23) Indiquez quand on utilise une boucle **for** et quand on utilise une boucle **while**.

Une boucle **for** est utilisée quand le nombre d'itérations (*Durchgänge*) est connu d'avance. Si le nombre d'itérations n'est pas connu alors on utilise une boucle **while**.

- 24) Donnez un exemple d'une boucle vide (aucune itération) et d'une boucle infinie (infinité d'itérations).

Boucle vide :

```
for (int i=1; i<1; i++)
    System.out.println("Hello");
```

OU

```
while (false)
    System.out.println("Hello");
```

- 25) En analysant les deux premiers cas d'exécution de la méthode **gcd** du point 21, réalisez une méthode **gcd** plus performante.

```
public int gcd()
{
    int a = Math.abs(numerator);
    int b = Math.abs(denominator);

    // Déterminer le minimum de a et b pour éviter des
    // itérations inutiles dans la boucle while
    int min = a;
    if (b<a)
        min=b;
    //ou aussi : Math.min(a,b);

    // En utilisant une boucle while descendante
    // on peut s'arrêter dès qu'on a trouvé un diviseur commun
    while ((a % min != 0) || (b % min != 0))
        min--;
    return min;
}
Attention, la condition a été inversée logiquement d'après la loi 'de Morgan' (→ voir cours de 11e)
```

- 26) Réalisez une méthode **gcd** encore plus performante en utilisant la méthode de la division euclidienne.

Exemple :

PGCD(42, 27) = 3

a	b	a % b
42	27	15
27	15	12
15	12	3
12	3	0 => on a trouvé le résultat

```

public int gcd()
{
    int a = Math.abs(numerator);
    int b = Math.abs(denominator);
    int help;
    while (a%b != 0)
    {
        help = a % b;
        a = b;
        b = help;
    }
    return b;
}

```

- 27) Réalisez la méthode **lcm** (pour *least common multiple*) qui permet de calculer le plus petit commun multiple (ou PPCM) du numérateur et du dénominateur d'une fraction sachant que:

$$PPCM(a, b) = \frac{|a \cdot b|}{PGCD(a, b)}$$

```

public int lcm()
{
    return Math.abs(numerator*denominator)/gcd();
}

```

- 28) Réalisez la méthode **reduce** qui permet de simplifier la fraction. Veillez à ce que la méthode fonctionne dans tous les cas !

```

public void reduce()
{
    if (denominator != 0)
    {
        int g=gcd();
        numerator = numerator / g;
        denominator = denominator / g;
    }
}

```

Attention : IL FAUT utiliser une variable pour le pgcd, sinon le dénominateur est divisé par un autre gcd que le numérateur... ==> résultat incorrect!

- 29) Réalisez la méthode **reciprocal** qui inverse la fraction.

Par exemple : la fraction $\frac{5}{11}$ est transformée en $\frac{11}{5}$

```

public void reciprocal()
{
    int temp = numerator;
    numerator = denominator;
    denominator = temp;
}

```

Attention : IL FAUT utiliser une variable temporaire pour permuter le numérateur et le dénominateur.

Utilisation d'une classe comme paramètre

- 30) Réalisez la méthode **add** qui permet d'additionner à la fraction une fraction passée comme paramètre. Réalisez également les méthodes **subtract**, **multiply** et **divide**.

```

public void add(Fraction pFract)
{
    numerator    = numerator    * pFract.denominator +
                    denominator * pFract.numerator;
    denominator = denominator * pFract.denominator;
    reduce();
}

public void subtract(Fraction pFract)
{
    numerator    = numerator    * pFract.denominator -
                    denominator * pFract.numerator;
    denominator = denominator * pFract.denominator;
    reduce();
}

public void multiply(Fraction pFract)
{
    numerator    = numerator    * pFract.numerator;
    denominator = denominator * pFract.denominator;
    reduce();
}

public void divide(Fraction pFract)
{
    numerator    = numerator    * pFract.denominator;
    denominator = denominator * pFract.numerator;
    reduce();
}

```

Pour avancés

31) Vous voulez pouvoir créer des fractions à partir de nombres décimaux.

Exemple: $0,25 \Rightarrow \frac{1}{4}$

Comment pouvez-vous procéder ? Effectuez les réalisations nécessaires.

Indications:

- Procédez par étapes ! Exemple: $0,25 \Rightarrow \frac{25}{100} \Rightarrow \frac{1}{4}$
- Pour obtenir la partie entière d'un nombre décimal de type **double** vous pouvez utiliser la conversion suivante : **(int) decimal**
- Testez votre constructeur avec beaucoup de valeurs différentes. Est-ce qu'il fonctionne toujours correctement? Si non essayez de déterminer une explication ou même une solution.

Nous réalisons un deuxième constructeur pour la classe **Fraction**:

```

public Fraction(double pDecimal){
    int d=1;
    while ( (int)pDecimal != pDecimal ) {
        pDecimal = pDecimal * 10;
        d = d * 10;
    }
    numerator    = (int)pDecimal;
    denominator = d;
    reduce();
}

```

Problèmes:

- dépassement du type int possible
- imprécision de la représentation de double dans la mémoire:
p.ex. $75.32 * 10 \rightarrow 753.1999999999999999$
ainsi on peut obtenir des boucles infinies, avec dépassement.

Solution possible:

- Ajouter des conditions pour éviter le dépassement :

```
public Fraction(double pDecimal) {  
    long d = 1; //conditions pour éviter le dépassement  
    while (pDecimal - (int) pDecimal != 0 &&  
           d * 10 < Integer.MAX_VALUE &&  
           Math.abs(pDecimal * 10) < Integer.MAX_VALUE )  
    {  
        pDecimal = pDecimal * 10;  
        d = d * 10;  
    }  
    numerator = (int) pDecimal;  
    denominator = (int) d;  
    reduce();  
}
```

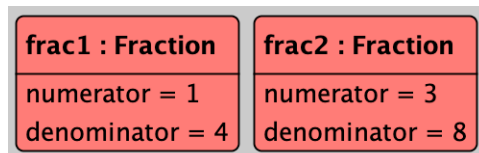
Exercice A.2 : Addition de deux fractions

Copier la solution de l'exercice A.1 et renommer le projet (le dossier) en **A02_Addition_Fractions_new**.

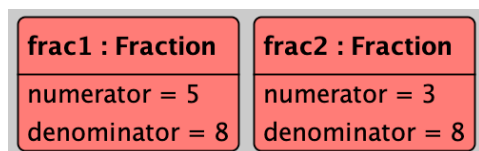
Dans cet exercice vous allez créer des objets de la classe **Fraction** à l'aide de l'instruction

new et effectuer l'addition suivante : $\frac{1}{4} + \frac{3}{8}$

1. Créez manuellement dans Unimozer des objets de la classe **Fraction** représentant les deux fractions.

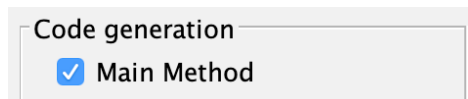


Utilisez ensuite la méthode **add** de la fraction **frac1** pour additionner la fraction **frac2**.



2. Maintenant vous allez programmer les actions que vous avez effectuées sous le point 1.

Ajoutez tout d'abord la classe **Test** au projet. Lors de la création de la classe sélectionnez dans le dialogue l'option de génération d'une méthode **main**.



C'est dans cette méthode que vous allez effectuer vos développements.

```
public static void main(String[] args)
{
}
}
```

Ajoutez dans cette méthode le code effectuant les mêmes actions que sous le point 1. en déclarant et en utilisant les variables **frac1** et **frac2**.

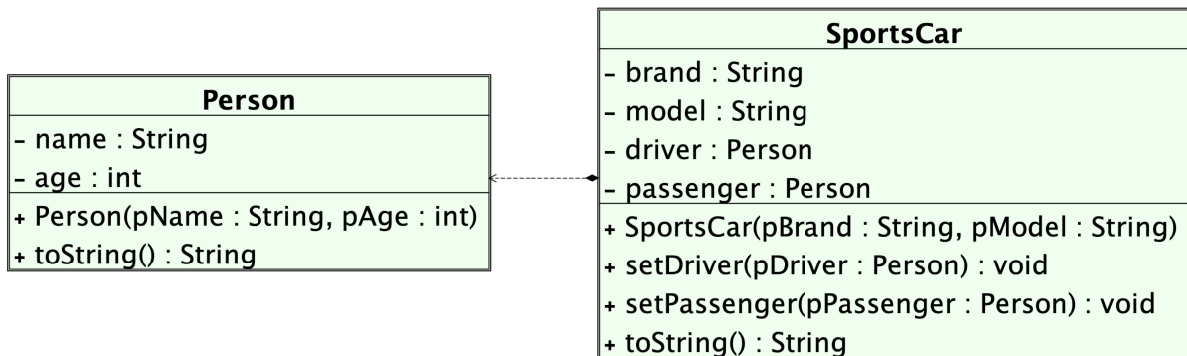
Vous pouvez utiliser la méthode **System.out.println()** pour afficher le contenu des variables dans la console ou la méthode **Unimozer.monitor()** pour afficher les objets des variables dans le panneau des objets (indiquer la variable et son nom comme arguments p.ex. **Unimozer.monitor(frac1, "frac1")**).

3. Représenter l'exécution de la méthode à l'aide d'un tableau des variables en y dessinant les objets.

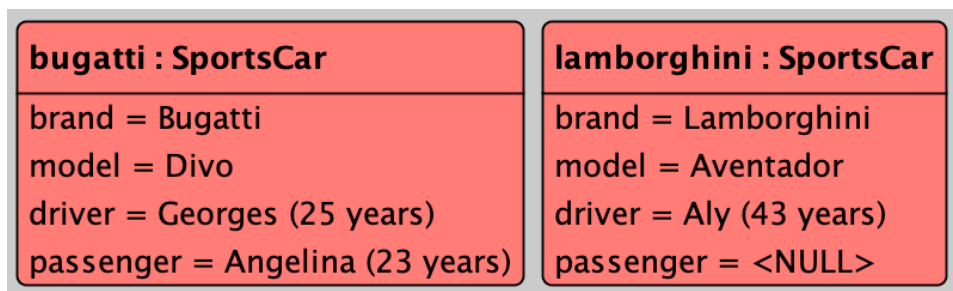
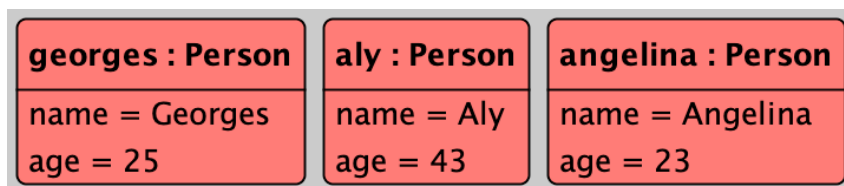
Notions requises : Création de nouveaux objets avec l'instruction **new**. Tableaux des variables et diagrammes objet.

Exercice A.3 : Voitures de sport

Ouvrez le projet **A03_Sportscars**. Ce projet contient les deux classes suivantes :



1. Créez une classe test et développez la méthode pour créer les objets suivants :



2. Représenter l'exécution de la méthode à l'aide d'un tableau des variables en y dessinant les objets.

Notions requises : Création de nouveaux objets avec l'instruction **new**. Tableaux des variables et diagrammes objet.

Exercice A.4 : Exercice de compréhension

Voici le code d'une méthode créant des instances (objets) de la classe **Fraction**.

```
public static void main(String[] args)
{
    Fraction f1;

    f1 = new Fraction(0.75);

    System.out.println("1: f1="+f1.toString());

    Fraction f2 = new Fraction(4, 12);

    System.out.println("2: f2="+f2.toString());

    f1 = f2;

    System.out.println("3: f1="+f1.toString()+" ,f2="+f2.toString());

    f1.reduce();

    System.out.println("4: f1="+f1.toString()+" ,f2="+f2.toString());

    f2 = null;

    System.out.println("5: f2="+f2.toString());

}
```

Complétez le tableau des variables suivant en dessinant les objets de la classe **Fraction**.

Notions requises : Création de nouveaux objets avec l'instruction **new**. Références vers les objets. Tableaux des variables et diagrammes objet.